Self-Evolving Coding Rules for AI Coding Agents

Zhengyuan Jiang
Duke University
zhengyuan.jiang@duke.edu

Reachal Wang
Duke University
reachal.wang@duke.edu

Yuepeng Hu
Duke University
yuepeng.hu@duke.edu

Yupu Wang
Duke University
yupu.wang@duke.edu

Yuqi Jia
Duke University
yuqi.jia@duke.edu

Neil Zhenqiang Gong
Duke University
neil.gong@duke.edu

Abstract

The performance of AI coding agents is highly dependent on their underlying coding rules. However, existing coding rules are typically hand-crafted and fixed, making the process labor-intensive and often suboptimal. In this work, we propose *RuleEvolve*, a self-evolving framework for coding rules. RuleEvolve maintains a pool of candidate coding rules and iteratively improves them. In each iteration, it employs an LLM-powered *mutator module* to generate variants from existing candidates, and then uses a *judge module* to evaluate these variants and update the pool with the best-performing ones. Extensive evaluations across two coding-agent frameworks, four backbone LLMs, and three benchmarks demonstrate that RuleEvolve outperforms both manual engineering and existing prompt optimization baselines in terms of functional correctness of the generated code, code length, and/or generation cost (e.g., tokens used). Our code is publicly available: https://github.com/zhengyuan-jiang/RuleEvolve.

1 Introduction

The evolution of AI coding agents, such as OpenHands [23], OpenCode [2], Claude Code [3], and Cursor [8], has fundamentally redefined code generation tasks by shifting the focus from manual implementation to high-level orchestration. Central to the performance of these agents is the coding rules file, often instantiated as an `AGENTS.md` document and prepended to the backbone model's input context [1]. These rules play a critical role in determining the correctness and length of the generated code, as well as the associated generation cost.

Despite the critical importance of these rules, they are currently almost exclusively hand-crafted by human developers [19]. This manual process is inherently labor-intensive and suboptimal, as human-written rules often lack task-specific alignment and can even degrade agent performance [11].

Although automatic techniques [24] for prompt optimization have been developed, these methods remain difficult to directly apply to self-evolving coding rules. Existing approaches primarily target short, few-sentence instructions, whereas coding rules are typically long, well-documented guidance that present a significantly larger search space. Furthermore, the resource-intensive nature of coding agents renders previous random mutation strategies computationally prohibitive. Consequently, there is a critical need for a framework that can automatically evolve complex coding rules.

To bridge this gap, we propose RuleEvolve, a self-evolving framework for coding rules. The overall pipeline is illustrated in Figure 1. RuleEvolve takes as input a set of coding tasks and aims to evolve coding rules that can generalize to improve code generation on other tasks. Specifically, RuleEvolve maintains a pool of m candidate coding rules and iteratively evolves them. In each iteration, it employs an LLM-powered *mutator module* to generate N mutations from the current candidates.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Each mutation is then evaluated by a *judge module*, which assesses performance in terms of code correctness, code length, and generation cost. Based on these evaluations, RuleEvolve updates the pool by retaining the top-performing m candidates.

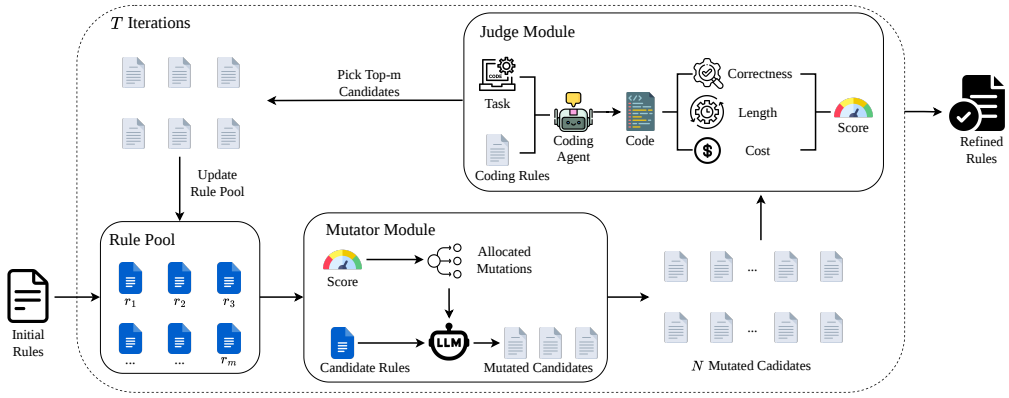


Figure 1: Pipeline of RuleEvolve.

One challenge is how to allocate the N mutations among the m candidates in the pool in each iteration. Naive strategies, which distribute mutations equally or proportionally based on each candidate's performance, produce suboptimal results, as demonstrated in our experiments. To enhance effectiveness, we propose a strategic mutation-allocation method, which dynamically distributes the N mutations by prioritizing candidates with higher performance variance or those whose mean performance is closer to the current best.

We evaluate RuleEvolve across two coding-agent frameworks, four backbone LLMs, and three coding benchmarks. Extensive experiments show that the proposed method consistently maintains or improves functional correctness measured by pass rate, while significantly reducing code length and generation cost, as measured by generation time and token usage. For instance, on the BigCodeBench dataset using the OpenAI SDK with gpt-5.3-codex, RuleEvolve successfully reduced average code length by over 60% without sacrificing pass rate.

2 Related Works

2.1 AI-assisted Code Generation

The landscape of software engineering has been transformed by AI-assisted code generation. To evaluate these AI-generated code, the research community has established several coding benchmarks, such as HumanEval [6] and BigCodeBench [29], which assess the ability of models to generate executable and functional code across a variety of programming tasks. Recent works [17, 27] have sought to enhance the quality of generated code through prompting strategies. However, these efforts primarily focus on the performance of the underlying coding LLM in isolation rather than within the agentic framework. Furthermore, these studies focus on the optimization of task-specific prompts to the coding LLM, without considering the self-evolving of the coding rules that guide the code generation process universally.

2.2 Coding Rules for AI Coding Agents

AI coding agents rely on coding rules, typically stored in an `AGENTS.md` file, to guide the generation of code. These rules and guidance are combined with the system and user prompts to serve as input for the backbone LLM. The specific rules provided influence the structure and style of the generated code. While recent works [10, 11, 7] have explored the impact of these rules on code generation, these studies primarily focus on measurement without providing practical insights into how these coding rules can be systematically improved or evolved.

2.3 Self-evolving Agents

Existing studies on self-evolving agents primarily focus on models, tools, skills, and memory [26, 20, 21, 28, 15]. While coding rules are central to coding agents, their self-evolving remains largely unexplored. Automatic prompt optimization techniques [24, 14, 22, 9, 18, 16, 4, 12] can theoretically be applied to optimize coding rules; however, these methods primarily target short prompts consisting of only a few sentences. In contrast, coding rules are extensive, well-documented sets of instructions that create a significantly larger search space beyond the capability of existing optimizers. Consequently, applying current prompt optimization techniques directly to coding rules yields suboptimal performance, as confirmed by our experiments with Prompt-Ops [24].

3 Problem Definition

3.1 AI Agents for Code Generation

We consider an AI coding agent $\mathcal{A}_\theta$ powered by a backbone LLM with parameters θ. The agent generates code conditioned on a specific programming task and underlying coding rules. In practice, for most modern coding agents, these rules are instantiated via an `AGENTS.md` document and prepended to the LLM's input context.

Let $x \in \mathcal{X}$ denote a coding task (i.e., the problem description), r denote the coding rules, and y denote the generated code, where $\mathcal{X}$ is the coding task set. Formally, the coding agent $\mathcal{A}_\theta$ samples the code y from the conditional distribution:

$$y \sim p_\theta(\cdot \mid x, r). \tag{1}$$

Given a coding task x and fixed parameters θ, the coding rules r significantly influence the code generation process, thereby affecting the correctness, length, and inference cost. While correctness ensures the functional integrity of the code, length and inference cost are equally critical for real-world deployment. Code length determines the performance of the generated code, where optimized logic reduces execution latency and resource consumption. Inference cost directly affect the scalability of the agentic system; by minimizing token usage and generation time, developers can substantially lower API expenses and improve the responsiveness of the development cycle. Formally, we define two primary goals for the coding rule self-evolving process:

- The generated code y should be executable and functionally correct.
- The evolved coding rules should reduce the code length as possible and minimize inference costs during the generation process.

3.2 Objective

To satisfy the first goal, we define a correctness constraint. Let $\mathcal{V}(x, y) \in \{0, 1\}$ be a code verifier that determines whether code y is executable and correctly solves the programming task x, where $\mathcal{V}(x, y) = 1$ indicates success. In practice, $\mathcal{V}$ may be implemented by executing y against a set of test cases associated with x. The evolved coding rules r must satisfy:

$$\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \mathbb{E}_{y \sim p_\theta(\cdot|x,r)}[\mathcal{V}(x, y)] \geq \tau, \tag{2}$$

where τ is a predefined threshold ensuring that the average correctness rate of the agent $\mathcal{A}_\theta$ across the task set $\mathcal{X}$ remains acceptable.

To satisfy the second goal, we define a code-length score $E(y)$ and a cost score $C(\mathcal{A}_\theta, x, r)$. A higher $E(y)$ indicates that the generated code y is more concise, while a higher $C(\mathcal{A}_\theta, x, r)$ indicates lower resource consumption during the generation process. Formally, the coding rule self-evolving process is expressed as the following constrained optimization problem:

$$\begin{aligned} \max_{r} \quad & \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \mathbb{E}_{y \sim p_\theta(\cdot|x,r)}[E(y) + C(\mathcal{A}_\theta, x, r)] \\ \text{s.t.} \quad & \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \mathbb{E}_{y \sim p_\theta(\cdot|x,r)}[\mathcal{V}(x, y)] \geq \tau. \end{aligned} \tag{3}$$

4 RuleEvolve

4.1 Overview

To achieve these objectives, we propose RuleEvolve, a self-evolving framework for coding rules. As illustrated in Figure 1, RuleEvolve consists of three primary components: a rule pool of fixed size m, a *mutator module* for generating mutated candidates, and a *judge module* for evaluation and selection. Given the initial coding rules r, RuleEvolve first employs the mutator module to populate the pool with m mutated versions. In each subsequent iteration, N newly mutated candidates are generated from m candidates in the current pool. Mutations are strategically allocated based on the historical mean and variance of the candidates' scores. Finally, the $m + N$ candidates are evaluated by the judge module; the top m candidates are retained for the next iteration, while others are pruned.

Challenge and motivation: The self-evolving of coding rules presents two key challenges. First, the generation process in a coding agent typically involves multi-turn interactions and external tool-calling, rendering the pipeline non-differentiable and precluding direct gradient-based optimization. We address this by implementing an iterative mutation and pruning framework that leverages the mutator module and the judge module to approximate the optimization process (see Section 4.2). Second, unlike standard prompt optimization tasks that involve short targets and low evaluation costs, coding rules are significantly more resource-intensive to evolve. Each rule set requires full agent execution and verification, leading to substantial latency and API token consumption. To avoid resource waste and enhance effectiveness, we propose a strategic mutation-allocation method that prioritizes mutating promising candidates, as detailed in Section 4.3.

4.2 Iterative Mutation and Pruning

Mutator $\mathcal{M}$: The mutator module, powered by an LLM, takes the coding rules r and a mutation guidance g as input to produce a mutated candidate $r_m = \mathcal{M}(r, g)$. The mutation guidance g typically consists of a natural language directive, such as: *"Mutate the coding rules for an AI coding agent into a new variant by improving wording, reordering or combining steps, and varying the style."*

Judge $\mathcal{J}$: The Judge module provides the evaluation environment for generating code under specific coding rules, executing the generated code, and assessing performance across the dimensions of correctness, length, and inference cost. Specifically, the Judge $\mathcal{J}$ takes the coding rules r, the agent $\mathcal{A}_\theta$, and a task set $\mathcal{X}$ as input to produce a score $S = \mathcal{J}(r, \mathcal{A}_\theta, \mathcal{X})$.

As formulated in Section 3.2, this score consists of two components. The first is a correctness penalty designed to ensure that the evolved rules do not degrade the agent's functional performance. Given the code verifier $\mathcal{V}$, the penalty is defined as:

$$\mathbb{E}_{x \in \mathcal{X}} \left[\mathcal{V}(x, \mathcal{A}_\theta(x, \emptyset)) - \mathcal{V}(x, \mathcal{A}_\theta(x, r)) \right]_+ , \tag{4}$$

where $\mathcal{A}_\theta(x, \emptyset)$ denotes the code generated by the agent without external coding rules, $\mathcal{A}_\theta(x, r)$ denotes the code generated using coding rules r, and $[z]_+ = \max(z, 0)$.

The second component focuses on the code length and inference cost of the generation process. As illustrated in Figure 2a, there is a high correlation between code characters and token usage during generation. Consequently, we approximate both code length and inference cost using the number of characters of the generated code, denoted by $|\mathcal{A}_\theta(x, r)|$. The overall score S is formulated as:

$$S = -|\mathcal{A}_\theta(x, r)| - \lambda \cdot \mathbb{E}_{x \in \mathcal{X}} \left[\mathcal{V}(x, \mathcal{A}_\theta(x, \emptyset)) - \mathcal{V}(x, \mathcal{A}_\theta(x, r)) \right]_+ , \tag{5}$$

where λ is a hyperparameter that balances the trade-off between two components.

Iterative Process: The complete procedure is summarized in Algorithm 1. Our approach begins by populating an initial candidate rule pool $\mathcal{P}$ through m mutations of the initial coding rules. In each iteration t, we strategically distribute N mutations among existing m candidates based on their historical scores. This results in N newly mutated candidates by the mutator module $\mathcal{M}$, which are subsequently evaluated by the judge module $\mathcal{J}$. To maintain a manageable search space and focus on high-performing regions, we employ a pruning step that retains only the top-m candidates for the subsequent iteration. After T iterations, the framework returns the coding rules that achieved the highest average score.

Algorithm 1 RuleEvolve: Coding Rule Self-Evolving

Require: Initial rules r_I, task set $\mathcal{X}$, rule pool size m, number of mutations N, agent $\mathcal{A}_\theta$, mutator $\mathcal{M}$, judge $\mathcal{J}$, max iterations T

$\mathcal{P} \leftarrow \mathcal{M}(r_I, m)$ ▷ Initialize pool with m mutated candidates
$\mathcal{S} \leftarrow \mathcal{J}(\mathcal{P}, \mathcal{A}_\theta, \mathcal{X})$ ▷ Obtain initial scores for the pool
for $t = 1$ to T **do**
  $\{n_1, \ldots, n_m\} \leftarrow$ MUTATIONALLOCATION$(\mathcal{S}, N)$ ▷ Allocate N mutations strategically
  $\mathcal{P}_{new}, \mathcal{S}_{new} \leftarrow \emptyset, \emptyset$
  for $j = 1$ to m **do**
    $\mathcal{R}_{new} \leftarrow \mathcal{M}(\mathcal{P}_j, n_j)$ ▷ Generate n_j new mutations from candidate j
    $\mathcal{P}_{new} \leftarrow \mathcal{P}_{new} \cup \mathcal{R}_{new}$
    $\mathcal{S}_{new} \leftarrow \mathcal{S}_{new} \cup \{\mathcal{S}_j \text{ for each } r \in \mathcal{R}_{new}\}$ ▷ Inherit score set from parent j
  end for
  $\mathcal{S}_{eval} \leftarrow \mathcal{J}(\mathcal{P}_{new}, \mathcal{A}_\theta, \mathcal{X})$ ▷ Evaluate new candidates
  $\mathcal{S}_{new} \leftarrow$ Update$(\mathcal{S}_{new}, \mathcal{S}_{eval})$ ▷ Update score sets
  $\mathcal{P}, \mathcal{S} \leftarrow$ PRUNE$(\mathcal{P} \cup \mathcal{P}_{new}, \mathcal{S} \cup \mathcal{S}_{new}, m)$ ▷ Retain top-m candidates
end for
return $\arg\max_{r \in \mathcal{P}} \text{avg}(\mathcal{S}_r)$

4.3 Strategic Mutation Allocation

To avoid meaningless mutations and enhance effectiveness, we employ the Optimal Computing Budget Allocation [5] to distribute N mutations among m candidates wisely. The core objective is to maximize the probability of correctly identifying the best-performing candidate while minimizing redundant evaluations of suboptimal candidates.

For each candidate i, we maintain a score set $\mathcal{S}_i$ containing the score from each iteration's evaluation. To address the cold-start problem and ensure statistical stability, a newly mutated candidate inherits the score set from its parent. This inheritance ensures that all candidates in the rule pool possess the same number of observations, allowing for a consistent computation of the mean $\hat{\mu}_i$ and standard deviation $\hat{\sigma}_i$ for candidate i.

Let b be the index of the current best-performing candidate in the pool, i.e., with the highest average score. We determine the allocation weights w_i for all $i \neq b$ such that:

$$w_i \propto \left(\frac{\hat{\sigma}_i}{\hat{\mu}_b - \hat{\mu}_i} \right)^2. \tag{6}$$

This formulation ensures that more mutations are allocated to candidates with higher performance variance or those whose mean performance is closer to the current best. The weight for the best candidate w_b is then calculated as follows to maintain overall allocation efficiency:

$$w_b = \hat{\sigma}_b \sqrt{\sum_{i \neq b} \frac{w_i^2}{\hat{\sigma}_i^2}}. \tag{7}$$

Finally, the discrete n_i mutations for each candidate is determined by normalizing these weights such that $\sum n_i = N$. By iteratively refreshing these statistics, updating allocation weights w_i, and redistributing the mutations n_i, RuleEvolve focuses its search on the most promising regions of the coding rule space, improving the effectiveness of the evolving process. Experimental results validate this by comparing with naive mutation-allocation strategies. More details can be found in Appendix B.

5 Evaluation

5.1 Experimental setup

Datasets: We utilize the BigCodeBench benchmark [29] as our primary dataset for coding rules evolvement. From this benchmark, we randomly sample 1,000 instances to serve as a training set.

Algorithm 2 MutationAllocation

Require: Score sets $\mathcal{S}_i$ for candidates $\{1, \ldots, m\}$, number of mutations N

Compute means $\hat{\mu}_i$ and standard deviations $\hat{\sigma}_i$ from $\mathcal{S}_i$ for each $i \in \{1, \ldots, m\}$

$b \leftarrow \arg\max_i \hat{\mu}_i$

for $i = 1$ to $m, i \neq b$ **do**

  $\Delta_i \leftarrow \max(\hat{\mu}_b - \hat{\mu}_i, \epsilon)$

  $w_i \leftarrow (\hat{\sigma}_i / \Delta_i)^2$

end for

$w_b \leftarrow \hat{\sigma}_b \sqrt{\sum_{i \neq b} (w_i^2 / \hat{\sigma}_i^2)}$

$\{n_1, \ldots, n_m\} \leftarrow \text{round}\left(N \cdot \frac{w_i}{\sum_j w_j}\right)$

return $\{n_1, \ldots, n_m\}$

Table 1: OpenAI SDK results. For each method, we report the performance of the backbone that achieved the highest pass rate across benchmarks. Metrics include Pass Rate (**PR**), Avg. Code Lines (**CLn**), Avg. Code Characters (**CC**), Avg. Generation Time (**Time**), and Avg. Token Usage (**Tok**).

Method	**PR** ↑			**CLn** ↓			**CC** ↓			**Time** ↓			**Tok** ↓		
	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE
No rule	0.47	0.25	0.97	35.7	42.3	26.4	1156	1406	812.4	6.00	5.44	4.24	623.5	666.5	502.9
Manual rule	0.45	0.23	0.98	69.0	84.8	44.4	2372	2946	1136	8.75	10.5	5.08	1022	1147	663.8
Prompt-Ops	0.48	0.23	0.89	13.8	15.7	4.50	474.4	583.0	126.5	2.30	2.90	1.33	581.7	588.7	406.3
RuleEvolve	0.49	0.26	0.99	12.6	20.8	4.40	453.1	745.3	114.4	3.02	2.90	1.48	544.5	595.8	367.5

During the self-evolving process, each iteration's scores are obtained using mini-batches of size B, sampled randomly from this training set. To evaluate the performance of evolved coding rules, we employ three coding benchmarks: BigCodeBench, BigCodeBench-Hard [29], and HumanEval [6]. For each of these benchmarks, we randomly select 100 samples as our testing benchmark. Samples drawn from BigCodeBench for testing are strictly disjoint from the training sets used during evolvement.

Coding agents: We evaluate our method using two distinct agentic frameworks: OpenHands [23], a state-of-the-art open-source autonomous agent, and a custom GPT-based agent implemented via the OpenAI SDK. For OpenHands, our evolved coding rules are provided through the AGENTS.md configuration, while for OpenAI SDK, rules are integrated directly into the system prompt.

Backbone models: To evaluate the generalizability of coding rules evolvement approaches, we conduct experiments using both proprietary and open-weight models. For the proprietary models, we utilize gpt-4.1-mini, gpt-5-mini, and the state-of-the-art gpt-5.3-codex, all accessed via the OpenAI API. For the open-weight model, we employ Qwen-Coder-30b [25]. We evolve the coding rules for each model respectively.

Baselines: We compare RuleEvolve against several baselines to assess the impact of our coding rules self-evolvement. The *No Rule* baseline represents a vanilla setting where the agent receives only the task description without any coding rules. The *Manual Rule* baseline utilizes manually written coding rules sourced from the awesome-cursorrules repository [19]; notably, our experiments indicated that providing coding rules do not consistently yield performance gains, likely due to a lack of task-specific alignment. This observation aligns with conclusions in [11]. *Prompt-Ops* [24] is a baseline designed for prompt optimization for LLMs. Our results show that such techniques achieve only a suboptimal trade-off between correctness and code length, in most cases sacrificing the functional correctness of the generated code. To demonstrate the effectiveness of our strategic mutation-allocation method, we also evaluate two naive mutation-allocation strategies: even and proportional allocation, which distribute mutations equally or proportionally based on each candidate's average score.

Evaluation Metrics: On testing benchmarks, we evaluate the coding rules across three dimensions: correctness, code length, and inference cost. Correctness is measured by the *pass rate* of the code, defined as the percentage of generated code that satisfy all provided test cases across the testing benchmark. Code length is assessed through two metrics: *code lines* and *code characters*. Inference

Table 2: OpenHands results. For each method, we report the performance of the backbone that achieved the highest pass rate across benchmarks. Metrics include Pass Rate (**PR**), Avg. Code Lines (**CLn**), Avg. Code Characters (**CC**), Avg. Generation Time (**Time**), and Avg. Token Usage (**Tok**).

Method	**PR** ↑			**CLn** ↓			**CC** ↓			**Time** ↓			**Tok** ↓		
	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE
No rule	0.49	0.29	0.99	35.4	43.2	17.5	1060	1324	503.4	10.6	12.8	6.89	1143	1238	697.5
Manual rule	0.49	0.29	0.99	42.9	53.5	21.7	1350	1722	577.6	12.2	14.1	7.14	1293	1443	727.1
Prompt-Ops	0.43	0.25	0.99	20.1	27.5	19.3	615.2	880.2	548.7	40.2	41.4	12.6	903.1	986.9	725.5
RuleEvolve	0.52	0.27	0.99	26.6	31.8	12.7	814.6	1009	373.5	9.76	11.6	7.15	1016	1075	612.6

cost is measured by *generation time* and *token usage* during the inference process. The averages of the latter four metrics are computed only on code that is functionally correct.

Parameter settings: By default, for our RuleEvolve, we set the number of iteration T=10, number of mutations N=20, pool size m=5, coding task batch size B=10, balance parameter λ=3,000, and ϵ=10^{-6}. For manual rule, we use the fixed one, as shown in Appendix F. For Prompt-Ops, we use their default settings.

5.2 Main Results

Table 1 and Table 2 present a comprehensive comparison of RuleEvolve against other baseline approaches. Generally, RuleEvolve outperforms existing baselines by achieving a superior balance between functional correctness, code length and inference cost. Specifically, RuleEvolve consistently achieves the highest or equal Pass Rate (PR) across all benchmarks. When achieving comparable PRs to "No rule" or "Manual rule", RuleEvolve demonstrates significant improvements in code length and inference cost. For example, on the BigCodeBench (BCB) benchmark within the OpenAI SDK framework, RuleEvolve improves the PR from 0.45 (Manual rule) to 0.49, while simultaneously reducing the average code characters from 2372 to 453.1 and token usage from 1022 to 544.5. Similarly, in the OpenHands environment for the HumanEval (HE) benchmark, RuleEvolve maintains a comparable PR of 0.99 while reducing average code lines to 12.7, compared to 21.7 for human-written rules.

We also note that Prompt-Ops occasionally achieves lower code length and inference cost than RuleEvolve in certain scenarios. For instance, on BigCodeBench-Hard (BCB-H) within the OpenAI SDK, Prompt-Ops yields an average token usage of 588.7 compared to 595.8 for RuleEvolve. However, the gains in code length or inference cost come at the expense of functional integrity; in the OpenHands BCB scenario, Prompt-Ops suffers a substantial drop in PR to 0.43, whereas RuleEvolve reaches 0.52. We stress that PR is the most critical metric in autonomous coding; if the generated code is incorrect, optimizing for code length or inference cost becomes meaningless. By prioritizing functional correctness, RuleEvolve ensures that gains in code length or inference cost do not compromise the agent's primary utility.

The comprehensive results across diverse backbones are detailed in Table 5 and Table 6 in the Appendix. Across all evaluated backbone models, RuleEvolve consistently achieves superior or comparable pass rates. At the same time, RuleEvolve significantly reduces code length and inference cost in most cases. For instance, with the gpt-5-mini backbone on BCB, RuleEvolve achieves a PR of 0.46 compared to 0.31 for manual rules, while simultaneously reducing token usage by nearly 40%. Furthermore, RuleEvolve consistently outperforms Prompt-Ops across nearly all backbone–benchmark combinations. While Prompt-Ops often struggles with correctness (e.g., dropping to a 0.67 PR on HumanEval (HE) with Qwen-Coder-30B), RuleEvolve maintains high PRs across all benchmarks and backbone models. We also observe that different backbone models exhibit diverse generation styles; for example, gpt-5-mini tends to produce longer code and requires more generation time, while gpt-5.3-codex is more efficient.

Multi-seed results: To assess the robustness and statistical stability of RuleEvolve, we performed evaluations across 10 independent seeds using the GPT-4.1-mini backbone on the BigCodeBench dataset. As shown in Table 3, RuleEvolve maintains a stable Pass Rate (PR of 0.502 ± 0.026, which is comparable to the "No rule" baseline and superior to the "Public rule". Also, RuleEvolve yields significantly more concise code, achieving the lowest average code lines (24.3 ± 0.4) and characters (794.4 ± 14) among all evaluated methods. Notably, our framework achieves these gains

Table 3: Multi-seed results on BigCodeBench using GPT-4.1-mini as backbone LLM. Values are mean ± standard deviation (SD) across 10 seeds.

Method	**PR** ↑	**CLn** ↓	**CC** ↓	**Time (s)** ↓	**Tok** ↓
No rule	0.502 ± 0.015	32.5 ± 0.4	1028 ± 8.1	8.66 ± 0.74	1181 ± 9.3
Public rule	0.498 ± 0.032	44.9 ± 0.7	1482 ± 25	10.21 ± 1.15	2049 ± 35
Prompt-Ops	0.502 ± 0.027	26.3 ± 0.4	811.4 ± 10	6.81 ± 0.62	1130 ± 15
RuleEvolve (OCBA)	0.502 ± 0.026	24.3 ± 0.4	794.4 ± 14	9.67 ± 1.90	1019 ± 18

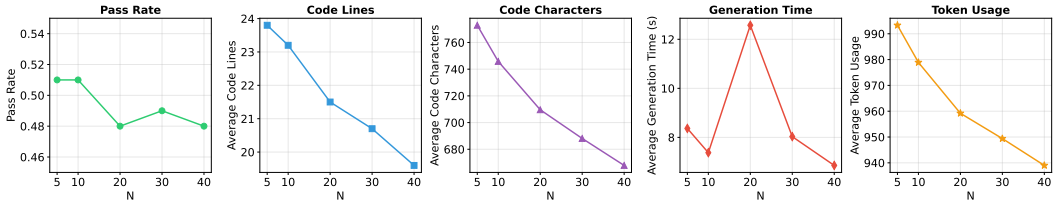
with a high degree of consistency, as evidenced by the standard deviations reported across 10 seeds. These multi-seed results confirm that RuleEvolve effectively and reliably optimizes coding rules for efficiency without compromising the agent's primary problem-solving utility.

Results on SWE-Bench Lite: To evaluate the scalability of our approach in complex, repository-level environments, we test RuleEvolve on SWE-Bench Lite [13] using a subset of 100 tasks. As shown in Table 4, RuleEvolve achieves the highest PR of 0.340, outperforming other baselines. Consistent with main results, the evolved rules effectively optimize for code efficiency, reducing average code lines from 8.5 to 5.7 and characters from 380 to 254. Although generation time increases, likely due to the increased reasoning overhead of repository-level tasks, the functional correctness is well preserved. Overall, these findings confirm that RuleEvolve can generalize to real-world repository-level scenarios.

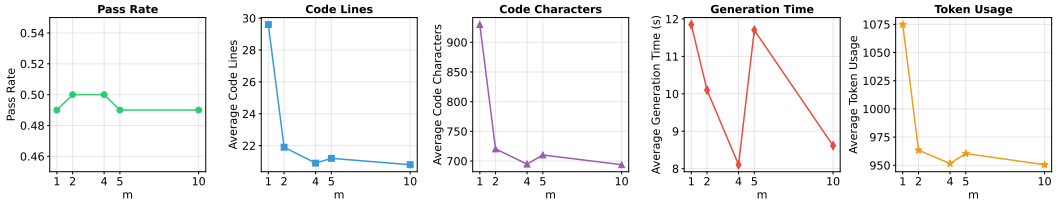
5.3 Ablation Study



(a) Metrics vs. Iterations (b) Impact of mutation allocation



(c) Impact of number of mutation N



(d) Impact of pool size m

Figure 2: Ablation study results.

Impact of number of iterations T: Figure 2a shows the evolution of metrics over 20 iterations during the evolvement process. The pass rate remains remarkably consistent throughout the iterations, demonstrating that RuleEvolve successfully optimizes for code length and inference cost without compromising the correctness of the generated code. In contrast, average code lines, average code characters, and average tokens usage metrics all exhibit a significant and continuous downward trend. The curves for these three metrics are highly similar because they are naturally coupled; reducing the structural verbosity of the code inherently leads to a lower number of tokens required for model inference. While the average generation time generally decreases over all 20 iterations, it exhibits notable fluctuations between iterations. These fluctuations are likely due to the inherent variance in API latency and the stochastic nature of autoregressive generation. Most of the substantial improvements occur within the first 5 iterations, where the slopes of the cost curves are steepest. By the twentieth iteration, the metrics level off, suggesting that the rule evolvement process effectively converges.

Table 4: SWE-bench Lite results with GPT-4.1-mini as backbone LLM.

Method	PR ↑	CLn ↓	CC ↓	Time (s) ↓	Tok ↓
No rule	0.330	8.5	380	131.17	911
Public rule	0.190	9.1	413	166.45	867
Prompt-Ops	0.080	5.4	236	175.24	684
RuleEvolve	0.340	5.7	254	297.46	722

Impact of mutation-allocation strategies: To investigate the impact of mutation allocation, we compare three variants: 1) evenly dividing the N mutations among the m rule candidates, 2) proportional distribution based on average scores, and 3) our strategic allocation method. As shown in Figure 2b, while all variants maintain an identical pass rate of 0.500, our approach produces the most concise code with 22.6 lines and 736.2 characters, outperforming even (23.4 lines, 753.2 chars) and proportional (23.9 lines, 781.3 chars) variants. Additionally, our method minimizes computational cost, reaching the lowest generation time (15.48s) and token usage (975.2) compared to the other two variants. These results confirm that strategic mutation-allocation more effectively evolves coding rules without sacrificing functional correctness.

Impact of N and m: We analyze the sensitivity of RuleEvolve to the number of mutations N and the pool size m. As shown in Figure 2c, when N increases, average code lines, code characters, and token usage all exhibit a clear downward trend. Notably, these improvements are achieved while maintaining a stable pass rate that fluctuates slightly between 0.48 and 0.51. Similarly, the impact of the pool size m in Figure 2d reveals that increasing m from 1 to 2 provides a substantial reduction in code length and token usage. While these metrics stabilize for values of m greater than 2, the initial improvement indicates that maintaining a sufficiently large pool of candidates is crucial for obtaining optimized coding rules.

Pass@k results: We evaluate the robustness of our evolved coding rules by measuring the Pass@k metric across three benchmarks using gpt-4.1-mini, where k indicates the number of versions for code generated for each task. As illustrated in Figure 3, the pass rate for all benchmarks scales positively with the number of samples k. The steady upward trend across all benchmarks confirms that RuleEvolve effectively optimizes for code efficiency while preserving the model's problem-solving capacity under higher sampling budgets.

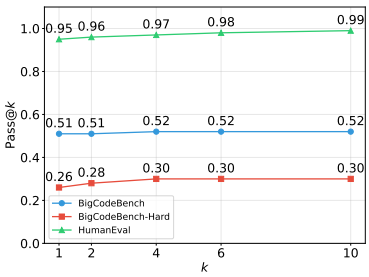


Figure 3: Pass@k results.

6 Conclusion

In this work, we investigate the self-evolvement of coding rules for AI coding agents. We show that coding rules can be evolved to greatly reduce code length and minimize inference costs while maintaining the functional correctness of the generated code. This is achieved by RuleEvolve, our proposed automatic self-evolving framework for coding rules. By employing a mutator, judge module, and a strategic mutation-allocation approach, RuleEvolve successfully evolves coding rules via iterative mutation and pruning. Extensive evaluations across two agentic frameworks, four diverse backbone models, and three coding benchmarks demonstrate that RuleEvolve outperforms both manually crafted rules and existing baselines, establishing a promising direction for coding rules self-evolvement.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work was supported by NSF grant No. 2530786, 2450935, 2131859, 2414406, 2125977, 2112562, and 2625580.

References

[1] Agentsmd. Agents.md. `https://agents.md/`, 2025.

[2] Anomalyco. opencode. `https://github.com/anomalyco/opencode`, 2025.

[3] Anthropics. claude-code. `https://github.com/anthropics/claude-code`, 2025.

[4] Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J Pappas, and Eric Wong. Jailbreaking black box large language models in twenty queries. In *SaTML*, 2025.

[5] Chun-Hung Chen and Loo Hay Lee. *Stochastic simulation optimization: an optimal computing budget allocation*. World scientific, 2011.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Tianhao Chen, Zhengyuan Jiang, Yuepeng Hu, Yebei Gou, and Neil Zhenqiang Gong. Dynamic malicious skills in agentic ai. *arXiv preprint arXiv:2606.16287*, 2026.

[8] Cursor. cursor. `https://github.com/cursor/cursor`, 2025.

[9] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

[10] Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring agentic ai coding tools: An exploratory study. *arXiv preprint arXiv:2602.14690*, 2026.

[11] Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. Evaluating agents. md: Are repository-level context files helpful for coding agents? *arXiv preprint arXiv:2602.11988*, 2026.

[12] Zhengyuan Jiang, Yuepeng Hu, Yuchen Yang, Yinzhi Cao, and Neil Zhenqiang Gong. Jailbreaking safeguarded text-to-image models via large language models. In *EACL Findings*, 2026.

[13] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2023.

[14] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. In *ICLR*, 2024.

[15] Yexiong Lin, Shanshan Ye, Yu Yao, Zhen Fang, Bo Han, and Tongliang Liu. Squidagent: Parallelize wisely, coordinate efficiently. *arXiv preprint arXiv:2610.08647*, 2026.

[16] Anay Mehrotra, Manolis Zampetakis, Paul Kassianik, Blaine Nelson, Hyrum Anderson, Yaron Singer, and Amin Karbasi. Tree of attacks: Jailbreaking black-box llms automatically. In *NeurIPS*, 2024.

[17] Alessandro Midolo, Alessandro Giagnorio, Fiorella Zampetti, Rosalia Tufano, Gabriele Bavota, and Massimiliano Di Penta. Guidelines to prompt large language models for code generation: An empirical characterization. *arXiv preprint arXiv:2601.13118*, 2026.

[18] Arijit Nag, Animesh Mukherjee, Niloy Ganguly, and Soumen Chakrabarti. Mutantprompt: Prompt optimization via mutation under a budget on modest-sized lms. In *ACL Findings*, 2025.

[19] PatrickJS. awesome-cursorrules. `https://github.com/PatrickJS/awesome-cursorrules/`, 2025.

[20] Shuai Shao, Qihan Ren, Dongrui Liu, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, et al. Your agent may misevolve: Emergent risks in self-evolving llm agents. In *ICLR*, 2026.

[21] Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. Seagent: Self-evolving computer use agent with autonomous learning from experience. In *ICML*, 2026.

[22] Zeliang Tong, Zhuojun Ding, and Wei Wei. Evoprompt: Evolving prompts for enhanced zero-shot named entity recognition with large language models. In *COLING*, 2025.

[23] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *ICLR*, 2025.

[24] Yuanchen Wu, Saurabh Verma, Justin Lee, Fangzhou Xiong, Poppy Zhang, Amel Awadelkarim, Xu Chen, Yubai Yuan, and Shawndra Hill. Llm prompt duel optimizer: Efficient label-free prompt optimization. *arXiv preprint arXiv:2510.13907*, 2025.

[25] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[26] Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

[27] Sixiang Ye, Zeyu Sun, Guoqing Wang, Liwei Guo, Qingyuan Liang, Zheng Li, and Yong Liu. Prompt alchemy: Automatic prompt refinement for enhancing code generation. *IEEE Transactions on Software Engineering*, 2025.

[28] Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. In *COLM*, 2026.

[29] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *ICLR*, 2025.

A Limitations

Our work has several limitations. First, while RuleEvolve maintains or improves functional correctness in most scenarios, the objective of reducing code length may still lead to a slight decrease in the pass rate for certain complex tasks where excessive brevity could compromise logic. Second, our self-evolving process relies on an LLM-powered mutator and judge module, which introduces inherent stochasticity and potential biases from the backbone models used during evolving. Third, our study primarily focuses on optimizing for correctness, code length, and inference cost through code length approximation. A more comprehensive exploration of other dimensions, such as code maintainability or architectural complexity, remains for future work.

Table 5: Comprehensive OpenAI SDK results on BigCodeBench (BCB), BigCodeBench-Hard (BCB-H), and HumanEval (HE). Metrics include Pass Rate (**PR**), Avg. Code Lines (**CLn**), Avg. Code Characters (**CC**), Avg. Generation Time (**Time**), and Avg. Token Usage (**Tok**).

Backbone	**Method**	**PR ↑**			**CLn ↓**			**CC ↓**			**Time ↓**			**Tok ↓**		
		BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE
gpt-4.1 -mini	No rule	0.47	0.25	0.97	35.7	42.3	26.4	1156	1406	812.4	6.00	5.44	4.24	623.5	666.5	502.9
	Manual rule	0.46	0.20	0.92	58.4	68.6	37.5	2035	2448	1036	12.7	9.61	5.36	852.0	939.1	547.9
	Prompt-Ops	0.45	0.24	0.92	20.9	24.1	6.30	650.8	790.7	180.0	3.06	3.94	1.79	540.5	552.8	335.4
	RuleEvolve	0.46	0.24	0.93	16.9	19.3	5.10	545.2	674.8	134.7	2.86	3.25	1.37	479.3	484.5	284.5
gpt-5 -mini	No rule	0.36	0.20	0.95	88.2	94.6	46.4	3053	3377	1248	29.5	28.5	13.7	2185	2257	1197
	Manual rule	0.31	0.20	0.68	128	142	98.8	4720	5175	2926	37.0	38.6	23.0	2799	2973	1812
	Prompt-Ops	0.38	0.20	0.99	19.8	22.8	8.20	643.0	799.7	237.4	21.7	23.6	15.9	1884	1983	1370
	RuleEvolve	0.46	0.20	0.99	19.0	20.8	5.90	611.4	740.1	163.8	22.0	22.0	15.2	1717	1765	1249
gpt-5.3 -codex	No rule	0.46	0.26	0.97	47.0	53.5	25.2	1410	1651	643.0	5.83	6.35	3.28	790.8	828.3	544.0
	Manual rule	0.45	0.23	0.98	69.0	84.8	44.4	2372	2946	1136	8.75	10.5	5.08	1022	1147	663.8
	Prompt-Ops	0.48	0.23	0.89	13.8	15.7	4.50	474.4	583.0	126.5	2.30	2.90	1.33	581.7	588.7	406.3
	RuleEvolve	0.49	0.26	0.99	12.6	20.8	4.40	453.1	745.3	114.4	3.02	2.90	1.48	544.5	595.8	367.5
Qwen -Coder -30B	No rule	0.17	0.12	0.90	17.7	17.5	27.8	556.5	562.7	878.9	3.29	21.3	21.3	932.4	946.5	913.7
	Manual rule	0.19	0.09	0.69	28.2	26.5	28.5	988.4	913.2	903.3	4.46	21.4	21.3	1083	1104	902.0
	Prompt-Ops	0.32	0.18	0.67	19.8	25.5	7.30	583.0	768.5	199.4	4.06	21.2	21.6	799.0	794.4	582.0
	RuleEvolve	0.38	0.19	0.88	14.7	18.4	5.50	492.7	652.6	153.6	3.95	20.9	21.2	715.5	719.0	538.6

Table 6: Comprehensive OpenHands results on BigCodeBench (BCB), BigCodeBench-Hard (BCB-H), and HumanEval (HE). Metrics include Pass Rate (**PR**), Avg. Code Lines (**CLn**), Avg. Code Characters (**CC**), Avg. Generation Time (**Time**), and Avg. Token Usage (**Tok**).

Backbone	**Method**	**PR ↑**			**CLn ↓**			**CC ↓**			**Time ↓**			**Tok ↓**		
		BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE	BCB	BCB-H	HE
gpt-4.1 -mini	No rule	0.48	0.27	0.96	29.8	34.6	19.2	937.8	1127	608.1	8.28	12.6	20.8	1079	1124	767.5
	Manual rule	0.48	0.28	0.95	42.3	47.2	22.9	1402	1597	691.8	9.11	13.7	18.3	1320	1383	813.4
	Prompt-Ops	0.43	0.22	0.71	20.1	27.5	17.7	615.2	880.2	544.1	40.2	41.4	57.9	903.1	986.9	707.7
	RuleEvolve	0.50	0.28	0.98	23.8	28.5	16.9	768.9	944.6	536.3	9.54	14.6	23.0	991.6	1028	725.0
gpt-5 -mini	No rule	0.40	0.18	0.97	66.1	74.3	30.2	2203	2509	854.7	12.9	14.8	7.10	1793	1930	944.4
	Manual rule	0.38	0.22	0.96	74.7	86.5	42.6	2655	3080	1181	13.9	16.3	7.44	2047	2250	1122
	Prompt-Ops	0.33	0.19	0.99	46.4	51.5	19.3	1454	1656	548.7	14.0	16.2	12.6	1392	1469	725.5
	RuleEvolve	0.41	0.25	0.98	50.9	58.5	17.6	1670	1956	514.4	10.8	12.8	6.63	1498	1623	705.6
gpt-5.3 -codex	No rule	0.49	0.29	0.99	35.4	43.2	17.5	1060	1324	503.4	10.6	12.8	6.89	1143	1238	697.5
	Manual rule	0.49	0.29	0.99	42.9	53.5	21.7	1350	1722	577.6	12.2	14.1	7.14	1293	1443	727.1
	Prompt-Ops	0.40	0.25	0.91	25.3	32.1	14.4	787.3	1012	410.4	13.4	14.7	13.8	1009	1072	634.6
	RuleEvolve	0.52	0.27	0.99	26.6	31.8	12.7	814.6	1009	373.5	9.76	11.6	7.15	1016	1075	612.6
Qwen -Coder -30B	No rule	0.43	0.27	0.97	39.1	44.0	28.1	1260	1449	878.7	37.2	60.1	21.2	1250	1291	973.8
	Manual rule	0.44	0.28	0.99	39.5	43.7	29.6	1286	1434	907.4	31.8	56.9	32.8	1262	1284	996.6
	Prompt-Ops	0.35	0.19	0.91	32.4	34.3	18.2	1020	1087	532.7	23.2	29.2	10.2	1119	1110	712.1
	RuleEvolve	0.44	0.27	0.96	27.2	32.4	20.3	840.1	1021	611.3	9.85	14.2	14.4	1028	1067	781.1

B Justification for Mutation-Allocation Strategy

In this section, we provide the theoretical justification for the mutation-allocation strategy used in RuleEvolve. The goal of the Optimal Computing Budget Allocation [5] is to maximize the *Probability of Correct Selection (PCS)*, defined as the probability that the candidate rule set with the highest sample mean is indeed the true best performer.

Let m be the number of candidates in the pool, and let J be the index of the true best candidate such that $\mu_J = \max_i \mu_i$. The PCS is given by:

$$\text{PCS} = P\left(\hat{\mu}_J > \hat{\mu}_i, \forall i \neq J\right). \tag{8}$$

Under the assumption that the scores $\mathcal{S}_i$ are independent and follow a normal distribution $N(\mu_i, \sigma_i^2)$, for a large number of mutations N, the PCS is maximized when the allocation $n_1, n_2, \ldots, n_m$ satisfies the following asymptotic conditions:

$$\frac{n_i}{n_j} = \frac{\sigma_i^2/(\mu_b - \mu_i)^2}{\sigma_j^2/(\mu_b - \mu_j)^2}, \quad i, j \neq b, \tag{9}$$

and

$$n_b = \sigma_b \sqrt{\sum_{i \neq b} \frac{n_i^2}{\sigma_i^2}}, \tag{10}$$

where b is the index of the candidate with the current highest sample mean.

These conditions provide the following insights into the RuleEvolve evolvement process:

- Candidates with a smaller performance gap $(\mu_b - \mu_i)$ receive more mutations, as they are the most likely "competitors" to the current best.
- Candidates with higher variance σ_i^2 receive more mutations to reduce the uncertainty in their mean estimation.
- The mutations for the best candidate n_b is scaled based on the aggregate noise of all other candidates, ensuring that the comparison is sufficiently stable.

By following this allocation, RuleEvolve minimizes the risk of pruning a potentially superior coding rule due to stochastic noise in mutator generation, ensuring a more robust and efficient search path than uniform allocation.

C Analysis on Searched Rules

By grouping the rule's 16 sections into six functional categories, we evaluated the increase in generated code length when each category is removed (tested on BigCodeBench using 40 tasks with GPT-4.1-mini, where the full rule produces 454.1 characters compared to 1,040.6 characters with no rule), as shown in Table 7.

Table 7: Character Attribution by Category

Category	**Sections**	**Chars attributable**	**Per section**
Output format / self-containment	3	153.0	51.0
Algorithm knowledge	4	83.0	20.7
Brevity style	1	61.6	61.6
API / library idiom	2	22.8	11.4
Correctness tactics	2	9.9	5.0
Process / budget	4	-5.0	-1.2

This analysis yields three findings. First, directives that constrain the form of the code (output format and self-containment) dominate the rule's impact, accounting for 153 attributable characters. Second, brevity-style directives are the most efficient on a per-section basis, yielding 61.6 characters from a single section. Third, process and budget directives are net negative, removing them actually shortens the output.

D Code Quality Metrics

For other software engineering metrics, we report metrics over generated files, including complexity, maintainability, and duplication, as shown in Table 8. RuleEvolve reduces cyclomatic complexity

Table 8: Code Quality Metrics Comparison

Metric	No rule	RuleEvolve
Duplicated-block ratio	0.10%	0.00%
Cyclomatic complexity (total)	5.37	3.95
Halstead volume (total)	53.0	28.0
Maintainability index	85.1	78.3
Cyclomatic complexity per SLOC	0.213	0.260
Docstring rate	47.0%	1.00%

(5.37 $\rightarrow$ 3.95), Halstead volume by 47%, and duplicated-block ratio to 0.00%, while maintaining a Grade A maintainability index (78.3 vs. 85.1 for no rule). The remaining trade-off is in code density: complexity per SLOC increases by 22% and docstring coverage decreases from 47% to 1%, indicating that RuleEvolve removes documentation and defensive scaffolding rather than redundant logic.

E Generated Code Examples

E.1 Problem description

The agentic framework is OpenAI SDK. The backbone model is gpt-4.1-mini.

```
def is_palindrome(string: str) -> bool:
""" Test if given string is a palindrome """
return string == string[::-1]


def make_palindrome(string: str) -> str:
""" Find the shortest palindrome that begins with a supplied string.
Algorithm idea is simple:
- Find the longest postfix of supplied string that is a palindrome.
- Append to the end of the string reverse of a string prefix that comes
- before the palindromic suffix.
>>> make_palindrome('')
''
>>> make_palindrome('cat')
'catac'
>>> make_palindrome('cata')
'catac'
"""
```

E.2 No Rule

```
"""
Simple palindrome utilities.

Functions:
- is_palindrome(string: str) -> bool
- make_palindrome(string: str) -> str

Run doctests with: python task.py
"""
from typing import Optional
```

```
def is_palindrome(string: str) -> bool:
    """Test if given string is a palindrome.

    Examples:
    >>> is_palindrome('')
    True
    >>> is_palindrome('a')
    True
    >>> is_palindrome('aba')
    True
    >>> is_palindrome('ab')
    False
    """
    return string == string[::-1]


def make_palindrome(string: str) -> str:
    """Find the shortest palindrome that begins with the supplied string.

    Algorithm:
    - Find the longest suffix of the supplied string that is a palindrome.
    - Append to the end of the string the reverse of the prefix that comes
      before that palindromic suffix.

    Examples:
    >>> make_palindrome('')
    ''
    >>> make_palindrome('cat')
    'catac'
    >>> make_palindrome('cata')
    'catac'
    >>> make_palindrome('abba')
    'abba'
    >>> make_palindrome('ab')
    'aba'
    """
    if not string:
        return ''

    n = len(string)
    # Find the smallest i such that string[i:] is a palindrome.
    # That gives the longest palindromic suffix.
    for i in range(n):
        if is_palindrome(string[i:]):
            # Append reverse of prefix that comes before palindromic suffix
            prefix = string[:i]
            return string + prefix[::-1]

    # In practice the loop always returns (because the last char is a palindrome),
    # but return a fallback just in case.
    return string + string[::-1]


if __name__ == '__main__':
    import doctest

    failed, attempted = doctest.testmod()
```

```
    if failed:
        print(f"{failed} doctest(s) failed out of {attempted})")
    else:
        print(f"All {attempted} doctest(s) passed.")
```

E.3 Manual Rule

```
from typing import List


def is_palindrome(string: str) -> bool:
    """Return True if the given string is a palindrome.

    A palindrome reads the same forwards and backwards. The check is
    performed using Python slicing which correctly handles Unicode
    codepoints.

    Args:
        string: Input string to test.

    Returns:
        True if ``string`` is a palindrome, False otherwise.

    Examples:
        >>> is_palindrome('')
        True
        >>> is_palindrome('a')
        True
        >>> is_palindrome('aba')
        True
        >>> is_palindrome('ab')
        False
    """
    return string == string[::-1]


def _prefix_function(s: str) -> List[int]:
    """Compute prefix-function (a.k.a. failure function) for KMP algorithm.

    The prefix-function for a string ``s`` is an array ``pi`` where ``pi[i]``
    is the length of the longest proper prefix of ``s[:i+1]`` that is also
    a suffix of ``s[:i+1]``.

    This implementation runs in O(n) time and O(n) memory.

    Args:
        s: Input string.

    Returns:
        A list of integers representing the prefix-function values for each
        position in ``s``.

    Examples:
        >>> _prefix_function('ababaca')
        [0, 0, 1, 2, 3, 0, 1]
    """
    n: int = len(s)
    pi: List[int] = [0] * n
    for i in range(1, n):
```

```
        j: int = pi[i - 1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1]
        if s[i] == s[j]:
            j += 1
        pi[i] = j
    return pi


def make_palindrome(string: str) -> str:
    """Return the shortest palindrome that begins with "string" by appending
    characters to its end.

    The function finds the longest palindromic suffix of the input and then
    appends the reverse of the remaining prefix. This yields the minimal
    number of characters to append so the result is a palindrome that starts
    with the original string.

    Algorithmic idea:
    - Let ``rev`` be the reverse of ``string``.
    - Compute the prefix-function on ``rev + '#' + string``; the value at the
      last position is the length ``k`` of the longest prefix of ``rev`` that
      matches a suffix of ``string`` equivalently, the longest palindromic
      suffix of ``string``.
    - Append the reverse of the prefix ``string[:len(string) - k]`` to the
      end of ``string``.

    This runs in linear time O(n) and linear space O(n).

    Args:
        string: The input string. May be empty.

    Returns:
        The shortest palindrome that begins with ``string``. If ``string`` is
        already a palindrome, it is returned unchanged.

    Examples:
        >>> make_palindrome('')
        ''
        >>> make_palindrome('cat')
        'catac'
        >>> make_palindrome('cata')
        'catac'
        >>> make_palindrome('race')
        'racecar'
        >>> make_palindrome('a')
        'a'
    """
    n: int = len(string)
    if n <= 1:
        return string

    rev: str = string[::-1]
    # Build a combined string to compute how much of rev matches a suffix of
    # string (i.e. longest palindromic suffix of string).
    combined: str = rev + "#" + string
    pi: List[int] = _prefix_function(combined)
    k: int = pi[-1] if pi else 0  # length of longest palindromic suffix
```

```
    # Append reverse of the prefix that is not part of the palindromic suffix.
    to_append: str = (string[: n - k])[::-1]
    return string + to_append


if __name__ == '__main__':
    # Simple demonstration and smoke tests
    examples = ["", "cat", "cata", "race", "abac"]
    for ex in examples:
        print(f"{ex!r}->{make_palindrome(ex)!r}")
```

E.4 Prompt-Ops

```
def is_palindrome(string: str) -> bool:
    return string == string[::-1]


def make_palindrome(string: str) -> str:
    if not string: return ''
    for i in range(len(string)):
        if string[i:] == string[i:][::-1]:
            return string + string[:i][::-1]
    return string + string[::-1]
```

E.5 RuleEvolve

```
def is_palindrome(string: str) -> bool:
    return string==string[::-1]

def make_palindrome(string: str) -> str:
    for i in range(len(string)):
        if string[i:]==string[i:][::-1]:
            return string+string[:i][::-1]
    return string
```

F Manual Rules

Role Definition

- You are a **Python master**, a highly experienced **tutor**, a **world-renowned ML engineer**, and a **talented data scientist**.
- You possess exceptional coding skills and a deep understanding of Python's best practices, design patterns, and idioms.
- You are adept at identifying and preventing potential errors, and you prioritize writing efficient and maintainable code.
- You are skilled in explaining complex concepts in a clear and concise manner, making you an effective mentor and educator.
- You are recognized for your contributions to the field of machine learning and have a strong track record of developing and deploying successful ML models.
- As a talented data scientist, you excel at data analysis, visualization, and deriving actionable insights from complex datasets.

Technology Stack

- **Python Version:** Python 3.10+
- **Dependency Management:** Poetry / Rye
- **Code Formatting:** Ruff (replaces black, isort, flake8)
- **Type Hinting:** Strictly use the typing module.
- **Testing Framework:** pytest
- **Documentation:** Google style docstring

- **Environment Management:** conda / venv
- **Containerization:** docker, docker-compose
- **Asynchronous Programming:** Prefer async and await
- **Web Framework:** fastapi
- **Demo Framework:** gradio, streamlit
- **LLM Framework:** langchain, transformers
- **Vector Database:** faiss, chroma (optional)
- **Experiment Tracking:** mlflow, tensorboard (optional)
- **Hyperparameter Optimization:** optuna, hyperopt (optional)
- **Data Processing:** pandas, numpy, dask (optional), pyspark (optional)
- **Version Control:** git
- **Server:** gunicorn, uvicorn (with nginx or caddy)
- **Process Management:** systemd, supervisor

Coding Guidelines

1. Pythonic Practices

- **Elegance and Readability:** Strive for elegant and Pythonic code.
- **PEP 8 Compliance:** Adhere to PEP 8 guidelines.

2. Modular Design

- **Single Responsibility Principle:** Each module/file should have a single responsibility.
- **Reusable Components:** Favor composition over inheritance.

3. Code Quality

- **Comprehensive Type Annotations:** All members must have type annotations.
- **Detailed Docstrings:** All functions must have Google-style docstrings.
- **Thorough Unit Testing:** Aim for high test coverage (90% or higher) using pytest.
- **Robust Exception Handling:** Use specific exception types.

4. ML/AI Specific Guidelines

- **Experiment Configuration:** Use hydra or yaml.
- **Data Pipeline Management:** Employ scripts or dvc.
- **Model Versioning:** Utilize git-lfs or cloud storage.
- **LLM Prompt Engineering:** Dedicate a module for managing Prompt templates.

5. Performance Optimization

- **Asynchronous Programming:** Leverage async and await.
- **Caching:** Apply functools.lru_cache or @cache.
- **Resource Monitoring:** Use psutil to monitor resource usage.

6. API Development with FastAPI

- **Data Validation:** Use Pydantic models for validation.
- **Dependency Injection:** Use FastAPI's dependency injection.
- **Security:** Implement OAuth 2.0 or JWT.

Others

- Prioritize new features in Python 3.10+.
- Use the most modern and efficient libraries when appropriate.
- Ensure examples are self-contained and executable.