Let's Revise Step-by-Step: A Unified Local Search Framework for Code Generation with LLMs

Zhiyi Lyu[1] **Jianguo Huang**[1] **Yanchen Deng**[1]*

Steven Hoi[2] **Bo An**[1]

[1] Nanyang Technological University [2] Alibaba Group

Abstract

Large Language Models (LLMs) with inference-time scaling techniques show promise for code generation, yet face notable efficiency and scalability challenges. Construction-based tree-search methods suffer from rapid growth in tree size, high token consumption, and lack of anytime property. In contrast, improvement-based methods offer better performance but often struggle with uninformative reward signals and inefficient search strategies. In this work, we propose **ReLoc**, a unified local search framework which effectively performs step-by-step code revision. Specifically, ReLoc explores a series of local revisions through four key algorithmic components: initial code drafting, neighborhood code generation, candidate evaluation, and incumbent code updating, each of which can be instantiated with specific decision rules to realize different local search algorithms such as Hill Climbing (HC) or Genetic Algorithm (GA). Furthermore, we develop a specialized revision reward model that evaluates code quality based on revision distance to produce fine-grained preferences that guide the local search toward more promising candidates. Finally, our extensive experimental results demonstrate that our approach achieves superior performance across diverse code generation tasks, significantly outperforming both construction-based tree search as well as the state-of-the-art improvement-based code generation methods. The code is available at https://github.com/alphatogo/ReLoc.

1 Introduction

Large Language Models (LLMs) like GPT-4 (Achiam et al., 2023) and Claude (Wang et al., 2024a) have demonstrated remarkable capabilities in code-related tasks, including code generation (Chen et al., 2021; Austin et al., 2021), repair (Xia and Zhang, 2022; Jiang et al., 2023; Jin et al., 2023), and optimization (Shypula et al., 2023; Cummins et al., 2023). However, when facing challenging tasks, their auto-regressive token generation process prohibits the use of additional computational resources to achieve better performance (Yao et al., 2023; Snell et al., 2024). To fully unleash the power of LLMs, recent studies have focused on inference-time scaling techniques like construction-based tree-search algorithms (Feng et al., 2023; Wang et al., 2024b), which incrementally build a high-quality full response via exploring a tree of intermediate reasoning steps guided by a value model (Wang et al., 2023) or a Process-based Reward Model (PRM) (Lightman et al., 2023).

Despite their potential, construction-based tree-search methods suffer from a rapid increase in tree size and excessive token consumption as the number of reasoning steps grows, which inevitably leads to insufficient exploration given a practical budget. Besides, these methods do not hold the *anytime*

*Correspondence to: ycdeng@ntu.edu.sg

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

property (Zilberstein, 1996), since they cannot return a response until they find a reasoning path from the root to a leaf in the search tree. In contrast, recent approaches (Li et al., 2024a; Light et al., 2024) that utilize multi-turn improvements (Zheng et al., 2024) on complete responses for code generation have shown promise. These multi-turn approaches explore a series of local revisions on the code by feeding the execution feedback from public test cases (Xia et al., 2024) back to the LLM, which mirrors how humans iteratively refine code drafts and enjoys the anytime property.

However, the existing improvement-based approaches still face significant challenges in efficiently finding high-quality codes (Olausson et al., 2023). First, many of those methods rely on *ad-hoc* reward functions (e.g., the pass rate on the public test cases or simply an LLM's self-evaluated score) to measure the code quality, which may fail to provide informative direction to guide the search process (cf. Figure 1). In more detail, since the number of public test cases is usually small (e.g., 2-3 test cases per task), the pass rate often collapse to binary signals (i.e., 0% or 100%), offering little guidance in selecting promising code revisions. LLM-based self-evaluation, on the other hand, is prone to hallucinations (Zhang et al., 2024a) and can mislead the search by inaccurately assessing a code revision's potential. Second, the inefficient code revision generation and search algorithms exacerbate the token consumption. For example, the agentic methods (Li et al., 2024a; Wang et al., 2024a) exploit complex workflows which would consume a large number of tokens even over a few improvement iterations. Besides, search algorithms like Monte Carlo Tree Search (MCTS) (Li et al., 2024b) often require a significant number of improvement iterations to balance exploration and exploitation, leading to excessive computational overhead, e.g., running at least 50 iterations for comparable performance which consumes over 20,000 tokens on a single task. An extended discussion of related work is available in the Appendix.

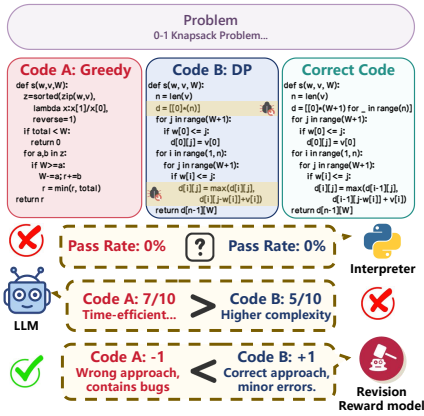


Figure 1: In the 0-1 Knapsack problem, the LLM favors an efficient but incorrect greedy solution (Code A) over a conceptually correct yet buggy DP version (Code B). As the tied pass rate offers no guidance, a revision reward model instead prioritizes candidates that are easier to revise into correct solutions.

In light of this, we introduce a lightweight, unified local search framework for improvement-based code generation with LLMs. The core idea behind our approach is to explore the neighborhood of the incumbent code with simple-yet-effective decision rules. To this end, we first frame the iterative improvement process within a local search framework with four key algorithmic components: initial code drafting, neighborhood code generation, candidate evaluation and incumbent code updating. Each component can be instantiated with a specific decision rule, allowing the development of different local search algorithms for code generation with LLMs. Furthermore, to facilitate candidate evaluation in each iteration, we develop a specialized revision reward model tailored for local search which is trained to prefer the code with a smaller *revision distance*, i.e., the minimum number of revision steps required to transform it into a corrected version. Intuitively, instead of solely maximizing the pass rate on the public test cases which sometimes can be uninformative, our reward model works directly on the textual space and guides the local search to explore the codes *close to* the correct ones. Specifically, our main contributions are summarized as follows.

1) We propose ReLoc, a lightweight and unified local search framework for code generation. ReLoc can be effectively instantiated into different local search algorithms such as Hill Climbing (HC) (Russell and Norvig, 2016) and Genetic Algorithm (GA) (Mitchell, 1998) by implementing each algorithmic component with a specific decision rule.
2) We develop a revision reward model trained with pairwise supervisions derived from revision distance comparisons. By constructing win/loss pairs based on which candidate is closer to the corrected code, we train the reward model using the Bradley–Terry framework (Bradley and Terry, 1952) to produce fine-grained preferences that guide the local search toward promising candidates, even when explicit correctness signals are uninformative.
3) We conduct extensive experimental evaluations on popular code generation benchmarks including LiveCodeBench (Jain et al., 2024) and TACO (Li et al., 2023). The results demon-

strate that our local search approaches consistently outperform both construction-based tree-search methods and existing improvement-based approaches, achieving a **33.8%→38.4%** improvement in Pass@1 on LiveCodeBench and an **11.5%→15.3%** gain on TACO over the strongest baseline, while reducing token consumption by **37%** (cf. Figure 4).

2 Preliminaries

Code generation tasks. A code generation task can be defined as a triple $\mathcal{T}_i = \langle x_i, u_i, v_i \rangle$ where where x_i is the problem statement given in natural language, u_i is a set of public test cases, and v_i is a set of private test cases. A code sample a is deemed correct if it passes all test cases in both u_i and v_i. Given a set of code generation tasks $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_N\}$, the performance of a code generation policy π is measured by $\text{Pass@}k = \mathbb{E}_{\mathcal{T}_i \sim \mathcal{T}} \left[1 - \frac{\binom{n-c_i}{k}}{\binom{n}{k}} \right]$, where n is the total number of codes sampled for each task with policy π and c_i is the number of correct code samples for task $\mathcal{T}_i$ (Chen et al., 2021).

Improvement-based code generation. The improvement-based code generation process can be formalized as an episodic Partially Observable Markov Decision Process (POMDP), defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, p, r, T \rangle$. The state $s_t \in \mathcal{S}$ includes the code generation task $\mathcal{T}_i$, the current code sample a_t, the execution feedback from public test cases $f(a_t; u)$ and the one from private test cases $f(a_t; v)$ for each time step t. Particularly, the initial state $s_0 = \mathcal{T}_i$. The action $a_t \in \mathcal{A}$ is a token sequence which constitutes a code sample. The state transition function $p : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ deterministically updates the state by evaluating the code sample a_{t+1}, yielding s_{t+1}. The observation $o_t \in \mathcal{O}$ is a subset of s_t, consisting of $\langle x_i, u_i \rangle$, a_t and $f(a_t; u)$, as the LLM can only access public test cases and the corresponding feedback. The reward function r assigns a reward of 1 if the last time step code sample a_T passes all test cases, where T is the time horizon. The history $\tau_t = (o_0, a_1, o_1, \ldots, o_t)$ captures the whole trajectory of actions and observations up to the t-th time step. Finally, the policy π outputs code revision based on the history for each time step, i.e., $a_{t+1} \sim \pi(\cdot|\tau_t)$.

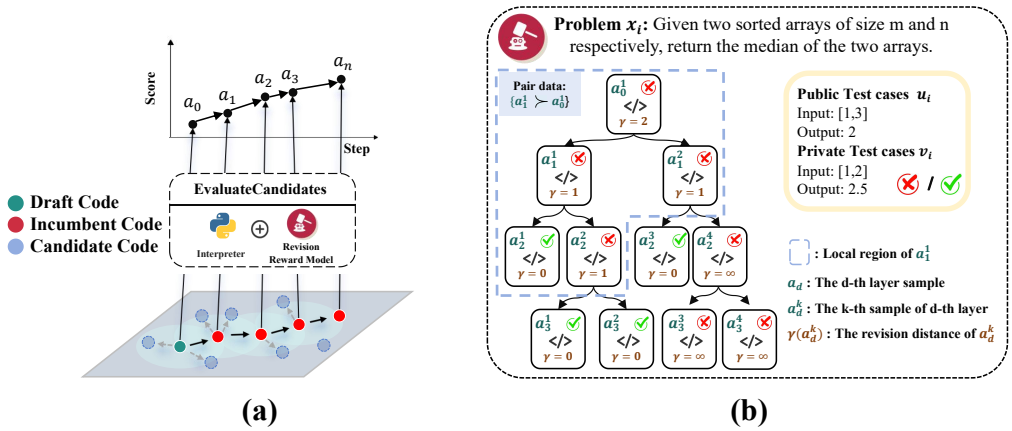
Local search. Local search (Pirlot, 1996) is an important class of heuristic methods to solve computationally challenging optimization problems. Instead of systematically exploring the whole solution space, local search iteratively improves an incumbent solution by exploring its local neighborhood until a given termination condition is met. Local search naturally enjoys the anytime property (Zilberstein, 1996), in the sense that it can return a solution at anytime and the solution quality is monotonically non-decreasing over time by simply caching the best solution found so far.

3 Methodology

While existing improvement-based methods offer the advantage of iterative code revision through multi-turn interactions, they suffer from two critical limitations: (1) inefficient exploration due to complex revision workflows that consume excessive tokens, and (2) inaccurate reward functions that provide limited guidance for selecting promising candidates. To address these challenges, we introduce a lightweight and unified **Re**vision **Lo**cal Sear**ch** (ReLoc) framework, which leverages simple-yet-efficient decision rules to perform step-by-step code revision (cf. Figure 2). In Section 3.1, we elaborate the essentials and key algorithmic components of ReLoc. To address the limitations of prior ad-hoc evaluation heuristics, we further introduce a revision reward model trained to rank code candidates according to their revision distance in Section 3.2. Finally, we show the flexibility and expressiveness of our ReLoc framework by implementing two well-known local search algorithms (i.e., Hill Climbing and Genetic Algorithm) in Section 3.3.

3.1 Local Search Framework

As shown in Algorithm 1, our ReLoc framework consists of four algorithmic components that can be instantiated with different decision rules: (1) DRAFTCODE, which generates an initial code sample population P_0 based on the problem statement x_i and public test cases u_i of code generation task $\mathcal{T}_i$; (2) GENERATENEIGHBORHOOD, which constructs a set of neighborhood code samples P_t by prompting the LLM π to propose new code revisions based on the history and the execution feedback



(a) **(b)**

Figure 2: **(a) Overview of the ReLoc framework.** Starting from an initial draft code, ReLoc iteratively generates a neighborhood of candidate code samples around the current incumbent code. Each candidate is assessed by the EVALUATECANDIDATES, which utilizes an interpreter and our revision reward model. **(b) Training the Revision Reward Model.** A code tree is constructed by iteratively revising an initial incorrect code. Each code sample in the tree is labeled with its revision distance, i.e., the minimum number of revisions needed to reach a correct version. Pairwise comparisons in local region create preference data.

from the public test cases; (3) EVALUATECANDIDATES, which assigns each candidate in current neighborhood P_t with a score measuring its quality; and (4) UPDATEINCUMBENT, which implements the move strategy by selecting the next incumbent solution a_t from the neighborhood P_t. Finally, we maintain the best-so-far code sample a^* with score e^* to enforce the anytime property.

Algorithm 1 ReLoc: Revision local search framework for code generation with LLMs

Input: Code generation task $\mathcal{T}_i$, LLM π, iteration limit T
Output: Code sample a^*

$P_0 \leftarrow$ generate initial code sample population with DRAFTCODE
$E_0 \leftarrow$ evaluate code samples P_0 with EVALUATECANDIDATES
$a_0 \leftarrow$ select a code sample from P_0 with UPDATEINCUMBENT, $a^* \leftarrow a_0$, $e^* \leftarrow E_0[a_0]$
for $t = 1, \ldots, T$ **do**
  $P_t \leftarrow$ generate neighborhood code samples with GENERATENEIGHBORHOOD
  $E_t \leftarrow$ evaluate code samples P_t with EVALUATECANDIDATES
  $a_t \leftarrow$ select a code sample from P_t with UPDATEINCUMBENT
  if $E_t[a_t] > e^*$ **then**
    $a^* \leftarrow a_t$, $e^* \leftarrow E_t[a_t]$
return a^*

3.2 Revision Reward Model

A key question in implementing ReLoc is how to evaluate the quality of the generated code candidates (cf. EVALUATECANDIDATES), which determines the search direction for each iteration. As we will show in Section 4.2, simple heuristics like pass rate on public test cases or LLM-based self-evaluation scores fail to effectively guide the search direction, since they often either collapse to binary signals or are prone to hallucinations. Outcome-based reward model (Shen and Zhang, 2024), on the other hand, solely focuses on the correctness of the code samples rather than how likely an incorrect candidate will be revised into a correct code in future steps, which is also not applicable to our scenario.

Instead of directly assessing the correctness, we train a specialized revision reward model to rank code samples according to their *revision distance*, i.e., the minimal number of revision steps required to transform it into the correct version. This way, in addition to prioritizing correct code samples, we also effectively differentiate incorrect ones and enable local search to focus on promising candidates with smaller revision distance, thus guaranteeing the overall search efficiency even when the correctness signals are uninformative (e.g., 0% pass rate on public test cases for all candidates).

To train our reward model, given a training task $\mathcal{T}_i$, we build a special code tree (cf. Figure 2(b)) where the root is an **incorrect** code a_0^1 sampled from the LLM policy. Then we incrementally expand the tree in a **breadth-first** fashion up to a depth limit $d_{\max}$. That is, for each incorrect code sample a_{d-1}^j in the $(d-1)$-th layer, we prompt the LLM π to generate K code revisions for d-th layer:

$$a_d^{K(j-1)+k} \sim \pi\left(\cdot|REVISE\_PT\left(x_i, a_{d-1}^j, f(a_{d-1}^j; u_i)\right)\right) \qquad k \in \{1, \ldots, K\}, \tag{1}$$

where $REVISE\_PT$ is the prompt template instructing the LLM π to revise the code sample a_{d-1}^j according to the problem statement x_i and execution feedback $f(a_{d-1}^j; u_i)$ on public test cases, and the generated code revision $a_d^{K(j-1)+k}$ is then inserted to the tree as a child of a_{d-1}^j.

Once the code tree is built, we recursively label the revision distance of each code sample in the tree according to the following rule:

$$\gamma(a_d^j) = \begin{cases} 0, & a_d^j \text{ is correct;} \\ \infty, & a_d^j \text{ is incorrect } \wedge\ Ch(a_d^j) = \emptyset; \\ 1 + \min_{a \in Ch(a_d^j)} \gamma(a), & \text{otherwise,} \end{cases} \tag{2}$$

where $Ch(a_d^j)$ is the children of a_d^j in the code tree. Note that a code sample is considered correct if and only if it passes all public test cases u_i and private test cases v_i. After that, we proceed to build win/loss pairs of code samples and train our reward model with Bradley–Terry framework (Bradley and Terry, 1952; Ouyang et al., 2022). Particularly, we confine the comparison within a small region around a code sample in the code tree to reflect the *locality* of the local search. Specifically, for code sample a_d^j, we consider the following neighborhood code samples:

$$\mathcal{N}(a_d^j) = \{Pa(a_d^j)\} \cup Ch(a_d^j) \cup Sib(a_d^j), \tag{3}$$

where $Pa(a_d^j)$ is its parent and $Sib(a_d^j) = \{a | a \in Ch(Pa(a_d^j)) \wedge a \neq a_d^j\}$ is the siblings of a_d^j. Then for each code sample $a' \in \mathcal{N}(a_d^j)$ with $\gamma(a_d^j) < \gamma(a')$, we model the preference probability as

$$\mathbb{P}(a_d^j \succ a' | x_i) = \sigma\left(R_\phi(a_d^j | x_i) - R_\phi(a' | x_i)\right), \tag{4}$$

where $\sigma(\cdot)$ is the sigmoid function and $R_\phi(a|x_i)$ represents the learned reward score for code samples a given problem statement x_i. The learning objective of the reward model is to maximize the expected log-probability:

$$\max_\phi \mathbb{E}_{(x_i, a, a') \sim \mathcal{D}}[\log \mathbb{P}(a \succ a' | x_i)], \tag{5}$$

where the pair dataset $\mathcal{D}$ is constructed by collecting the pairs of code samples in each code tree.

3.3 Case Study

To demonstrate the flexibility and expressiveness of our ReLoc framework, we now present two well-known local search algorithms, i.e., Hill Climbing (HC) (Russell and Norvig, 2016) and Genetic Algorithm (GA) (Mitchell, 1998) for improvement-based code generation with LLMs by instantiating each algorithmic component with specific decision rules.

DRAFTCODE. It is widely acknowledged that the quality of the initial solution has a significant impact on the performance of local search (Lourenço et al., 2018). Therefore, to guarantee the quality of the initial code sample population P_0, we adopt a Plan-then-Generate paradigm by firstly prompting the LLM to enumerate N diverse natural language plans that outline different algorithmic strategies. Then for each plan, we prompt the LLM to synthesize a corresponding code implementation, forming a candidate pool $P_0 = \{a_0^1, \ldots, a_0^N\}$.

GENERATENEIGHBORHOOD. For each iteration t, we generate neighborhood code samples P_t according to the following rules.

- **Hill Climbing.** Given the incumbent code sample a_{t-1} and the feedback $f(a_{t-1}; u_i)$ from public test cases u_i, we generate the neighborhood code revisions by first prompting the

LLM to propose K natural language revision strategies $Q_t = \{q_t^1, \ldots, q_t^K\}$ (e.g., "fix condition logic", "refactor loop"). Then for each strategy q_t^k, we prompt the LLM π to generate a candidate code revision:

$$P_t = \{a_t^1, \ldots, a_t^K\}, \quad a_t^k \sim \pi\left(\cdot \mid HC\_PT\left(x_i, a_{t-1}, f(a_{t-1}; u_i), q_t^k\right)\right), \tag{6}$$

where HC_PT is the prompt template instructing the LLM to generate a code revision based on problem statement x_i, previous code a_{t-1}, execution feedback $f(a_{t-1}; u_i)$ and revision strategy q_t^k.

- **Genetic Algorithm.** For each iteration $t > 1$, we select two parent code samples a, a' from the history $(a_0, \ldots, a_{t-1})$ according to their *fitness*, i.e., the scores evaluated by EVALUATECANDIDATES. Particularly, we select parent code samples from P_0 when $t = 1$. Then we prompt the LLM to generate K new candidates:

$$P_t = \{a_t^1, \ldots, a_t^K\}, \quad a_t^k \sim \pi\left(\cdot \mid GA\_PT\left(x_i, a, f(a; u_i), a', f(a'; u_i)\right)\right), \tag{7}$$

where GA_PT is the prompt template[2] instructing the LLM to generate a code revision by combining the strengths or addressing the shared weaknesses of the parent code samples. To maintain diversity, we also implement an *aging* mechanism where each code sample is disqualified from being selected as a parent after it has been used in this role 3 times.

EVALUATECANDIDATES. To evaluate the candidate code samples P_t, we propose a synergistic approach that leverages both public test cases and the learned revision reward model. Let $P_{\text{pass}} = \{a \in P_t \mid f(a; u_i) = \texttt{pass}\}$ be the subset of candidates that pass all public test cases. Then, the evaluation score of candidate a is defined as:

$$E_t[a] = \begin{cases} R_\phi(a|x_i), & \text{if } P_{\text{pass}} = \emptyset; \\ R_\phi(a|x_i), & \text{if } a \in P_{\text{pass}}; \\ -\infty, & \text{otherwise.} \end{cases} \tag{8}$$

That is, when no candidate passes all public tests, we fall back to using the reward model R_ϕ to score all candidates. If there are successful candidates, we score them using R_ϕ, while assigning the rest with a score of $-\infty$. This way, we provide fine-grained preferences that guide the local search toward promising candidates by explicitly differentiating the candidates with the same correctness signal.

UPDATEINCUMBENT. Given the current code samples P_t and the corresponding evaluation scores, UPDATEINCUMBENT aims to select a code sample as the incumbent solution for iteration t. Technically, decision rules like ϵ-greedy, Boltzmann distribution (Landau and Lifshitz, 2013) or more complex simulated annealing acceptance rule (Delahaye et al., 2018) can be applied. Here we choose to greedily select the one with the maximum evaluation score for simplicity and efficiency, i.e., $a_t = \arg\max_{a \in P_t} E_t[a]$, where ties are broken alphabetically.

4 Experiments

In this section, we present extensive empirical evaluations to demonstrate our superiority across diverse code-related tasks. Our experiments aim to answer the following research questions:

- **RQ1:** How well do our local search methods perform on code-related tasks compared to state-of-the-art construction-based and improvement-based approaches?
- **RQ2:** Can our proposed revision reward model provide more effective guidance for local search than heuristic rewards or outcome-based reward model?
- **RQ3:** How does the performance of our local search methods scale with increasing token budgets at inference time?
- **RQ4:** What are the individual contributions of planning, revision strategies, and execution feedback to the overall performance of our local search methods?

[2]All prompt templates are provided in the Appendix.

Table 1: Pass@1 accuracy (%) of different methods on the LiveCodeBench and TACO benchmarks. All methods are evaluated under the same token budget (7K) to ensure fair comparison. Methods are categorized as construction-based (Con.) or improvement-based (Imp.), and further distinguished by reward functions: PRM (), self-evaluation (), pass rate (), and revision reward model ().

Methods	Cat.	Rew.	LiveCodeBench		TACO	
			Code gen	Code repair	Code gen	Code repair
RAP	Con.		22.9	21.2	5.4	4.1
TOT	Con.		25.6	20.4	6.8	3.7
Code Tree	Imp.		27.7	24.1	8.2	5.6
Reflexion	Imp.		25.6	23.9	7.1	6.1
Plan Search	Imp.		32.7	31.3	11.2	8.2
BoN	Imp.		30.2	30.5	10.8	6.3
SFS	Imp.		32.1	27.3	10.5	8.1
ORPS	Imp.		28.8	26.6	9.8	7.8
ReLoc_HC **(Ours)**	Imp.		**38.4**	**33.4**	13.3	9.7
ReLoc_GA **(Ours)**	Imp.		35.7	29.9	**15.3**	**11.5**

4.1 Experimental Setup

Benchmarks. We evaluate our methods on two benchmarks: **LiveCodeBench** (Jain et al., 2024), using 511 problems from May 2023–May 2024 for training and 268 problems from Jul 2024–Jan 2025 for testing; and **TACO** (Li et al., 2023), which aggregates problems from CodeContests, APPS (Hendrycks et al., 2021), and other sources. From TACO's original 25,443 training and 1,000 test problems, we randomly sample 4,000 and 200 respectively due to computational constraints. Both datasets provide 2–3 public test cases per problem. For code repair, we construct a buggy-code dataset by sampling incorrect solutions from diverse models (`Qwen2.5-7B/32B/70B`, `GPT-4o`) to ensure a range of error types and complexities.

Implementation details. Our base model throughout the experiments is `Qwen2.5-32B-Instruct`. During training, we use the training splits of LiveCodeBench and TACO to construct code revision pairs as described in Section 3.2, where we expand code trees via breadth-first search up to a maximum depth of $d_{max} = 5$, with $K = 3$ revisions per node. These pairs are used to train a specialized revision reward model based on `Qwen2.5-7B-Instruct`, following the reward modeling procedure from (von Werra et al., 2020). The distribution of pairwise comparisons is shown in Figure 3. For inference, we adopt decoding settings consistent with (Jain et al., 2024), using a temperature of 0.2 and top-p of 0.95. To ensure fair comparison across all methods, we fix the token budget to 7K tokens per problem. Additional implementation and training details are provided in the Appendix.

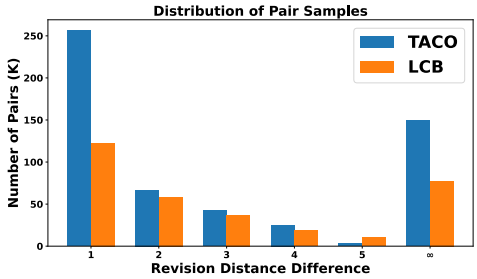


Figure 3: Distribution of 872K training pairs by revision distance difference (γ(loss) − γ(win)).

Baselines. We compare our methods with the following approaches. (1) **Construction-based** approaches like ToT (Yao et al., 2023) and RAP (Hao et al., 2023), which build solutions via search (BFS or MCTS); (2) **Improvement-based methods with self-evaluation**, which leverage internal assessment mechanisms to guide search, such as CodeTree (Li et al., 2024a) and Reflexion (Shinn et al., 2023), which use multi-agent evaluation or test case generation with reflective learning; and Plan-and-Search (Wang et al., 2024c), which prompts LLMs to search among candidate solution plans expressed in natural language. (3) **Improvement-based methods with pass rate**, which utilize performance on public test cases to provide reward signals, including Best-of-N (BoN) (Cobbe et al.,

2021), which randomly samples N solutions and filters according to the pass rate; SFS (Light et al., 2024), which employs MCTS to revise code using pass rate as the reward function; and OPRS (Yu et al., 2024), which combines self-scoring and pass rate to guide beam search.

4.2 Empirical Results

Performance comparison. We systematically compare our local search methods against state-of-the-art baselines and present the results in Table 1. It can be concluded that ReLoc consistently outperforms all baseline methods across both LiveCodeBench and TACO benchmarks on the Pass@1 metric. Specifically, ReLoc_HC achieves the best performance of **38.4%** and **33.4%** on LiveCodeBench, while ReLoc_GA reaches **15.3%** and **11.5%** on TACO. It is interesting to find that ReLoc_HC outperforms ReLoc_GA on the easier LiveCodeBench benchmark, while the reverse is observed on the more challenging TACO tasks. This phenomenon highlights the distinct merits of each algorithm: ReLoc_HC is particularly effective for straightforward tasks where the underlying solution is relatively obvious and the primary challenge lies in fixing minor bugs or syntax errors, while ReLoc_GA excels in more complex scenarios by strategically integrating the advantages from parent code samples to discover non-trivial solutions for conceptually challenging tasks in TACO.

Compared to construction-based methods like ToT and RAP, ReLoc exhibits significantly superior performance under the same computational budget, thanks to the anytime property of local search algorithms. On the other hand, improvement-based methods with self-evaluation often are prone to hallucinations, leading to unreliable assessments of code quality. While such methods can be somewhat effective for guiding high-level search, as evidenced by Plan Search, they cannot fully leverage the detailed execution feedback. Unlike Plan Search that explores high-level strategies in an abstract plan space, ReLoc implements fine-grained planning and code-level revision, achieving an average improvement of **4.9%** over Plan Search on Code gen tasks.

Finally, the improvement-based techniques using pass rates from public test cases struggle with sparse or binary reward signals, which also offer insufficient guidance for search. SFS-based MCTS methods require extensive exploration with high computational costs, while ORPS employing beam search suffers from low exploration efficiency, unable to select the most promising code candidates. ReLoc adeptly navigates these challenges through lightweight decision rules, guided by a revision reward model, achieving an average improvement of **5.6%** over pass rate-based methods across different tasks.

Effectiveness of revision reward model. To evaluate the effectiveness of our revision reward model when guiding local search, we conduct controlled experiments on LiveCodeBench and TACO on ReLoc_HC with different reward functions. Specifically, we compare revision reward model against five baselines: public test case pass rate, LLM-generated test case pass rate, LLM self-evaluation, Skywork-27B reward model (Liu et al., 2024), and ORM-7B, i.e., an outcome-based reward model trained with the same architecture as the revision reward model (`Qwen2.5-7B`).

As shown in Table 2, pass rate and self-evaluation heuristics offer weak guidance. That is not surprising because execution-based scores are often coarse or binary, while self-evaluated scores often suffer from hallucinations, leading to unreliable rankings. In contrast, methods using a reward model perform better, with our revision reward model outperforming ORM by **35.9%**→**38.4%** and **9.7%**→**13.3%** on LiveCodeBench and TACO. We attribute this to the ability of the revision reward model to differentiate incorrect candidates based on their likelihood of future correction, which is a property the ORM lacks due to its exclusive focus on code correctness.

Inference-time scaling law. To further evaluate the scaling performance of ReLoc_HC with increasing computational resources, we vary the token budget from 1K to 15K. We compare our method against three representative baselines: BoN, ORPS, and ToT. Figure 4 illustrates the scaling behavior of different methods on the LiveCodeBench and TACO benchmarks. Notably, as the token budget increases, ReLoc_HC demonstrates a faster improvement in terms of Pass@1, highlighting the benefit of guided local search and our learned revision reward model. This is particularly evident in the LiveCodeBench, where ReLoc_HC reaches over **40%** Pass@1 under a 15K token budget, outperforming the BoN with the same budget by a significant margin.

Table 2: Pass@1 accuracy of ReLoc_HC with different reward functions under a 7K token budget. Our revision reward model achieves the highest Pass@1 on both LiveCodeBench and TACO, demonstrating strong inference-time performance without relying on test case generation or self-evaluation.

Reward Function	**Gen Test Case**	**Self Score**	**Reward Model**	**LiveCodeBench**	**TACO**
Pass Rate	✗	✗	✗	33.5	10.3
w/ Gen Test case	✓	✗	✗	29.0	8.4
Self Evaluation	✓	✓	✗	29.3	9.2
Skywork-27B	✗	✗	✓	31.9	9.4
ORM-7B	✗	✗	✓	35.9	9.7
Revision Reward Model **(Ours)**	✗	✗	✓	**38.4**	**13.3**

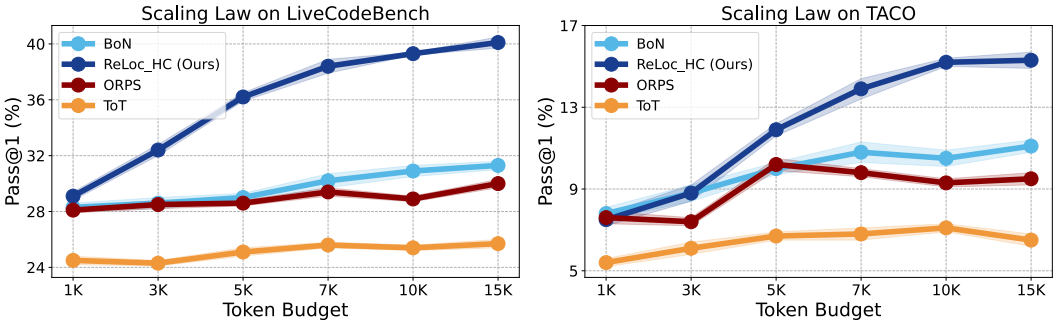


Figure 4: **Scaling Law.** Pass@1 accuracy of ReLoc_HC **(Ours)** compared to baselines (BoN, ORPS, ToT) on LiveCodeBench and TACO benchmarks as token budget increases from 1K to 15K.

Table 3: **Ablation study of ReLoc.** We evaluate the impact of core components in both **ReLoc_HC** and **ReLoc_GA**. Removing natural language plans, revision strategies, or execution feedback.

Method	**LiveCodeBench**		**TACO**	
	Pass@1 (%)	**Tokens (1K)**	**Pass@1 (%)**	**Tokens (1K)**
ReLoc_HC	38.4	7.1	13.3	7.6
w/o Natural Language Plans	36.9	6.5	13.7	7.1
w/o Revision Strategies	35.9	4.7	11.7	5.5
w/o Execution Feedback	34.8	7.5	11.4	7.7
ReLoc_GA	35.7	6.8	15.3	7.7
w/o Natural Language Plans	34.3	4.9	12.9	6.6
w/o Execution Feedback	34.6	5.8	13.5	6.8

Ablation study. We conduct an ablation study on both **ReLoc_HC** and **ReLoc_GA** to assess the importance of each design choice. As shown in Table 3, removing natural language plans during initialization and replacing them with randomly sampled code reduces performance. Notably, eliminating revision strategies significantly reduces the number of generated tokens (e.g., from 7.1K to 4.7K in ReLoc_HC). However, this also limits the diversity of candidate code samples explored during the search, ultimately resulting in inferior performance. Furthermore, execution feedback plays a particularly crucial role, as it enables precise and targeted revision in each iteration, improving the overall Pass@1 accuracy by 2.1%.

5 Conclusion

In this work, we present ReLoc, a lightweight and unified local search framework for improvement-based code generation with LLMs. Unlike computationally expensive construction-based inference-time scaling methods like ToT and MCTS, ReLoc finds high-quality solutions and enjoys the anytime property by exploring a series of local revisions of an established code sample. Besides, compared to the existing improvement-based methods, ReLoc leverages simple yet effective decision rules to navigate the search space. Furthermore, a specialized revision reward model effectively differentiates code samples based on the potential of each code sample being corrected in future steps, which provides fine-grained preferences when the correctness signal is uninformative. Finally, we show the flexibility and expressiveness of ReLoc by developing two well-known local search algorithms, i.e., Hill Climbing and Genetic Algorithm. Extensive experiments on benchmarks like LiveCodeBench and TACO validate the effectiveness of ReLoc in significantly improving code generation performance while reducing computational cost.

Acknowledgments

This research is supported by the RIE2025 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) (Award I2301E0026), administered by A*STAR, as well as supported by Alibaba Group and NTU Singapore through Alibaba-NTU Global e-Sustainability CorpLab (ANGEL).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

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*, 2024a.

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.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *ESEC/FSE*, pages 959–971, 2022.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *ICSE*, pages 1430–1442. IEEE, 2023.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with LLMs. In *ESEC/FSE*, pages 1646–1656, 2023.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36:11809–11822, 2023.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*, 2023.

Chaojie Wang, Yanchen Deng, Zhiyi Lyu, Liang Zeng, Jujie He, Shuicheng Yan, and Bo An. Q*: Improving multi-step reasoning for LLMs with deliberative planning. *arXiv preprint arXiv:2406.14283*, 2024b.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*, 2023.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *ICLR*, 2023.

Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–73, 1996.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329*, 2024a.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, Wei Cheng, et al. Scattered forest search: Smarter code space exploration with LLMs. *arXiv preprint arXiv:2411.05010*, 2024.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin Negrevergne, and Gabriel Synnaeve. What makes large language models reason in (multi-turn) code generation? *arXiv preprint arXiv:2410.08105*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2023.

Xiaoying Zhang, Baolin Peng, Ye Tian, Jingyan Zhou, Lifeng Jin, Linfeng Song, Haitao Mi, and Helen Meng. Self-alignment for factuality: Mitigating hallucinations in LLMs via self-evaluation. *arXiv preprint arXiv:2402.09267*, 2024a.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. RethinkMCTS: Refining erroneous thoughts in monte carlo tree search for code generation. *arXiv preprint arXiv:2409.09584*, 2024b.

Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. pearson, 2016.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT press, 1998.

Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. TACO: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

Marc Pirlot. General local search methods. *European Journal of Operational Research*, 92(3): 493–511, 1996.

Wei Shen and Chuheng Zhang. Policy filtration in rlhf to fine-tune llm for code generation. *arXiv preprint arXiv:2409.06957*, 2024.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *NeurIPS*, pages 27730–27744, 2022.

Helena Ramalhinho Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search: Framework and applications. In *Handbook of Metaheuristics*, pages 129–168. Springer, 2018.

Lev Davidovich Landau and Evgenii Mikhailovich Lifshitz. *Course of Theoretical Physics*. Elsevier, 2013.

Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. Simulated annealing: From basics to applications. In *Handbook of Metaheuristics*, pages 1–35. Springer, 2018.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*, 36:8634–8652, 2023.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves LLM search for code generation. *arXiv preprint arXiv:2409.03733*, 2024c.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. Outcome-refining process supervision for code generation. *arXiv preprint arXiv:2412.15118*, 2024.

Chris Yuhao Liu, Liang Zeng, Jiacai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. Skywork-reward: Bag of tricks for reward modeling in LLMs. *arXiv preprint arXiv:2410.18451*, 2024.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*, 2022.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024b.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354*, 2024.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, et al. Rlaif vs. rlhf: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. *arXiv preprint arXiv:2501.07301*, 2025.

Jinhao Jiang, Zhipeng Chen, Yingqian Min, Jie Chen, Xiaoxue Cheng, Jiapeng Wang, Yiru Tang, Haoxiang Sun, Jia Deng, Wayne Xin Zhao, et al. Technical report: Enhancing llm reasoning with reward-guided tree search. *arXiv preprint arXiv:2411.11694*, 2024.

Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. Generative reward models. *arXiv preprint arXiv:2410.12832*, 2024.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. Llm critics help catch llm bugs. *arXiv preprint arXiv:2407.00215*, 2024.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*, 2024c.

Xiusi Chen, Gaotang Li, Ziqi Wang, Bowen Jin, Cheng Qian, Yu Wang, Hongru Wang, Yu Zhang, Denghui Zhang, Tong Zhang, et al. Rm-r1: Reward modeling as reasoning. *arXiv preprint arXiv:2505.02387*, 2025.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Both the abstract and introduction sections outline our research scope, methodology, motivations, experimental findings, and key contributions. Refer to Section 1

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: In our work on ReLoc, due to limited resources, we did not utilize more powerful models like Qwen2.5-32B-instruct to train the revision reward model. Additionally, we only explored two different local search algorithms. Please refer to the appendix for more details.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: This paper does not include assumptions and proof.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We introduce all details of our method in Section 4.1 and appendix.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We submit our code and data as supplemental materials. We provide instructions that contain the exact command and environment needed to reproduce the results.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://nips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://nips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We specify all demonstration selection and test details in the Section 4.1. The full details can be found in our code, which be provided as supplemental material.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report average results and standard deviation to illustrate the statistical significance of our method in Figure 4

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Detailed compute resource specifications are provided in the appendix.

   Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: We confirm that our work fully complies with the NeurIPS Code of Ethics in all respects.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: This research aims to advance the fields of large language models and code generation. While our work may have various societal implications, we have chosen not to emphasize any specific impacts within the scope of this paper.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: The models and datasets presented in this paper are publicly accessible via HuggingFace and GitHub

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We cite the original paper that produced the code package or dataset.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [NA]

Justification: This work does not release new assets.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.

- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: : Our paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: We do not use LLMs for developing our core idea and methodology.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (https://neurips.cc/Conferences/2025/LLM) for what should or should not be described.

A Related Work

Code Generation with Large Language Models. Recent advances in large language models (LLMs) have significantly boosted code generation by leveraging pretraining on large-scale code corpora (Lu et al., 2021; Christopoulou et al., 2022; Guo et al., 2024; Hui et al., 2024). At inference time, two main strategies have emerged to further enhance performance. Construction-based methods generate solutions step-by-step, often guided by value models (Yao et al., 2023; Wang et al., 2024b), process-based reward models (Lightman et al., 2023), or planning techniques like Monte Carlo Tree Search (Hao et al., 2023; Zhou et al., 2023). In contrast, improvement-based approaches iteratively refine full code drafts using multi-turn updates, agentic workflows (Wang et al., 2024a; Zhong et al., 2024; Zhang et al., 2024b), and test-time feedback, sometimes with multi-agent collaboration (Li et al., 2024a; Zan et al., 2024). While effective, many of these methods are resource-intensive and complex. Our work introduces a lightweight local search framework that streamlines key components into an efficient and scalable loop.

Reward Models. Reward models play a central role in RLHF (Ouyang et al., 2022), providing learning signals for policy optimization. To reduce dependence on human-labeled data, RLAIF (Lee et al., 2023) proposed an automated reward data pipeline. More recently, reward models have been extended to reasoning tasks. Math-Shepherd (Wang et al., 2023) and others (Zhang et al., 2025) trained process-based reward models to guide inference-time strategies, while outcome-based models have supported tree search (Jiang et al., 2024). Generative Reward Models (GRMs) (Mahan et al., 2024; McAleese et al., 2024; Zhang et al., 2024c; Chen et al., 2025) further leverage CoT-based self-critique for scoring outputs. Distinct from these paradigms, we propose a reward model trained to estimate *revision distance*, capturing the minimal steps needed to reach a correct solution. This enables more efficient candidate evaluation and improves the effectiveness of iterative code refinement.

B Experimental Setup.

B.1 Revision Reward Model

This section outlines the hyperparameters and settings used during the training phase of the revision reward model. We trained the revision reward model on `Qwen2.5-7B-Instruct` using the TRL library (von Werra et al., 2020) with DeepSpeed ZeRO Stage 2 on 4 NVIDIA H100 GPUs. The training was conducted for one epoch on a combined dataset of LiveCodeBench and TACO. Table 4 details the training configuration.

Table 4: Revision reward model training Configuration

Parameter	**Value**
Mixed Precision	bf16
Batch Size per Device	8
Number of Epochs	1
Gradient Checkpointing	True
Learning Rate	5.0e-6
Logging Steps	25
Evaluation Strategy	Steps
Evaluation Interval	Every 500 steps
Save Interval	Every 3000 steps
Max Sequence Length	2048
Push to Hub	False
Optimizer	paged_adamw_32bit
Warmup Ratio	0.05
Learning Rate Scheduler	Cosine
Number of GPUs	4 × NVIDIA H100

B.2 Local Search Hyperparameters

We use `Qwen2.5-32B-Instruct` as the inference model throughout all experiments, with a decoding temperature of 0.2 and a top-p value of 0.95. All algorithms are run under a fixed token budget of 7,000 tokens per task.

For **Hill Climbing (HC)**, we initialize with 5 draft codes and expand 3 neighbors for each candidate during each improvement iteration.

For the **Genetic Algorithm (GA)**, we similarly maintain a population of 5 draft codes. In each iteration, we select 2 codes from the candidate pool as parent codes, with each code allowed to be selected as a parent up to 3 times.

C Additional Evaluation on GPT-4o

To further validate the effectiveness of ReLoc, we conduct experiments on the closed-source model `gpt-4o-2024-1120`. Specifically, we evaluate ReLoc and several baselines, including the state-of-the-art *Plan Search* algorithm and *Best of N* sampling, on the LiveCodeBench benchmark. For ReLoc, we employ the revision reward model trained as described in Section B.1 to guide the search process.

In our evaluation, the revision reward model guiding ReLoc's local search was trained entirely on data sampled from the open-source `Qwen2.5-32B-Instruct` model, a setting that differs in both distribution and model family from the target inference model, GPT-4o. Remarkably, as shown in Table 5, ReLoc achieves the highest Pass@1 score (44.2%) while consuming only 8.7K tokens on average—representing a 52% reduction in token usage compared to *Plan Search*. This underscores the efficiency of ReLoc's local search mechanism.

More importantly, these results demonstrate that the revision reward model, trained on Qwen-generated code trajectories, generalizes robustly to guide search on GPT-4o. This transferability is non-trivial: GPT-4o may exhibit different stylistic tendencies, error patterns, and semantic representations compared to Qwen2.5-32B. Yet, the reward model still provides reliable signals for ranking candidate revisions, suggesting that it captures model-agnostic features of code quality—such as syntactic closeness to correct solutions, functional coherence, and local editability.

Such robustness to distributional shifts suggests broader applicability of our approach. It indicates that ReLoc, and particularly its reward model component, can serve as a plug-and-play module to improve inference-time performance across diverse LLMs, without the need for costly re-annotation or model-specific retraining. This is particularly valuable for deployment in scenarios involving closed-source or frequently updated models, where direct supervision signals or fine-tuning access are unavailable.

Table 5: Performance comparison on GPT-4o (`gpt-4o-2024-1120`).

Methods	**Rew.**	**Pass@1 (%)**	**Tokens (1K)**
Plan Search	self-evaluation	42.7	16.7
BoN	Pass Rate	41.8	12.2
ReLoc_HC **(Ours)**	Revision Reward Model	**44.2**	**8.7**

D Prompts

Table 6: Write diverse observations based on the problem to guide the generation of draft code.

Attribute	Description
DRAFTCODE	You are an expert Python programmer. You will be given a competitive programming question (problem specification). You will return 5 useful, non-obvious, and correct observations about the problem, like hints to solve the problem. You will NOT return any code. Be as creative as possible, going beyond what you think is intuitively correct. For each observation should be between tag [OBSERVATION] and [/OBSERVATION].
Outputs	**[OBSERVATION]** Observation 1: A permutation can be fixed with at most one swap if and only if it has at most 2 inversions. **[/OBSERVATION]** **[OBSERVATION]** Observation 2: Only 3-cycles (like "bca" and "cab") cannot be solved with a single swap operation. **[/OBSERVATION]** ...

Table 7: ReLoc_HC generates diverse modification strategies for incumbent code

Attribute	Description
HC_NEIGHBOR	You are an expert in debugging Python code. You will be provided with a code snippet that requires debugging, along with a revision history for reference. Your task is to begin by writing a brief textual explanation of the current code—summarize its intended behavior and any evident issues between [explanation] and [/explanation]. Next, propose three refinement directions that could help improve the code. Please put each direction between tag [direction] and [/direction]. Important: Do not include any Python code in your response—only the explanation and the three refinement directions.
Outputs	**[explanation]** The feedback indicates that the main issue is the function returns only the first element of min-k and max-k, instead of the entire lists. **[/explanation]** **[direction]** Modify the return statement to return the full min-k and max-k lists. **[/direction]** **[direction]** Include input validation to ensure K is a non-negative integer. **[/direction]** ...

Table 8: ReLoc_HC revises code according to different strategies

Attribute	Description
HC_NEIGHBOR	You are a helpful programming assistant and an expert in Python. The user has written code that contains errors. You will be provided with a Python programming problem, the user's code, revision history, an explanation, and directions for refinement. Your task is to debug and revise the code to correctly solve the problem. Generate a corrected version of the complete program, incorporating the explanation, refinement directions, and revision history. Output only one corrected program enclosed within a single pair of code delimiters. Do not include any additional commentary or text.

Outputs	`def minimumBoxes(apple_packs, box_capacities):`
	`  for box_capacity in box_capacities:`
	`  current_capacity += box_capacity`
	`  boxes_used += 1`
	`  if current_capacity >= total_apples:`
	`  return boxes_used`
	`  return boxes_used`
	
	`apple_packs = [1, 3, 2]`
	`box_capacities = [4, 3, 1, 5, 2]`
	`print(minimumBoxes(apple_packs, box_capacities))`
	
	`apple_packs = [5, 5, 5]`
	`box_capacities = [2, 4, 2, 7]`
	`print(minimumBoxes(apple_packs, box_capacities))`

Table 9: ReLoc_GA directly prompts LLM to generate neighborhood by referencing parent code

Attribute	Description
GA_NEIGHBOR	You are a helpful programming assistant and an expert in Python. The user has written code that contains errors. You will be provided with a Python programming problem and the user's code intended to solve it. Your task is to refer to the input code and revise it to correctly solve the problem. Generate a corrected version of the complete program. Output only one corrected program enclosed within a single pair of code delimiters. Do not include any additional commentary or text.

Outputs	`def minimumBoxes(apple_packs, box_capacities):`
	`  for box_capacity in box_capacities:`
	`  current_capacity += box_capacity`
	`  boxes_used += 1`
	`  if current_capacity >= total_apples:`
	`  return boxes_used`
	`  return boxes_used`
	
	`apple_packs = [1, 3, 2]`
	`box_capacities = [4, 3, 1, 5, 2]`
	`print(minimumBoxes(apple_packs, box_capacities))`
	
	`apple_packs = [5, 5, 5]`
	`box_capacities = [2, 4, 2, 7]`
	`print(minimumBoxes(apple_packs, box_capacities))`

E Step by Step Revisions

Below, we demonstrate how ReLoc guides `Qwen2.5-32B-Instruct` step by step to revise completely incorrect draft code into correct code.

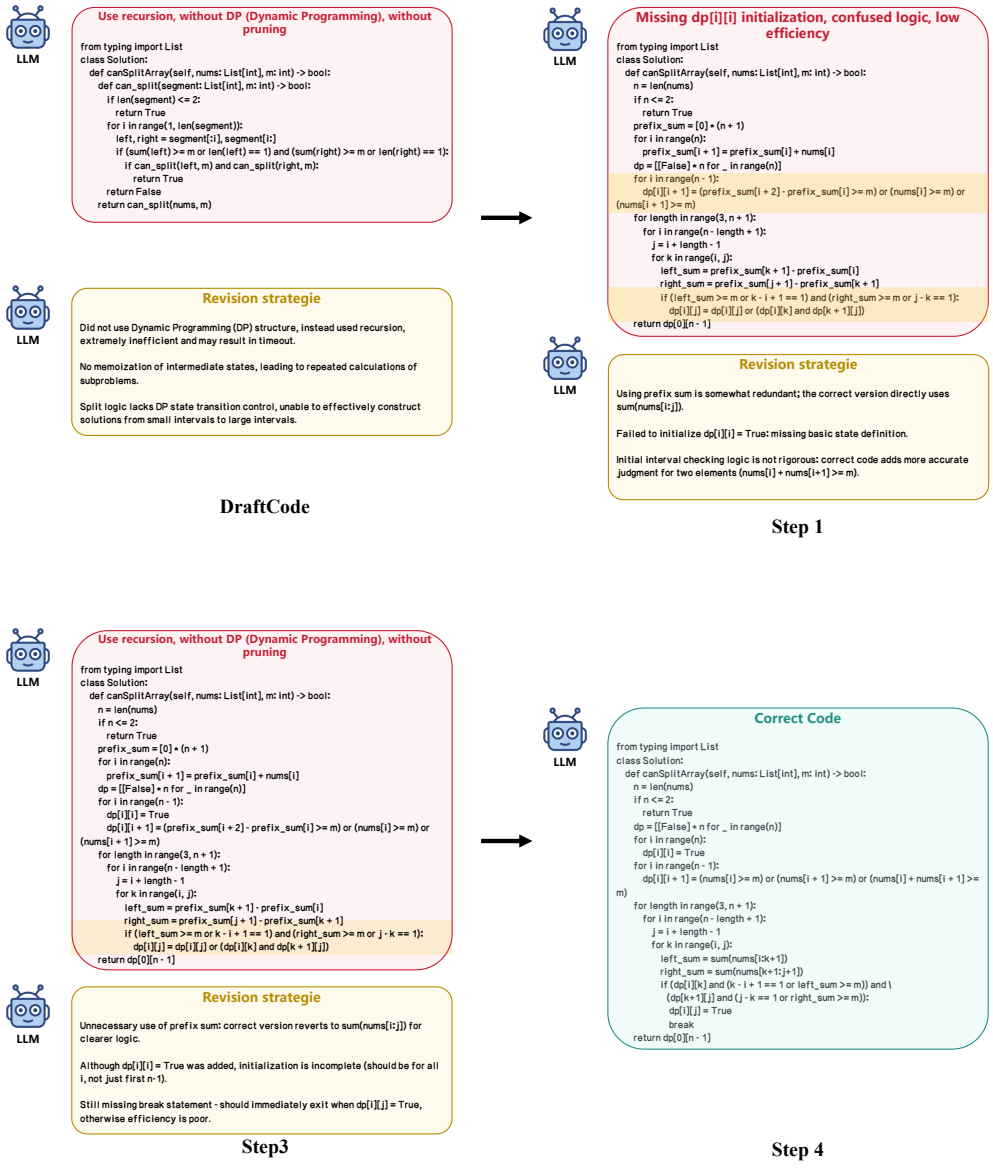


Figure 5: ReLoc step-by-step revise incorrect code