DOCE: Finding the Sweet Spot for Execution-Based Code Generation

Anonymous ACL submission

Abstract

A diverse set of decoding, reranking, and self-improvement procedures have been shown effective for code generation with LLMs. However, a unified framework that links and experimentally compares these methods is missing. We address this by proposing **D**ecoding **O**bjectives for **C**ode **E**xecution, a framework with candidate generation, n-best reranking, minimum Bayes risk (MBR) decoding, and self-debug as the core components. We then systematically study the contributions of these components through execution-based evaluation. Our findings suggest the importance of sampling with high temperature, reranking with execution-based methods using only a few high-quality unit test inputs, and robust self-debugging on multiple candidates. Specifically, we highlight filtering based on trial unit tests, a simple and effective strategy that has often been overlooked in prior works.

1 Introduction

Despite impressive performances of large language models (LLMs) in code generation (Chen et al., 2021; Li et al., 2022, 2023; Rozière et al., 2023; Luo et al., 2024; Guo et al., 2024a; Lozhkov et al., 2024; DeepSeek-AI et al., 2024) on execution-based metrics (Hendrycks et al., 2021; Chen et al., 2021; Austin et al., 2021; Lai et al., 2023; Jain et al., 2024; Zhuo et al., 2024) based on exploration of training approaches, predominant methods during inference still heavily rely on techniques such as greedy decoding, beam search, and p-nucleus sampling (Holtzman et al., 2020) followed directly by evaluation on execution accuracy.

An alternative approach involves *sampling* multiple candidates with an LLM and selecting one using n-best reranking (Ng et al., 2019; Bhattacharyya et al., 2021; Cobbe et al., 2021; Shen et al., 2021) or Minimum Bayes Risk (MBR) decoding (Eikema and Aziz, 2020, 2022), with potential improvements on candidates using the same LLM (Madaan et al., 2023). These methods have been systematically studied in domains like machine translation (Eikema and Aziz, 2020; Freitag et al., 2022; Fernandes et al., 2022; Farinhas et al., 2023). However, despite the proposal of reranking methods, based on likelihoods (Zhang et al., 2023), execution agreements (Shi et al., 2022) or predicting execution accuracy (Inala et al., 2022; Ni et al., 2023), and improvement methods simulating the debugging process (Olausson et al., 2024; Chen et al., 2024), a systematic study that helps understand the effects and provide an apples-to-apples comparison of these methods is still missing.

Bridging this gap is crucial to current research. First of all, lately released open-source code LLMs (Rozière et al., 2023; Luo et al., 2024; Guo et al., 2024a; DeepSeek-AI et al., 2024) excelling on a range of tasks lead to a demand for a renewed understanding of the effects of sampling and reranking methods. Example methods include choosing a relatively low sampling temperature (Li et al., 2022; Shi et al., 2022; Huang et al., 2024; Chen et al., 2023), which is originally suggested with LLMs having lower performances on these tasks (Fried et al., 2023; Nijkamp et al., 2023), and filtering on trial unit tests (Li et al., 2022), a method originally proposed to retain sensible programs when a code LLM can hardly generate. Additionally, strong open-source code LLMs also allow a systematic study of sampling, reranking, and candidate improvement with a low budget compared to models requesting API calls (OpenAI, 2023). Last but not least, lately proposed neural metrics for code (Zhang et al., 2023; Zhou et al., 2023; Dong et al., 2023) provide an opportunity to systematically understand the execution component of reranking compared with execution-free approaches.

Therefore, we propose a unified framework for execution-based code generation to improve decoding performance on execution-based metrics. Our contributions are listed below:

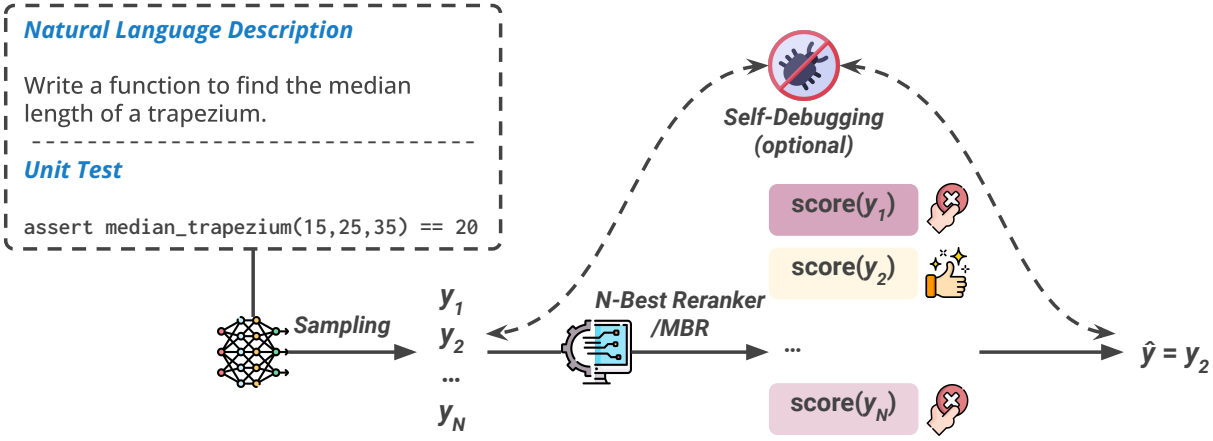


Figure 1: The **D**ecoding **O**bjectives for **C**ode **E**xecution (**DOCE**) Framework. Firstly, multiple candidates are generated through sampling. Each candidate then is assigned a score using an n-best reranker or MBR, before the candidate with the highest score is returned. Self-Debug can be applied to multiple candidates before scoring as we propose, or the highest score candidate as proposed by Chen et al. (2024).

- We propose the **D**ecoding **O**bjectives for **C**ode **E**xecution (**DOCE**) framework for code generation (see Figure 1), including candidate generation, reranking, and self-debug.
- We provide analysis on the number of generated candidates and sampling temperature. Specifically, sampling with previously unseen high temperatures allows high oracle and reranking performance.
- With experiments on reranking, we emphasize the importance of execution-based approaches. Specifically, our analysis reveals the importance of filtering based on trial unit tests, a commonly used technique whose effect has been rarely reported in previous works.
- Execution-based methods lead to close-to-Oracle performance, with further improvements using Self-Debug. Effective and robust Self-Debug only happens when applied to multiple candidates before choosing the final candidate.

2 Candidate Generation, Self-Debugging, and Reranking

2.1 Candidate generation

An LLM-based code generation model defines a probability distribution $p_{\theta_{\text{LLM}}}(y|x)$ over the space of possible programs, conditioned on a description of the task x, where θ_{LLM} refers to learned parameters of the LLM. An x often contains a natural language description; some unit test cases, namely *trial unit tests*, are used to regularize and generate the corresponding code.

Generated code is usually predicted using greedy decoding or p-nucleus sampling (Holtzman et al., 2020). The latter approach allows obtaining diverse generated candidates $\mathcal{Y}$ compared to greedy decoding and is often accompanied by temperature sampling (Dabre and Fujita, 2021). Previous studies (Shi et al., 2022; Chen et al., 2023) found that sampling leads to higher *oracle* performance, denoted by *pass@k*, assuming always choosing the best among k candidates. However, deciding which candidate to pick is still unclear.

2.2 Reranking

We now assume access to $\mathcal{Y}$ containing N generated candidates for a given x, as mentioned in Section 2.1. To provide a final output $\hat{y}$, we explore how to rank and pick candidates for code generation effectively[1].

2.2.1 *n*-Best Reranking.

In its simplest form, n-best reranking uses a single feature f and ranks candidates, with the chosen candidate maximizing this feature:

$$\hat{y} = \underset{y' \in \mathcal{Y}}{\text{argmax}}\, f(y'). \quad (1)$$

Likelihood-based Feature. We can use the likelihood as the feature $f(y') = p_{\theta_{LLM}}(y'|x)$. In practice of code generation, the likelihood is often

[1]With outputs that have the same score, we choose the output with the smallest index.

normalized by sequence length $p_{\theta_{LLM}}(y'|x)^{\frac{1}{len(y')}}$. Alternatively, Zhang et al. (2023) proposed an average of likelihood combined with a "reviewer-likelihood", normalized by sequence length as well:

$$f(y') = (p_{\theta_{LLM}}(y'|x)p_{\theta_{LLM}}(x|I, y'))^{\frac{1}{len(y')}}, \quad (2)$$

where I is the set of the few-shot demonstrations.

Execution-based Feature. Previous works (Li et al., 2022; Chen et al., 2024) have experimented with filtering based on trial unit tests. It filters out outputs that do not pass any single unit test $t(y) \in \{\text{pass,fail}\}$ of the set of trial unit tests T_{trial}:

$$f(y') = \prod_{t \in T_{\text{trial}}} \mathbb{1}_{t(y')=\text{pass}}. \quad (3)$$

Scores from External Models. We can also use a single model θ_{ext}. Specifically, previous efforts include training models using execution success/-failure as signals (Inala et al., 2022; Ni et al., 2023), given the input x and generated candidate y'. During inference, a score $f_{\theta_{ext}}(x, y')$ is given for every generated candidate.

2.2.2 MBR Decoding

While n-best list reranking can rank candidates based on *reference-free* features, it is also possible to rank candidates with *reference-based* metrics using MBR decoding. 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:

$$\hat{y} = \underset{y' \in \mathcal{Y}}{\text{argmax}} \frac{1}{|\mathcal{Y}|} \sum_{y \in \mathcal{Y}} U(y, y'). \quad (4)$$

Execution-based Metrics. If we can feed unit test inputs to candidates, a basic utility function (Shi et al., 2022; Li et al., 2022) is to consider if a candidate matches the outputs of some reference code with the set of unit tests used for evaluation $\mathcal{E}_{\mathcal{T}}$, the utility function is the exact match of outputs between the reference and the candidates

$$U(y, y') = \mathbb{1}_{\mathcal{E}_{\mathcal{T}}(y)=\mathcal{E}_{\mathcal{T}}(y')}. \quad (5)$$

In practice, this utility function reduces the MBR decision to a majority vote for the candidate that produces outputs that agree the most with other candidates' outputs (Wang et al., 2023a).

Note that in Shi et al. (2022) execution-based MBR both on the cases of using only inputs from trial unit tests only and from all unit tests used for evaluation. Since we decide to use these unit tests as filtering, we use all unit tests given but only their inputs, which aligns with previous practices (Shi et al., 2022; Ni et al., 2023; Chen et al., 2024).

External Models for MBR. We can also use an external neural model (Zhou et al., 2023; Dong et al., 2023) used for evaluation θ_{ext} to compute $U_{\theta_{ext}}(x, y, y')$, obtaining the utility by inputting input x and two generated candidates y and y'.

2.3 Self-Debugging

Reranking is oracleed by the best possible candidate's performance. An approach for further quality improvement of generated codes is to *debug* it with the same model used for generation (Chen et al., 2024). Given few-shot demonstrations I, an input x, a candidate y', and its execution feedback with the trial unit tests $exec(y', T_{\text{trial}})$, we obtain a corrected generation y'_d using θ_{LLM}:

$$y'_d = \text{debug}_{\theta_{\text{LLM}}}(I, x, y', exec(y', T_{\text{trial}})). \quad (6)$$

2.4 DOCE Framework

We define **DOCE** framework. Firstly, with only reranking and MBR, we have

$$\hat{y} = \underset{y' \in \mathcal{Y}}{\text{argmax}} \frac{f(y')}{|\mathcal{Y}|} \sum_{y \in \mathcal{Y}} U(y, y'). \quad (7)$$

Self-Debugging with Single Selected Candidate from Reranking. If we only debug selected candidates from reranking, namely **SD-1** (Chen et al., 2024), we have the final code answer $debug_\theta(x, \hat{y}, T_{\text{trial}})$.

Self-Debugging on All Candidates before Reranking. We can also perform self-debugging on *all* generated candidates before reranking, namely **SD-Multi**. We presume this outperforms self-debugging on one candidate with more candidates passing trial unit tests, leading to better candidates to be reranked and possibly better reranking results. The final code answer is $\text{argmax}_{y'_d \in \mathcal{Y}_d} \frac{f(y'_d)}{|\mathcal{Y}_d|} \sum_{y_d \in \mathcal{Y}_d} U(y_d, y'_d)$.

3 Experiments

3.1 Datasets

We first conduct experiments using two widely recognized execution-based datasets: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). We employ the sanitized, higher-quality subset of the MBPP dataset, namely MBPP-S. We utilize EvalPlus (Liu et al., 2023) that includes more than 35 times unit tests than the original benchmark, ensuring rigorous execution-based evaluation. We then extend experiments to competitive programming using LiveCodeBench (Jain et al., 2024), a dataset containing up-to-date competitive programming problems to avoid data contamination. See Appendix A for further details.

3.2 Code LLMs

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

3.3 Candidate 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 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 candidate generation and reranking, we present the results from CodeLlama-7B-Instruct.

3.4 Reranking

For n-best reranking, we study three metrics types: likelihood-based, execution-based, and neural. For likelihood-based metrics, we use the likelihood feature and CoderReviewer, both normalized by sequence length. For execution-based metrics, we consider filtering based on trial unit tests. For neural metrics, we use CodeScore (Dong et al., 2023) as it is so far the only publicly available general-purpose neural metric trained to predict execution accuracy. We do not consider task-specific and model-specific neural metrics (Inala et al., 2022; Ni et al., 2023).

For MBR, we consider execution-based metrics and neural metrics. Specifically, among execution-free metrics, we include CodeBertScore (Zhou et al., 2023), which is not trained with execution signals, and CodeScore (Dong et al., 2023), which is trained with execution signals. For execution-based metrics, we consider MBR-Exec (Shi et al., 2022) mentioned in Equation 5. Note that we assume access to the *inputs* of unit tests used for evaluation, which aligns with one setting of MBR-Exec (Shi et al., 2022) and the setting of LEVER (Ni et al., 2023) and Self-Debug (Chen et al., 2024).

3.5 Self-Debug

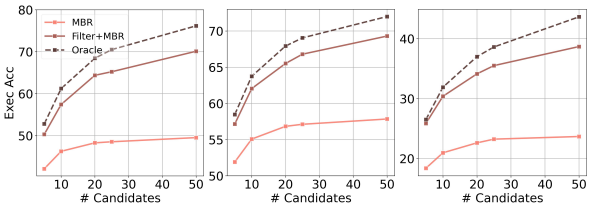
We consider the simple setting of Self-Debug proposed by Chen et al. (2024), using only unit test (UT) feedback, i.e., the feedback obtained from execution when a generated candidate is tested on trial unit tests. We only consider this simple setting because 1) our main focus is to test generally the 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 for inference 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.

4 Results

In this section, we provide results and discuss the effects of sampling, reranking, and Self-Debug. Firstly, we analyze the impact of candidate generation on the model's *oracle* performance (assuming we pick the best possible candidate) and the performance of execution-based reranking methods. We then compare different reranking methods and discuss the importance of execution for reranking. With the answers to these questions in mind, we study the effect of Self-Debug and discuss the best way to combine all these methods to improve reranking performance. We provide a subset of all our experimental results to answer these research questions. For results on more experiments, see Appendix B.

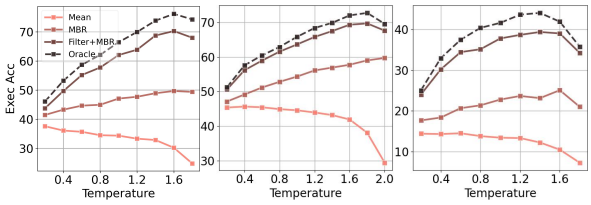
4.1 Candidate Generation

First, we study the impact of candidate generation varied by candidate numbers and sampling temper-



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

Figure 2: Performance of reranking and oracle performance over different numbers of generated candidates using CodeLlama-7B-Instruct with temperature 1.6 for HumanEval+ and MBPP+, and 1.2 for LiveCodeBench. Results are averaged across at least 2 runs for LiveCodeBench and 4 runs for the rest.



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

Figure 3: Performance of reranking and oracle over sampling temperatures using CodeLlama-7B-Instruct with 50 generated candidates over 4 runs.

ature using CodeLlama-7B-Instruct. We provide average execution accuracies, reranking results using MBR-Exec with and without filtering based on trial unit tests, and estimates of reranking oracle using *Pass@k*. We hypothesize that filtering gives reranking better alignment with the (potential) ground-truth function as it is a strong signal.

What is the impact of the number of candidates? We generate candidates using CodeLlama-7B-Instruct by setting temperature 1.6 and p 0.95.

Figure 2 shows the oracle performance and the reranking performance. First of all, when filtering is not included, MBR-Exec only shows moderate improvement with more generated candidates across different numbers of unit tests for evaluation and two benchmarks, aligning with the analysis from Shi et al. (2022). However, with filtering, MBR-Exec shows considerable improvements over the case without filtering, with performance scaling much better with more candidates generated, which aligns with our hypothesis. Secondly, the performance of MBR-Exec with filtering consistently approaches the oracle performance even as we increase the number of candidates, providing further evidence supporting our hypothesis.

What is the impact of sampling temperature? We compare generations with temperatures using CodeLlama-7B-Instruct with nucleus p 0.95 50 generated candidates. We show results in Figure 3.

We find 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 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). Additionally, when filtering is applied, reranking performance approaches the oracle, which again supports our aforementioned hypothesis.

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[2].

4.2 Comparing Reranking Methods

With the best setting we observe from Section 4.1, we now try to provide an apples-to-apples comparison between different reranking methods that either directly utilize execution, predict executions, or utilize other information to understand the effect of filtering or execution more generally in reranking and provide suggestions for reranking methods.

We conduct experiments with n-best reranking and MBR to compare different raking approaches with and without filtering based on trial unit tests. We show results in Table 4.

Filtering is highlighted. According to the group "N-Best Reranking" (see Table 4), filtering based on trial unit tests gives the largest gain from the random baseline compared to the likelihood feature, CoderReviewer, and CodeScore.

Not only is filtering powerful in itself, but it also shows further improvements when combined with n-best reranking methods. Therefore, we again emphasize the importance of filtering based on unit tests for reranking.

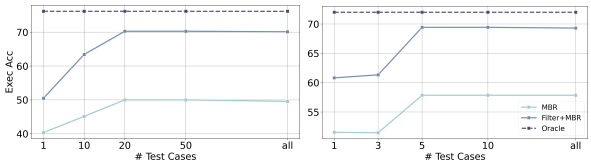
[2]With temperature >1.2 on DeepSeek models, sampling is more likely to generate token indices that are not defined in the vocabulary, see https://github.com/vllm-project/vllm/pull/3685.

	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
Filter	59.7	**60.8**	29.6
Filter + LL	60.4	59.7	**32.5**
Filter + CR	**61.3**	60.2	32.2
Filter + CS	61.0	-	30.2
MBR			
MBR-CBS	35.5	45.4	-
MBR-CS	31.9	-	-
MBR-Exec	49.5	57.8	23.7
N-Best Reranking + MBR			
LL + MBR-Exec	49.4	58.1	24.2
CR + MBR-Exec	48.9	58.7	24.2
Filter + MBR-CBS	60.1	61.3	-
Filter + MBR-CS	59.9	-	-
Filter + MBR-Exec	70.1	69.3	**38.7**
Filter + LL + MBR-Exec	**70.6**	69.2	33.4
Filter + CR + MBR-Exec	70.0	**69.4**	33.2

Table 1: Comparison of 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, CodeBertScore (CBS), and MBR-Exec. We highlight **best** and second best reranking results of the class of reranking methods.

MBR-Exec with filtering still rocks. Zhang et al. (2023) shows that Coder-Reviewer outperforms MBR-Exec with a weaker type of filtering, i.e., retaining a code candidate if it can produce outputs given the inputs provided in trial unit tests. In our experimental settings that include a more sensible definition of filtering and assume access to all unit test inputs for evaluation, likelihood feature and CoderReviewer underperform MBR-Exec with filtering. Additionally, when combined with filtering and MBR-Exec, they do not substantially outperform MBR-Exec with filtering either, suggesting that the likelihood feature and CoderReviewer (Zhang et al., 2023) do not substantially help reranking when sufficient direct execution is applied, giving close-to-oracle performances.

Additionally, when combined with filtering, MBR based on neural metrics does not significantly outperform filtering itself on basic programming tasks (HumanEval and MBPP). Therefore, we do not recommend compared to using execution-based MBR. Acknowleding the huge computational cost of applying neural metrics for MBR, we do not apply it to competitive programming with long generated sequences.



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

Figure 4: Performance of MBR-Exec with fewer unit tests.

4.3 Efficient Reranking with Fewer Unit Tests

While MBR-Exec performs well with filtering, it requires access to execution unit test inputs, which is impractical considering the execution cost. Therefore, we ask: *how many high-quality unit tests should we execute candidates on?* We experiment with 50 generated candidates using MBR-Exec using increasing unit tests. Note that as we use the extended unit tests (Liu et al., 2023), we should also evaluate cases with the extended unit tests. We do not test this on LiveCodeBench due to the limit of unit tests. Additionally, when the number of unit tests is lower than those given in the original dataset, we only use unit tests from the original dataset instead of the extended partition.

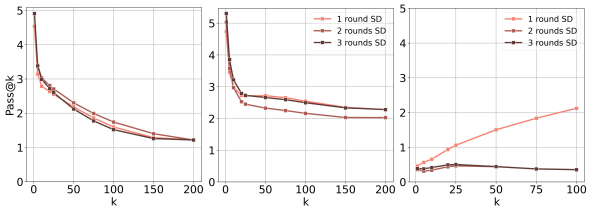
With only a few more unit tests than from the original partition, MBR-Exec with filtering reaches peak performance. According to Figure 4, for HumanEval+, we only need 20 unit tests, while for MBPP-S+, MBR-Exec with 5 unit tests already gives optimal performance. This suggests that MBR-Exec can already reach its optimal performance with a few unit tests provided.

4.4 Improving Oracle with Self-Debug

Applying MBR-Exec with filtering gives close-to-oracle performances, suggesting that improving the oracle performance with Self-Debug (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.

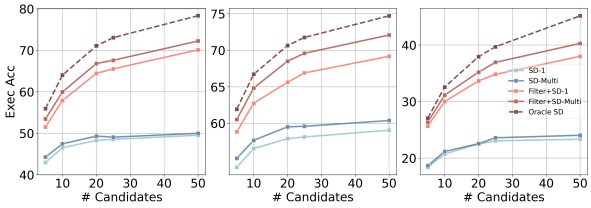
Model	HumanEval+			MBPP-S+			LiveCodeBench		
	MBR	SD-1	SD-Multi	MBR	SD-1	SD-Multi	MBR	SD-1	SD-Multi
∼7B Scale									
CL-7B-Instruct	49.5	49.5	**50.0**	57.8	59.1	**60.4**	23.7	23.3	**24.0**
+ Filtering	70.1	70.1	**72.3**	69.3	69.2	**72.1**	38.7	38.0	**39.9**
DS-6.7B-Instruct	85.5	86.0	**87.0**	78.6	78.8	**79.3**	30.6	30.6	**31.3**
+ Filtering	**90.7**	90.4	90.4	82.1	81.7	**82.8**	41.9	41.2	**42.0**
∼13B Scale									
CL-13B-Instruct	59.6	60.1	**61.4**	67.2	67.0	**68.4**	27.0	**27.6**	**27.6**
+ Filtering	76.1	76.5	**77.6**	74.7	74.2	**76.2**	40.6	40.8	**46.8**
∼16B Scale									
DS-V2-Lite-Instruct	83.5	83.4	**85.1**	80.1	81.0	**81.4**	45.8	45.4	**46.6**
+ Filtering	**89.6**	88.4	**89.6**	81.8	82.2	**82.8**	**58.7**	58.3	58.5

Table 2: Comparison of self-debugging methods with 50 candidates generated by CodeLlama-{7,13}B-Instruct and DeepSeekCoder-{6.7B,V2-Lite}-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+ (c) LiveCodeBench

Figure 5: Improvement in Pass@k of CodeLlama-7B-Instruct after Self-Debug compared to no Self-Debug applied.



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

Figure 6: Comparison of Self-Debug methods over different numbers of candidates generated by CodeLlama-7B-Instruct, and debugged over {1, **Multi**} candidates. Results are averaged across at least 4 runs (and 2 runs for LiveCodeBench).

Self-Debug improves oracle performance of reranking, and one single round is enough. According to Figure 5, Self-Debug helps improve the oracle of reranking, represented by the improvement on *Pass@k* across different *k*s. However, later rounds of Self-Debug do not show substantial improvement in the oracle compared to the first round. Moreover, Self-Debug 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 Self-Debug regenerating 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, Self-Debug regenerates candidates close to greedy decoding.

SD-Multi outperforms SD-1. According to Table 2, SD-Multi consistently outperforms SD-1 in most experiments, with better-performing models having smaller margins of improvement with Self-Debug. The only exceptions we find are on experiments with candidates generated and self-debugged with DeepSeekCoder-{6.7B, V2-Lite}-Instruct, where either the task is considered more difficult compared to basic programming, or that the baseline performance with MBR-Exec is already above 91.

Filtering matters in Self-Debug. With 50 candidates generated and self-debugged, more experiments of MBR-Exec outperforming SD-1 with filtering compared to without filtering (see Table 2), suggesting that improvement brought by Self-Debug can be partly accomplished by filtering, a finding that is missing from Chen et al. (2024) where filtering is also applied but not analyzed. Additionally, when filtering is applied, SD-1 only outperforms MBR-Exec with samples generated and debugged with CodeLlama-13B-Instruct.

Figure 6 shows that SD-Multi (1 round) consistently outperforms SD-1 (3 rounds) with different numbers of candidates generated. Last but not least, Self-Debug leaves a larger gap between SD-Multi and the oracle.

5 Related Work

Large Language Models for Programming Languages. Recent years have witnessed the success of pretrained large language models for code generation (Chen et al., 2021; Li et al., 2022, 2023) that have been successful on multiple execution-based benchmarks (Hendrycks et al., 2021; Austin et al., 2021; Chen et al., 2021; Liu et al., 2023; Lai et al., 2023; Wang et al., 2023b). More recently, strong-performing open-source pretrained Code LLMs (Rozière et al., 2023; Luo et al., 2024; Guo et al., 2024a; Lozhkov et al., 2024; DeepSeek-AI et al., 2024) have been available, and they can perform tasks such as filling-in-the-middle (Rozière et al., 2023), code debugging (Guo et al., 2024b). Our works use open-source code LLMs and instruction tuning to study reranking and Self-Debug.

Reranking in Code Generation Shi et al. (2022) and Li et al. (2022) proposed MBR decoding based on agreement on execution results. Li et al. (2022) further proposed filtering based on trial unit tests, and Shi et al. (2022) proposed filtering out samples that cannot get execution outputs, followed by Zhang et al. (2023) combining log-likelihoods of the code candidate and the instruction. Our work systematically these methods

Other works adopt a setting without unit test inputs for evaluation available, and proposed methods with generated unit test cases utilizing generated unit tests (Li et al., 2022; Chen et al., 2023; To et al., 2024) for reranking and Huang et al. (2024) incorporating both generated unit test cases and functional specifications Our work differs from these efforts as we adopt a setting with unit test inputs for evaluation available.

Additionally, previous works have also studied training N-Best rerankers. Early efforts include training classifiers that predict program success in execution (Inala et al., 2022), followed by Ni et al. (2023) proposing to train both model-specific and task-specific classifiers for reranking. Our work differs from Ni et al. (2023) as we evaluate with more general-purpose metrics (Dong et al., 2023) that do not rely on execution during inference time.

Generally, our work differs as we systematically study the reranking with representative methods, along with sampling strategies and Self-Debug.

Self-Debug with LLMs Chen et al. (2024) studied the effect of using the same Code LLM to debug a single generated code candidate by varying feedback information such as execution errors and code explanation. Olausson et al. (2024) also studied Self-Debug but focus on different sampling methods and their impacts on Self-Debug. Zhong et al. (2024) additionally integrated information regarding intermediate states to provide additional information for debugging. Our work differs from theirs as we study the joint effect of both Self-Debug and reranking. We also perform Self-Debug on multiple candidates.

6 Conclusion and Future Work

We propose **DOCE**, a unified inference framework for code generation. We systematically study candidate generation, reranking, and self-debugging. Importantly, we highlight the impact of high-temperature sampling, execution-based reranking with few high-quality unit tests, and self-debugging with multiple candidates. Potential future works include analyzing unit test quality for reranking and proposing better methods for generating code candidates, exploring the current framework with generated unit test cases, and exploring general-purpose rerankers.

Limitation

First of all, our paper is limited by the scale of experiments as we cannot experiment with all available large language models due to the limit of computation, and our solution is to select representative classes of open-source models and experiments on models with parameters at most 16 billions. Additionally, this paper is limited by potential risks of data contamination from the dataset, despite that HumanEval and MBPP are commonly used in evaluating code generation models. Our solution is to include experiments with LiveCodeBench, which contains a lower risk of data contamination.

Ethical Considerations

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

Sumanta Bhattacharyya, Amirmohammad Rooshenas, Subhajit Naskar, Simeng Sun, Mohit Iyyer, and Andrew McCallum. 2021. Energy-based reranking: Improving neural machine translation using energy-based models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4528–4537. 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.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

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.

Bryan Eikema and Wilker Aziz. 2020. Is MAP decoding all you need? the inadequacy of the mode in neural machine translation. In *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 4506–4520. International Committee on Computational Linguistics.

Bryan Eikema and Wilker Aziz. 2022. Sampling-based approximations to minimum bayes risk decoding for neural machine translation. 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 10978–10993. Association for Computational Linguistics.

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.

Markus Freitag, David Grangier, Qijun Tan, and Bowen Liang. 2022. High quality rather than high model probability: Minimum bayes risk decoding with neural metrics. *Trans. Assoc. Comput. Linguistics*, 10:811–825.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

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. 2024a. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. 2024b. Codeeditorbench: Evaluating code editing capability of large language models. *CoRR*, abs/2404.03543.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

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.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codas, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

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.

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.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science 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 18319–18345. PMLR.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

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*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur

Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. 2019. Facebook fair's WMT19 news translation task submission. In *Proceedings of the Fourth Conference on Machine Translation, WMT 2019, Florence, Italy, August 1-2, 2019 - Volume 2: Shared Task Papers, Day 1*, pages 314–319. Association for Computational Linguistics.

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.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. *CoRR*, arXiv:2303.08774.

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.

Jianhao Shen, Yichun Yin, Lin Li, Lifeng Shang, Xin Jiang, Ming Zhang, and Qun Liu. 2021. Generate & rank: A multi-task framework for math word problems. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2269–2279. Association for Computational Linguistics.

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.

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.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023a. 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.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023b. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 1271–1290. Association for Computational Linguistics.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023. 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.

Lily Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A large language model debugger via verifying runtime execution step-by-step. *CoRR*, abs/2402.16906.

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.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

A 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. We adopt a simpler evaluation by eliminating all test cases with expected output being extreme values and set 20 unit test cases per problem at most. For easy implementation, we only include problems presented as function completion, obtaining 283 problems, among which 88 are easy problems, 135 are medium-level problems, and 60 are hard problems.

B Experimental Results

B.1 Candidate Generation

We present the effect of the number of candidates with candidates generated by CodeLlama-{7,13}B-Instruct (see Figure 7 and Figure 9) DeepSeekCoder-{6.7B, V2-Lite}-Instruct (see Figure 8 and Figure 10) on both the original unit tests and extended test cases in EvalPlus (Liu et al., 2023). We also provide all cases for LiveCodeBench (see Figure 11). Most results align with those we present in Figure 2, where filtering based on trial unit tests improves MBR-Exec considerably and approaches the oracle performance when combined with MBR-Exec. The only exception happens on candidates generated by DeepSeekCoder-6.7B-Instruct on HumanEval (see Figure 8a) where the execution accuracy using MBR-Exec with filtering on no less than 10 candidates is already above 90.

To validate our choice of temperature, we also present the choice of sampling temperature of CodeLlama-7B-Instruct on both the original and extended test cases in Figure 12. 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 3. 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-Exec allows constant improvement in execution accuracy when sampling with higher temperatures, which is not guaranteed without filtering.

B.2 Comparing Reranking Methods

We compare reranking methods with candidates generated by DeepSeekCoder-6.7B-Instruct (see Table 5) and CodeLlama-13B-Instruct (see Table 6). Findings are similar to Section 4.2, where filtering based on trial unit tests is important itself and helps to boost other reranking methods, and MBR-Exec remains the best-performing method, giving close-to-oracle performance when combined with filtering. Log-likelihoods of the generated candidate and the input do not necessarily help better reranking.

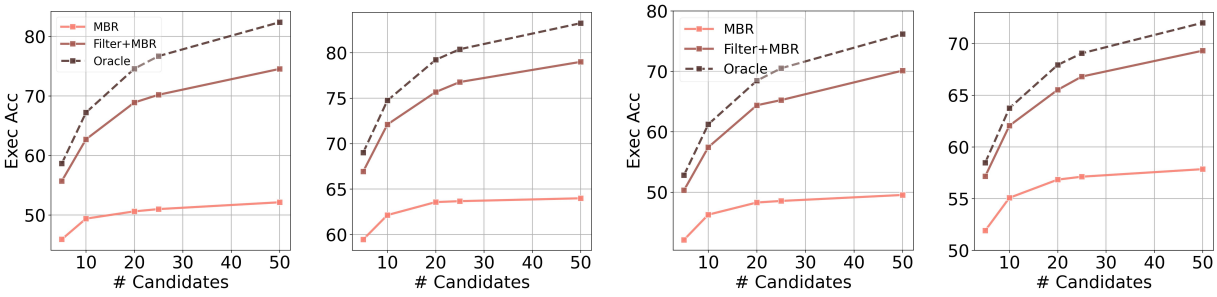
B.3 MBR-Exec with Fewer Unit Tests

We present results using fewer unit tests using candidates generated from DeepSeekCoder-6.7B-Instruct Figure 13 and CodeLlama-13B-Instruct in Figure 14. Similar to Figure 4, we only need 20 unit test cases for HumanEval+, and 5 unit test cases for MBPP-S+ to obtain optimal performance with MBR-Exec.

B.4 Improving Oracle with Self-Debugging

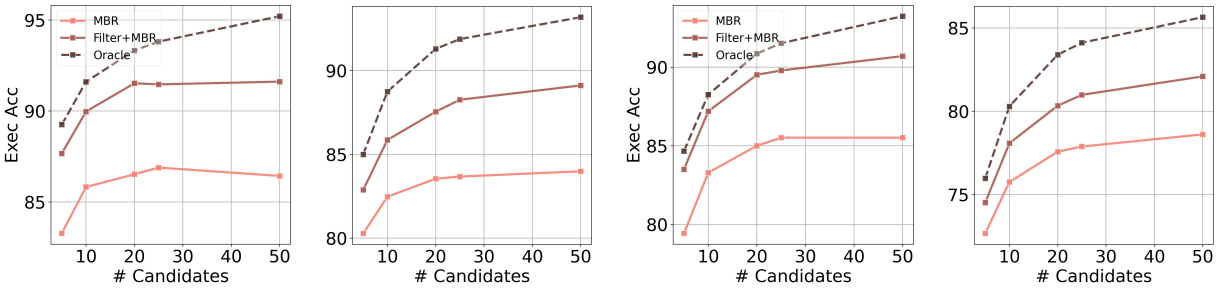
We first present results of improvement of *Pass@k* that estimates oracle improvement using candidates generated and self-debugged by CodeLlama-7B-Instruct (see Figure 15), DeepSeekCoder-6.7B-Instruct (see Figure 16), CodeLlama-13B-Instruct (see Figure 17), and DeepSeekCoder-V2-Lite-Instruct (see Figure 18). We also provide cases for LiveCodeBench (see Figure 19). Our findings align with Section 4.4 as one round of self-debugging is enough to improve the oracle.

We then present results over different numbers of candidates generated and debugged by CodeLlama-7B-Instruct (see Figure 20), DeepSeekCoder-6.7B-Instruct (see Figure 21), CodeLlama-13B-Instruct (see TFigure 22), and DeepSeekCoder-V2-Lite-Instruct (see Figure 23). We also provide all cases for LiveCodeBench (see Figure 24). We find that SD-Multi outperforms SD-1 constantly except for HumanEval(+) with candidates generated and debugged by DeepSeekCoder-6.7B-Instruct with 25 generated candidates.



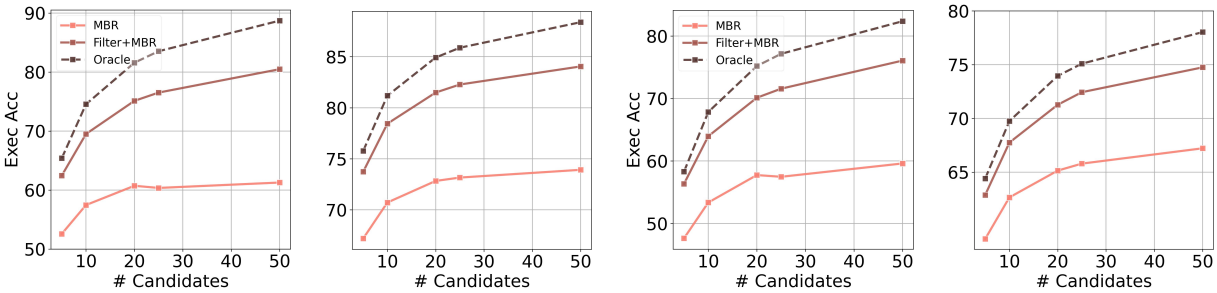
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 7: Performance of MBR-Exec with and without filtering over different numbers of generated candidates using CodeLlama-7B-Instruct. We also provide the oracle with solid lines, denoted as *Pass@k*. Results that end with + mean that it's evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



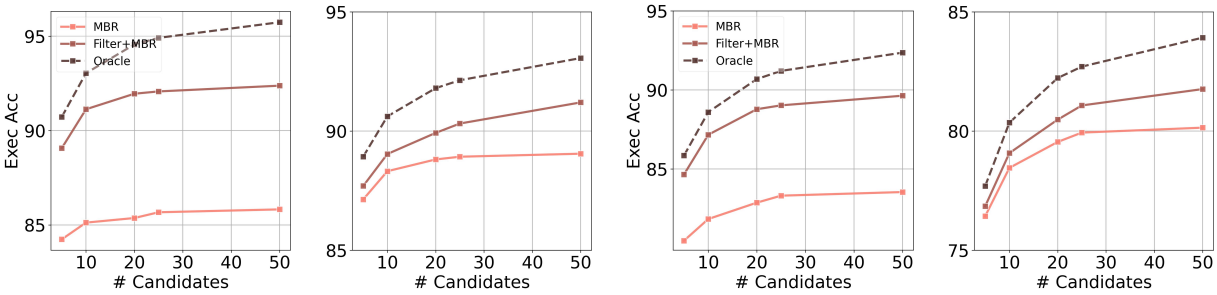
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 8: Performance of MBR-Exec with and without filtering over different numbers of generated candidates using DeepSeekCoder-6.7B-Instruct. We also provide the oracle with solid lines, denoted as *Pass@k*. Results that end with + mean that it's evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



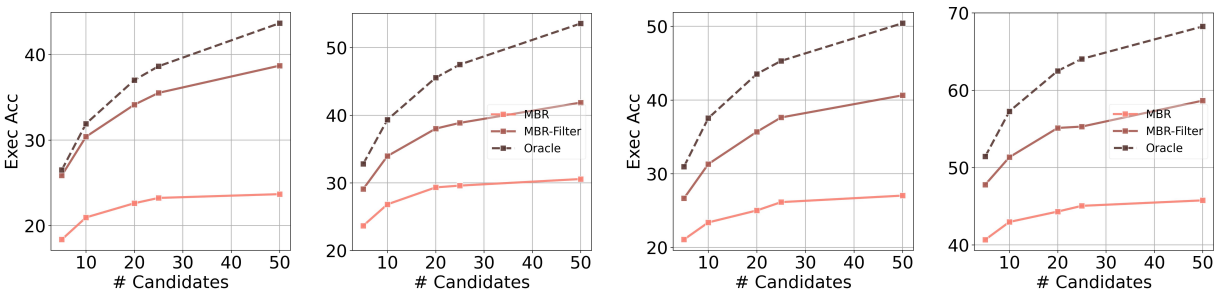
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 9: Performance of MBR-Exec with and without filtering over different numbers of generated candidates using CodeLlama-13B-Instruct. We also provide the oracle with solid lines, denoted as *Pass@k*. Results that end with + mean that it's evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



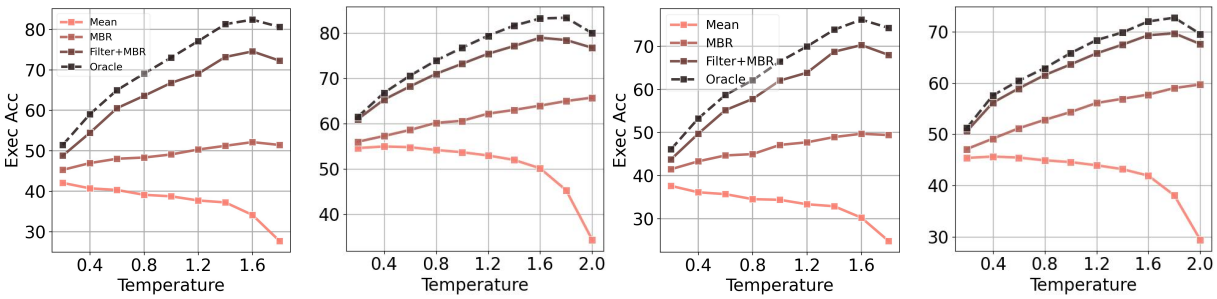
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 10: Performance of MBR-Exec with and without filtering over different numbers of generated candidates using DeepSeekCoder-V2-Lite-Instruct. We also provide the oracle with solid lines, denoted as *Pass@k*. Results that end with + mean that it's evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



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

Figure 11: Performance of MBR-Exec with and without filtering over different numbers of generated candidates using all models and tested on LiveCodeBench. We also provide the oracle with solid lines, denoted as *Pass@k*. Results are averaged across at least 2 runs.

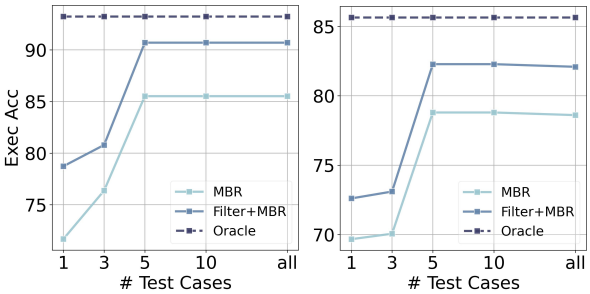


(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 12: Performance of MBR-Exec with and without filtering over sampling temperatures using CodeLlama-7B-Instruct with 50 generated candidates. We also provide the oracle of reranking with dashed lines. Results that end with + mean that it's evaluated on the plus with extended test cases, otherwise it's evaluated on the original test cases. Results are averaged across 4 runs.

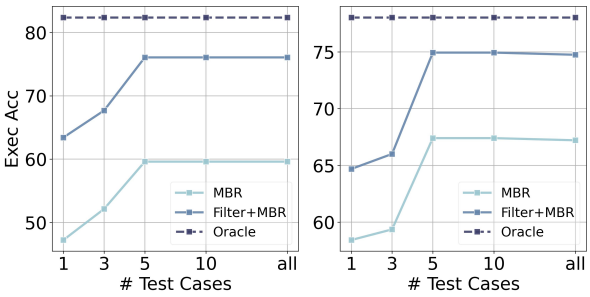
Model	DS-6.7B-Instruct		CL-13B-Instruct		DS-V2-Lite-Instruct	
	temp = 0.8	temp = 1.2	temp = 0.8	temp = 1.6 (1.2 for LCB)	temp = 0.8	temp = 1.2
HumanEval(+)						
Mean	77.3 (70.6)	72.5 -4.8 (65.1 -5.5)	45.7 (40.0)	35.8 -9.9 (31.1 -8.9)	81.7 (76.6)	81.7 -0 (76.6 -0)
MBR	87.0 (85.8)	86.4 -0.6 (85.5 -0.3)	59.0 (54.4)	61.3 +2.3 (59.6 +5.2)	84.8 (81.3)	85.8 +1.0 (83.5 +2.2)
Filter + MBR	90.7 (89.8)	91.6 +0.9 (90.7 +0.9)	76.5 (69.2)	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)	67.5 (60.7)	73.9 +6.4 (67.2 +6.5)	88.6 (78.5)	89.1 +0.5 (80.1 +1.6)
Filter + 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
Filter + 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 3: Comparison of performance of sampling and reranking using temperature 0.8 and those chosen for further experiments. We report mean execution accuracies, MBR-Exec 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.



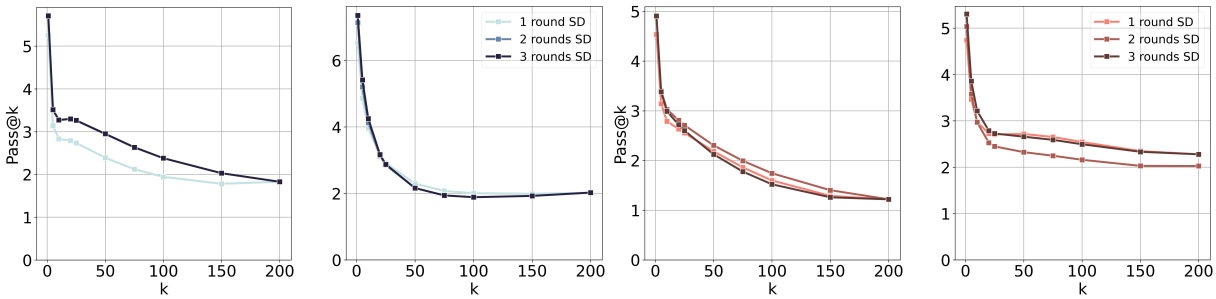
(a) HumanEval+ (b) MBPP-S+

Figure 13: Performance of MBR-Exec with fewer unit tests using candidates generated from DeepSeekCoder-6.7B-Instruct.



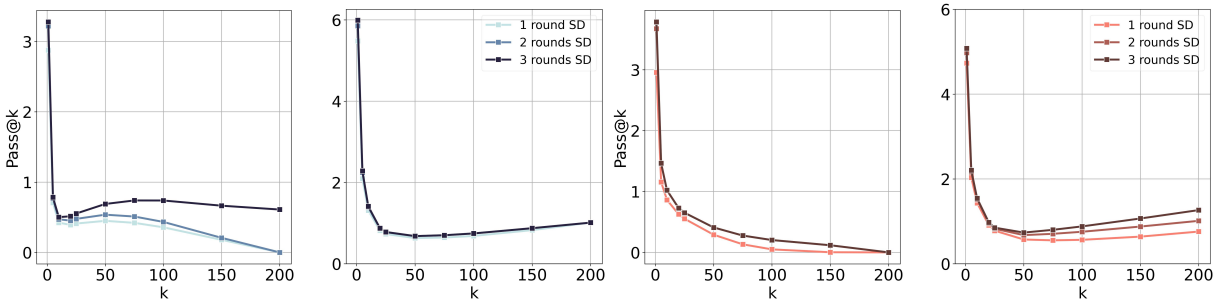
(a) HumanEval+ (b) MBPP-S+

Figure 14: Performance of MBR-Exec with fewer unit tests using candidates generated from CodeLlama-13B-Instruct.



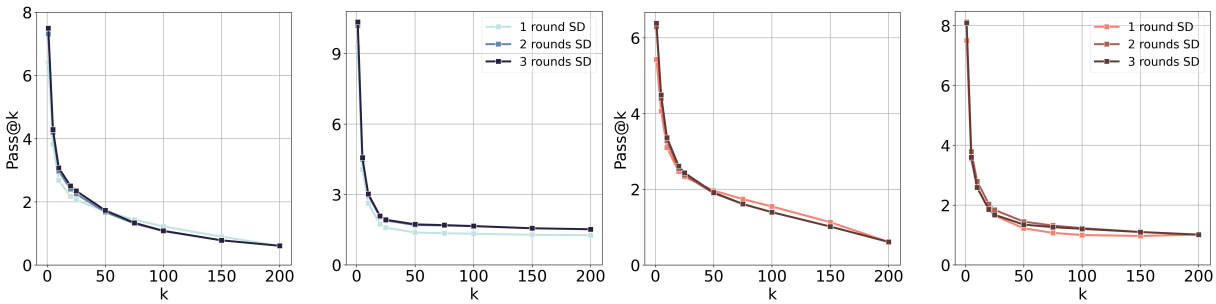
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 15: Improvement in Pass@k of CodeLlama-7B-Instruct after self-debugging compared to no self-debugging applied.



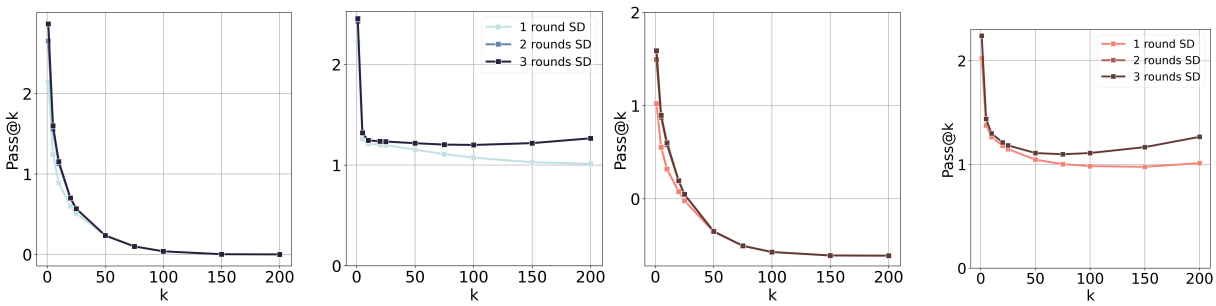
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 16: Improvement in Pass@k of DeepSeekCoder-6.7B-Instruct after self-debugging compared to no self-debugging applied.



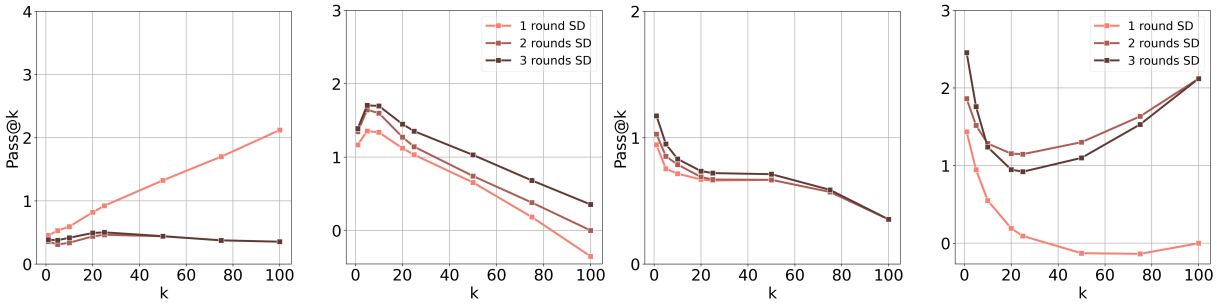
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 17: Improvement in Pass@k of CodeLlama-13B-Instruct after self-debugging compared to no self-debugging applied.



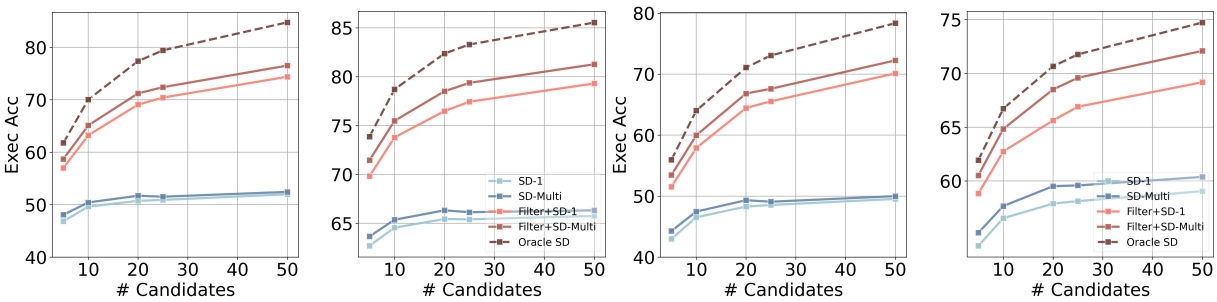
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 18: Improvement in Pass@k of DeepSeekCoder-V2-Lite-Instruct after self-debugging compared to no self-debugging applied.



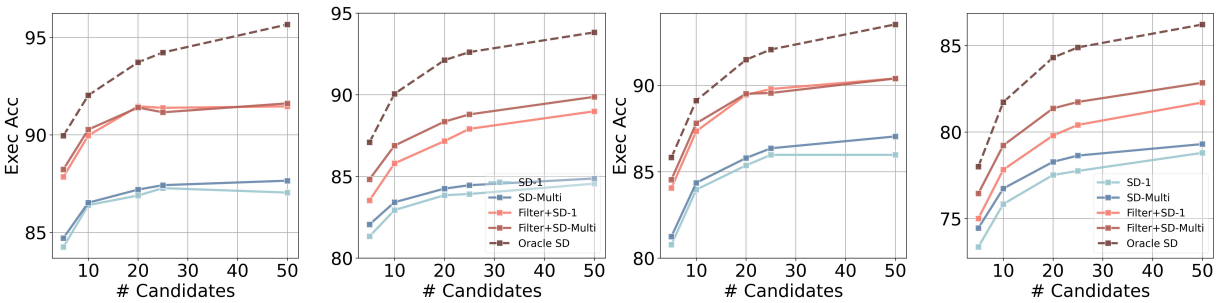
(a) CL-7B (b) DS-6.7B (c) CL-13B (d) DS-V2-Lite

Figure 19: Improvement in Pass@k on LiveCodeBench.



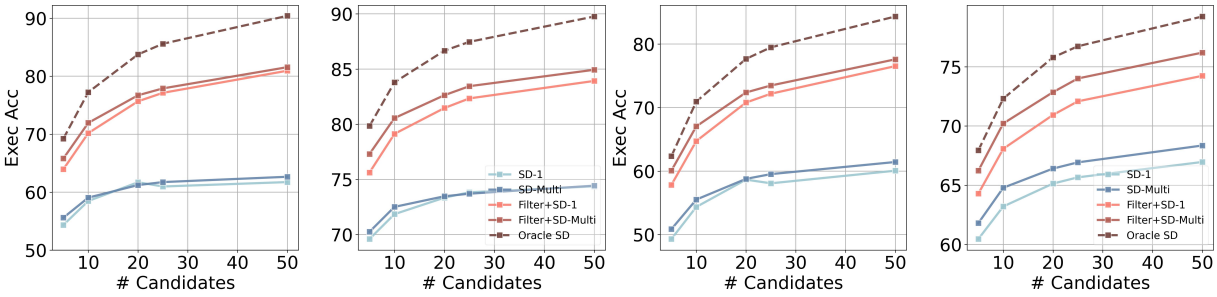
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 20: Comparison of self-debugging methods over different numbers of candidates generated by CodeLlama-7B-Instruct, and debugged over $\{1, \textbf{Multi}\}$ candidates using the same LLM. We also provide the oracle after self-debugging for one round. Note that results ending with $+$ mean it is evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



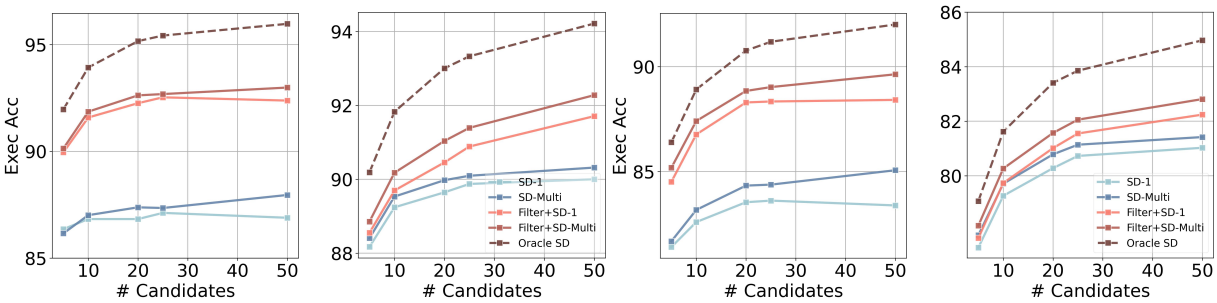
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 21: Comparison of self-debugging methods over different numbers of candidates generated by DeepSeekCoder-6.7B-Instruct, and debugged over $\{1, \textbf{Multi}\}$ candidates using the same LLM. We also provide the oracle after self-debugging for one round. Note that results ending with $+$ mean it is evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



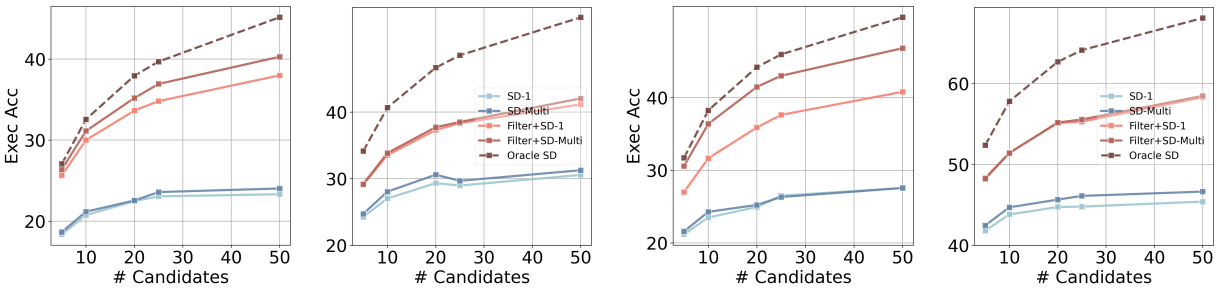
(a) HumanEval (b) MBPP-S (c) HumanEval+ (d) MBPP-S+

Figure 22: Comparison of self-debugging methods over different numbers of candidates generated by CodeLlama-13B-Instruct, and debugged over $\{1, \textbf{Multi}\}$ candidates using the same LLM. We also provide the oracle after self-debugging for one round. Note that results ending with + mean it is evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



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

Figure 23: Comparison of self-debugging methods over different numbers of candidates generated by DeepSeekCoder-V2-Lite-Instruct, and debugged over $\{1, \textbf{Multi}\}$ candidates using the same LLM. We also provide the oracle after self-debugging for one round. Note that results ending with + mean it is evaluated on the plus with extended test cases. Results are averaged across at least 4 runs.



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

Figure 24: Comparison of self-debugging methods over different numbers of candidates on LiveCodeBench, and debugged over $\{1, \textbf{Multi}\}$ candidates using the same LLM. We also provide the oracle after self-debugging for one round. Note that results ending with + mean it is evaluated on the plus with extended test cases. Results are averaged across at least 2 runs.

	HE (+)	MBPP-S (+)
Random	34.1 (30.2)	50.1 (41.9)
Greedy	43.3 (39.0)	53.2 (44.8)
Oracle	82.4 (76.2)	83.2 (72.0)
N-Best Reranking		
LL	45.1 (38.7)	54.2 (42.6)
CR	46.0 (40.5)	53.1 (43.1)
CS	33.1 (30.2)	-
Filter	70.0 (59.7)	77.4 (**60.8**)
Filter + LL	71.0 (60.4)	78.1 (59.7)
Filter + CR	**71.3** (**61.3**)	**78.5** (60.2)
Filter + CS	71.2 (61.0)	-
MBR		
MBR-CBS	40.5 (35.5)	55.3 (45.4)
MBR-CS	37.5 (31.9)	-
MBR-Exec	52.1 (49.5)	64.0 (57.8)
N-Best Reranking + MBR		
LL + MBR-Exec	51.8 (49.4)	64.4 (58.1)
CR + MBR-Exec	51.4 (48.9)	64.8 (58.7)
Filter + MBR-CBS	69.8 (60.1)	77.3 (61.3)
Filter + MBR-CS	71.5 (59.9)	-
Filter + MBR-Exec	**74.5** (70.1)	79.0 (69.3)
Filter + LL + MBR-Exec	**74.5** (**70.6**)	78.9 (69.2)
Filter + CR + MBR-Exec	74.0 (70.0)	**79.3** (**69.4**)

Table 4: Comparison of reranking methods using CodeLlama-7B-Instruct on HumanEval(+) and MBPP(+). For N-Best Reranking, we compare log-likelihood (LL), Coder-Reviewer (CR), and CodeScore (CS). For MBR, we compare CodeScore, CodeBertScore (CBS), and MBR-Exec. **HE** stands for the HumanEval benchmark, + refers to evaluation on extended test cases on EvalPlus. We also highlighted **best** and second best reranking results of the class of reranking methods.

	HE (+)	MBPP-S (+)
Random	72.5 (65.1)	68.6 (58.6)
Greedy	82.3 (76.8)	72.2 (64.3)
Oracle	95.2 (93.2)	93.2 (85.6)
N-Best Reranking		
LL	81.4 (71.8)	70.1 (58.9)
CR	78.7 (70.3)	76.1 (64.0)
CS	70.1 (63.6)	-
Filter	88.4 (78.7)	87.4 (72.6)
Filter + LL	**90.5** (79.1)	**88.5** (**72.8**)
Filter + CR	89.5 (**79.6**)	88.2 (**72.8**)
Filter + CS	87.7 (78.4)	-
MBR		
MBR-CBS	81.1 (73.0)	72.6 (63.4)
MBR-CS	74.2 (67.5)	-
MBR-Exec	86.4 (85.5)	84.0 (78.6)
N-Best Reranking + MBR		
LL + MBR-Exec	86.6 (85.7)	84.4 (79.0)
CR + MBR-Exec	86.6 (85.7)	85.4 (79.7)
Filter + MBR-CBS	90.7 (81.6)	88.1 (75.3)
Filter + MBR-CS	89.6 (80.3)	-
Filter + MBR-Exec	91.6 (90.7)	89.1 (82.1)
Filter + LL + MBR-Exec	**91.9** (**90.9**)	89.5 (82.4)
Filter + CR + MBR-Exec	**91.9** (**90.9**)	**89.6** (**82.5**)

Table 5: Comparison of reranking methods using DeepSeekCoder-6.7B-Instruct on HumanEval(+) and MBPP(+). For N-Best Reranking, we compare log-likelihood (LL), Coder-Reviewer (CR), and CodeScore (CS). For MBR, we compare CodeScore, CodeBertScore (CBS), and MBR-Exec. **HE** stands for the HumanEval benchmark, + refers to evaluation on extended test cases on EvalPlus. We also highlighted **best** and second best reranking results of the class of reranking methods.

	HE (+)	MBPP-S (+)
Random	35.8 (31.1)	52.5 (43.5)
Greedy	45.7 (39.6)	62.5 (52.7)
Oracle	88.7 (82.4)	88.4 (78.0)
N-Best Reranking		
LL	54.0 (45.6)	57.9 (47.2)
CR	51.1 (43.8)	60.4 (49.5)
CS	35.8 (31.7)	-
Filter	75.4 (63.4)	81.9 (64.7)
Filter + LL	**77.4** (**65.7**)	**82.8** (64.1)
Filter + CR	76.8 (**65.7**)	82.7 (**65.0**)
Filter + CS	76.7 (**65.7**)	-
MBR		
MBR-CBS	47.3 (40.2)	64.4 (53.5)
MBR-CS	37.5 (33.4)	-
MBR-Exec	61.3 (59.6)	73.9 (67.2)
N-Best Reranking + MBR		
LL + MBR-Exec	61.6 (59.8)	73.8 (67.0)
CR + MBR-Exec	62.5 (60.5)	75.1 (68.2)
Filter + MBR-CBS	75.9 (64.5)	82.5 (65.8)
Filter + MBR-CS	77.0 (65.7)	-
Filter + MBR-Exec	**80.5** (**76.1**)	84.1 (74.7)
Filter + LL + MBR-Exec	79.9 (75.5)	83.9 (74.5)
Filter + CR + MBR-Exec	79.6 (75.3)	**84.2** (**74.8**)

Table 6: Comparison of reranking methods using CodeLlama-13B-Instruct on HumanEval(+) and MBPP(+). For N-Best Reranking, we compare log-likelihood (LL), Coder-Reviewer (CR), and CodeScore (CS). For MBR, we compare CodeScore, CodeBertScore (CBS), and MBR-Exec. **HE** stands for the HumanEval benchmark, + refers to evaluation on extended test cases on EvalPlus. We also highlighted **best** and second best reranking results of the class of reranking methods.