# Reason-Code: Reliable Code Generation via Test-Driven Monte Carlo Tree Search

**Zixu Li**[1]* † **Zhiqi Peng**[2]*
[1]Sun Yat-sen University
[2]Independent Researcher
zixu.li915@gmail.com, zhiqi.peng16@gmail.com

## Abstract

Large Language Models (LLMs) are widely used for code generation, but their performance degrades on tasks requiring multi-step logical reasoning. In practice, reliability is often improved through multi-sample inference, but its cost grows linearly with the sample size, making it impractical under strict latency constraints. To address this, we propose Reason-Code, an inference-time framework that formulates code generation as a search process guided by execution feedback. It integrates Monte Carlo Tree Search (MCTS) with a lightweight execution sandbox, where candidate programs are evaluated via unit tests. To control inference cost, Reason-Code adopts a conditional budgeting strategy that activates search only when greedy generation fails. Compared with large-sample Best-of-$N$ sampling, Reason-Code is designed to improve reliability without paying the full linear cost of additional sampling under strict latency budgets. Experiments on HumanEval and MBPP show that Reason-Code matches strong sampling baselines (e.g., Best-of-10) with lower token cost and no regression. Additional matched-budget analyses show that execution-guided adaptive inference improves over independent sampling/filtering baselines, while differences between UCB-guided search and simpler iterative repair are limited at low budget.

## 1 Introduction

Large Language Models (LLMs) have enabled automated code generation, yet they still struggle with logical consistency on complex tasks. In practice, reliability is often improved via Best-of-$N$ rejection sampling; however, inference costs scale linearly with the sample size, leading to diminishing marginal returns. Moreover, increasing the sample size alone does not directly improve reasoning quality, and extensive external validation introduces additional latency and engineering overhead in practical systems. While recent work suggests that increasing inference-time computation can improve reasoning performance, efficiency and scalability remain key challenges (Lyu et al., 2025).

Our key observation is that improving code generation requires better-guided samples, where execution feedback guides search instead of increasing sample volume. Inspired by test-driven development, we treat execution not as a post-hoc filter but as a real-time feedback signal during generation.

Reason-Code is designed for deployment-constrained settings, where average latency, token cost, and robustness matter more than maximizing unconstrained search depth. Table 1 and Figure 2 show that Reason-Code achieves comparable accuracy at substantially lower cost.

We formulate code generation as a structured search problem with two practical design choices. First, we employ a model-agnostic UCB-based search strategy that does not rely on token-level log probabilities, ensuring compatibility with quantized or black-box models. Second, inspired by the Reflexion mechanism (Nathani et al., 2023), execution errors are converted into natural language feedback to guide refinement. Search is activated only when greedy decoding ($T = 0$) fails, and refinement uses controlled sampling ($T = 0.5$), avoiding unnecessary computation on simple tasks. Experiments on HumanEval and MBPP demonstrate that Reason-Code achieves strong performance under strict cost constraints. On the HumanEval benchmark, it matches the accuracy of 10× sampling with only 1.5× relative cost under the original deployment-oriented comparison. On MBPP, we observe zero regression compared to the greedy baseline, suggesting that conditional search improves performance on hard cases without degrading results on simple ones. We observe statistically significant improvements on MBPP under the same token budget, indicating that execution

* Equal contribution.
† Corresponding author.

| Method | Inference Strategy | Cost (Rel.) | MBPP Pass@1 | HumanEval Pass@1 |
|---|---|---|---|---|
| *Baselines* | | | | |
| Greedy Decoding | Direct Generation | $1.0\times$ | 71.2 | 86.6 |
| Best-of-$N$ Sampling | Stochastic ($N = 10$) | $10.0\times$ | - | 87.8 |
| *Best-of-$N$ (Oracle)* | *Oracle Est. ($N = 5$)* | $5.0\times$ | ***72.8*** | - |
| *Ours (Reason-Code)* | | | | |
| MCTS (Static) | Always-on ($N = 3$) | $\sim 3.5\times$ | **72.8** | 86.0 |
| **MCTS (Adaptive)** | **Conditional Budget** | $\sim 1.5\times$ | **72.8** | **88.4** |

Table 1: **Deployment-Oriented Main Results.** Adaptive Reason-Code provides a favorable cost–accuracy trade-off: it matches the Best-of-5 oracle-style reference on MBPP and reaches the strongest HumanEval result at $\sim$1.5$\times$ average cost, significantly lower than large-sample sampling. "Cost" denotes relative token consumption vs. Greedy ($1.0\times$).

feedback provides a reliable search signal.

Our contributions are as follows:

- We present Reason-Code, an inference-time search framework integrating MCTS with execution feedback.
- We introduce a conditional budgeting strategy that prevents regressions while minimizing overhead.
- Controlled matched-budget analyses show that execution-guided adaptive inference is stronger than independent sampling/filtering baselines on MBPP, while the difference between UCB-guided and simpler tree or iterative repair strategies is not statistically significant at the small budget studied here.

## 2 Related Work

### 2.1 Test-Driven Reasoning Search

Recent work has explored combining LLMs with structured search to improve difficult multi-step reasoning (Yao et al., 2023; Besta et al., 2024). In code generation, related directions include large-scale candidate sampling (Li et al., 2022), generic planning-style tree search (Zhou et al., 2024), and execution-aware refinement or process supervision (Li et al., 2025a,b; Zhang et al., 2025). These approaches primarily focus on improving search effectiveness or pushing the performance frontier under relatively generous compute budgets.

Our setting is distinct in its emphasis on adaptive compute. We focus on low-budget, deployment-constrained inference, where the practical question is not only whether search helps, but also when it should be activated and how much additional computation it can justify under a strict total budget.

### 2.2 Execution-Guided Refinement and Verification

Execution feedback has become a common signal for improving code generation, including self-debugging (Chen et al., 2024), execution-based verification (Ni et al., 2023), verbal feedback-/refinement (Shinn et al., 2024; Nathani et al., 2023; Madaan et al., 2023), and reward-model or reinforcement-learning approaches (Li et al., 2025a; Zhong et al., 2024; Le et al., 2022). These methods differ in whether execution is used as a post-hoc filter, as a training signal, or as an online refinement signal during inference.

Reason-Code uses execution strictly at inference time and does not require additional training, learned reward models, or token-level confidence estimates. In this sense, our approach is closest to execution-guided online repair, but with an explicit search controller and a conditional activation mechanism.

### 2.3 Efficient Inference and Edge Computing

As LLM applications shift to edge devices, balancing reasoning capability with hardware constraints has become central (Liu et al., 2023). Recent work has pursued several inference-time strategies. For example, Chen et al. (Chen et al., 2025) proposed a Model Cascading strategy that routes queries between small and large models using self-generated tests as a quality gate. Similarly, Speculative Decoding (Leviathan et al., 2023) accelerates generation via draft models. Our work follows this trend by prioritizing average-case efficiency through conditional search on a single local model.

Our conditional budgeting strategy enables fully offline, on-device reasoning under strict resource

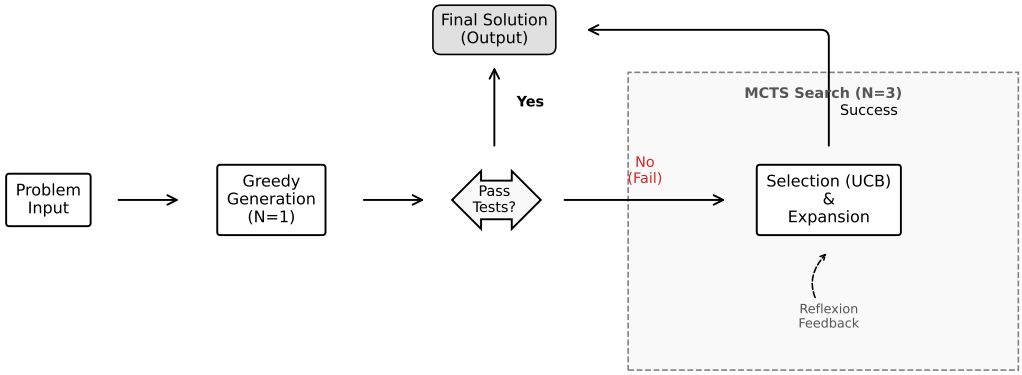


Figure 1: **System Overview.** Reason-Code employs a conditional workflow. Simple tasks are resolved via greedy decoding ($N = 1$). The MCTS loop ($N = 3$) is activated only upon execution failure, using execution feedback to guide exploration. This design ensures computational efficiency and prevents performance regression.

constraints. Unlike cascading systems that rely on cloud-based API latency (Xu et al., 2024), our framework offers a viable technical path for fully offline, high-performance coding assistants. Distinct from cascading or filtering-based approaches, our method explicitly treats inference as a resource-constrained tree search problem, where execution feedback is reused across structured partial programs under a fixed budget.

## 3 Reason-Code System Design

### 3.1 System Overview

Reason-Code is designed to capture inference-time cost characteristics that are relevant in practical deployment settings. As illustrated in Figure 1, we formulate code generation as a Markov Decision Process (MDP) and search for an optimal solution in the program space using Monte Carlo Tree Search (MCTS). It is crucial to note that Reason-Code operates purely at inference time without any additional parameter updates or fine-tuning. The system comprises two core feedback loops: a generation loop in which the LLM proposes code repair steps, and a verification loop that utilizes sandbox execution feedback to compute rewards and activate a reflexion mechanism.

Implementation-wise, each search node corresponds to a complete executable program candidate together with its execution outcome summary (e.g., pass/fail and runtime error information). Nodes are therefore not token-level states. An edge corresponds to a whole-program repair proposal conditioned on the parent node's execution feedback and repair instructions.

### 3.2 Problem Formulation

We formulate code generation as a sequential decision process defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$. This formulation is specifically designed for scenarios where token-level probabilities, gradients, or internal model states are unavailable, such as closed-source APIs or edge devices.

- **State** $s_t \in \mathcal{S}$: The state at the timestep $t$ represents the current problem description $x$ and the history of accumulated repair attempts $(a_0, a_1, \ldots, a_{t-1})$. Formally, it is defined as:

$$s_t = \langle x, (a_i)_{i=0}^{t-1} \rangle,$$

  where $s_0 = \langle x, \emptyset \rangle$ represents the initial state.
- **Action** $a_t \in \mathcal{A}$: An action represents a coherent code block or a refinement step generated by the LLM, rather than an individual token. The transition to the next state is governed by a deterministic transition function: $s_{t+1} = s_t \oplus a_t$.
  **Implementation Note.** Each action corresponds to a self-contained code block that can be executed once concatenated.
- **Reward** $\mathcal{R}(s)$: The system evaluates only terminal states using a black-box execution sandbox. Therefore, we intentionally adopt a strict binary reward. Given a test suite $\mathcal{T}$, the reward is defined as:

$$\mathcal{R}(s) = \begin{cases} 1, & \text{if all tests in } \mathcal{T} \text{ pass,} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

  In industrial settings, partially correct programs are typically unacceptable, and correctness is verified through execution.

### 3.3 Test-Driven MCTS

We utilize MCTS to explore the program space under strict execution-based evaluation. In deployment scenarios, token-level log probabilities are often unavailable, unstable, or expensive to query. Hence, we adopt a model-agnostic UCB1 selection strategy rather than PUCT, as the latter requires prior probabilities that are inaccessible in black-box or quantized settings. The search process consists of the following steps:

1. **Selection**: Starting from the root, the algorithm recursively selects a child node $j$ of the current state $s$ that maximizes the UCB1 score:

$$j^* = \arg\max_{j \in \text{children}(s)} \left( \bar{v}_j + C\sqrt{\frac{\ln n_s}{n_j}} \right)$$

   Here, $\bar{v}_j = w_j/n_j$ is the empirical success rate of child $j$, $n_s$ is the parent visit count, and $C$ controls the exploration-exploitation trade-off(set to $\sqrt{2}$ in our implementation). Unvisited nodes (where $n_j = 0$) are assigned infinite priority to ensure initial exploration coverage.

2. **Expansion & Evaluation**: At a leaf node $s_{\text{leaf}}$, we generate a small set of candidate actions $\{a^{(1)}, \ldots, a^{(k)}\}$, where each action corresponds to a repair proposal. For each action, we form a new state $s' = s_{\text{leaf}} \oplus a$ which constitutes a complete executable program candidate. The program is then executed in a sandboxed environment to obtain a binary reward according to Eq. 1. When execution fails, reflexion is applied during the expansion phase by conditioning subsequent action proposals on the resulting execution errors, while the tree policy and reward definition remain unchanged.

3. **Backpropagation**: The reward $r \in \{0, 1\}$ is propagated along the search path $\mathcal{P}$ from $s'$ back to the root. For each node $i \in \mathcal{P}$, the node statistics are updated as:

$$w_i \leftarrow w_i + r, \quad n_i \leftarrow n_i + 1.$$

   This design ensures that search decisions are guided strictly by functional correctness rather than unreliable internal heuristic scores.

**Stochastic Baseline.** To evaluate the effectiveness of UCB1 guidance, we implement a Random Tree Search (RTS) baseline that utilizes the same tree structure, simulation budget, and execution feedback as our method. The critical distinction is that it selects actions uniformly at random from the expansion candidates, bypassing the UCB1 selection policy. This design allows us to isolate the effect of informed exploration from the inherent benefits of the tree search structure and execution-guided repair.

### 3.4 Reflexion with Execution Feedback

Execution-based rewards are sparse when $\mathcal{R}(s) = 0$, as feedback is only available at terminal states. To address this sparsity, we introduce a text-based reflexion mechanism that utilizes execution errors as auxiliary guidance. When code execution fails, the system captures the standard error output $E$ (such as AssertionError or Traceback). We model the state transition as

$$s' = s \oplus \text{Refine}(E),$$

where Refine denotes a prompting function that instructs the model to generate a refined action based on the error message. Crucially, Reflexion operates as a topology-altering operator, not a reward-shaping mechanism. It injects new candidates into the search space based on runtime diagnostics, thereby improving search without corrupting the binary reward objective.

### 3.5 Conditional Budgeting

To optimize token efficiency for industrial deployment, Reason-Code implements an adaptive search strategy. For a given problem $x$, the system initially performs greedy decoding to generate a baseline solution $s_{\text{greedy}}$. The MCTS process, with a predefined computational budget $B$ (representing the total number of simulations), is activated if and only if $\mathcal{R}(s_{\text{greedy}}) = 0$. This tiered approach ensures that additional reasoning tokens are only expended on complex problems that a single pass cannot solve. This strategy aligns with practical deployment constraints, where average latency and inference cost are prioritized over worst-case performance.

## 4 Experimental Setup

### 4.1 Datasets and Metrics

To evaluate the performance of Reason-Code under practical industrial constraints, we select two estab-

lished code-generation benchmarks: HumanEval and MBPP-Sanitized. We specifically utilize the Sanitized version of MBPP to mitigate potential data contamination and ensure the integrity of the evaluation. Our evaluation framework reports the standard Pass@1 accuracy along with two key indicators designed to assess the viability of industrial deployment:

1. Token Efficiency (TE): Defined as the number of problems successfully solved per 1,000 tokens consumed. This metric accounts for the total token expenditure across all reasoning paths, including the initial greedy pass, reflexion prompts, and MCTS simulations (both successful and failed).

2. Relative Compute Cost (RCC): Quantified as the total token consumption of the Reason-Code process normalized against the consumption of a single-pass greedy decoding (set as $1.0\times$). Formally,

$$RCC = \frac{T_{\text{total}}}{T_{\text{greedy}}},$$

where $T$ denotes the token count.

### 4.2 Matched-Budget Controlled Comparisons

In addition to the original deployment-oriented evaluation, we report a matched-budget controlled comparison to isolate what is gained from execution-guided adaptive inference versus the specific choice of search controller. All methods use the same model/tokenizer, task IDs, evaluator, temperature (0.5), per-call token cap (600), and comparable hard call budgets. We compare Best-of-$N$ ($N = 3$), Reflexion-only (max 4 model calls), and UCB-MCTS (max 4 model calls). A task is counted as passed only if the generated program passes the full official unit test suite; syntax errors, runtime errors, and timeouts are all counted as failures.

### 4.3 Baselines and Comparison Methods

We employ Qwen2.5-Coder-7B-Instruct as our base model. Reason-Code is evaluated against the following three reasoning strategies:

1. **Greedy Decoding** ($T = 0$): Denoting as the minimal latency and token usage.

2. **Best-of-$N$ Sampling** ($T = 0.8$): We sample $N \in \{3, 5, 10\}$ candidates and select the final program via test execution. This represents the conventional industrial approach for improving functional coverage.

3. **Test-only Filtering**: To rigorously attribute any performance improvement specifically to the tree search mechanism, we introduce a Test-only Filtering baseline. In this setup, we first generate $k = 3$ independent programs under an equivalent token budget constraint and return the first one that passes the test suite. This baseline serves as a controlled comparison that isolates the effect of guided state reuse (MCTS) from stochastic sampling. It distinguishes the value of guided state reuse (MCTS) from the simple probability amplification of stochastic sampling.

4. **Oracle-Style Estimate**: To estimate the theoretical performance upper bound under a fixed budget, we establish an Oracle baseline using Best-of-5 sampling. A task is considered successful if at least one of the 5 samples passes all tests.

### 4.4 Implementation Details

For Reason-Code, the maximum number of MCTS simulations is set to 3 (i.e., $N = 3$) to evaluate the efficiency of "low-budget search." The exploration constant $C$ is set to 1.4 (around $\sqrt{2}$) to encourage sufficient exploration during the initial search phase.

**Hybrid Temperature Strategy:** To balance generation precision and diversity, we employ different sampling parameters across stages:

- Root Expansion: Greedy decoding ($T = 0$) is used at the root node to ensure baseline quality and minimize unnecessary initial search cost.
- Reflexion/Refinement: Sampling with $T = 0.5$ is employed to diversify the output and mitigate recurring error patterns observed in earlier execution failures. The context window is dynamically adjusted based on the task phase: standard generation is limited to 640 tokens, while reflexion generation including error messages is extended to 800 tokens to accommodate diagnostic feedback.

### 4.5 Hardware & Environment

To verify the system's practicality and low deployment overhead on consumer-grade hardware, all experiments were conducted on local workstations equipped with Apple Silicon (e.g., M2/M3 Max).

| Benchmark | Best-of-$N$ | Reflexion-only | UCB-MCTS |
|---|---|---|---|
| MBPP | 71.21 | 73.93 | 73.54 |
| HumanEval | 86.59 | 88.41 | 87.20 |

Table 2: **Matched-Budget Controlled Comparison.** All three methods use the same model, task IDs, evaluator, temperature (0.5), per-call token cap (600), and comparable hard call budgets. Best-of-$N$ uses $N = 3$; Reflexion-only and UCB-MCTS are capped at 4 model calls.

We utilized the MLX framework for efficient inference, employing a 4-bit quantized variant of the base model. This setup not only minimizes computational costs but also demonstrates the feasibility of privacy-preserving, on-device reasoning on consumer-grade hardware.

## 5 Results and Analysis

### 5.1 Main Results under Edge Deployment Constraints

- **MBPP (Complex Logic Tasks).** With $N = 3$, Reason-Code achieves 72.8% Pass@1 on MBPP, matching the Best-of-5 oracle-style reference while operating at a much lower average cost.
- **HumanEval (Basic Tasks).** Under the original deployment-oriented comparison, adaptive Reason-Code reaches 88.4% on HumanEval at roughly 1.5× average cost. On MBPP, it matches the oracle-style reference at much lower cost. Together, these results support the deployment value of conditional budgeting. We additionally report a matched-budget controlled comparison to separate the value of execution-guided adaptive inference from the specific choice of search controller.

### 5.2 Matched-Budget Controlled Comparison

Table 2 isolates the low-budget controlled setting. On MBPP, both Reflexion-only and UCB-MCTS improve over Best-of-$N$, indicating that execution-guided adaptive inference is more effective than independent sampling/filtering at the same budget. On HumanEval, Reflexion-only achieves the highest raw pass rate, while pairwise differences are not statistically significant under conservative paired testing. We therefore do not claim that tree search is uniformly better than simpler iterative repair at this budget. For HumanEval, this controlled comparison uses the seed42 canonical rerun for internal consistency with paired significance analysis.

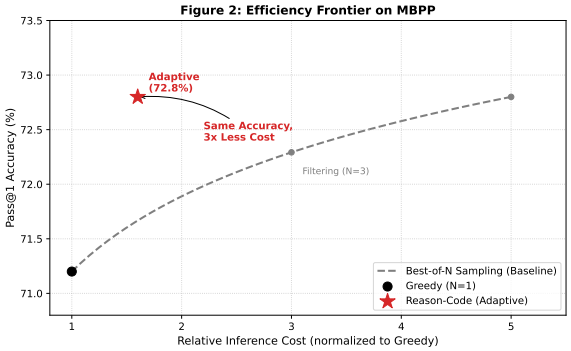


Figure 2: **Efficiency Frontier.** Reason-Code matches the Best-of-5 oracle-style reference on MBPP (72.8%) at approximately $\mathbf{1.6}\times$ cost.

### 5.3 Efficiency Analysis: Pareto Frontier and On-Device Feasibility

- **Cost efficiency**: Reason-Code reduces token consumption by approximately 40% compared to standard baselines while achieving 72.8% accuracy on the MBPP benchmark. The conditional strategy attains the performance level of the Best-of-10 baseline (10.0x cost) at only 1.5x[1] average cost on the HumanEval benchmark.
- **Implications for On-Device Deployment**: Since all experiments were conducted on Apple Silicon hardware, high token efficiency implies reduced battery consumption and less thermal throttling. These attributes are crucial for practical local deployment of coding agents, enabling highly capable code generation assistants to run on standard laptop hardware without cloud dependency.

### 5.4 Robustness and Regression Analysis

Beyond aggregate accuracy, we analyze whether search introduces regressions on solved instances. On HumanEval, the static search ($N = 3$) caused slight regression (141 vs. 142 solved), over-complicating simple tasks. However, our conditional budgeting strategy eliminated this by bypassing search for trivial cases, achieving a net gain of +3 with no observed regression. On MBPP, Reason-Code fixed 4 complex logical errors (e.g., Woodall numbers) without breaking any correct baseline solutions, yielding a 100% net positive gain, which are detailed in Appendix D and E

[1] $Cost \approx P_{pass} \cdot 1.0 + (1 - P_{pass}) \cdot (1.0 + N)$. With $P_{pass} = 0.866$ and $N = 3$, the effective cost is $\sim 1.5\times$.

| Method | MBPP | HumanEval |
|---|---|---|
| Independent Filtering | 71.21 | 86.59 |
| Random Tree Search | 73.54 | 87.20 |
| UCB-MCTS | 73.54 | 87.20 |

Table 3: **Mechanism Ablation under Fixed Low Budget.** Pass@1 is reported in percentage points.

### 5.5 Efficiency and Ablation Analysis

We conduct ablation studies on MBPP rather than HumanEval, as MBPP contains a larger proportion of multi-step programming tasks where structured search provides consistent benefits. On HumanEval, excessive search may introduce unnecessary modifications to initially correct solutions, which would confound the experimental analysis.

**Impact of Search Strategy** Table 3 reports the mechanism ablation under the same low-budget setting. On MBPP, both tree-based methods outperform independent filtering, indicating that execution-guided tree search is more effective than independent sampling/filtering at this budget. However, UCB-MCTS and random tree search are tied, and we therefore do not claim a consistent UCB-specific advantage at this budget. On HumanEval, the differences are likewise small.

**Latency Analysis** Reason-Code introduces additional inference overhead due to multi-step search. On MBPP, the average latency is 5.71 seconds, approximately $3.2\times$ that of greedy decoding. Timing decomposition shows that generation dominates end-to-end latency (approximately 93–99%), sandbox execution is a minority component (approximately 1–7%), and controller overhead remains below 0.2%. Latency scales close to linearly with the number of simulations.

Importantly, this cost is not uniformly incurred across all instances. Figure 3 plots the inference latency distribution of adaptive MCTS on MBPP. While the mean latency is 7.3 seconds, the median is only 4.7 seconds. This gap indicates a long-tailed distribution: most problems are solved quickly with minimal overhead, and additional compute is selectively allocated to a small subset of hard cases. Unlike brute-force sampling, which pays a fixed cost for every input, Reason-Code adapts its inference effort to problem difficulty.

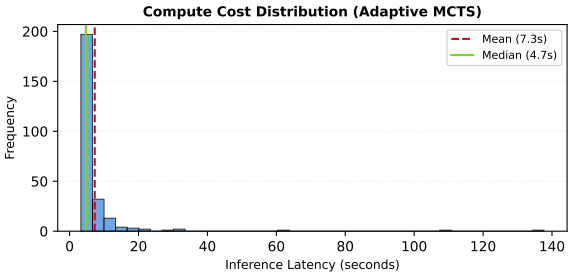


Figure 3: **Latency Profile.** The gap between median (4.7s) and mean (7.3s) shows that additional compute is concentrated on harder tasks.

## 6 Conclusion

Overall, our results support Reason-Code primarily as a practical inference-time reliability controller under deployment constraints. The strongest evidence is for the value of execution-guided adaptive inference relative to independent sampling/filtering baselines. At the small budget studied here, we do not claim a consistent UCB-specific advantage over simpler iterative repair or random tree selection.

## 7 Limitations

While Reason-Code shows strong practical results, it has several limitations:

1. **Test Dependence.** Reliability is bounded by the coverage and informativeness of the available tests. Under sparse or incomplete tests, the binary reward may overestimate correctness, and execution-based evaluation also introduces a non-negligible latency floor.

2. **Limited Empirical Breadth.** Our evaluation is concentrated on MBPP and HumanEval and primarily on one model family. Broader validation across models and more modern benchmarks remains future work.

3. **Search Space Approximation.** Although UCB balances exploration and exploitation, the action space of code generation is effectively unbounded. Our current block-level expansion may miss solutions that require finer-grained edits.

4. **Controller-Specific Gains at Low Budget.** At the small budget studied here, we observe clearer evidence for the value of execution-guided adaptive inference than for a consistent UCB-specific advantage over simpler tree or linear repair strategies.

## Acknowledgments

We thank three anonymous reviewers for their constructive feedback, which helped improve the clarity and evaluation of this work.

## References

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Nolle, Patrick Butler, and 1 others. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.

Boyuan Chen, Mingzhi Zhu, Brendan Dolan-Gavitt, Muhammad Shafique, and Siddharth Garg. 2025. Model cascading for code: A cascaded black-box multi-model framework for cost-efficient code completion with self-testing. *Preprint*, arXiv:2405.15842.

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

Hung Le, Yue Wang, Silvio Gotmare, Akhilesh D and-Savarese, and Steven CH Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025a. CodePRM: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182, Vienna, Austria. Association for Computational Linguistics.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. 2025b. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *Preprint*, arXiv:2409.09584.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Simiao Liu, Yanchen Guo, Zihan Liu, and Lin Song. 2023. Llmops: A survey of foundation model operations. *arXiv preprint arXiv:2312.15286*.

Zhiyi Lyu, Jianguo Huang, Yanchen Deng, Steven Hoi, and Bo An. 2025. Let's revise step-by-step: A unified local search framework for code generation with llms. *Preprint*, arXiv:2411.07434. Preprint. arXiv:2411.07434.

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. *Preprint*, arXiv:2303.17651.

Deepak Nathani, David Wang, Liangming Pan, and William Yang Wang. 2023. Maf: Multi-aspect feedback for improving reasoning in large language models. *Preprint*, arXiv:2310.12426.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Xiaolong Xu, Yanyu Li, Hancheng Li, Weixing Wang, Yujie Fu, Tao Yang, Zhaolong Li, and Ye Yuan. 2024. A survey on edge intelligence: emerging trends, state-of-the-art and future directions. *Artificial Intelligence Review*, 57(5):112.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Xiaoqing Zhang, Yuhan Liu, Flood Sung, Xiuying Chen, Shuo Shang, and Rui Yan. 2025. Thinking before running! efficient code generation with thorough exploration and optimal refinement. *Preprint*, arXiv:2502.17442.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Annual Meeting of the 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. *Preprint*, arXiv:2310.04406.

# A Artifact Availability

All experiments were conducted using local model inference on consumer-grade hardware. We release the full set of evaluation scripts and execution logs used to compute all reported metrics. This enables exact reproduction of the results without re-running model inference.

# B Additional Experimental Analyses

## B.1 Paired Statistics for the Matched-Budget Comparison

We report paired task-level significance for the matched-budget controlled comparison using exact McNemar tests. For HumanEval, all inferential claims use the canonical seed42 controlled rerun. For MBPP, the same seed42 task-level counts are reported; repeated runs with seeds 42/43/44 were identical in pass@1 and token usage and are therefore treated as reproducibility checks rather than independent variance estimates.

The MBPP results show that both Reflexion-only and UCB-MCTS outperform Best-of-$N$ under this budget, while UCB-MCTS and Reflexion-only are statistically indistinguishable. On HumanEval, pairwise differences are small and not statistically significant under the same conservative test.

## B.2 Reproducibility Across Seeds

We reran the matched-budget comparison with seeds 42/43/44. Under the current backend path, outputs were identical across seeds in both pass@1 and total token usage for all reported methods on MBPP and HumanEval. We therefore use these reruns as reproducibility checks rather than variance estimates and base inferential claims on paired task-level statistics from the canonical seed42 run.

## B.3 Mechanism Ablation

To separate the value of execution-guided adaptive inference from the specific choice of UCB selection, we additionally compare Independent Filtering, Random Tree Search, and UCB-MCTS under the same budget.

On MBPP, both tree-based methods outperform Independent Filtering under the same budget (both $p = 0.0313$), while UCB-MCTS and Random Tree Search are tied ($p = 1.0$). On HumanEval, all pairwise differences in this ablation are small and not statistically significant. These results support a narrower claim than the original draft: tree-structured adaptive search is more effective than independent filtering on MBPP, but we do not observe a consistent UCB-specific advantage at this low budget.

## B.4 Test Sensitivity and Timing Decomposition

We evaluate sensitivity to test-suite strength by comparing the full official tests with a deterministic reduced-assert setting (`reduced_half_asserts`). We also report timing decomposition into generation, execution, and controller components.

**Test-suite sensitivity.** Absolute pass rates rise slightly under weaker suites, while relative ordering remains broadly stable. On MBPP, all three methods show the same absolute increase (+1.17), indicating that the sensitivity effect is method-agnostic in that setting.

**Timing decomposition.**

# C System Prompts

We provide the core prompts used in Reason-Code. The following prompt is used to convert execution errors into corrective feedback. The prompt does not modify the reward function and is not tuned for specific benchmarks.

## C.1 Reflexion Prompt

The following prompt is used to guide the model to fix errors based on sandbox feedback:

**Reflexion Prompt Template**

```
The current code failed to pass the tests.
[Current Code]
{code}
[Error Message]
{error_message}
[Instruction]
Fix the bugs in the code above based on the
error message. Return the COMPLETE fixed
function. Do not output explanations. Wrap
the code in “‘python ... “‘.
```

# D Extended Case Studies

This appendix provides detailed case studies of execution-driven correction. We present four scenarios where Reason-Code successfully resolved errors that the baseline ($N = 1$) failed to handle.

## D.1 Case 1: Correcting Mathematical Hallucination (MBPP/20)

**Problem:** Check if a number is a Woodall number ($W_n = n \cdot 2^n - 1$).

| Benchmark | Pair | Δ Pass@1 | $p$-value | A wins | B wins | Ties |
|---|---|---|---|---|---|---|
| MBPP | Reflexion-only vs Best-of-$N$ | +2.72 | 0.0156 | 7 | 0 | 250 |
| MBPP | UCB-MCTS vs Best-of-$N$ | +2.33 | 0.0313 | 6 | 0 | 251 |
| MBPP | UCB-MCTS vs Reflexion-only | -0.39 | 1.0000 | 3 | 4 | 250 |
| HumanEval | Reflexion-only vs Best-of-$N$ | +1.83 | 0.2500 | 3 | 0 | 161 |
| HumanEval | UCB-MCTS vs Best-of-$N$ | +0.61 | 1.0000 | 4 | 3 | 157 |
| HumanEval | UCB-MCTS vs Reflexion-only | -1.22 | 0.6875 | 2 | 4 | 158 |

Table 4: **Paired statistics for the matched-budget controlled comparison.** Δ Pass@1 is reported in percentage points. For each pair, "A wins" and "B wins" count tasks solved only by the first or second method, respectively.

| Benchmark | Pass@1 identical | Tokens identical |
|---|---|---|
| MBPP | Yes | Yes |
| HumanEval | Yes | Yes |

Table 5: **Reproducibility across seeds 42/43/44.** Under the current backend path, matched-budget reruns are bit-identical in task outcomes and total token usage.

| Benchmark | Independent Filtering | Random Tree Search | UCB-MCTS |
|---|---|---|---|
| MBPP | 71.21 | 73.54 | 73.54 |
| HumanEval | 86.59 | 87.20 | 87.20 |

Table 6: **Mechanism ablation under fixed low budget.** Pass@1 in percentage points.

- **Baseline Error:** The model hallucinated a heuristic ("$x + 1$ is a power of 2"), failing for input 23.
- **Reason-Code Fix:** MCTS discarded the heuristic after execution failure and implemented the iterative definition.

**Baseline (Failed)**

```
def is_woodall(n):
    # <Hallucination: Incorrect heuristic>
    # Incorrectly assumes Woodall numbers are related
      to power of 2 check
    if (n + 1) & n == 0:
        return True
    return False
```

**Reason-Code (Fixed)**

```
def is_woodall(n):
    if n <= 0: return False
    i = 1
    while True:
        # <Correction: Strict mathematical definition>
        curr = i * (2 ** i) - 1
        if curr == n:
            return True
        if curr > n:
            return False
        i += 1
```

### D.2 Case 2: Fixing Interface Mismatch (MBPP/419)

**Problem:** Round and sum list elements. Prompt asks to "print" but test expects return.

- **Baseline Error:** Strictly followed "print" instruction, returning `None`.
- **Reason-Code Fix:** Caught `AssertionError` (None != Expected) and switched to `return`.

**Baseline (Failed)**

```
def round_and_sum(list1):
    total = sum(round(i) for i in list1)
    print(total * len(list1))
    # <Error: Printed instead of returned>
```

**Reason-Code (Fixed)**

```
def round_and_sum(list1):
    total = sum(round(i) for i in list1)
    return total * len(list1)
    # <Fixed: Detected output mismatch>
```

### D.3 Case 3: Missing Return Statement (MBPP/71)

**Problem:** Implement Comb Sort.

- **Baseline Error:** The model implemented the logic correctly but failed to include a `return` statement.
- **Reason-Code Fix:** The system appended `return nums` based on the assertion failure feedback.

| Benchmark | Method | Full | Reduced | $\Delta$ | Mean asserts (full) | Mean asserts (reduced) |
|---|---|---|---|---|---|---|
| MBPP | Best-of-$N$ | 71.21 | 72.37 | +1.17 | 3.02 | 2.00 |
| MBPP | Reflexion-only | 73.93 | 75.10 | +1.17 | 3.02 | 2.00 |
| MBPP | UCB-MCTS | 73.54 | 74.71 | +1.17 | 3.02 | 2.00 |
| HumanEval | Best-of-$N$ | 86.59 | 87.20 | +0.61 | 7.20 | 3.89 |
| HumanEval | Reflexion-only | 88.41 | 89.02 | +0.61 | 7.20 | 3.89 |
| HumanEval | UCB-MCTS | 87.20 | 87.20 | +0.00 | 7.20 | 3.89 |

Table 7: **Test-suite sensitivity under reduced assertion coverage.** "Full" and "Reduced" denote pass@1 under the original official tests and deterministic reduced-half-asserts setting, respectively. $\Delta$ is Reduced $-$ Full in percentage points.

| Benchmark | Method | Generation (%) | Execution (%) | Controller (%) | Mean total time (s) |
|---|---|---|---|---|---|
| MBPP | Best-of-$N$ | 93.42 | 6.49 | 0.09 | 5.53 |
| MBPP | Reflexion-only | 93.92 | 5.99 | 0.09 | 3.64 |
| MBPP | UCB-MCTS | 94.57 | 5.33 | 0.11 | 7.92 |
| HumanEval | Best-of-$N$ | 95.87 | 4.04 | 0.09 | 8.99 |
| HumanEval | Reflexion-only | 95.85 | 4.05 | 0.09 | 4.82 |
| HumanEval | UCB-MCTS | 96.18 | 3.72 | 0.10 | 13.03 |

Table 8: **Timing decomposition for the canonical matched-budget runs.** Percentages are computed from mean per-task wall-clock time. Across the reported settings, generation dominates end-to-end latency, sandbox execution is a minority component, and controller overhead remains negligible.

**Baseline (Failed)**

```
def comb_sort(nums):
    gap = len(nums)
    shrink = 1.3
    sorted = False
    while not sorted:
        gap = int(gap / shrink)
        if gap <= 1:
            gap = 1
            sorted = True
        i = 0
        while i + gap < len(nums):
            if nums[i] > nums[i + gap]:
                nums[i], nums[i + gap] = nums[i +
  gap], nums[i]
                sorted = False
            i += 1
    # <Error: Missing return statement>
```

**Reason-Code (Fixed)**

```
def comb_sort(nums):
    # ... (same sorting logic as above) ...
            i += 1
    return nums  # <Fixed by MCTS>
```

### D.4 Case 4: Structural Refactoring (MBPP/428)

**Problem:** Implement Shell Sort.

- **Baseline Error:** The baseline defined a nested helper function. This multi-function structure caused execution scope issues in the sandbox.
- **Reason-Code Fix:** MCTS refactored the code into a flat, self-contained nested loop.

**Baseline (Failed)**

```
def shell_sort(my_list):
    sub = len(my_list) // 2
    while sub > 0:
        for start in range(sub):
            # <Risk: Nested Helper Function Scope
  Issue>
            gap_insertion_sort(my_list, start, sub)
        sub //= 2

# Helper defined outside logic scope which caused
    NameError in strict sandbox
def gap_insertion_sort(list, start, gap):
    for i in range(start + gap, len(list), gap):
        current_value = list[i]
        position = i
        while position >= gap and list[position -
  gap] > current_value:
            list[position] = list[position - gap]
            position = position - gap
        list[position] = current_value
```

**Reason-Code (Fixed)**

```
def shell_sort(my_list):
    n = len(my_list)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = my_list[i]
            j = i
            # <Refactored: Self-contained Logic>
            # NOTE: Long lines will wrap automatically
            while j >= gap and my_list[j - gap] >
  temp:
                my_list[j] = my_list[j - gap]
                j -= gap
            my_list[j] = temp
        gap //= 2
    return my_list
```

## E Additional Qualitative Reproducibility Details

To facilitate reproducibility and further analysis of the "Zero Regression" phenomenon, we provide the full list of MBPP tasks where Reason-Code outperformed the Greedy Baseline.

**Fixed Cases (Net Gain):** Table 9 lists the 4 tasks that were failed by the Baseline ($N = 1$) but solved by Reason-Code ($N = 3$). These cases represent the specific logic gaps bridged by our search algorithm.

Table 9: **Qualitative Analysis of Fixed Cases (MBPP).** Examples of specific error types solved by Reason-Code ($N = 3$) that were failed by the Baseline.

| Task ID | Problem Type | Error Type Fixed |
|---|---|---|
| MBPP/20 | Math (Woodall) | Logic Hallucination |
| MBPP/71 | Sorting (Comb) | Missing Return Stmt |
| MBPP/419 | Math (Rounding) | Interface Mismatch |
| MBPP/428 | Sorting (Shell) | Structural/Scope Issue |

**Potential Regressions (Filtered):** The static MCTS ($N = 3$) configuration induced regressions in four HumanEval tasks: `HumanEval/20`, `95`, `135`, and `142`. These cases represent relatively simple problems where excessive reasoning introduced errors into otherwise correct logic. Crucially, our conditional budgeting strategy successfully identified that the greedy baseline had already satisfied the test requirements, thereby preventing the MCTS process from overriding valid solutions. This validates the necessity of the "gatekeeper" mechanism within our architecture to ensure reliability.

**Zero Regression Verification:** We verify that for all remaining 253 tasks in the MBPP-Sanitized test set, Reason-Code preserved the correct solutions generated by the baseline. Consequently, the set of problems solved by Reason-Code is a strict superset of those solved by the greedy baseline.