Formalizing Test-Time Compute for Function-Level Code Generation

Anonymous ACL submission

Abstract

Test-time compute has emerged as a powerful paradigm in function-level code generation. However, previous proposed strategies have been viewed as disparate, thus lacking a fair apples-to-apples analysis enabling understanding of their operational mechanisms in execution-based benchmarks. Therefore, we present a mathematical framework that unifies generation and reranking with theoretical justifications through the lens of Minimum Bayes Risk (MBR) decoding. Our proposed framework leads to key research questions regarding the effectiveness of using parallel and/or iterative sampling, design choices of reranking signals and soft/hard MBR utility functions, and behaviors of the final selected program across different methods. Our empirical findings highlight the importance of the diversity of sampled candidates (over self-improvement), reranking with simple and high-quality signals, and the effectiveness of test-time compute to select programs that manifest general and edge test case robustness. We will open-source our analysis toolkit and implementation to enable reproducible research.

1 Introduction

Increasing *test-time compute* (TTC) has been shown to be a promising alternative to scaling training compute to further improve the performance of large language models (LLMs) on math-related downstream tasks (Snell et al., 2024). One way to scale TTC is to prompt models to *generate* multiple candidates and *rerank* them, for example, by selecting the one that has the highest *consistency* with other candidates (Bertsch et al., 2023).

However, unlike domains such as mathematical reasoning and machine translation, where exact matching and lexical metrics can be both used for evaluation and reranking (Wang et al., 2023; Fernandes et al., 2022; Farinhas et al., 2023), function-level code generation tasks adopt execution-based evaluation with unit tests to measure the correctness of generated programs. Therefore, TTC practices on this task are different and non-trivial.

Previous works on TTC have tried to utilize reranking signals from likelihood features (Zhang et al., 2023b), trial unit tests provided in the prompt (Shi et al., 2022; Li et al., 2022), and generated unit tests (Chen et al., 2023; To et al., 2024), with Chen et al. (2024) focusing on improving the quality of candidates reranked through self-improvement. However, despite the improving performance on existing benchmarks, the operational mechanism of scaling TTC using unit tests is poorly understood.

We attribute this to two reasons. Firstly, there is a strong inconsistency behind the mathematical formulation of all these methods, with each of them defining the problem independently, making understanding TTC prohibitive. Additionally, due to the lack of unified experimental assumptions, e.g., whether there is access to inputs to the unit test for evaluation, analysis on the decisive decision choices for better quality is missing.

The lack of mathematical and empirical formalizations thus calls for a unified framework to better analyze TTC on function-level code generation. In this paper, we present such a framework (See Figure 1) with empirical findings. Our contributions are listed as follows:

- Mathematically, we unify TTC for function-level code generation. We put forward candidate generation as the basis, with an emphasis on the reranking stage. Importantly, we prove that previous reranking works with unit testing can be mathematically justified through the lens of Minimum Bayes Risk (MBR) decoding, manifesting that the key differences between methods lie in some decision choices in reranking/utility functions.

- We design experiments under two contexts: the user has access to private unit tests or can

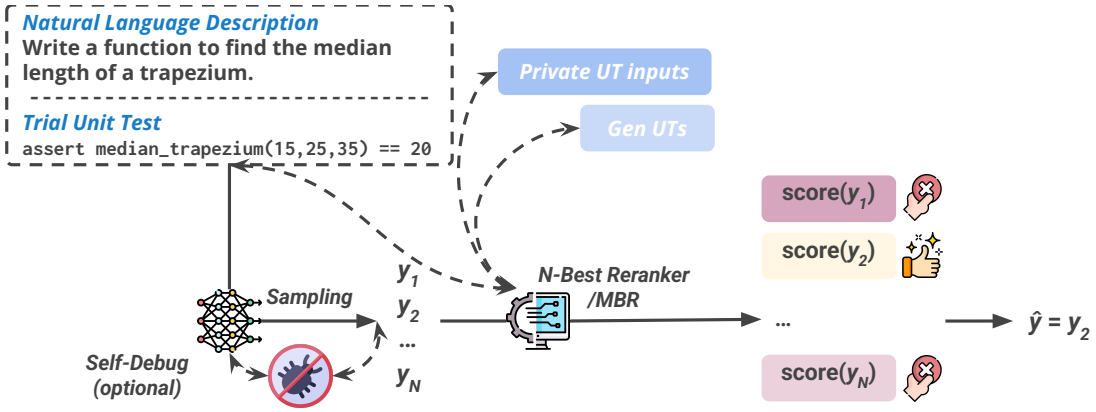


Figure 1: Our proposed framework. Given a natural language description and trial unit tests, a generative model performs parallel sampling and iterative sampling (as self-debug). Generated candidates are then reranked with signals from trial, private, and generated unit tests according to different contexts, before a final solution is selected.

only use generated unit tests. Following our mathematical framework, we analyze the effect of different decision choices in different contexts.

- We observe that candidate diversity should be emphasized more than self-improvement in the generative stage, and the optimal decision choices vary under different contexts. That being said, simple high-quality reranking signals like trial unit tests are always preferred. Through behavioral analysis, we observe that the selected candidates with the best decision choices are general and edge case robust.

- We provide further analysis by scaling unit testing and combining iterative sampling with reranking. Results support our findings on reranking decision choices and emphasis on parallel over iterative sampling.

We hope our findings provide practical insights for implementing test-time compute strategies in code generation tasks, particularly the importance of matching unit test behavior types and using soft reranking methods when leveraging automatically generated unit tests. We will open-source our analysis toolkit and implementation to enable future principled research.

2 Formalizing Test-Time Compute for Function-Level Code Generation

In this section, we provide a formal definition of test-time compute in function-level code generation. We first introduce the sampling and evaluation of the task. We then provide a formulation from the perspective of MBR decoding of reranking of generated candidates.

2.1 Evaluating and Sampling for Function-Level Code Generation

Evaluation. We evaluate a generated program y using a set of private unit tests $T_{\text{private}} = \{(i_1, o_1), \ldots, (i_M, o_M)\}$ that remain inaccessible to the LLM. Specifically, a generated program is considered correct if and only if it satisfies all private tests, i.e., $\forall (i, o) \in T_{\text{private}}, y(i) = o$. Following Chen et al. (2021), we measure performance using *Pass@N*, defined as the probability that at least one of N generated programs successfully passes all unit tests.

Sampling from Code LLMs. A code generation LLM defines a conditional probability distribution $p_{\theta_{\text{LLM}}}(y \mid x)$ over possible programs, conditioned on input x. Typically, x includes a natural language description nl and a set of trial unit tests T_{trial}.[1] Sampling usually takes two forms:

- **Parallel Sampling.** The most common method is ancestral sampling that generates N programs $\mathcal{Y} = \{y_1, \ldots, y_N\}$ independently by controlling temperature (Dabre and Fujita, 2021) and top-p (Holtzman et al., 2020).

- **Iterative Sampling.** This is a broader term in general. However, we refer to iterative sampling as self-debug (Chen et al., 2024) in this paper, where a generated program is iteratively fed into the same LLM, along with

[1] Trial unit tests may overlap with private tests; however, the private tests should never be entirely contained within the trial tests.

Method	$f(y)$
T_{trial} Filtering (FT)	$\prod_{(i,o)\in T_{\text{trial}}} \mathbb{1}\{y(i) = o\}$
Exec. Filtering (FE)	$\prod_{(i,o)\in I'_{\text{private}}} \mathbb{1}\{y(i) \neq \texttt{NULL}\}$
Test Scoring (TS)	$\frac{1}{\lvert T_{\text{gen}}\rvert} \sum_{(i,o)\in T_{\text{gen}}} \mathbb{1}\{y(i) = o\}$

Table 1: Summary of n-best reranking scoring functions. Note that NULL means the program cannot yield an output given a test input.

execution feedback running this program on T_{trial}. Iterative sampling stops once the generated program has passed T_{trial} or a maximum number of turns has been reached.

Note that these two types of sampling can be combined, with multiple generated programs being resubmitted multiple times.

2.2 Rerank with Unit Tests

With N sampled candidates, choosing the best candidate is crucial after introducing the variation of contexts. We now introduce a clear unification of reranking strategies using n-best Reranking and MBR Reranking. Importantly, we provide theoretical justifications to show that previous works lies in either one of these strategies or a combination of these, as their claimed innovations are no more than decision choices from our formalization of n-Best/MBR reranking.

2.2.1 Access to Unit Tests

For function-level code generation, a set of examples as trial unit tests is usually provided by annotators to help both annotators and models understand the program (Chen et al., 2021; Austin et al., 2021; Jain et al., 2024). However, there are two different cases depending on the context.

In some cases of online assessment, interviewees have access to some of the input from private tests $I_{\text{private}} = \{i_o, \ldots, i_K\}, K \leq M$ to help them verify the validity of the written program (Chen et al., 2024). In other cases, reranking requires models to generate unit tests T_{gen} (Li et al., 2022; Chen et al., 2023; To et al., 2024). Note that there are also strategies utilizing inputs of trial unit tests (Shi et al., 2022; Zhang et al., 2023b), which we do not include in this paper as they provide weaker reranking signals compared to simply filtering with the entire trial unit tests.

2.2.2 *n*-Best Reranking with Unit Tests

In its simplest form, n-best reranking selects the candidate maximizing a scoring function f:

Method	$U(y, y')$
MBR-i-H	$\prod_{(i,o)\in T} \mathbb{1}\{y(i) = y'(i)\}$
MBR-i-S	$\frac{1}{\lvert T\rvert} \sum_{(i,o)\in T} \mathbb{1}\{y(i) = y'(i)\}$
MBR-io-H	$\prod_{(i,o)\in T_{\text{gen}}} \mathbb{1}\{y(i) = o\} \cdot \mathbb{1}\{y'(i) = o\}$
MBR-io-S	$\frac{1}{\lvert T_{\text{gen}}\rvert} \sum_{(i,o)\in T_{\text{gen}}} \mathbb{1}\{y(i) = o\} \cdot \mathbb{1}\{y'(i) = o\}$

Table 2: Choice of utility functions and test inputs. "i" and "io" refer to using only test inputs and using the entire input-output pairs. "H" and "S" refer to hard and soft utilities, respectively. Note that MBR-io can only be applied with generated test cases, obviously.

$$\hat{y} = \operatorname*{argmax}_{y\in\mathcal{Y}} f(y). \tag{1}$$

Table 1 summarizes reranking strategies defined by their respective scoring functions. Note that executability can only be done with access to I'_{private}.

2.2.3 Formalization of MBR Decoding

While MBR decoding has been formalized in many generation tasks, including machine translation (Fernandes et al., 2022; Farinhas et al., 2023) and mathematical reasoning (Wang et al., 2023), the formalization in code generation is not clear. Specifically, the only effort we are aware of is Shi et al. (2022), which has not been experimented on with inputs of T_{trial}.

With a utility function $U(y^\star, y)$ measuring the *similarity* between a candidate y and some *reference* (correct) code $y^\star$, MBR selects the candidate in $\mathcal{Y}$ that maximizes the *expected* utility considering all candidates as possible references:

$$\hat{y} = \operatorname*{argmax}_{y'\in\mathcal{Y}} \frac{1}{N} \sum_{y\in\mathcal{Y}} U(y, y'). \tag{2}$$

Choice of Utility Functions and Unit Testing. We categorize utility functions based on two dimensions: soft/hard, and input-only/input-output pairs. Table 2 summarizes the utility functions of these decision choices. Note that most previous works utilizes MBR-i-H (Shi et al., 2022; Li et al., 2022; Chen et al., 2024) except for CodeT (Chen et al., 2023) and SRank (To et al., 2024). We now show that these two methods can also be recovered as MBR decoding.

CodeT (Chen et al., 2023) is MBR-io-H. Chen et al. (2023) introduced a clustering method to form different "consensus sets" of the generated programs that pass the same set of generated unit tests. Concretely, a consensus set $S = \{(y, (i, o))|y \in$

Works	Context	FT	FE	Iter. Sampling
Shi et al. (2022)	Inputs of T_{trial}	✗	✓	✗
Li et al. (2022)	T_{gen}	✓	✗	✗
Chen et al. (2023)	T_{gen}	✗	✗	✗
To et al. (2024)	T_{gen}	✗	✗	✗
Chen et al. (2024)	I'_{private}	✓	✓	on single candidate
Ours	T_{gen} and I'_{private}	✓	✓	✓

Table 3: Unification for analysis in our work compared with experimental settings from previous works.

$S_y, (i, o) \in S_{T_{\text{gen}}}$, where $\forall y \in S_y, (i, o) \in S_{T_{\text{gen}}}, y(i) = o$, which is equivalent to MBR-io-H combined with generated test scoring. We leave the proof to Appendix A.1.

SRank (To et al., 2024) is MBR-i-H. According to To et al. (2024), after generated programs are clustered by output agreement into K clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$ given inputs of generated tests. By defining an interaction matrix $I \in \mathbb{R}^{K \times K}$ and a cluster feature $V \in \mathbb{R}^{K \times 1}$. This is equivalent to MBR-i-S. We leave the proof to Appendix A.2.

3 Experiments

3.1 Systematic Analysis

Table 3 illustrates the experimental design of previous works and our analysis. We are mainly motivated to combine disparate contexts, i.e., whether having access to private test inputs. Notably, we also find that most previous works omitted simple but effective reranking features like FT, which we are motivated to include in our analysis as well.

3.2 Datasets and Models

We conduct experiments using three widely recognized execution-based datasets: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCodeBench (Jain et al., 2024). Concretely, we utilize EvalPlus (Liu et al., 2023), which includes more than 35 times more private unit tests than the original benchmark, enabling our study of reranking using both generated unit tests and private test inputs. See Appendix B for further details.

We generate candidates with the CodeLlama-{7B,13B,34}-Instruct (Rozière et al., 2023) and DeepSeekCoder-{6.7B,V2-Lite,33B}-Instruct (Guo et al., 2024; DeepSeek-AI et al., 2024). We use instruction-tuned models that enable simple processing of generated codes for both candidate generation and iterative sampling (self-debug).

3.3 Generation

We generate 5 to 50 candidates for each task. We first explore sampling temperature by varying it between 0.2 and 2.0 with p-nucleus sampling p 0.95, before deciding what sampling temperature we use for subsequent experiments. With MBPP-S and LiveCodeBench, we vary the temperature between 0.2 and 1.8. When generating multiple candidates, we use the open-source vLLM (Kwon et al., 2023) for fast inference. To answer research questions related to generation and reranking, we present the results from CodeLlama-7B-Instruct. The final temperature selection is based on the best performance with FT-only to ensure fairness.

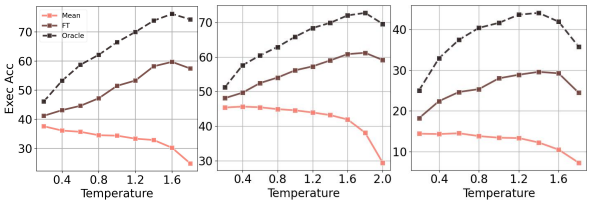
For iteratively sampling, i.e., self-debug, we consider the simple setting proposed by Chen et al. (2024), using only unit test (UT) feedback, i.e., the feedback obtained from execution when a generated candidate is tested on trial unit tests. We only consider this simple setting because 1) our main focus is to test generally the impact of self-debug on improving the oracle's performance, 2) it does not require steps of generation using the LLM other than self-debug, thus requires less computation at test-time and 3) it is the feedback that gives the largest gain to execution accuracy post-debugging according to Chen et al. (2024). We perform 3 rounds of self-debug for candidates generated with sampling because Chen et al. (2024) demonstrated that most debugging can be finished in 3 rounds. In each round, the LLM debugs the candidates generated in the last round.

3.4 Reranking

For a fair comparison across contexts, we use either 100 generated tests or 20 private test inputs for most experiments. For experiments on HumanEval, we use 200 generated tests or 50 private test inputs.

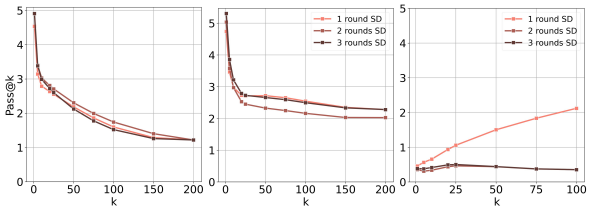
For n-best reranking, we consider two types of filtering, i.e., filtering on trial unit tests and filtering executability on private unit tests. For MBR decoding, we considered all utility functions we present, providing a more panoramic view of reranking with generated or private tests. Note that we do not independently test generated test scoring as it is integrated into the original implementation of CodeT (Chen et al., 2023).

Note that we only include execution-based signals here. We also include reranking with non-execution-based signals in Appendix C.3.



(a) HumanEval+ (b) MBPP-S+ (c) LiveCodeBench

Figure 2: Performance of reranking and oracle over sampling temperatures using CodeLlama-7B-Instruct with 50 generated candidates over 4 runs.



(a) HumanEval+ (b) MBPP-S+ (c) LiveCodeBench

Figure 3: Improvement in Pass@k of CodeLlama-7B-Instruct after iterative sampling (SD) compared to no iterative sampling applied.

3.5 Classifying Behaviors of Unit Testing

A unit test, generated or written down by experts, tests a specific behavior of a generated program. Therefore, inspired by Langr et al. (2015), we develop a unit test profiling dataset that classifies unit tests into one of the following categories: **General case**, which tests general behaviors of the generated code **Cardinality edge case**, which tests if a program can handle an input of length or size that is either zero or one; **Extreme edge case**, which tests if a program can handle inputs that require significantly more memory or time compared to normal cases; **Other edge cases,** which test if a program can handle other types of edge cases, dependent on the task itself.

In practice, we ask DeepSeek-V3 to generate the profiling functions of different tasks, followed by refinement from three experts with at least 8 years of software engineering experience. Note that we ask experts to discuss with each other before making a final decision.

With this dataset, we analyze behaviors of different types of unit test cases and the performance of reranked programs on these unit tests.

4 Results

In this section, we provide results of our experiments and discuss the operational mechanism behind test-time compute. We propose research questions (RQs) with respect to generation, reranking, and behaviors that reranked programs manifest. For results on more experiments, see Appendix C.

	HE+	MBPP-S+	LCB
Random	30.2	41.9	13.3
Greedy	39.0	44.8	13.4
Oracle	76.2	72.0	44.1
FT	59.7	60.8	29.6
Reranking w/ T_{gen}			
MBR-i-S	46.0	48.7	20.8
MBR-i-H	36.1	46.8	18.4
MBR-io-S	37.0	44.9	15.7
MBR-io-H	41.6	48.8	19.3
MBR-io-S + TS	34.9	47.3	16.6
MBR-io-H + TS	43.3	49.4	19.6
FT + MBR-i-S	**64.2**	**61.6**	**32.0**
FT + MBR-i-H	46.8	54.8	27.0
FT + MBR-io-S	54.6	56.2	29.3
FT + MBR-io-H	58.1	58.9	29.3
FT + MBR-io-S + TS	56.6	58.6	27.7
FT + MBR-io-H + TS	58.7	58.9	29.9
Reranking w/ I'_{private}			
FE	69.8	68.3	14.8
MBR-i-S	34.5	43.1	19.4
MBR-i-H	69.5	68.1	21.6
FT + MBR-i-S	52.4	55.2	30.4
FT + MBR-i-H	71.8	69.4	33.0
FT + FE	71.0	**70.0**	31.6
FE + MBR-i-S	67.5	65.9	23.0
FE + MBR-i-H	70.9	68.6	23.7
FT + FE + MBR-i-S	71.8	69.4	32.3
FT + FE + MBR-i-H	**72.0**	69.7	**33.2**

Table 4: Comparison of reranking methods on HumanEval (HE)+, MBPP-S+, and LiveCodeBench (LCB). Some results are centered as they do not use either generated or private unit tests. We highlight **best** and second best reranking results.

4.1 RQ1: How should we configure sampling for better candidates?

We answer this question by focusing on studying sampling temperature for parallel sampling and the number of debugging rounds for iterative sampling.

Selecting high temperature is helpful, and reranking performance with FT is indicative. We show results in Figure 2, finding that sampling with a previously unseen high gives a higher oracle of reranking performance, peaking between 1.6 and 1.8 for HumanEval+ and MBPP+, and between 1.2 and 1.4 for LiveCodeBench. This suggests that sampling with a lower temperature is suggested when applied to tasks that are more difficult, e.g., competitive programming, when compared to basic programming. However, sampling in practice

	Avg	MBR-i-S-Gen Base	MBR-i-S-Gen +FT	MBR-i-H-P + FE Base	MBR-i-H-P + FE +FT
		HumanEval+			
Gen.	37.2	52.2	69.6	76.1	76.7
		+40.2%	+87.5%	+104.4%	+106.1%
Ext.	52.6	65.7	77.1	81.9	81.9
		+25.0%	+46.7%	+55.8%	+55.8%
Card.	58.3	71.1	82.6	87.2	87.2
		+22.0%	+41.5%	+49.6%	+49.6%
Other	40.1	54.3	71.5	80.0	80.1
		+35.3%	+78.2%	+99.7%	+99.7%
		MBPP+			
Gen.	44.8	52.7	68.2	71.1	72.2
		+17.7%	+52.3%	+58.7%	+61.2%
Ext.	51.7	61.5	68.8	73.6	73.6
		+19.1%	+33.0%	+42.3%	+42.3%
Card.	63.6	73.1	80.1	84.3	84.3
		+14.8%	+25.9%	+32.4%	+32.4%
Other	55.4	64.5	74.2	78.7	79.0
		+16.5%	+34.0%	+42.1%	+42.7%

Table 5: Pass rates (*pass@1*) by category using the best methods in Section 4.2. The improvement percentages are highlighted in green.

generally can be done with a higher temperature than previously suggested (Chen et al., 2021; Li et al., 2022; Shi et al., 2022; Liu et al., 2023; Jain et al., 2024).

Moreover, we observe substantial differences in oracle and FT performances when different temperatures are selected for generation, highlighting the importance of controlling this hyperparameter. Last but not least, we find that selecting temperature with FT performances is indicative, as it closely follows changes of oracle performance[2].

Iterative sampling improves oracle performance of reranking, and one single round is enough. According to Figure 3, iterative sampling helps improve the oracle of reranking, represented by the improvement on *Pass@k* across different *k*s. However, later rounds of iterative sampling do not show substantial improvement in the oracle compared to the first round. Moreover, iterative sampling with one single round LiveCodeBench shows larger *Pass@k* improvement with larger *k*s. A sensible guess for this trend is related to the higher level of difficulty of problems in LiveCodeBench, with iterative sampling regenerating a few sensible programs on problems that originally had no sensible candidates. The disappearance of this improvement suggests that, since most code LLMs have not been trained extensively on debugging objectives, iterative sampling regenerates candidates close to greedy decoding.

[2]For later experiments, we choose the number of generated candidates to be 50. We adopt sampling temperature 1.6 for HumanEval and MBPP-S and 1.2 for LiveCodeBench, with nucleus p 0.95. For DeepSeek models, we use temperature 1.2, as we observe that with temperature >1.2 on DeepSeek models, sampling is more likely to generate token indices that are not defined in the vocabulary, see https://github.com/vllm-project/vllm/pull/3685.

4.2 RQ2: What are the best decision choices under different contexts?

We now try to provide an apples-to-apples comparison between different reranking methods utilizing information from executing unit tests.

Soft and hard MBR utility functions have different best cases. According to Table 4, the performance using soft versus hard utility functions varies in different cases. In the case of using only test inputs for reranking, MBR-i-S outperforms MBR-i-Hard when these test cases are generated once. In contrast, MBR-i-H leads to higher Pass@1 when high-quality test cases are provided. This finding is contrary to the claims from Kumar and Byrne (2004), implicating the difference in nature between function-level code generation and other tasks. We attribute this to the nature of this task, where evaluation is performed by executing a generated program on private unit test cases. Our findings are different from Shi et al. (2022), who claimed little difference between MBR-i-S and MBR-i-H. Our findings also illustrated that reaching best performance with SRank (To et al., 2024) is no more than utilizing a soft utility function.

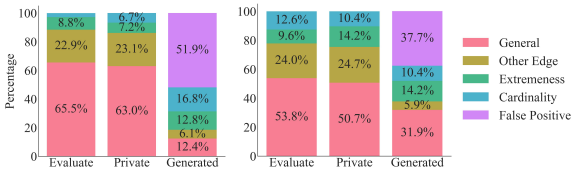
Better programs are selected with simpler signals. First, we find that using only generated unit test inputs, when applied with soft utility functions, outperforms using both generated inputs and outputs, supporting the hypothesis that generating more tokens as test outputs induces more hallucination. This being said, when high-quality unit test inputs are provided, simply applying FE and MBR with a hard utility function is even a better choice. The same applies to FT, which already outperforms all previously proposed methods that lie in the domain of MBR decoding.

4.3 RQ3: What behavior improvements do selected unit tests manifest?

Even though we experiment with as many as 5 times the generated unit tests, the performance of MBR-i with them is still short of that of one using fewer private test inputs. Figure 5 further showcases the pass rate (as **pass@1**) of reranking methods on different types of unit tests. We only

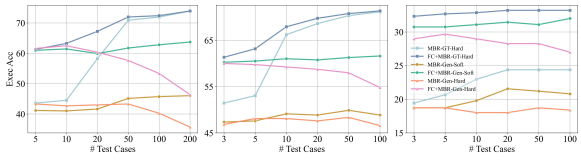
Model	HumanEval+ MBR	HumanEval+ SD-1	HumanEval+ SD-Multi	MBPP-S+ MBR	MBPP-S+ SD-1	MBPP-S+ SD-Multi	LiveCodeBench MBR	LiveCodeBench SD-1	LiveCodeBench SD-Multi
∼7B Scale									
CL-7B-Ins (P)	69.5	69.5	**70.5**	68.1	68.8	**70.4**	21.6	21.3	**24.0**
+ FT + FE	72.0	71.1	**73.3**	69.7	69.2	**72.1**	33.2	33.0	**34.0**
(Gen)	46.0	46.0	**46.6**	48.7	49.0	**50.6**	20.8	20.3	**21.2**
(Gen) + FT + FE	64.2	64.2	**65.3**	61.6	62.0	**63.5**	20.8	20.4	**22.0**
DS-6.7B-Ins (P)	85.5	86.0	**87.0**	78.6	78.8	**79.3**	30.6	30.6	**31.3**
+ FT + FE	**90.7**	90.4	90.4	82.1	81.7	**82.8**	41.9	41.2	**42.0**
(Gen)	83.5	84.0	**85.2**	75.2	75.3	**75.6**	29.3	29.1	**29.9**
(Gen) + FT + FE	**88.0**	87.0	87.0	79.2	79.3	**81.0**	40.8	39.9	**41.1**
∼13B Scale									
CL-13B-Ins (P)	72.6	73.1	**74.4**	71.2	71.0	**72.4**	27.0	**27.6**	**27.6**
+ FT + FE	76.1	76.5	**77.6**	74.7	74.2	**76.2**	40.6	40.8	**46.8**
∼16B Scale									
DS-V2-Lite-Ins (P)	83.5	83.4	**85.1**	80.1	81.0	**81.4**	45.8	45.4	**46.6**
+ FT	**89.6**	88.4	**89.6**	81.8	82.2	**82.8**	**58.7**	58.3	58.5
∼33B Scale									
CL-34B-Ins (P)	74.1	74.5	**75.9**	72.6	72.6	**74.5**	27.5	27.9	**28.3**
+ FT	77.1	77.8	**78.0**	75.2	76.2	**76.8**	44.0	44.0	**48.1**
DS-33B-Ins (P)	84.8	83.9	**86.8**	80.3	82.0	**82.2**	42.8	42.6	**42.9**
+ FT	90.2	89.0	**90.8**	81.7	83.0	**83.5**	**55.9**	55.1	55.6

Table 6: Comparison of self-debugging methods with 50 candidates generated by CodeLlama-{7,13,34}B-Instruct and DeepSeekCoder-{6.7B,V2-Lite,33B}-Instruct, and debugged over {1, **Multi**} candidates. We also provide the upper bound after debugging. Results are averaged across 2 runs for LiveCodeBench and 4 runs for the rest.



(a) HumanEval+ (b) MBPP-S+

Figure 4: Percentage of different test cases. "Evaluate" refers to test cases used for evaluation, and "Private" refers to private test cases used for reranking by subsampling from "Evaluate" ones.



(a) HumanEval+ (b) MBPP-S+ (c) LCB

Figure 5: Scaling unit test cases for reranking.

show results applied to HumanEval+ and MBPP+ since only these two benchmarks provide canonical solutions for all problems.

MBR-i-S improves reranking performance on all types of unit tests. According to Figure 5, MBR-i-S works not only because it can improve performance in general, but also because it helps select candidates that can pass general test cases and different types of edge cases. FT further boosts this reranking, showcasing robustness over different types of test cases used for evaluation.

Improvement on the reranking methods related to the types of the unit test distribution. According to Figure 4, private test cases for reranking show high similarity to the ones for evaluation, especially in the category of general tests and task-related edge tests. This is opposite to the distribution of generated tests, with false-positive test cases replacing a large portion of general tests and task-related edge tests detected with Evalplus's false-positive classifier (Liu et al., 2023).

5 Further Analysis

5.1 Scaling Unit Test Numbers for Reranking

Following our findings in Section 4.2, we now analyze the effect of reranking performances by scaling unit tests.

When it comes to generated unit tests, improvements of scaling only happen when it's done with MBR-i-S instead of an exact match. According to Figure 5, different trends happen when applying exact matches versus MBR-i-S. For MBR-i-S, we see an increasing trend with more test cases used for reranking, while for MBR-i-H, we see the opposite. This finding further indicates that adopting soft utility functions when using generated unit tests is helpful, challenging existing practices using hard utility functions (Li et al., 2022; Shi et al.,

2022; Ni et al., 2023; Chen et al., 2024).

5.2 Combining Both Sampling Strategies with Reranking

Applying MBR-i-H on private test inputs with FT gives close-to-oracle performances, suggesting that improving the oracle performance with iterative sampling (Chen et al., 2024) is a more sensible choice than providing better reranking methods. In this section, we first analyze the improvement in Oracle performance. We then compare our proposed SD-Multi and SD-1 proposed by Chen et al. (2024). For SD-Multi, we only use results with one round of debugging, while we use results with 3 rounds of debugging for SD-1. When using generated test cases, we experiment with 7B models.

SD-Multi outperforms SD-1. According to Table 6, SD-Multi consistently outperforms SD-1 in most experiments, with better-performing models having smaller margins of improvement with iterative sampling. The only exceptions we find are on experiments with candidates generated and self-improved with DeepSeekCoder-{6.7B, V2-Lite}-Instruct, where either the task is considered more difficult compared to basic programming, or the baseline performance with MBR-i-H on private test inputs is already above 90.

6 Related Work

Reranking in function-level code generation. Shi et al. (2022) and Li et al. (2022) proposed MBR decoding using agreement on execution outputs. Chen et al. (2023) extended the framework utilizing generated input-output pairs. Huang et al. (2024) utilized formal verifications as part of reranking. Ma et al. (2025) further trained test case generators to help reranking. Li et al. (2025) utilized an execution agreement assisted by a stronger LLM. Our work differs from these works as our focus lies in the formalization of TTC instead of proposing methods. Specifically, while Ma et al. (2025) studies unit test generation, their generation focuses on the "unittest" Python class, making it difficult to study from our MBR decoding framework with nuanced decision choices. Additionally, we do not include Li et al. (2025) into our MBR framework as we do not think utilizing a stronger LLM leads to a fair comparison in our experimental settings.

Reranking on Other Execution-Dependent Tasks. One related domain is Text-to-SQL generation, with Gao et al. (2024) including self-consistency (Wang et al., 2023) into studying TTC performances. Ehrlich et al. (2025) employed TTC on repository-level software engineering tasks. While all these tasks are related, we do not perform analysis on these tasks due to their nature. Text-to-SQL generation can be explicitly reranked through exact-matching without requesting any test inputs, and repository-level tasks focus on passing a new set of unit tests without sacrificing current ones.

Generation strategies. Zhang et al. (2023a) proposed Monte Carlo Tree Search on code generation. Wang et al. (2025) studied concept planning based on LLMs before generation. Zheng et al. (2025) then incorporates multi-turn code generation with execution feedback into model training. Our work differs from these efforts as we only treat generation as a basis for TTC. While these efforts do manifest impressive performances, we do not think they will lead to many changes in our findings, in which decision choices of reranking are a key part.

7 Conclusion and Future Work

We propose a formalization of test-time compute for functional code generation. Mathematically, we unify previous approaches through a generation-reranking framework and provide theoretical justifications of our reranking module, recovering previous works. Empirically, we ask key research questions with respect to generation, reranking, and behavior testing for this task. Our empirical findings highlight the importance of configuring generation parameters, reranking with appropriate and high-quality signals, and different types of test cases reranking methods manifest robustness on.

Limitation

First of all, our paper is limited by the scale of experiments as we cannot experiment with all the large language models due to the limit of computation and the vast set of analysis, and our solution is to select representative classes of open-source models and experiments. Additionally, the paper is limited by the inclusion of various sampling strategies while we do not think it will be a game-changing module for us to analyze TTC in our framework with a major focus on reranking stage.

Ethical Considerations

We do not consider the existence of ethical issues related to the paper, due to the nature of code gen-

eration and our usage of publicly available datasets that have been verified. However, we noticed the risk of ethical concern due to our choice of sampling temperature. We checked generations and found no ethical issues in the generated content.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Amanda Bertsch, Alex Xie, Graham Neubig, and Matthew Gormley. 2023. It's MBR all the way down: Modern generation techniques through the lens of minimum Bayes risk. In *Proceedings of the Big Picture Workshop*, pages 108–122, Singapore. 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 *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/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, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Raj Dabre and Atsushi Fujita. 2021. Investigating softmax tempering for training neural machine translation models. In *Proceedings of the 18th Biennial Machine Translation Summit - Volume 1: Research Track, MTSummit 2021 Virtual, August 16-20, 2021*, pages 114–126. Association for Machine Translation in the Americas.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931.

Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. *CoRR*, abs/2301.09043.

Ryan Ehrlich, Bradley C. A. Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. Codemonkeys: Scaling test-time compute for software engineering. *CoRR*, abs/2501.14723.

António Farinhas, José Guilherme Camargo de Souza, and André F. T. Martins. 2023. An empirical study of translation hypothesis ensembling with large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 11956–11970. Association for Computational Linguistics.

Patrick Fernandes, António Farinhas, Ricardo Rei, José Guilherme Camargo de Souza, Perez Ogayo, Graham Neubig, and André F. T. Martins. 2022. Quality-aware decoding for neural machine translation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, pages 1396–1412. Association for Computational Linguistics.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. 2024. Enhancing large language models in coding

through multi-perspective self-consistency. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 1429–1450. Association for Computational Linguistics.

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

Shankar Kumar and William Byrne. 2004. Minimum Bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 169–176, Boston, Massachusetts, USA. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM.

Jeff Langr, Andy Hunt, and Dave Thomas. 2015. *Pragmatic Unit Testing in Java 8 with JUnit*, 1st edition. Pragmatic Bookshelf.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. *CoRR*, abs/2502.14382.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. *CoRR*, abs/2501.01054.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin 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, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 3533–3546. Association for Computational Linguistics.

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

Hung To, Minh Nguyen, and Nghi Bui. 2024. Functional overlap reranking for neural code generation. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 3686–3704. Association for Computational Linguistics.

Evan Z Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean M. 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*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023a. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang.

2023b. Coder reviewer reranking for code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, benjamin negrevergne, and Gabriel Synnaeve. 2025. What makes large language models reason in (multi-turn) code generation? In *The Thirteenth International Conference on Learning Representations*.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 13921–13937. Association for Computational Linguistics.

A Proofing the Equivalence between MBR and other reranking methods

A.1 Proof: Equivalence between CodeT and MBR-io-H

(Chen et al., 2023) defined a consensus set $S = \{(y, (i, o))|y \in S_y, (i, o) \in S_{T_{\text{gen}}}\}$, where $\forall y \in S_y, (i, o) \in S_{T_{\text{gen}}}, y(i) = o$. The final score for a generated program $y \in S_y$ is

$$|S_y| \cdot |S_{T_{\text{gen}}}|. \tag{3}$$

Obviously,

$$|S_y| = \sum_{y' \in \mathcal{Y}} \prod_{(i,o) \in T_{\text{gen}}} \mathbb{1}\{y(i) = o\} \cdot \mathbb{1}\{y'(i) = o\}, \tag{4}$$

and

$$|S_{T_{\text{gen}}}| = \frac{1}{|T_{\text{gen}}|} \sum_{(i,o) \in T_{\text{gen}}} \mathbb{1}\{y(i) = o\}. \tag{5}$$

Therefore, MBR-io-S with the scoring on trial unit tests is recovered.

A.2 Proof: Equivalence between SRank and MBR-io-H

To et al. (2024) defined K clusters $\{C_1, \ldots, C_k\}$ based on exact match of test outputs given M generated test inputs $I_{\textbf{gen}} = \{i_1, \ldots, i_M\}$, i.e. $\forall y, y' \in C_i$,

$$1 = \prod_{i \in I_{\textbf{gen}}} \mathbb{1}(y(i) = y'(i)). \tag{6}$$

The interaction matrix $I \in \mathbb{R}^{K \times K}$ is defined as

$$I_{i,j} = \frac{1}{M} \sum_{m=1}^{M} \mathbb{1}(y(i_m) = y'(i_m)), \tag{7}$$

where $y \in C_i$ and $y' \in C_j$.

To et al. (2024) also defined a cluster feature $V \in \mathbb{R}^{K \times 1}$. Note that we only include the cluster size as the feature, i.e. $V_i = |C_i|$.

After all, the final reranking score is $R = I \cdot V$. Specifically, the score that C_i receive is

$$R_i = I_{i,1} \cdot |C_1| + \cdots + I_{i,1} \cdot |C_K|. \tag{8}$$

Specifically, for $y_i \in C_i$,

$$I_{i,j} \cdot |C_j| = \frac{1}{M} \sum_{y_j \in C_j} \sum_{m=1}^{M} \mathbb{1}(y_i(i_m) = y_j(i_m)) \tag{9}$$

Note that any generated program must belong to a cluster, and any two different clusters are mutually exclusive. Therefore,

$$R_i = \frac{1}{M} \sum_{y_j \in \mathcal{Y}} \sum_{m=1}^{M} \mathbb{1}(y_i(i_m) = y_j(i_m)). \tag{10}$$

Thus, the score for any candidate y is

$$\frac{1}{M} \sum_{y' \in \mathcal{Y}} \sum_{m=1}^{M} \mathbb{1}(y'(i_m) = y'(i_m)). \tag{11}$$

Regarless of the normalization across N generated programs, MBR-i-S is recovered.

B Dataset Statistics

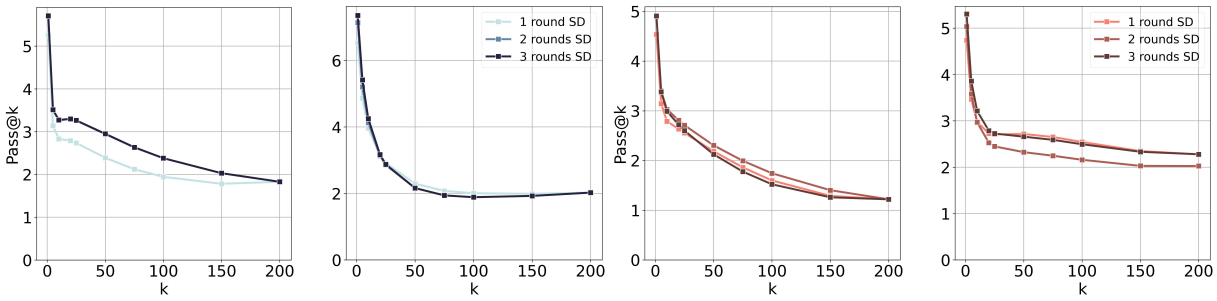
We use the HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) datasets in EvalPlus (Liu et al., 2023), consisting of 164 and 395 problems respectively. For LiveCodeBench (Jain et al., 2024), we use the version that includes competitive programming problems from July 2023 to September 2024 to balance the risk of data contamination and size of the dataset. For fair MBR decoding, we only include problems presented as "functional", obtaining 283 problems, among which 88/135/60 are easy/mid/hard problems.

Note that when prompting the model to generate programs for MBPP, EvalPlus adopts the format that uses the first private unit test as the trial unit test. The average numbers of trial unit tests in HumanEval/MBPP/LiveCodeBench are 2.8/1/2.47.

C Experimental Results

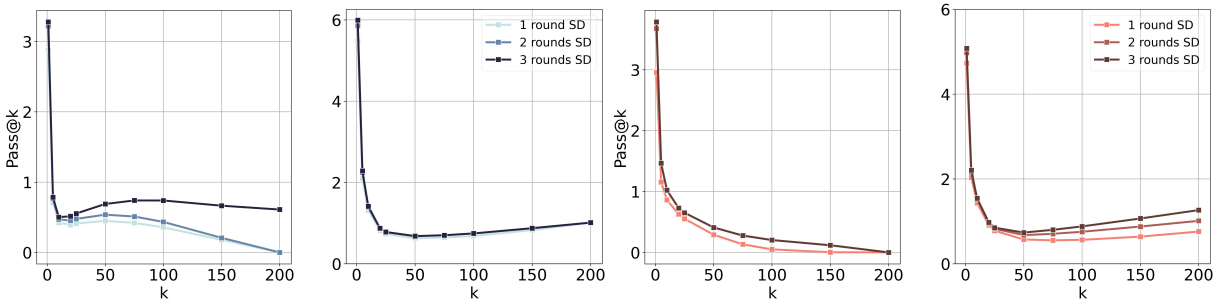
C.1 Candidate Generation

To validate our choice of temperature, we present the choice of sampling temperature of using other models. For CodeLlama-13B-Instruct and DeepSeekCoder-6.7B-Instruct, we compare results with our choice of temperature for further experiments with results using temperature 0.8. Results are presented in Table 7. All models we experiment with allow a sampling temperature over 1, with lower mean execution accuracy but higher oracle performance. Combined with filtering on trial unit tests, MBR-i allows constant improvement in execution accuracy when sampling with higher temperatures, which is not guaranteed without filtering.



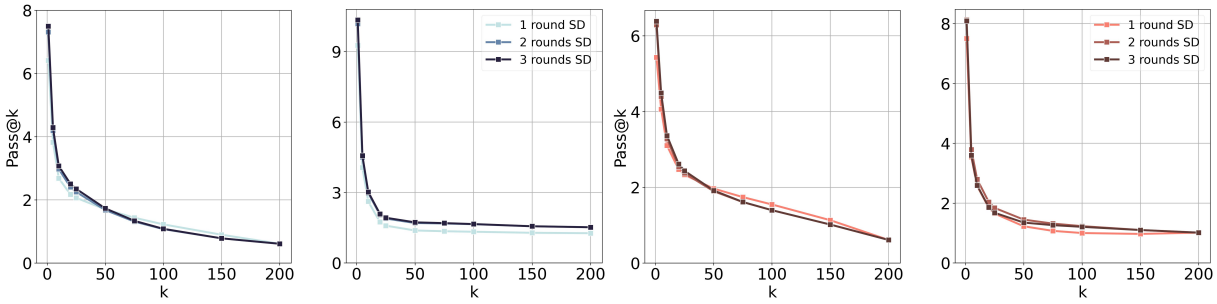
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 6: Improvement in Pass@k of CodeLlama-7B-Instruct after self-debugging compared to no self-debugging applied.



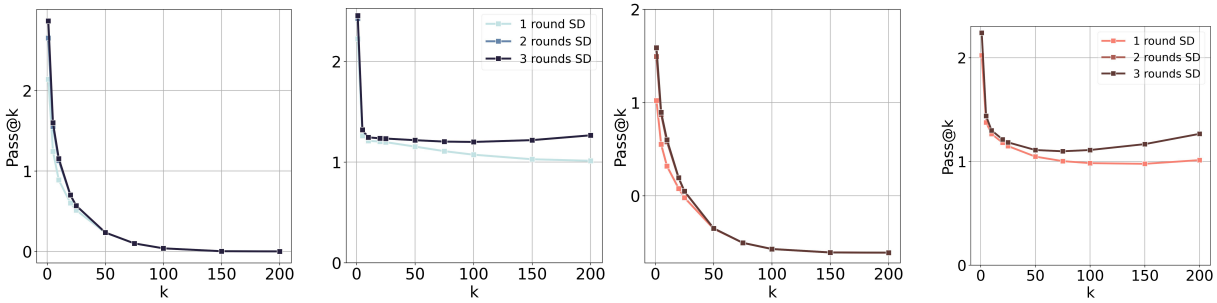
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 7: Improvement in Pass@k of DeepSeekCoder-6.7B-Instruct after self-debugging compared to no self-debugging applied.



(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 8: Improvement in Pass@k of CodeLlama-13B-Instruct after self-debugging compared to no self-debugging applied.

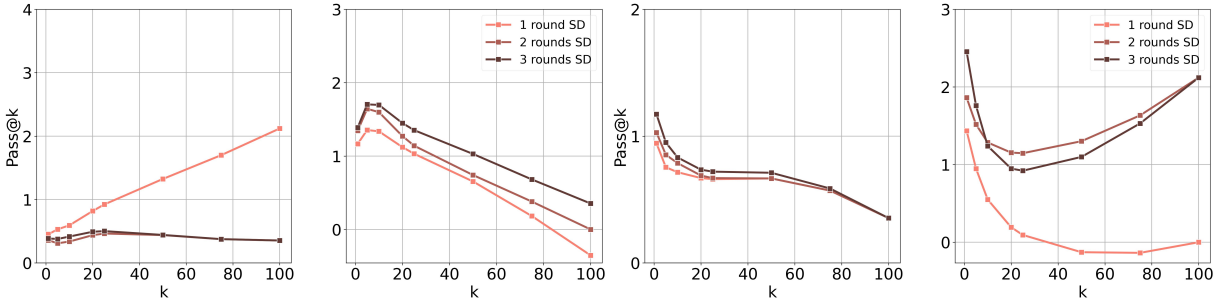


(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 9: Improvement in Pass@k of DeepSeekCoder-V2-Lite-Instruct after self-debugging compared to no self-debugging applied.

Model	DS-6.7B-Instruct temp = 0.8	DS-6.7B-Instruct temp = 1.2	CL-13B-Instruct temp = 0.8	CL-13B-Instruct temp = 1.6 (1.2 for LCB)	DS-V2-Lite-Instruct temp = 0.8	DS-V2-Lite-Instruct temp = 1.2
HumanEval(+)						
Mean	77.3 (70.6)	72.5 -4.8 (65.1 -5.5)	45.7 (40.0)	35.8 -9.9 (31.1 -8.9)	81.7 (76.6)	81.7 -0 (76.6 -0)
MBR	87.0 (85.8)	86.4 -0.6 (85.5 -0.3)	72.0 (67.4)	74.3 +2.3 (72.6 +5.2)	84.8 (81.3)	85.8 +1.0 (83.5 +2.2)
FT + FE + MBR	90.7 (89.8)	91.6 +0.9 (90.7 +0.9)	76.5 (76.1)	80.5 +4.0 (76.1 +6.9)	91.5 (87.2)	92.4 +1.0 (89.6 +2.4)
Oracle	92.9 (91.5)	95.2 +2.3 (93.2 +1.7)	82.8 (75.1)	88.7 +5.9 (82.4 +7.3)	94.5 (89.7)	95.7 +1.2 (92.3 +2.6)
MBPP(+)						
Mean	71.4 (61.8)	68.6 -2.8 (58.6 -3.2)	60.7 (51.0)	52.5 -8.2 (43.5 -7.5)	79.7 (67.2)	79.1 -0.6 (66.8 -0.4)
MBR	83.7 (78.2)	84.0 +0.3 (78.6 +0.4)	71.5 (64.7)	77.9 +6.4 (71.2 +6.5)	88.6 (78.5)	89.1 +0.5 (80.1 +1.6)
FT + FE + MBR	87.3 (80.5)	89.1 +1.8 (82.1 +1.6)	78.0 (67.8)	84.1 +6.1 (74.7 +6.9)	89.7 (79.6)	91.2 +1.5 (81.8 +2.2)
Oracle	91.2 (82.8)	93.2 +2.0 (85.6 +2.8)	82.0 (70.4)	88.4 +6.4 (78.0 +7.4)	91.3 (80.9)	93.1 +1.8 (83.9 +3.0)
LiveCodeBench						
Mean	19.9	17.5 -2.4	17.4	15.9 -1.5	36.1	35.4 -0.7
MBR	30.0	30.6 +0.6	24.9	27.0 +2.1	42.6	45.8 +3.2
FT + FE + MBR	43.1	41.9 -1.2	35.7	40.6 +4.9	57.8	58.7 +0.9
Oracle	52.5	53.6 +1.1	46.8	50.4 +3.6	63.1	68.2 +5.1

Table 7: Comparison of performance of sampling and reranking using temperature 0.8 and those chosen for further experiments. We report mean execution accuracies, MBR-i results, and oracle performances of 50 candidates generated by DeepSeekCoder-{6.7B, V2-Lite}-Instruct or CodeLlama-13B-Instruct. Results that end with + mean that it is evaluated on the plus with extended test cases, otherwise, it is evaluated on the original test cases. We also show decreases and improvements of results in our choice of temperature over 0.8. Results are averaged across 4 runs for HumanEval(+) and MBPP-S(+), and 2 runs for LiveCodeBench.



(a) CL-7B (b) DS-6.7B (c) CL-13B (d) DS-V2-Lite

Figure 10: Improvement in Pass@k on LiveCodeBench.

C.2 Improving Oracle with Iterative Sampling

We first present results of improvement of *Pass@k* that estimates oracle improvement using candidates generated and self-debugged by CodeLlama-7B-Instruct (see Figure 6), DeepSeekCoder-6.7B-Instruct (see Figure 7), CodeLlama-13B-Instruct (see Figure 8), and DeepSeekCoder-V2-Lite-Instruct (see Figure 9). We also provide cases for LiveCodeBench (see Figure 10). Our findings align with Section 5.2 as one round of self-debugging is enough to improve the oracle.

C.3 Reranking with Non-Execution-Based Metrics

Table 8 presents our preliminary results on non-execution-based metrics including Coder-Reviewer (Zhang et al., 2023b), CodeScore (Dong et al., 2023), and CodeBertScore (Zhou et al., 2023). As we observe that they do not perform as ideal as execution-based metrics, we decide to focus on

	HE+	MBPP-S+	LCB
Random	30.2	41.9	13.3
Greedy	39.0	44.8	13.4
Oracle	76.2	72.0	44.1
N-Best Reranking			
LL	38.7	42.6	16.3
CR	40.5	43.1	15.9
CS	30.2	-	14.1
MBR			
MBR-CBS	35.5	45.4	-
MBR-CS	31.9	-	-

Table 8: Comparison of non-execution based reranking methods on HumanEval (HE)+, MBPP-S+, and LiveCodeBench(LCB). For N-Best Reranking, we compare Likelihood (LL), Coder-Reviewer (CR), and CodeScore (CS). For MBR, we compare CodeScore and CodeBertScore (CBS). We highlight **best** and second best reranking results of the class of reranking methods.

execution-based metrics.