ReMCTS: Reflection-Enhanced Monte Carlo Tree Search for Code Generation

Wang Huifei[1], Huang Xinying[2], Sun Yiheng[1], Yifan Yuan[2*]
[1]College of Computer Science and Software Engineering, Shenzhen University
[2]School of Artificial Intelligence, Shenzhen University
2023040114@email.szu.edu.cn, doris.70huang@gmail.com
2023040450@email.szu.edu.cn, yifanyuan@szu.edu.cn

Abstract

Open-weight large language models (LLMs) can generate function-level programs from natural-language prompts, but plausible candidates still fail on hidden semantics and repeat mistakes across repair attempts. We present ReMCTS, an execution-grounded, memory-augmented, LLM-guided MCTS-style search framework. It organizes program candidates as tree states, retains branch-local debugging context, retrieves failure experience across branches, and distinguishes failed checks from unavailable evidence. On HumanEval and MBPP-Sanitized, visible-test ReMCTS improves over direct generation in 8 of 10 model–dataset pairs under held-out evaluation, whereas proxy-only search is less stable. Controlled tree-search, sampling, repair, and memory ablations characterize the source and limits of these gains. A 30-task HumanEval-X C++ pilot further demonstrates compatibility with compiler-backed execution, but does not constitute a broad multilingual evaluation.

1 Introduction

Code generation is not merely a one-pass text completion task, but rather a program search problem under partially observable feedback. Even when a model generates syntactically correct code, the program can still fail on boundary conditions, interface constraints, hidden inputs, or complex logic. These failures make feedback-driven revision a necessary part of the generation process. This view is close to how human programmers solve coding tasks, in which they write an initial implementation, run it, inspect errors, reflect on the cause, revise the code, and try again. Current LLM code generation often lacks this closed-loop workflow. If the first solution follows a wrong design, later local repair may only patch symptoms while the underlying algorithm remains wrong. Therefore, an effective inference-time system should explore multiple candidate programs, execute them, record failure experience, and reuse that experience across later attempts. In real-world environments, developers usually verify programs indirectly. Prior work models this practice with compiler feedback (Bi et al., 2024), execution-and-refinement feedback (Zheng et al., 2024), unit-test feedback (Liu et al., 2023), iterative test-and-repair feedback (Tang et al., 2026), static-analysis and test loops (Ravi et al., 2025), and interactive feedback for research-code tasks (Miao et al., 2025). These studies suggest that feedback is essential for code generation. Practical open-source code models benefit from feedback when refining candidates over multiple attempts.

Existing approaches improve code generation from several perspectives. Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and Self-Debugging (Chen et al., 2024) repair programs with natural language feedback or execution feedback. LATS (Zhou et al., 2024) and MapCoder (Islam et al., 2024) combine language-model reasoning with search, planning, or multi-agent collaboration. CodeRL (Le et al., 2022), RLTF (Liu et al., 2023), and CODERL+ (Jiang et al., 2026) use unit tests or execution semantics to build reward signals. These studies show that feedback, search, planning, and reflection can improve code generation. However, existing approaches still leave three gaps. 1) Single-trajectory self-repair can remain trapped in an early wrong algorithmic direction. This occurs because later revisions are conditioned on the same trajectory and often patch local symptoms rather than reconsider alternative algorithms. 2) Tree-search methods often rely on numerical rewards and underuse cross-branch natural-language error experience. Scalar rewards can rank candidates, but they do not preserve the textual diagnoses needed to help sibling branches avoid similar failures. 3) No-test tasks require distinguishing missing evidence from genuine success. Without explicit tests, semantic-only acceptance can be over-optimistic,

*Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 6227–6247
October 24-29, 2026 ©2026 Association for Computational Linguistics

allowing plausible but weakly checked programs to terminate the search too early.

To address these limitations, we propose ReMCTS, an execution-grounded, memory-augmented, LLM-guided MCTS-style framework for code generation. ReMCTS maintains multiple candidate programs in a search tree instead of repairing one candidate along a single trajectory. The multi-branch design helps the system recover from an early wrong algorithmic direction. Each tree node stores the candidate program, visit statistics, reward information, execution metadata, and path reflection memory. The node states let the search use both program behavior and prior repair experience. A memory-aware selection strategy combines numerical rewards, exploration terms, and semantic memory to allocate the search budget. Sandbox execution and multi-source evidence provide dense rewards that distinguish runnable, partially correct, and risky programs. Failure diagnoses are stored in both path memory and a global memory repository, so later branches can avoid rediscovering the same failure modes. The method inherits the tree policy, UCB selection, and backpropagation structure of MCTS, but uses LLM-generated expansions and no classical random rollout; we use "MCTS-style" in this precise sense.

Our contributions are summarized as follows:

- We formalize execution-grounded program search in which program candidates are tree states, and structured failure diagnoses become reusable path-level and cross-branch search artifacts. This interface reduces dependence on a single repair trajectory.
- We propose an evidence-driven reward and acceptance mechanism. The mechanism integrates execution, proxy, semantic, and risk evidence. The design supports both explicit-test and no-explicit-test settings while distinguishing missing evidence from negative evidence.
- We evaluate ReMCTS through controlled comparisons on multiple non-GPT models, separating search feedback from hidden evaluation. Paired tests, budget-up baselines, component ablations, and a limited C++ pilot characterize when the method helps and where it does not.

The source code and reproducibility materials are publicly available at https://github.com/AkieSUKI/ReMCTS.

2 Related Work

2.1 One-Shot Code Generation and Candidate Selection

Function-level code generation is often studied as prompt-to-code generation and evaluated with benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Many methods improve this setting by generating multiple candidates and then selecting among them. CodeT (Chen et al., 2023) generates tests and uses test agreement to select code candidates. LEVER (Ni et al., 2023) learns to verify language-to-code outputs with execution results. Coder-Reviewer (Zhang et al., 2023) reranks generated code with a reviewer model. These methods improve candidate selection, but they mainly operate after candidates have been generated. They do not maintain a shared search state where execution feedback and reflection from failed candidates guide later branches.

2.2 Feedback-Based Code Repair

Feedback-based methods repair programs after errors are observed. Self-Refine (Madaan et al., 2023) iteratively improves model outputs with self-feedback. Reflexion (Shinn et al., 2023) stores verbal feedback from failures, and Self-Debugging (Chen et al., 2024) uses execution feedback and explanation to repair programs. OpenCodeInterpreter (Zheng et al., 2024) builds execution-and-refinement data for code models. Other work turns tests or execution semantics into reward signals. RLTF (Liu et al., 2023) extracts rewards from unit-test feedback, and CODERL+ (Jiang et al., 2026) aligns code generation with execution semantics. These studies show that feedback is useful for code generation. However, many repair methods still follow a single trajectory, or use feedback mainly for training, filtering, or reranking. Olausson et al. (2024) further show that self-repair depends strongly on feedback quality and on the model's ability to use that feedback.

2.3 Search-Based and Tree-Based Code Generation

Search and planning methods explore more than one generation path. LATS (Zhou et al., 2024) combines language agents with Monte Carlo tree search. MapCoder (Islam et al., 2024) and AgentCoder (Huang et al., 2023) use multiple agents for retrieval, planning, coding, testing, or debugging. PlanSearch (Wang et al., 2025) searches

over natural-language plans before code generation. S* (Li et al., 2025a) studies test-time scaling for code generation. RethinkMCTS (Li et al., 2025b) applies Monte Carlo tree search to refine erroneous thoughts in code generation. These methods are closest to ReMCTS because they use search, planning, or test-time compute. ReMCTS instead uses generated programs as the primary search states; each node carries attempted-execution evidence, reward traces, and path reflections, including when a candidate does not compile or run. Diagnoses from one branch can be retrieved by later branches through a task-local global memory repository. Candidate evaluation is also availability-aware: missing tests are marked as unavailable rather than negative evidence. The distinguishing contribution is therefore the integration of program-state tree search, within-task reusable error memory, and availability-aware evidence, rather than tree search alone.

3 ReMCTS Methodology

3.1 Overview

Given a natural-language task description P and an optional visible test set $\mathcal{T}_{vis}$, code generation's goal is to use a base code model f_θ to produce a candidate program $S^\star$ that passes the final hidden tests $\mathcal{T}_{hid}$. During search, the system can only access P and the feedback allowed by the setting. We consider two search-feedback settings. In the `known` setting, the system can use $\mathcal{T}_{vis}$ as execution feedback. In the `no_tests` setting, no explicit tests are available. The system then relies on proxy evidence, such as contracts, static checks, automated probes, semantic reviews, and risk scans.

ReMCTS formulates the problem as test-time tree search. The root node corresponds to the original task. Edges correspond to generation or repair actions from the model. Child nodes correspond to new candidate programs. Unlike single-trajectory repair, ReMCTS keeps multiple candidate directions. It reallocates the search budget through reward backpropagation and reflection memory. Each search iteration follows a five-stage loop: selection, expansion, execution, reflection, and backpropagation. In the **selection** stage, the system starts from the root and selects a node for expansion with a memory-augmented Upper Confidence Bound score (UCB_M). Here UCB_M denotes a memory-aware version of the standard Upper Confidence Bound (UCB) rule. It balances exploitation and exploration while adding semantic memory. In the **expansion** stage, the code model generates b child candidates. The prompt includes the task, current program, execution feedback, path reflections, and global experience. In the **execution** stage, the sandbox runs each candidate and collects test, contract, static, probe, semantic, and risk evidence. In the **reflection** stage, the system generates a structured reflection when a candidate is not accepted. The reflection records the error type, root cause, and repair hint. In the **backpropagation** stage, the system propagates rewards along the search path and writes generated reflections into the global memory repository.

Figure 1 illustrates the leaf-expansion procedure. Selection, expansion, execution, reflection, and backpropagation form one closed loop. Execution feedback is converted into both numerical rewards and retrievable semantic experience. Reflection affects later node selection and expansion prompts; acceptance is determined by the evidence gate defined below.

3.2 Memory-Guided Monte Carlo Tree Design

Each node represents a candidate program. Each node also stores statistical value and semantic history. A node is defined as:

$$Node_i = [S_i, V_i, N_i, M^{(k)}_{path,i}, \Phi_i], \qquad (1)$$

where S_i denotes the current candidate program. V_i and N_i denote cumulative value and visit count. Φ_i denotes metadata, such as execution feedback, reward decomposition, risk information, and semantic reviews. The symbol v_i denotes the tree node whose state vector is $Node_i$. Let the path from the root to v_i be $\pi_i = (v_0, \ldots, v_i)$. Let M_j be the structured reflection produced at node v_j, when such a reflection exists. The k-hop path reflection memory is:

$$M^{(k)}_{path,i} = \mathrm{Tail}_k\big((M_j)_{v_j \in \pi_i,\ M_j \neq \varnothing}\big), \qquad (2)$$

where $\mathrm{Tail}_k(\cdot)$ denotes the operator that keeps the most recent k reflections. Here k denotes a fixed memory-window hyperparameter. $\varnothing$ denotes the case where no valid reflection is produced at a node. Each valid M_j denotes a structured record with error type, failure symptom, root cause, repair hint, and severity. Thus, a node records two kinds of information: the generated code and recent errors on the path with the corresponding repair hints.

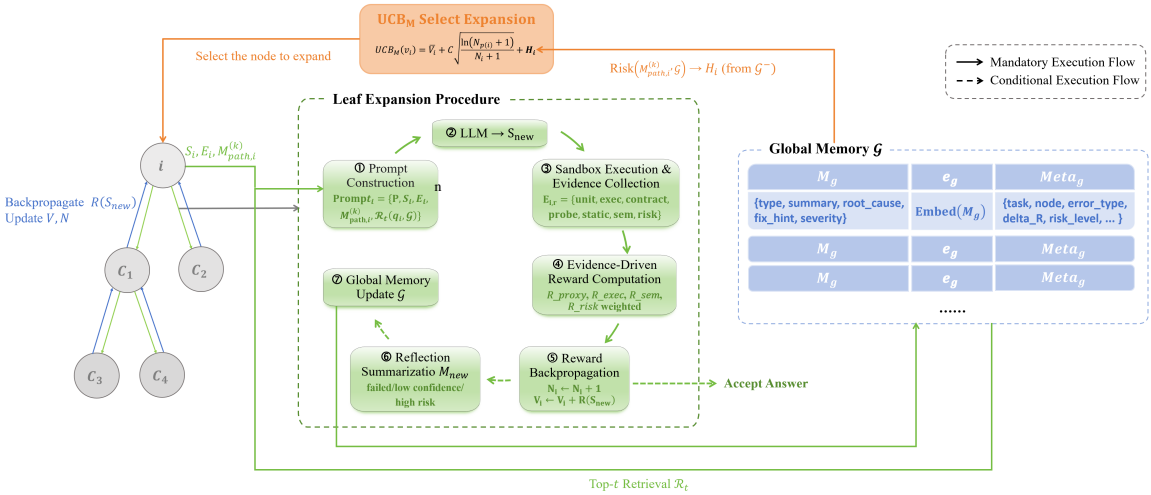

Figure 1: Overview of ReMCTS. The search proceeds through selection, expansion, execution, reflection, and backpropagation. Execution evidence determines rewards and acceptance, while path and global reflection memories guide later selection and expansion.

Standard MCTS uses UCB to balance high-value nodes and under-visited nodes. Error patterns in code generation are often semantically repetitive. Examples include entry-point mismatches, missing boundary cases, excessive complexity, and task misunderstanding. To exploit this structure, ReMCTS augments UCB with a **memory heuristic**:

$$UCB_M(v_i) = \bar{V}_i + C\sqrt{\frac{\ln(N_{p(i)}+1)}{N_i+1}} + H_i, \quad (3)$$

where $\bar{V}_i = V_i/(N_i+\epsilon)$ denotes the average node value. ϵ denotes a small constant for numerical stability. C denotes the exploration coefficient. $N_{p(i)}$ denotes the visit count of the parent node. Path reflections and global error memory jointly determine H_i. This paper instantiates the memory heuristic as:

$$H_i = \alpha Conf(M^{(k)}_{path,i}) - \beta Risk(M^{(k)}_{path,i}, \mathcal{G}). \quad (4)$$

For path records m, $Conf = \text{clip}_{[0,1]}[0.5(1-\overline{sev}) + 0.3\min(\overline{\max(\Delta R, 0)}, 1) + 0.2\overline{I_{hint}}]$: it rewards lower mean severity, positive reward changes, and the presence of repair hints; it is zero for an empty path memory. $Risk$ is the maximum cosine similarity to the top-t global records whose severity is at least 0.7, or zero if none exist. Thus, α and β control a repair-confidence bonus and a high-risk similarity penalty. In all main runs, $C = 1.4$, $\alpha = 0.4$, and $\beta = 0.6$; these are fixed design choices rather than theoretically optimal values.

To avoid repeated local edits around a wrong trajectory, ReMCTS uses two kinds of memory: path memory and global memory. The path memory $M^{(k)}_{path,i}$ stores the most recent k structured reflections on the current branch. Each reflection contains the error type, failure symptom, root cause, repair hint, and severity. During expansion, the model only reads the recent failure trajectory. This controls context length and keeps the repair direction coherent.

The task-local global memory repository $\mathcal{G}$ stores error experience across branches of the current task and is reinitialized for every benchmark item. It is represented as:

$$\mathcal{G} = \{(M_g, \mathbf{e}_g, Meta_g)\}_{g=1}^{|\mathcal{G}|}, \quad (5)$$

$$\mathbf{e}_g = Embed(M_g). \quad (6)$$

Here M_g denotes a reflection summary. $\mathbf{e}_g$ denotes its embedding vector. $Meta_g$ denotes metadata that stores the source task, node id, error type, reward change, and risk level. We instantiate $Embed(\cdot)$ as a deterministic 128-dimensional hashed token-count vector and rank memories by cosine similarity. Expansion retrieves the top $t = 2$ records without a hard similarity threshold. When a candidate is not accepted, the system writes its reflection summary, vector representation, and metadata into $\mathcal{G}$. The query uses the task, candidate code, and execution feedback. Retrieved memory enters the expansion prompt and semantic heuristic, but never deletes a branch; later execution evidence can override a misleading retrieval through numerical returns. This is a soft search signal, and the paired harm/benefit analysis in Section 4.5 shows

that it is useful in some harder settings rather than uniformly beneficial.

3.3 Evidence-Driven Reward and Acceptance Mechanism

This section defines an evidence-driven reward for code generation. ReMCTS evaluates each candidate with executable, interface, behavioral, semantic, and risk evidence, instead of relying on a single LLM score or a binary test result. Figure 2 summarizes how these signals are collected, aggregated into rewards, and used for final acceptance.

After a candidate program is generated, the sandbox executes it and collects weighted evidence:

$$\mathcal{E}_i = \{E_{unit}, E_{exec}, E_{contract}, E_{probe}, E_{meta}, E_{static}, E_{sem}, E_{risk}\}. \quad (7)$$

Here $\mathcal{E}_i$ denotes the evidence set collected for candidate program S_i. For each evidence item $E_k = (A_k, C_k, R_k)$, A_k marks availability, C_k is confidence, and R_k is a normalized score. Unit tests provide the strongest signal when available; execution, contract, and static checks verify runnability and interface consistency; probe and meta evidence provide lightweight behavioral constraints in `no_tests`; semantic review and risk checks support final filtering. Unavailable evidence is excluded from reward aggregation, so missing tests are not treated as failed tests. Contract evidence comes from AST rules for the required entry point and simple signature properties. In `no_tests`, four LLM-generated weak assertions are derived from the task text alone, sanitized to prohibit imports, randomness, and file, network, or system access, and then executed as probes. Appendix Table 5 gives the full evidence list and Section D gives implementation details.

In `no_tests`, the final node reward is:

$$R(S_i) = \frac{\sum_{g\in\mathcal{J}} \delta_g \omega_g A_g C_g R_g(S_i)}{\sum_{g\in\mathcal{J}} \omega_g A_g C_g + \epsilon}, \quad (8)$$

$$\mathcal{J} = \{p, e, s, r\}, \quad (9)$$

$$\delta_g = \begin{cases} 1, & g \in \{p, e, s\}, \\ -1, & g = r. \end{cases} \quad (10)$$

where p, e, s, r index proxy correctness, execution reachability, semantic review, and risk, respectively. A_g and C_g are group availability and confidence, ω_g is the group weight, and δ_g assigns a negative sign only to risk. The `no_tests` weights are $(\omega_p, \omega_e, \omega_s, \omega_r) = (0.4, 0.2, 0.3, 0.1)$. In `known`, the implementation instead uses $R_{known} = 0.6R^{obs}_{unit} + 0.2R_{exec} + 0.2R_{sem}$ and accepts a candidate only after all available visible tests pass, subject to the search-level stopping rules. For risk-aware no-test acceptance, the system further computes:

$$\mathrm{Score}_{acc}(S_i) = R(S_i) - \gamma_1 U_i - \gamma_2 R_{risk}(S_i) - \gamma_3 D_i, \quad (11)$$

Here U_i penalizes low-confidence evidence, R_{risk} penalizes unsafe behavior, and D_i is a code-length deviation penalty. If $\mathcal{Q}_i$ contains the available contract, static, meta, and probe items plus semantic and risk evidence, then $U_i = 1 - |\mathcal{Q}_i|^{-1} \sum_{q\in\mathcal{Q}_i} C_q$. The implementation sets D_i to 0, 0.15, or 0.35 for at most 80, 81–200, or more than 200 nonblank lines. In `no_tests`, acceptance requires Score_{acc}, proxy, execution, and semantic thresholds to be met and risk to stay below its threshold. The gate is evaluated for every generated sibling; with complete-branch-before-accept enabled, the best accepted sibling is returned only after the branch is complete and the minimum iteration is reached.

Reward Component Details. We next define the item-level rewards used to compute proxy correctness, execution reliability, semantic review, and risk-aware acceptance.

The observed reward from explicit unit tests is:

$$R^{obs}_{unit}(S_i) = \frac{1}{m}\sum_{j=1}^{m} p_j(S_i), \quad m > 0, \quad (12)$$

where m is the number of visible unit tests and $p_j(S_i)$ indicates whether S_i passes the j-th test. If $m = 0$, the method sets $A_{unit} = 0$, so missing unit-test evidence is ignored rather than counted as failure.

The integrated proxy-correctness reward is:

$$\mathcal{K}_{proxy} = \{\mathrm{contract}, \mathrm{probe}, \mathrm{meta}, \mathrm{static}\}, \quad (13)$$

$$R_{proxy}(S_i) = \frac{\sum_{k\in\mathcal{K}_{proxy}} A_k C_k \eta_k R_k(S_i)}{\sum_{k\in\mathcal{K}_{proxy}} A_k C_k \eta_k + \epsilon}, \quad (14)$$

where the fixed item weights for contract, probe, meta, and static evidence are $(0.35, 0.25, 0.15, 0.25)$. Explicit unit tests are handled by R^{obs}_{unit} in `known` and remain unavailable in `no_tests`.

Execution reward uses four observable indicators:

$$R_{exec}(S_i) = 0.3(I_c + I_r) + 0.2(I_t + I_o), \quad (15)$$

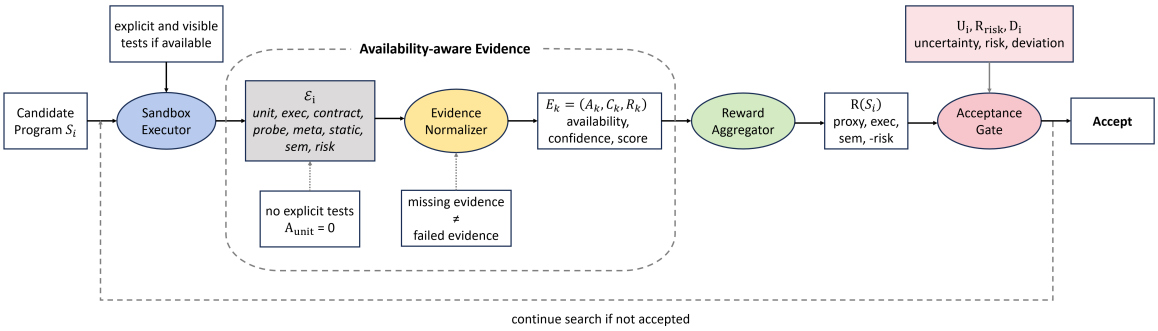

Figure 2: Evidence-driven reward and acceptance mechanism. A candidate program is first evaluated in the sandbox and converted into availability-aware evidence. When explicit tests are absent, unit-test evidence is marked unavailable ($A_{unit} = 0$) rather than failed. The reward aggregator combines proxy correctness, execution reachability, semantic review, and risk. The acceptance gate further penalizes uncertainty, risk, and code-length deviation; rejected candidates return to search.

where I_c, I_r, I_t, I_o mark successful compilation, execution, the presence of executable tests, and nonempty output or successful execution, respectively. In `no_tests`, let Q_{sem} combine the reviewer's requirement coverage, API match, context compatibility, and maintainability scores with weights $(0.35, 0.25, 0.20, 0.20)$. Given reviewer confidence C_{sem} and semantic risk q_{srisk}, the implemented score is

$$R_{sem} = C_{sem} Q_{sem}(1 - q_{srisk}). \tag{16}$$

The separate static risk scan sets $R_{risk} = \text{clip}(0.2 n_{danger} + 0.15 I_{>200}, 0, 1)$, where n_{danger} counts prohibited imports or calls; a syntax error receives risk 0.25 because a full scan is impossible. Semantic review is therefore confidence- and risk-weighted rather than treated as a correctness oracle.

4 Results

4.1 Experimental Setup

Datasets. The main study uses HumanEval (Chen et al., 2021) and MBPP-Sanitized (Austin et al., 2021), which contain 164 and 427 function-level Python tasks. The protocol separates search feedback from final hidden evaluation. Visible feedback is derived from HumanEval prompt doctests and the first assertion of each MBPP-Sanitized task. We additionally run a limited HumanEval-X C++ pilot (Zheng et al., 2023) on the first 30 tasks in benchmark order, without task selection. File names, hashes, derivation details, and the pilot protocol are reported in Appendix C.

Models and Methods. This paper mainly focuses on open-weight non-GPT code models. We evaluate four dense Qwen3 models, Qwen3-1.7B, Qwen3-8B, Qwen3-14B, and Qwen3-32B (Qwen Team, 2025)[1], together with DeepSeek-V4-Flash (DeepSeek-AI, 2026) [2]. The main methods are Direct `no_tests`, ReMCTS `no_tests`, and ReMCTS `known`. Supplementary baselines include Best-of-8, Reflexion-style, pure-MCTS, and MCTS-style, which test repeated sampling, single-trajectory reflection, tree search, and LLM semantic review. Budget-up controls use Best-of-32 and Reflexion-16 on representative MBPP `no_tests` settings. We also report two component ablations that remove global memory retrieval or path reflection memory. Appendix E describes model invocation, decoding settings, and budget conventions.

Search Parameters and Metrics. All main ReMCTS experiments use a fixed search budget: 8 iterations, branch factor 5, maximum depth 8, minimum acceptance iteration 3, and complete-branch-before-accept enabled. Experiments use two search-feedback configurations: `known` uses visible tests during search, while `no_tests` removes explicit tests and relies on proxy evidence. Direct uses one generation, whereas ReMCTS and tree-search baselines use extra test-time compute. We report hidden solve rate as the main metric, exact two-sided McNemar tests for paired outcomes, Wilson 95% intervals for solve rates, and token, call, wall-clock, and estimated monetary costs

[1] Qwen3 model collection: https://huggingface.co/collections/Qwen/Qwen3

[2] DeepSeek-V4-Flash model: https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash

where logs are comparable. Appendix E gives the definitions and pricing assumptions.

Controlled-comparison scope. Our goal is to isolate the inference-time search mechanism, not to claim a leaderboard comparison against systems with different base models, fine-tuning data, generated or private tests, tools, or evaluation harnesses. The controlled baselines therefore preserve the base model, hidden-test evaluator, feedback access, and, for MCTS-style, the tree-search budget while replacing the search strategy or disabling ReMCTS components.

4.2 Main Results

The main result is that ReMCTS is substantially more effective when reliable feedback is available. Because the headline comparison changes both feedback access and test-time compute, we treat it as mechanism validation rather than attributing the entire gap to search or memory. Table 1 reports hidden solve rates on HumanEval and MBPP-Sanitized. The `known` setting is a test-guided mechanism validation: visible feedback can be used during search, while final scoring still uses only hidden tests. Therefore, this setting is not equivalent to standard pass@1. Under this protocol, ReMCTS `known` improves over Direct `no_tests` in 8 out of 10 model–dataset pairs, ties in 1 pair, and slightly decreases in 1 pair. It also improves over ReMCTS `no_tests` in all pairs, showing that reliable feedback is more stable than proxy-only evidence.

Paired statistical evidence. Pooling the 2,955 model–task pairs, Direct `no_tests` solves 2,050 (69.37%) and ReMCTS `known` solves 2,292 (77.56%, Wilson 95% CI 76.02–79.03). There are 364 ReMCTS-only and 122 Direct-only successes (two-sided exact McNemar $p = 5.3 \times 10^{-29}$). Eight individual settings improve significantly at $p < 0.05$; Qwen3-14B HumanEval ties and Qwen3-14B MBPP decreases non-significantly. Appendix Table 14 reports all paired counts, confidence intervals, and p-values. The pooled test treats each model–task output as a paired observation; the individual tests avoid interpreting the pooled result as evidence about variation across models.

Model Capacity and Proxy Evidence. Two secondary patterns qualify the main gain: improvements depend on model capacity, and proxy-only evidence can still miss hidden semantics. The gaps in Table 1, search-tree statistics in Appendix K, and trajectories in Appendix Table 20 show that weaker models often benefit from execution evidence. Stronger models such as Qwen3-32B mainly recover residual errors. The Qwen3-14B drops and some MBPP `no_tests` cases expose the proxy-evidence boundary: plausible candidates can pass weak checks while still failing hidden semantics.

4.3 Cross-Language Pilot

To test executor portability, we ran a HumanEval-X C++ pilot on the first 30 tasks in published order, without filtering. Search used only the public `example_test`; the official `test` field was held out. With a C++17 executor and a smaller 3-iteration, branch-3 budget, ReMCTS improves hidden solve rate from 73.33 to 90.00 on Qwen3-8B, 83.33 to 96.67 on Qwen3-32B, and 60.00 to 96.67 on DeepSeek-V4-Flash. Pooling the 90 paired outputs gives 21 ReMCTS-only and 1 Direct-only success ($p = 1.10 \times 10^{-5}$). Appendix Table 7 reports the protocol, tokens, and costs. This limited pilot establishes compiler-backed compatibility, not broad multilingual capability.

4.4 Budget-Up and Cost Analysis

To test whether the main gains are explained by generating more candidates, we increased the budgets of the two strongest simple controls on representative MBPP `no_tests` runs. Best-of-32 generates 32 independent candidates, and Reflexion-16 permits up to 16 actor–evaluator–reflection rounds. These runs are not strict wall-clock matches; they test whether increasing sampling or repair budget closes the accuracy gap.

Best-of-32 remains 6.79 and 6.32 percentage points below ReMCTS for Qwen3-8B and Qwen3-32B, respectively. Reflexion-16 remains 4.44 and 1.40 points below; notably, Qwen3-8B Reflexion-16 uses more logged tokens than ReMCTS (15.8k vs. 10.3k) but is less accurate. ReMCTS therefore trades additional sequential search overhead for robustness and should not be interpreted as the cheapest or fastest method. The cost conclusion is setting-dependent: Reflexion-style is more cost-effective on known-feedback MBPP, where visible assertions are strong, whereas ReMCTS is more robust to incomplete feedback and outperforms the controlled memory-free tree-search baseline in the comparisons below.

Table 1: Hidden solve rates on HumanEval and MBPP-Sanitized. Parentheses in the two ReMCTS columns report percentage-point differences from Direct `no_tests`. TG–`no_tests` is the difference between ReMCTS `known` and ReMCTS `no_tests`. `known` results are used for mechanism validation and are not equivalent to standard pass@1.

Model	Dataset	Direct `no_tests`	ReMCTS `no_tests`	ReMCTS `known`	TG–`no_tests` (pp)
`Qwen3-1.7B`	HumanEval	60.37%	69.51% (+9.14)	83.54% (+23.17)	+14.03
`Qwen3-1.7B`	MBPP	53.40%	51.29% (-2.11)	63.47% (+10.07)	+12.18
`Qwen3-8B`	HumanEval	79.88%	86.59% (+6.71)	93.90% (+14.02)	+7.31
`Qwen3-8B`	MBPP	64.17%	66.74% (+2.57)	74.24% (+10.07)	+7.50
`Qwen3-14B`	HumanEval	82.93%	78.66% (-4.27)	82.93% (+0.00)	+4.27
`Qwen3-14B`	MBPP	66.98%	60.42% (-6.56)	65.57% (-1.41)	+5.15
`Qwen3-32B`	HumanEval	89.63%	89.63% (+0.00)	95.12% (+5.49)	+5.49
`Qwen3-32B`	MBPP	71.90%	70.49% (-1.41)	77.99% (+6.09)	+7.50
`DeepSeek-V4-Flash`	HumanEval	80.49%	95.12% (+14.63)	98.78% (+18.29)	+3.66
`DeepSeek-V4-Flash`	MBPP	72.60%	72.83% (+0.23)	81.03% (+8.43)	+8.20

Table 2: Budget-up baselines on MBPP `no_tests`. Wall-clock is total elapsed time over 427 tasks; tokens and calls are per task. Hidden tests are used only for final evaluation.

Model	Method	Hidden solve	Tok/task	Calls/task	Wall (s)	Cost (CNY)
`Qwen3-8B`	ReMCTS	66.74	10,295	19.7	1,980	3.72
`Qwen3-8B`	Best-of-32	59.95	8,290	44.7	1,502	3.18
`Qwen3-8B`	Reflexion-16	62.30	15,777	17.1	903	4.54
`Qwen3-32B`	ReMCTS	70.49	9,735	19.7	3,053	13.15
`Qwen3-32B`	Best-of-32	64.17	8,600	33.0	2,232	13.79
`Qwen3-32B`	Reflexion-16	69.09	11,072	12.4	1,147	12.94

Supplementary Baseline Setup. The supplementary baselines test whether repeated sampling, single-path reflection, or tree search alone explains the gains. We compare Best-of-8, Reflexion-style, pure-MCTS, and MCTS-style on the same three models, without hidden tests for search or selection. MCTS-style is the closest memory-free tree-search baseline: it adds LLM semantic review but disables path memory, global memory, semantic UCB, and LLM probes. Full results, acceptance rates, Oracle Pool upper bounds, and configuration details are in Appendix Tables 11 and 12.

Sampling and Reflection Baselines. In matched no-test runs, Best-of-8 trails ReMCTS on both Qwen MBPP settings, while Reflexion-style trails in five of six pairs (except Qwen3-32B HumanEval). Best-of-8 also trails ReMCTS `known` throughout, but that cross-feedback comparison is descriptive. The offline Oracle Pool sometimes contains a correct candidate that the baseline fails to select.

Tree-Search Baselines. Tree search alone is also insufficient: in `no_tests`, pure-MCTS and MCTS-style remain below full ReMCTS on all three MBPP settings. In `known`, full ReMCTS beats MCTS-style in all six pairs by 5.07 points on average. The strongest paired gains are Qwen3-8B HumanEval (13 vs. 1, $p = 0.00183$), Qwen3-8B MBPP (33 vs. 1, $p = 4.07 \times 10^{-9}$), Qwen3-32B MBPP (30 vs. 1, $p = 2.98 \times 10^{-8}$), and DeepSeek MBPP (27 vs. 5, $p = 1.13 \times 10^{-4}$); larger-model HumanEval comparisons are near ceiling. Appendix Tables 15 and 13 give all paired and token results.

The `known`-feedback Reflexion-style results show that visible tests can make single-path repair highly competitive on short assertion-style tasks. Full ReMCTS remains higher on HumanEval for all three models, but Reflexion-style is stronger on MBPP, reaching 81.97%, 86.89%, and 94.15% on Qwen3-8B, Qwen3-32B, and DeepSeek-V4-Flash. Thus, ReMCTS's advantage is not simply that reflection helps; multi-branch search and reusable memory mainly improve robustness when feedback is incomplete, noisy, or hard to exploit through one trajectory.

4.5 Ablation Results

Memory Ablations. Table 3 reports the core component results. Removing global retrieval lowers hidden solve rate by 0.97 points on average over twelve settings, with the clearest paired effects on Qwen3-8B MBPP-`known` (23 full-only vs. 3 no-global-only, $p = 8.8 \times 10^{-5}$) and Qwen3-32B MBPP-`known` (17 vs. 3, $p = 0.00258$). The effect is noisy on DeepSeek MBPP-`known` (12 vs. 10, $p = 0.83$), while path memory has a mixed 5-win/4-tie/3-loss pattern. Appendix Table 16 adds acceptance and tree statistics.

Table 3: Memory ablations under final hidden evaluation (%). Full denotes complete ReMCTS; the other columns disable one memory component.

Model	Dataset	Setting	Full	W/O global	W/O path
DeepSeek-V4-Flash	HumanEval	known	98.78	99.39	99.39
DeepSeek-V4-Flash	HumanEval	no_tests	95.12	94.51	94.51
DeepSeek-V4-Flash	MBPP	known	81.03	80.56	80.09
DeepSeek-V4-Flash	MBPP	no_tests	72.83	73.07	72.83
Qwen3-8B	HumanEval	known	93.90	92.07	95.12
Qwen3-8B	HumanEval	no_tests	86.59	85.37	86.59
Qwen3-8B	MBPP	known	74.24	69.56	73.54
Qwen3-8B	MBPP	no_tests	66.74	66.04	66.51
Qwen3-32B	HumanEval	known	95.12	95.12	95.12
Qwen3-32B	HumanEval	no_tests	89.63	90.85	89.63
Qwen3-32B	MBPP	known	77.99	74.71	78.69
Qwen3-32B	MBPP	no_tests	70.49	69.56	70.02

Search-Tree Behavior. Search-tree statistics indicate that ReMCTS usually performs shallow multi-branch exploration rather than long repair chains. Appendix Table 17 gives the full per-model values. The average tree size ranges from 3.389 to 11.593 nodes, the median mostly falls between 3 and 8 nodes, and the average maximum depth is about 1.23–1.61. A smaller actual tree does not mean the parameters are inconsistent: `branch_factor=5` is only an upper bound, and parsing failures, duplicate code, cross-node deduplication, and early acceptance all reduce the number of effective children.

Minimum Acceptance Iteration. The minimum-acceptance ablation further shows that no-test deployment needs stronger uncertainty control. Increasing `min_accept_iterations` from 1 to 3 improves hidden solve rate in 7 of 10 `no_tests` combinations, with an average gain of 1.55 points; full scores and search-scale changes are reported in Appendix Tables 18 and 19.

Qualitative Error Analysis. Representative trajectories (Appendix Table 20) make the aggregate patterns concrete. In known-feedback HumanEval, Qwen3-1.7B recovers an `intersperse` failure by exploring siblings after early candidates fail visible tests; the selected branch changes delimiter placement instead of repeatedly patching the same implementation. For Qwen3-8B `remove_duplicates`, path memory records a semantic confusion and a missing import before a later branch passes hidden tests. For Qwen3-32B `starts_one_ends`, the search repairs an inclusion–exclusion error and the $n = 1$ boundary case. These examples suggest that weaker models benefit from execution-guided filtering, whereas stronger models mainly use search to recover residual edge cases.

In `no_tests`, the failure case is different. A Qwen3-14B MBPP candidate is accepted after contract and static checks pass, despite a partially failing weak probe and over-optimistic semantic review; hidden tests later reject it. This illustrates why proxy-only gains are less stable and why delaying acceptance improves seven of ten combinations. More generally, automatic acceptance is a risk-controlled stopping rule, not a correctness oracle: hidden outcomes are never exposed during search, and offline hidden evaluation is used only to audit the returned candidate. The paired statistics therefore measure end-to-end success, while trajectories explain plausible mechanisms rather than establish causal mediation.

5 Conclusions

This paper presents ReMCTS, an execution-grounded, memory-augmented, LLM-guided MCTS-style framework for inference-time code search. On HumanEval and MBPP-Sanitized, visible feedback gives the strongest hidden-test gains; `no_tests` is positive on average but unstable. Paired tests and budget-up controls show that the gains are not explained by more samples alone. Global memory contributes more consistently than path memory. A 30-task C++ pilot suggests compiler-backed transfer, not broad multilingual or repository-level capability. Future work should strengthen proxy feedback and uncertainty calibration.

Limitations

Evaluation is limited to function-level benchmarks: known is test-guided mechanism validation rather than standard pass@1, while the 30-task C++ pilot cannot establish broad multilingual or repository-level capability. In both settings, hidden tests are used only for offline scoring and never guide search. Proxy signals can miss hidden boundary cases, the executor is not security-hardened, and sequential calls add cost. We do not evaluate workspace patches, cross-file dependencies, environment configuration, test selection, or long-horizon debugging, nor isolate every heuristic.

Cost comparisons should also be interpreted cautiously. API latency, batching, server load, and model-version changes can affect wall-clock time and token accounting. We therefore report tokens, calls, and elapsed time together as within-study diagnostics, rather than claiming deployment-wide efficiency.

Acknowledgments

This work is supported by the Intelligent Computing Center of Shenzhen University. During the preparation of this manuscript and its accompanying code, the authors used OpenAI ChatGPT/Codex for language editing, LaTeX organization, code and documentation assistance, and reference cross-checking. All AI-generated suggestions were reviewed and verified by the authors, who take full responsibility for the final content.

References

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

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2336–2353, Bangkok, Thailand. Association for Computational Linguistics.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*.

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

DeepSeek-AI. 2026. DeepSeek-V4: Towards highly efficient million-token context intelligence.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

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

Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Zhi Jin, Wenpin Jiao, and Ge Li. 2026. CodeRL+: Improving code generation via reinforcement with execution semantics alignment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3608–3622, San Diego, California, United States. Association for Computational Linguistics.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. 2025a. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 15964–15978, Suzhou, China. Association for Computational Linguistics.

Qingyao Li, Wei Xia, Xinyi Dai, Kounianhua Du, Weiwen Liu, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025b. RethinkMCTS: Refining erroneous thoughts in monte carlo tree search for code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8092–8110, Suzhou, China. Association for Computational Linguistics.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36.

Chunyu Miao, Henry Peng Zou, Yangning Li, Yankai Chen, Yibo Wang, Fangxin Wang, Yifan Li, Wooseong Yang, Bowei He, Xinni Zhang, Dianzhi Yu, Hanchen Yang, Hoang H. Nguyen, Yue Zhou, Jie Yang, Jizhou Guo, Wenzhe Fan, Chin-Yuan Yeh, Panpan Meng, and 12 others. 2025. RECODE-H: A benchmark for research code development with interactive human feedback. *Preprint*, arXiv:2510.06186.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. 2023. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. 2025. LLM-LOOP: Improving llm-generated code and tests through automated iterative feedback loops. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 930–934. IEEE.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Lingxiao Tang, Muyang Ye, Zhaoyang Chu, Xiaoxue Ren, Zhongxin Liu, Lingfeng Bao, and He Ye. 2026. An iterative test-and-repair framework for competitive code generation. *Preprint*, arXiv:2604.05560.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2025. Planning in natural language improves llm search for code generation. In *The Thirteenth International Conference on Learning Representations*.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684. Association for Computing Machinery.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12834–12859, Bangkok, Thailand. Association for Computational Linguistics.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62138–62160. PMLR.

A ReMCTS Pseudocode

Algorithm 1 summarizes the full search procedure. In `known`, EVIDENCEGATE requires all visible tests to pass; in `no_tests`, unit tests are unavailable and the gate applies the proxy, execution, semantic, risk, and acceptance-score thresholds. The algorithm evaluates the complete generated branch before returning its highest-reward accepted sibling, and does not stop before the minimum iteration.

Algorithm 1 ReMCTS search procedure

Require: Task P, optional visible tests $\mathcal{T}_{vis}$, model f_θ
Require: Budget B, branch factor b, max depth d, memory window k, minimum iteration i_{min}
Ensure: Final candidate program $S^\star$

```
Initialize root node v_0 and global memory G ← ∅
S*, S_acc ← ∅; r*, r_acc ← −∞
for iter = 1 to B do
    v ← SELECT(v_0, UCB_M, d)
    M_path ← PATHMEMORY(v, k)
    M_global ← RETRIEVE(G, P, S_v, Φ_v)
    X ← BUILDPROMPT(P, S_v, Φ_v, M_path, M_global)
    C ← GENERATE(f_θ, X, b)
    for all S_i ∈ C do
        E_i ← COLLECTEVIDENCE(P, S_i, T_vis)
        if T_vis = ∅ then
            Set A_unit ← 0
        end if
        R_i ← EVIDENCEREWARD(E_i)
        g_i ← EVIDENCEGATE(S_i, E_i, R_i)
        M_i ← ∅
        if not g_i then
            M_i ← REFLECT(P, S_i, E_i, M_path, M_global)
        end if
        u ← ADDCHILD(v, S_i, R_i, E_i, M_i)
        BACKPROPAGATE(u, R_i)
        if STOREABLE(M_i, E_i) then
            G ← G ∪ {(M_i, EMBED(M_i), Meta_i)}
        end if
        if R_i > r* then
            S* ← S_i, r* ← R_i
        end if
        if g_i and R_i > r_acc then
            S_acc ← S_i, r_acc ← R_i
        end if
    end for
    if S_acc ≠ ∅ and iter ≥ i_min then
        return S_acc
    end if
end for
return S*
```

B Reward and Evidence Details

Symbols in the method formulas fall into three categories. The first category is predefined hyperparameters, including the search budget, C, memory window k, global retrieval count t, semantic heuristic weights α, β, evidence and reward weights η_k, ω_g, acceptance penalties γ_j, and thresholds τ_*. The second category is runtime evidence quantities, including A_k, C_k, R_k and reward components derived from them. The third category is rule- or LLM-assisted scores, including H_i, semantic reviews, risk scans, and the deviation penalty D_i. Hyperparameters are fixed before evaluation and are not learned from tasks. No model or reward parameters are fitted; search-time state updates consist of tree structure and node statistics, current-best state, evidence metadata, and task-local reflection memory.

Table 4 summarizes these symbol categories and fixed hyperparameter settings. Table 5 lists the evidence items used to evaluate candidate programs. If an evidence item is unavailable, $A_k = 0$, and the missing item is ignored by the reward denominator.

C Datasets and Input Settings

Table 6 lists the data files, task counts, and source summaries. Complete SHA-256 hashes, byte counts, and derivation relations are recorded in `data/dataset_manifest.json`.

The experiments use existing public function-level code-generation benchmarks, including HumanEval, MBPP-Sanitized, and HumanEval-X, and do not collect human or user-generated data. We checked the released task files for obvious direct identifiers and offensive-content concerns. No personal identifiers are used in our analyses, and the benchmark materials are retained and redistributed only in accordance with their original source terms. No additional anonymization was required.

Cross-Language Pilot. Table 7 gives the full results for the first 30 HumanEval-X C++ tasks in published order. Search uses only `example_test`; the official `test` field is held out for final evaluation. The C++17 runs use 3 iterations, branch factor 3, maximum depth 4, minimum acceptance iteration 2, and an 8-second execution timeout.

D Implementation and Reproducibility

The complete release accompanying the camera-ready paper includes the search and evaluation framework, prompt templates, configuration files, dataset preprocessing and split scripts, the Python and C++ compiler-backed executors, hidden-evaluation scripts, anonymized per-task outputs, reflection/retrieval logs, and token/cost logs. API-dependent runs record the exact model alias, run date, decoding parameters, data hashes, and saved anonymized outputs so that hidden scores can be recomputed without re-querying a mutable hosted

Table 4: Symbol categories and hyperparameter settings.

Symbol	Type	Definition
C, k, t, α, β	Search and semantic-heuristic hyperparameters	$C = 1.4$, $k = 3$, $t = 2$; ReMCTS uses $\alpha = 0.4, \beta = 0.6$, while pure-MCTS/MCTS-style ablations set the memory heuristic to zero.
$\omega_g, \gamma_j, \tau_*$	Reward and acceptance hyperparameters	`known` uses unit/exec/semantic weights $(0.6, 0.2, 0.2)$; `no_tests` uses proxy/exec/semantic/risk weights $(0.4, 0.2, 0.3, 0.1)$. The `no_tests` acceptance penalties are $\gamma = (0.10, 0.35, 0.10)$, and its thresholds are $\tau_{proxy} = 0.50$, $\tau_{exec} = 0.60$, $\tau_{sem} = 0.45$, $\tau_{risk} = 0.25$, and $\tau_{accept} = 0.62$.
η_k	Proxy-evidence weights	Contract/probe/meta/static weights are $(0.35, 0.25, 0.15, 0.25)$; fixed before evaluation.
$A_k, C_k, R_k, I., R., U_i$	Runtime evidence quantities	Produced by visible tests, sandbox execution, contract checks, probes, property checks, static checks, risk scans, and LLM reviews; U_i is computed from the confidence of available evidence groups.
H_i, D_i, R_{sem}	Rule- or LLM-assisted scores	H_i combines path-reflection confidence and high-severity global-memory similarity; D_i depends on nonblank code length; R_{sem} is a confidence- and risk-weighted LLM review.
ϵ	Numerical stability term	Used only to avoid division by zero; not a tunable experimental parameter.

Table 5: Evidence items used in reward computation.

Evidence	Meaning	Role
Unit	Visible unit tests or assertions	Provides the highest-confidence functional feedback in `known`
Exec	Compilation, loading, entry call, timeout, and crash status	Checks whether the candidate is runnable and callable
Contract	Function name, argument count, return form, or schema	Prevents interface mismatches and entry-point errors
Probe	Four LLM-generated weak assertions from task text; fixed confidence 0.45	Provides low-cost behavioral signals without explicit tests after sanitizer checks
Meta	Properties or metamorphic relations extracted from the requirement; fixed confidence 0.35	Checks task properties such as sorting, length, idempotence, and reversal
Static	Compilation/execution status; fixed confidence 0.80	Filters obviously non-runnable code
Semantic	LLM semantic review	Estimates consistency between the candidate approach and task description
Risk	Dangerous system calls, unauthorized file access, abnormal complexity, etc.	Provides risk penalties for production-style acceptance

service. Secrets and provider credentials are excluded from the release.

The main Python runs use a 5-second subprocess timeout and the search settings in Table 9. The `no_tests` gate uses $w_{proxy}/w_{exec}/w_{semantic}/w_{risk} = 0.4/0.2/0.3/0.1$, thresholds $(\tau_{accept}, \tau_{proxy}, \tau_{exec}, \tau_{sem}, \tau_{risk}) = (0.62, 0.50, 0.60, 0.45, 0.25)$, and penalties $(\gamma_{uncertainty}, \gamma_{risk}, \gamma_{deviation}) = (0.10, 0.35, 0.10)$. Fixed evidence confidences are $C_{probe} = 0.45$, $C_{contract} = 0.75$, $C_{static} = 0.80$, and $C_{meta} = 0.35$. LLM calls use temperature 0.2 and at most 1024 output tokens. Global memory uses deterministic 128-dimensional hashed token-count embeddings and cosine similarity; it retrieves the top two memories without a hard similarity threshold. In `no_tests`, probe generation uses four weak assertions from task text only, sanitized against imports, randomness, file/network/system calls, and executed in the subprocess sandbox.

For cost accounting, we use the logged prompt and completion tokens and the pricing assumptions recorded with each run. The Qwen experiments use CNY 0.3 per million input tokens and CNY 1.2 per million output tokens; DeepSeek reports use the provider-specific rates saved in the corresponding report. Wall-clock values are total elapsed time for the complete task set, not per-task latency, and par-

Table 6: Experimental data files and usage.

File	Tasks	Use	Source or Description
`HumanEval_feedback.jsonl`	164	known input	Visible feedback parsed from doctest examples in HumanEval prompts; some tasks have no visible examples
`HumanEval_no_tests.jsonl`	164	`no_tests` input	Explicit tests removed; only the prompt and entry point are retained
`HumanEval_eval.jsonl`	164	hidden eval	Original HumanEval test fields retained only for final offline evaluation
`mbpp_sanitized_feedback_split.json`	427	known input	Converted from Google Research `sanitized-mbpp.json`; only the first assertion per task is used as search feedback
`mbpp_sanitized_no_tests.json`	427	`no_tests` input	Explicit assertions removed; only task description and entry point are retained
`mbpp_sanitized_eval_split.json`	427	hidden eval	Setup/imports and remaining assertions retained for final evaluation
`HumanEval-X C++ first 30`	30	cross-language pilot	Public `example_test` for search; official `test` retained only for final evaluation

Table 7: HumanEval-X C++ pilot on the first 30 tasks. Official tests are used only for final hidden evaluation. Costs use the logged-token pricing assumptions of the Python experiments.

Model	Method	Tasks	Visible pass	Hidden solve	Tok/task	Cost (CNY)
`Qwen3-8B`	Direct	30	73.33	73.33	335	0.0094
`Qwen3-8B`	ReMCTS	30	93.33	90.00	3,217	0.0818
`Qwen3-32B`	Direct	30	83.33	83.33	337	0.0381
`Qwen3-32B`	ReMCTS	30	96.67	96.67	3,245	0.3260
`DeepSeek-V4-Flash`	Direct	30	60.00	60.00	330	0.0123
`DeepSeek-V4-Flash`	ReMCTS	30	96.67	96.67	2,996	0.1063

allel workers can make them non-additive. These details explain why cost and wall time should be read as operational accounting rather than a strict compute-matched comparison.

E Search Parameters and Evaluation Metrics

Table 8 reports model invocation and decoding settings. All models are called through an OpenAI-compatible API. They are identified by the API aliases in our experiment records. API-served models are subject to server-side updates. Therefore, we fix the experimental protocol with aliases, run time, decoding parameters, and data hashes.

Table 9 gives the search budget used in the main experiments. To make `known` and `no_tests` comparable, they use the same maximum iterations, branch factor, maximum depth, and minimum acceptance iterations. Direct is a single-generation baseline. Best-of-8 generates exactly 8 candidates. Reflexion-style runs up to 8 single-path feedback rounds. ReMCTS, pure-MCTS, and MCTS-style use the same tree-search budget. If a tree has n nodes, then the $n-1$ non-root nodes correspond to generated and executed program candidates. Semantic reviews, reflections, and LLM probes add auxiliary calls when they are enabled. Different API services do not expose perfectly consistent token accounting, so the main analysis uses average iterations and search-tree size as compute-budget proxies across models and methods. For the direct ReMCTS/MCTS-style comparison, Appendix Table 13 additionally reports logged prompt-plus-completion token totals from the corresponding experiment reports.

Let $a_i \in \{0,1\}$ indicate whether the internal acceptor of a method automatically accepts task i. Let $h_i \in \{0,1\}$ indicate whether the final returned candidate passes hidden tests. Hidden Solve Rate is computed over all tasks as $\frac{1}{n}\sum_i h_i$. It does not depend on automatic acceptance. Accepted Rate is $\frac{1}{n}\sum_i a_i$. Acceptance Precision is $\frac{\sum_i a_i h_i}{\sum_i a_i + \epsilon}$. False Acceptance Rate is $\frac{\sum_i a_i(1-h_i)}{\sum_i a_i + \epsilon}$. For ReMCTS, pure-MCTS, and MCTS-style, a_i comes from their acceptance thresholds. For Best-of-8 and Reflexion-style, a_i means that self-generated tests or the evaluator accepted a candidate. If $a_i = 0$, the system still returns the current best candidate for hidden evaluation. Thus, the task affects Hidden Solve Rate, but it is not included in the denominator of acceptance precision or false acceptance. Table 10 summarizes these metric definitions.

Table 8: Model invocation and decoding settings.

Item	Setting
Model alias	`Qwen3-1.7B`, `Qwen3-8B`, `Qwen3-14B`, `Qwen3-32B`, `DeepSeek-V4-Flash`
Invocation	OpenAI-compatible API; main experiments were run in May 2026 and the budget-up/C++ pilots in July 2026
Direct, ReMCTS, and tree-search baselines	Code-generation temperature 0.2, maximum output length 1024 tokens, and API request timeout 90 seconds
Best-of-8	8 independent candidates per task, generation temperature 0.7, and 5 self-generated assertion tests for selection
Best-of-32	32 independent candidates per task with the same generation and proxy-selection protocol as Best-of-8
Reflexion-style	Up to 8 Actor–Evaluator–Self-Reflection rounds; Actor, Evaluator, and reflection temperatures are all 0.2
Reflexion-16	Up to 16 rounds with the same Actor–Evaluator–Self-Reflection protocol
Sandbox execution	Python timeout 5 seconds; C++17 pilot timeout 8 seconds; hidden tests are used only for final offline evaluation and Oracle Pool statistics

Table 9: Search parameters for the main ReMCTS experiments.

Parameter	Value
`max_iterations`	8
`branch_factor`	5
`max_depth`	8
`no_improve_patience`	8
`min_accept_iterations`	3
`complete_branch_before_accept`	true
Final metric	hidden solve rate

Table 10: Definitions of evaluation metrics.

Metric	Definition	Applicable Settings
Hidden Solve Rate	Fraction of tasks whose final candidate passes hidden tests	All experiments
Accepted Rate	Fraction of tasks whose candidate is automatically accepted	Methods/settings with an internal acceptor
Acceptance Precision	Fraction of automatically accepted candidates that pass hidden tests	Methods/settings with an internal acceptor
False Acceptance Rate	Fraction of automatically accepted candidates that fail hidden tests	Methods/settings with an internal acceptor
Avg. Iterations	Average search iterations per task	Search efficiency analysis
Avg. Tree Size	Average search-tree nodes per task	Search efficiency analysis
Oracle Pool Hidden Rate	Fraction of tasks where at least one candidate in the pool passes hidden tests	Offline analysis for multi-candidate and tree-search methods

F Supplementary Baseline Results

Table 11 reports the full supplementary results for Best-of-8, Reflexion-style, pure-MCTS, and MCTS-style. Hidden tests are used only for final offline evaluation. They are never used for candidate selection.

G Known-Feedback Baseline Results

Table 12 reports the completed known-feedback baseline results. These runs use visible tests as feedback during search and hidden tests only for final offline evaluation. Reflexion-style is a single-path Actor–Evaluator–Self-Reflection loop. MCTS-style keeps tree search but disables path memory, global memory retrieval, semantic UCB, and LLM probes.

H Logged Token Comparison

Table 13 compares the logged token use of full ReMCTS and MCTS-style on the same model–dataset–feedback settings used in the compact baseline comparison. Tokens are the sum of logged prompt and completion tokens in the experiment reports. Across all twelve comparable settings, ReMCTS uses 1.09× the total tokens of MCTS-style. The ratio is 1.14× in `no_tests` and 1.04× in `known`.

Table 11: Supplementary baselines for Best-of-8, Reflexion-style, pure-MCTS, and MCTS-style. Accepted is the number of tasks accepted by the method's internal acceptor. It is not the number of final returned tasks. Hidden Solve is always computed with the final returned candidate for each task. Oracle Pool is the offline upper bound indicating whether at least one candidate in the pool passes hidden tests. Best-of-8 uses 8 independent candidates. Reflexion-style uses all attempts. pure-MCTS and MCTS-style use non-root nodes generated in the search tree. DeepSeek abbreviates DeepSeek-V4-Flash.

Parameter note: Best-of-8 uses $n = 8$ independent candidates. It uses a candidate-generation temperature of 0.7 and 5 self-generated assertion tests for selection. Reflexion-style runs up to 8 Actor–Evaluator–Self-Reflection rounds. Actor and reflection generation temperatures are both 0.2. The Evaluator uses the same 5 self-generated assertion tests. pure-MCTS uses the same search budget as the main experiments. The settings are `max_iterations`=8, `branch_factor`=5, `max_depth`=8, `min_accept_iterations`=3, and `complete_branch_before_accept`=true. The UCB exploration coefficient is $C = 1.4$. pure-MCTS sets `path_memory_k`=0, `global_memory_top_t`=0, and semantic heuristic $\alpha = \beta = 0$. It also disables LLM semantic scoring and LLM probes. MCTS-style uses the same search budget as pure-MCTS. It also disables path memory, global memory, semantic UCB, and LLM probes. It enables LLM semantic scoring as a production-style reward and acceptance component. The settings are `w_semantic`=0.3 and `semantic_threshold`=0.45.

Model	Dataset	Method	Hidden Solve	Accepted	Acc. Prec.	False Acc.	Oracle Pool
Qwen3-8B	HumanEval	Best-of-8	82.93%	95/164	98.95%	1.05%	82.93%
Qwen3-8B	HumanEval	Reflexion	81.10%	99/164	96.97%	3.03%	84.15%
Qwen3-8B	HumanEval	pure-MCTS	84.76%	163/164	84.66%	15.34%	87.20%
Qwen3-8B	HumanEval	MCTS-style	85.98%	90/164	90.00%	10.00%	86.59%
Qwen3-8B	MBPP	Best-of-8	60.42%	190/427	80.53%	19.47%	61.83%
Qwen3-8B	MBPP	Reflexion	61.59%	234/427	74.36%	25.64%	64.17%
Qwen3-8B	MBPP	pure-MCTS	64.87%	427/427	64.87%	35.13%	66.51%
Qwen3-8B	MBPP	MCTS-style	64.87%	302/427	65.23%	34.77%	65.81%
Qwen3-32B	HumanEval	Best-of-8	88.41%	108/164	96.30%	3.70%	88.41%
Qwen3-32B	HumanEval	Reflexion	90.24%	118/164	97.46%	2.54%	90.85%
Qwen3-32B	HumanEval	pure-MCTS	89.02%	162/164	89.51%	10.49%	92.68%
Qwen3-32B	HumanEval	MCTS-style	90.24%	125/164	94.40%	5.60%	92.68%
Qwen3-32B	MBPP	Best-of-8	63.00%	207/427	80.68%	19.32%	63.47%
Qwen3-32B	MBPP	Reflexion	68.85%	289/427	78.55%	21.45%	70.26%
Qwen3-32B	MBPP	pure-MCTS	69.79%	427/427	69.79%	30.21%	70.49%
Qwen3-32B	MBPP	MCTS-style	69.56%	361/427	67.59%	32.41%	71.43%
DeepSeek	HumanEval	Best-of-8	95.73%	131/164	98.47%	1.53%	96.34%
DeepSeek	HumanEval	Reflexion	94.51%	138/164	97.83%	2.17%	96.95%
DeepSeek	HumanEval	pure-MCTS	92.07%	164/164	92.07%	7.93%	95.73%
DeepSeek	HumanEval	MCTS-style	94.51%	162/164	94.44%	5.56%	95.73%
DeepSeek	MBPP	Best-of-8	73.54%	322/427	78.88%	21.12%	77.52%
DeepSeek	MBPP	Reflexion	71.90%	356/427	77.81%	22.19%	74.47%
DeepSeek	MBPP	pure-MCTS	70.96%	427/427	70.96%	29.04%	76.35%
DeepSeek	MBPP	MCTS-style	71.19%	425/427	71.29%	28.71%	76.58%

Table 12: Known-feedback baseline completion results. Hidden Solve, Accepted, Acc. Prec., and False Acc. are percentages.

Model	Dataset	Method	Hidden Solve	Accepted	Acc. Prec.	False Acc.	Avg. Iter.
DeepSeek-V4-Flash	HumanEval	Reflexion-style	96.34	96.34	100.00	0.00	1.360
DeepSeek-V4-Flash	HumanEval	MCTS-style	96.34	96.34	100.00	0.00	3.183
DeepSeek-V4-Flash	MBPP	Reflexion-style	94.15	93.21	100.00	0.00	1.684
DeepSeek-V4-Flash	MBPP	MCTS-style	75.88	74.24	100.00	0.00	4.288
Qwen3-8B	HumanEval	Reflexion-style	86.59	86.59	100.00	0.00	2.098
Qwen3-8B	HumanEval	MCTS-style	86.59	86.59	100.00	0.00	3.445
Qwen3-8B	MBPP	Reflexion-style	81.97	79.86	100.00	0.00	2.611
Qwen3-8B	MBPP	MCTS-style	66.74	64.64	100.00	0.00	4.002
Qwen3-32B	HumanEval	Reflexion-style	93.29	93.29	100.00	0.00	1.604
Qwen3-32B	HumanEval	MCTS-style	93.90	93.90	100.00	0.00	3.244
Qwen3-32B	MBPP	Reflexion-style	86.89	85.71	100.00	0.00	2.150
Qwen3-32B	MBPP	MCTS-style	71.19	69.56	100.00	0.00	3.979

I Paired Significance Tests

All tests below are two-sided exact McNemar tests on paired per-task hidden outcomes. The Wilson interval is shown for the second method's solve rate. We report these tests descriptively; no claim of theoretical optimality is based on them.

J Memory Ablation Results

Table 16 reports the full memory ablation results. The no-global-memory variant sets `global_memory_top_t`=0. The no-path-memory variant sets `path_memory_k`=0. These ablations use the same final hidden-evaluation protocol as the main experiments.

Table 13: Logged token comparison between full ReMCTS and MCTS-style. Token counts are prompt-plus-completion totals in millions (M). Ratio is ReMCTS tokens divided by MCTS-style tokens.

Model	Dataset	Setting	ReMCTS Tok. (M)	MCTS-style Tok. (M)	Ratio
Qwen3-8B	HumanEval	no_tests	2.52	2.26	1.12
Qwen3-8B	HumanEval	known	1.07	1.74	0.61
Qwen3-8B	MBPP	no_tests	4.40	3.96	1.11
Qwen3-8B	MBPP	known	3.93	3.77	1.04
Qwen3-32B	HumanEval	no_tests	2.32	2.23	1.04
Qwen3-32B	HumanEval	known	0.80	1.72	0.46
Qwen3-32B	MBPP	no_tests	4.16	3.58	1.16
Qwen3-32B	MBPP	known	4.01	3.52	1.14
DeepSeek-V4-Flash	HumanEval	no_tests	3.01	2.74	1.10
DeepSeek-V4-Flash	HumanEval	known	0.80	1.64	0.49
DeepSeek-V4-Flash	MBPP	no_tests	7.43	6.20	1.20
DeepSeek-V4-Flash	MBPP	known	7.66	5.13	1.49
no_tests total			23.83	20.97	1.14
known total			18.26	17.51	1.04
Overall total			42.09	38.48	1.09

Table 14: Paired hidden-test comparisons between Direct no_tests and ReMCTS known. "Only" columns count tasks solved by exactly one method.

Model	Dataset	Direct	ReMCTS	ReMCTS-only	Direct-only	Exact p	ReMCTS 95% CI
Qwen3-1.7B	HumanEval	99/164	137/164	40	2	4.11×10^{-10}	77.11–88.43
Qwen3-1.7B	MBPP	228/427	271/427	62	19	1.77×10^{-6}	58.80–67.89
Qwen3-8B	HumanEval	131/164	154/164	25	2	5.65×10^{-6}	89.14–96.65
Qwen3-8B	MBPP	274/427	317/427	57	14	2.67×10^{-7}	69.89–78.16
Qwen3-14B	HumanEval	136/164	136/164	15	15	1.00	76.43–87.92
Qwen3-14B	MBPP	286/427	280/427	40	46	0.59	60.95–69.92
Qwen3-32B	HumanEval	147/164	156/164	13	4	0.0490	90.67–97.51
Qwen3-32B	MBPP	307/427	333/427	35	9	1.06×10^{-4}	73.82–81.66
DeepSeek-V4-Flash	HumanEval	132/164	162/164	30	0	1.86×10^{-9}	95.66–99.66
DeepSeek-V4-Flash	MBPP	310/427	346/427	47	11	2.03×10^{-6}	77.04–84.47

Table 15: Paired comparison of ReMCTS known with the same-search-budget MCTS-style baseline. MCTS-style disables path/global memory, semantic UCB, and LLM probes.

Model	Dataset	MCTS-style	ReMCTS	ReMCTS-only	MCTS-only	Exact p	ReMCTS 95% CI
Qwen3-8B	HumanEval	142/164	154/164	13	1	0.00183	89.14–96.65
Qwen3-8B	MBPP	285/427	317/427	33	1	4.07×10^{-9}	69.89–78.16
Qwen3-32B	HumanEval	154/164	156/164	5	3	0.727	90.67–97.51
Qwen3-32B	MBPP	304/427	333/427	30	1	2.98×10^{-8}	73.82–81.66
DeepSeek-V4-Flash	HumanEval	158/164	162/164	5	1	0.219	95.66–99.66
DeepSeek-V4-Flash	MBPP	324/427	346/427	27	5	1.13×10^{-4}	77.04–84.47

K Search-Tree Structure Statistics

Table 17 reports search-tree statistics for all main ReMCTS experiments. For known, we use the latest hidden-eval reruns. For no_tests, we use the v2 results adopted in the main table. Effective branch is the micro-average number of child nodes over non-leaf nodes. It equals total edges divided by total non-leaf nodes.

L Minimum Search Iteration Ablation

Tables 18 and 19 compare early experiments with the current main experiments. The early experiments set the minimum number of search iterations to 1. To make correctness metrics comparable, Score@1 for older known runs is obtained by rerunning hidden tests offline. These tests use the final candidate code recorded in the early run directories. Older no_tests runs directly use the hidden solve rates in the original reports. The current min_accept_iterations=3 generally gives more stable hidden solve rates in no_tests. In known, it mainly changes search scale and leaves scores nearly unchanged.

Table 16: Memory ablation results. Hidden Solve, Accepted, and False Acc. are percentages.

Model	Dataset	Method	Hidden Solve	Accepted	False Acc.	Avg. Iter.	Avg. Tree
DeepSeek-V4-Flash	HumanEval known	No global	99.39	99.39	0.00	3.043	7.165
DeepSeek-V4-Flash	HumanEval known	No path	99.39	99.39	0.00	3.030	7.238
DeepSeek-V4-Flash	HumanEval no_tests	No global	94.51	99.39	5.52	3.030	6.835
DeepSeek-V4-Flash	HumanEval no_tests	No path	94.51	100.00	5.49	3.000	6.695
DeepSeek-V4-Flash	MBPP known	No global	80.56	79.39	0.00	4.082	10.761
DeepSeek-V4-Flash	MBPP known	No path	80.09	78.45	0.00	4.082	11.354
DeepSeek-V4-Flash	MBPP no_tests	No global	73.07	99.77	26.76	3.014	7.862
DeepSeek-V4-Flash	MBPP no_tests	No path	72.83	100.00	27.17	3.000	7.621
Qwen3-8B	HumanEval known	No global	92.07	92.07	0.00	3.341	3.159
Qwen3-8B	HumanEval known	No path	95.12	95.12	0.00	3.213	3.366
Qwen3-8B	HumanEval no_tests	No global	85.37	87.80	13.19	3.354	3.262
Qwen3-8B	HumanEval no_tests	No path	86.59	88.41	12.41	3.518	3.689
Qwen3-8B	MBPP known	No global	69.56	67.68	0.00	4.124	3.471
Qwen3-8B	MBPP known	No path	73.54	71.43	0.00	4.389	4.370
Qwen3-8B	MBPP no_tests	No global	66.04	92.97	31.49	3.204	3.162
Qwen3-8B	MBPP no_tests	No path	66.51	94.85	31.60	3.239	3.290
Qwen3-32B	HumanEval known	No global	95.12	95.12	0.00	3.213	3.665
Qwen3-32B	HumanEval known	No path	95.12	95.12	0.00	3.226	3.811
Qwen3-32B	HumanEval no_tests	No global	90.85	97.56	8.13	3.098	3.567
Qwen3-32B	HumanEval no_tests	No path	89.63	98.17	8.70	3.091	3.677
Qwen3-32B	MBPP known	No global	74.71	72.83	0.00	4.115	4.052
Qwen3-32B	MBPP known	No path	78.69	77.28	0.00	4.145	4.934
Qwen3-32B	MBPP no_tests	No global	69.56	96.72	28.33	3.178	3.515
Qwen3-32B	MBPP no_tests	No path	70.02	98.59	28.98	3.098	3.548

Table 17: Search-tree structure statistics for all main ReMCTS experiments. Range gives the minimum and maximum number of tree nodes per task. Eff. Branch is the micro-average effective branch factor over non-leaf nodes.

Model	Dataset	Setting	Avg. Iter.	Avg. Tree	Median	Range	Avg. Depth	Eff. Branch
Qwen3-1.7B	HumanEval	no_tests	3.098	5.165	5	2–17	1.396	2.858
Qwen3-1.7B	HumanEval	known	3.835	5.762	5	2–21	1.384	2.861
Qwen3-1.7B	MBPP	no_tests	3.068	5.433	5	2–15	1.412	2.967
Qwen3-1.7B	MBPP	known	4.974	7.651	7	2–29	1.609	2.820
Qwen3-8B	HumanEval	no_tests	3.500	3.671	3	2–12	1.341	1.904
Qwen3-8B	HumanEval	known	3.354	3.537	3	2–21	1.366	1.741
Qwen3-8B	MBPP	no_tests	3.232	3.389	3	2–27	1.241	1.828
Qwen3-8B	MBPP	known	4.361	4.344	3	2–22	1.283	2.112
Qwen3-14B	HumanEval	no_tests	3.762	5.793	5	2–29	1.360	2.989
Qwen3-14B	HumanEval	known	3.823	5.335	4	2–27	1.317	2.938
Qwen3-14B	MBPP	no_tests	4.162	6.719	5	2–37	1.342	2.942
Qwen3-14B	MBPP	known	4.696	6.260	5	2–28	1.279	2.999
Qwen3-32B	HumanEval	no_tests	3.079	3.683	3	2–27	1.317	1.956
Qwen3-32B	HumanEval	known	3.256	3.604	3	2–16	1.317	1.889
Qwen3-32B	MBPP	no_tests	3.131	3.628	3	2–24	1.283	1.941
Qwen3-32B	MBPP	known	4.204	5.096	3	2–22	1.372	2.141
DeepSeek-V4-Flash	HumanEval	no_tests	3.006	7.116	6	2–21	1.232	4.438
DeepSeek-V4-Flash	HumanEval	known	3.061	7.463	6	2–31	1.293	4.435
DeepSeek-V4-Flash	MBPP	no_tests	3.030	8.019	6	3–33	1.365	4.440
DeepSeek-V4-Flash	MBPP	known	4.056	11.593	8	3–40	1.564	4.578

M Qualitative Case Analysis

The qualitative interpretation in Section 4.5 is not inferred from aggregate accuracy alone. We also inspect representative search trajectories where Direct and ReMCTS differ. These cases are illustrative. They are not a full manual audit. However, they show how the same mechanisms behind the aggregate trends appear in concrete tasks.

N Evaluation Protocol Notes

All tables use hidden tests as the final correctness metric. The known setting allows visible test feedback during search. Therefore, its results are interpreted as test-guided mechanism validation. They are not fully equivalent to standard pass@1. The no_tests setting uses no explicit tests. It is closer to production-style code generation without test cases. However, HumanEval and MBPP-Sanitized are still function-level benchmarks. They cannot fully cover cross-file context, dependency installation, permission control, or security risks in real software projects.

The Best-of-8, Reflexion-style, pure-MCTS, and MCTS-style baselines are run on Qwen3-8B, Qwen3-32B, and DeepSeek-V4-Flash. They test whether diverse sampling, single-path reflection, reflection-free tree search, or tree search with only

Table 18: Effect of changing the minimum search iterations from 1 to 3 on hidden solve rate. Score@1 and Score@3 are percentages. ΔScore is the percentage-point difference.

Model	Dataset	Setting	Score@1	Score@3	ΔScore
Qwen3-1.7B	HumanEval	no_tests	64.02	69.51	+5.49
Qwen3-1.7B	HumanEval	known	82.32	83.54	+1.22
Qwen3-1.7B	MBPP	no_tests	51.99	51.29	-0.70
Qwen3-1.7B	MBPP	known	63.70	63.47	-0.23
Qwen3-8B	HumanEval	no_tests	87.20	86.59	-0.61
Qwen3-8B	HumanEval	known	93.29	93.90	+0.61
Qwen3-8B	MBPP	no_tests	63.93	66.74	+2.81
Qwen3-8B	MBPP	known	74.47	74.24	-0.23
Qwen3-14B	HumanEval	no_tests	77.44	78.66	+1.22
Qwen3-14B	HumanEval	known	82.32	82.93	+0.61
Qwen3-14B	MBPP	no_tests	59.02	60.42	+1.41
Qwen3-14B	MBPP	known	66.28	65.57	-0.70
Qwen3-32B	HumanEval	no_tests	90.24	89.63	-0.61
Qwen3-32B	HumanEval	known	96.34	95.12	-1.22
Qwen3-32B	MBPP	no_tests	68.38	70.49	+2.11
Qwen3-32B	MBPP	known	78.92	77.99	-0.94
DeepSeek-V4-Flash	HumanEval	no_tests	93.29	95.12	+1.83
DeepSeek-V4-Flash	HumanEval	known	99.39	98.78	-0.61
DeepSeek-V4-Flash	MBPP	no_tests	70.26	72.83	+2.58
DeepSeek-V4-Flash	MBPP	known	81.03	81.03	+0.00

Table 19: Effect of changing the minimum search iterations from 1 to 3 on search scale. Iter@1/Tree@1 come from early experiments without the current lower bound. Iter@3/Tree@3 come from the current main experiments. ΔTree is the change in average search-tree size.

Model	Dataset	Setting	Iter@1	Iter@3	Tree@1	Tree@3	ΔTree
Qwen3-1.7B	HumanEval	no_tests	1.293	3.098	2.591	5.165	+2.573
Qwen3-1.7B	HumanEval	known	2.317	3.835	3.927	5.762	+1.835
Qwen3-1.7B	MBPP	no_tests	1.237	3.068	2.506	5.433	+2.927
Qwen3-1.7B	MBPP	known	3.864	4.974	6.129	7.651	+1.522
Qwen3-8B	HumanEval	no_tests	3.677	3.500	4.073	3.671	-0.402
Qwen3-8B	HumanEval	known	1.585	3.354	2.677	3.537	+0.860
Qwen3-8B	MBPP	no_tests	2.494	3.232	3.410	3.389	-0.021
Qwen3-8B	MBPP	known	2.993	4.361	3.710	4.344	+0.635
Qwen3-14B	HumanEval	no_tests	2.640	3.762	4.244	5.793	+1.549
Qwen3-14B	HumanEval	known	2.244	3.823	3.159	5.335	+2.177
Qwen3-14B	MBPP	no_tests	2.419	4.162	4.473	6.719	+2.246
Qwen3-14B	MBPP	known	3.471	4.696	4.799	6.260	+1.461
Qwen3-32B	HumanEval	no_tests	2.329	3.079	3.622	3.683	+0.061
Qwen3-32B	HumanEval	known	1.354	3.256	2.470	3.604	+1.134
Qwen3-32B	MBPP	no_tests	1.644	3.131	2.895	3.628	+0.733
Qwen3-32B	MBPP	known	2.705	4.204	4.098	5.096	+0.998
DeepSeek-V4-Flash	HumanEval	no_tests	1.104	3.006	2.463	7.116	+4.652
DeepSeek-V4-Flash	HumanEval	known	1.116	3.061	2.530	7.463	+4.933
DeepSeek-V4-Flash	MBPP	no_tests	1.014	3.030	2.136	8.019	+5.883
DeepSeek-V4-Flash	MBPP	known	2.525	4.056	7.005	11.593	+4.588

LLM semantic review can explain the gains of ReMCTS. None of the four methods uses hidden tests for candidate selection. Hidden tests are used only for final offline evaluation and Oracle Pool upper-bound statistics. For pure-MCTS and MCTS-style, Oracle Pool uses non-root nodes generated in the search tree as the candidate pool.

Table 20: Representative trajectories used to interpret the aggregate trends. The examples are selected from the latest main-result runs. They use hidden tests only for offline analysis.

Case	Observation
Qwen3-1.7B, HumanEval/5, `known`	Direct fails hidden evaluation on `intersperse`. In the ReMCTS run, the search tree has 10 nodes. Three root candidates fail all visible tests. Several sibling candidates pass all 3 visible tests. The selected candidate inserts the delimiter only between adjacent elements. It passes hidden evaluation. This supports the view that weaker models often benefit from sandbox filtering and evidence-based selection.
Qwen3-8B, HumanEval/26, `known`	Direct fails hidden evaluation on `remove_duplicates`. The ReMCTS run needs 7 iterations and reaches depth 3. Early nodes repeatedly fail visible tests. The reflection memory records two issues. The model confuses "keep first occurrence" with "keep elements that occur once". It also misses a `List` import. The final node counts element frequencies. It keeps only elements with count one. It passes all 3 visible tests and hidden evaluation. This shows how path reflections can prevent repeated repair mistakes.
Qwen3-32B, HumanEval/83, `known`	Direct fails hidden evaluation on `starts_one_ends`. It overcounts numbers ending in 1. Qwen3-32B Direct is already strong overall. Thus, this is a residual error rather than a broad failure mode. ReMCTS explores 10 nodes over 8 iterations. The reflection memory points to an inclusion–exclusion error and the special case $n = 1$. The selected candidate passes all 7 visible tests and hidden evaluation. This supports the claim that stronger models mainly benefit by recovering a smaller set of missed implementations.
Qwen3-14B, MBPP task 9, `no_tests`	The task asks for the minimum positive rotation needed to obtain the same string. In `no_tests`, ReMCTS accepts a candidate that returns `-1` for strings with no smaller period. Hidden evaluation expects full-length rotations, e.g., `find_Rotations("ab")==2` and `find_Rotations("abc")==3`. Probe evidence partially fails. Contract and static checks pass. The LLM semantic review remains optimistic. This produces a false acceptance. This case shows why proxy evidence can be insufficient on MBPP. It also shows why `no_tests` improvements are less stable.