Pragmatic Reasoning improves LLM Code Generation

Anonymous ACL submission

Abstract

Pragmatic reasoning is pervasive in human–human communication — it allows us to leverage shared knowledge and counterfactual reasoning in order to infer the intention of a conversational partner given their ambiguous or underspecified message. In human–computer communication, underspecified messages often represent a major challenge: for instance, translating natural language instructions into code is difficult when user instructions contain inherent ambiguities. In the present paper, we aim to scale up the pragmatic "Rational Speech Act" framework to naturalistic language-to-code problems, and propose a way of dealing with multiple meaning-equivalent instruction alternatives, an issue that does not arise in previous toy-scale problems. We evaluate our method, CodeRSA, with two recent LLMs (Llama-3-8B-Instruct and Qwen-2.5-7B-Instruct) on two widely used code generation benchmarks (HumanEval and MBPP). Our experimental results show that CodeRSA consistently outperforms common baselines, surpasses the state-of-the-art approach in most cases, and demonstrates robust overall performance. Qualitative analyses demonstrate that it exhibits the desired behaviour for the right reasons. These findings underscore the effectiveness of integrating pragmatic reasoning into a naturalistic complex communication task, language-to-code generation, offering a promising direction for enhancing code generation quality in LLMs and emphasizing the importance of pragmatic reasoning in complex communication settings. Our implementation is available at `https://anonymous.4open.science/r/CodeRSA-C76B`.

1 Introduction

Recent advances in generative large language models (LLMs) have demonstrated their impressive ability to generate program code from user-provided natural language instructions (Liu et al., 2024b; Coignion et al., 2024). However, given the intrinsic complexities of coding and the potential ambiguities in user input, producing code in a single attempt may fail to explore the vast solution space, overlooking correct or higher-quality solutions (Liu et al., 2024a). A standard practice to address this shortcoming is to sample multiple solutions, which we refer to as *code candidates* (Chen et al., 2021; Brown et al., 2024), and to rerank them.

When viewing code generation as a communicative process in which an LLM listens to the user's intentions (Ouyang et al., 2022), the different code candidates represent the different possible user intents and the instruction represents an ambiguous or underspecified message. Pragmatic reasoning then involves considering not only the different possible intended meanings, but also counterfactual reasoning about what instruction the speaker would have given if they had a specific intent. These ideas go back to (Grice, 1975) and have been mathematically formalized using a probabilistic model rooted in game-theoretic notions, as the "Rational Speech Act" (RSA) framework in Frank and Goodman (2012; 2016). Code reranking can thus be performed with respect to the RSA reasoning process. In previous research, Pu et al. (2020, 2024) demonstrated the effectiveness of the RSA framework for program generation in a simple domain (regular-expression synthesis), while Schuster et al. (2024) reported negative results on a spreadsheet domain. One aspect that has held back RSA models from scaling up to realistic use cases is the computational overhead (Pu et al., 2024): It requires reasoning about a whole set of alternative instructions that the speaker could have given and about the set of alternative pieces of code that could solve the problem, which is computationally expensive. Zhang et al. (2023a) therefore proposed *CoderReviewer Reranking* as a scalable approach that simplifies these probability estimation processes over alterna-

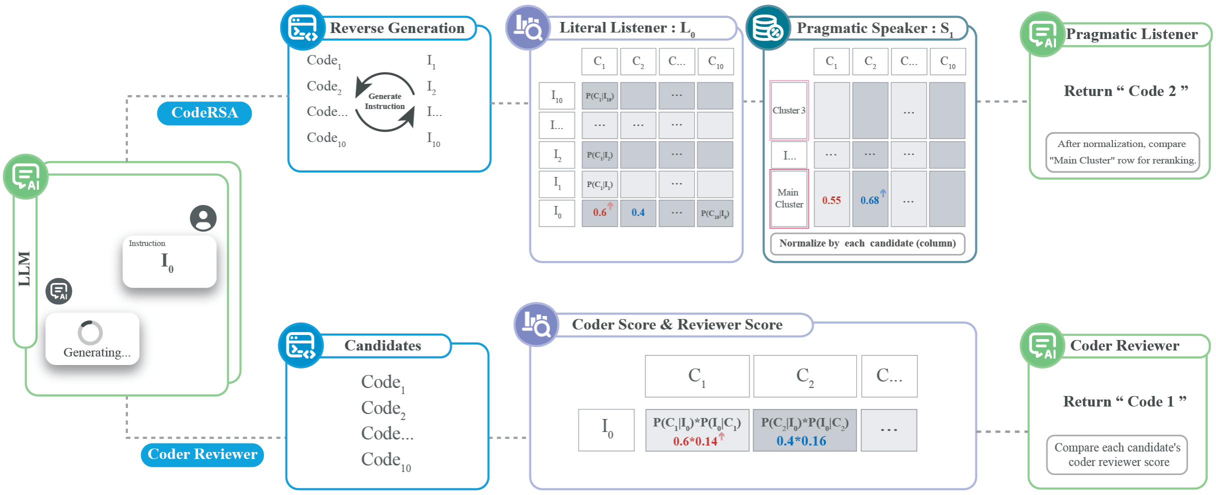


Figure 1: A comparison of our approach CodeRSA (top) compared to CoderReviewer (bottom).

tives. However, this comes at the cost of not fully modelling the dialogic, interactive reasoning that can emerge when speaker and listener exchange information.

In this paper, we propose CodeRSA, enabling LLMs to reason as pragmatic listeners and rank code candidates based on the user's underlying intentions. It addresses the probability estimates for the set of alternative code candidates and alternative utterances via a sampling approach. CodeRSA generates multiple code candidates, and then generates additional instructions for each candidate, forming a set of potential instructions (including the original one), as illustrated in Fig. 1.

Following the RSA framework, the literal listener L_0 first estimates the probability of each code candidate given each potential instruction. The pragmatic speaker S_1 then normalizes these probabilities to measure how specifically an instruction fits the generated code. Finally, by comparing these pragmatic speaker scores for the original instruction across all candidates, the pragmatic listener identifies the code candidate that best aligns with the user's intent, completing the reranking process (see Fig. 1).

A challenge arises when many instructions are semantically equivalent but differ only in surface form. Applying RSA directly in such cases can lead to an overinterpretation of the formulation choice: The reasoning process is forced to treat near-identical descriptions as distinct alternatives, which were chosen for a reason of differentiating from other meaning alternatives. This fragments probability mass and reduces accuracy. To mitigate this, CodeRSA employs a clustering step. It groups semantically equivalent descriptions using an LLM-based equivalence test, ensuring that pragmatic reasoning emphasizes genuine differences in meaning rather than superficial wording (see Fig. 1, Pragmatic Speaker Part, where the *main cluster* refers to the one containing the original description I_0).

We conducted experiments using CodeRSA with Llama-3-8B-Instruct, one of the latest language models from the Llama family (Grattafiori et al., 2024), and Qwen-2.5-7B-Instruct, a recent instruction-tuned model from the Qwen series (Yang et al., 2024), on two widely used code generation benchmarks: OpenAI's HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Our experimental results demonstrate that CodeRSA reliably outperforms the simpler Coder and Coder-Reviewer Reranking methods. Our qualitative analysis reveals how the CodeRSA enables better candidate selection, promoting a more comprehensive understanding of user intent.

2 Related Work

Natural Language to Code. Previous research has extensively explored generating code from natural language using neural network models (Ling et al., 2016; Rabinovich et al., 2017; Hayati et al., 2018). Recently, large language models (LLMs) have propelled significant advances in this area, driven by the transformer (Vaswani et al., 2017) architecture and large-scale pretraining. Their performance on code generation tasks often surpasses that of traditional models, and in many cases even rivals human programmers (Ni et al., 2024; Becker et al., 2023). A recent study shows that LLMs also exhibit strong performance in code summarization, effectively translating code snippets into text (Akib et al., 2024).

Code Reranking Methods. Researchers have proposed various reranking strategies for code candidates, broadly divided into *execution-driven* and *content-driven* approaches. Execution-driven reranking methods such as CodeT (Chen et al., 2022) and AgentCoder (Huang et al., 2024) evaluate code candidates by running them against automatically generated test suites. Although often effective, these methods rely on the availability and reliability of test suites, which are frequently incomplete or difficult to construct, and executing untrusted code can pose safety risks (Yetiştiren et al., 2023; Khoury et al., 2023). In contrast, content-driven reranking methods are far more versatile because they do not rely on execution and are not even confined to coding tasks. One such model is Coder Reranking (Chen et al., 2021).

Coder Reranking. Chen et al. (2021) rerank code candidates by estimating $\mathrm{P}(c \mid i)$, where c denotes the generated code candidate and i denotes the given instruction. This process can also be called Coder Reranking because the LLM is a mere Coder that estimates the candidate probability based on the corresponding instruction. When using an LLM to estimate conditional probabilities, we compute the probability of each token iteratively. For example, in Coder Reranking, the model processes a candidate's tokens from left to right: At each step, it calculates the probability of the current token given the instruction and the previously generated tokens, then appends that token to the context before moving on. The product of these sequential probabilities across all tokens yields the overall probability of the code candidate under the given instruction:

$$\mathrm{P}(c \mid i_0) \;=\; \prod_{t=1}^{|c|} \mathrm{P}_{\mathrm{LLM}}\big(c^{(t)} \mid i_0, c^{(<t)}\big), \quad (190)$$

where $c^{(t)}$ denotes the token at position t in the sequence c, and $c^{(<t)}$ represents the sequence of all tokens before position t.

CoderReviewer Reranking. Zhang et al. (2023a) introduced the idea of augmenting Coder Reranking with a reviewer, which jointly considers how likely a code candidate is under the instruction and how well the instruction is supported by the code. Formally, the CoderReviewer conditional probability is defined as:

$$\mathrm{P}_{\mathrm{CR}}(c \mid i) \;\propto\; \underbrace{\mathrm{P}_{\mathrm{LLM}}(c \mid i)}_{\text{(Coder)}} \cdot \underbrace{\mathrm{P}_{\mathrm{LLM}}(i \mid c)}_{\text{(Reviewer)}} \quad (201)$$

By switching the positions of the instruction and code in the conditional formulation, the second term can be interpreted as reformulating the code-generation task as an instruction-generation task. This bidirectional formulation can be viewed as a specialized form of maximum mutual information (Li and Jurafsky, 2016).

3 CodeRSA

In this section, we introduce CodeRSA, an approach that builds on the Rational Speech Act (RSA) framework to enhance the reranking of candidate code snippets. CodeRSA extends the models proposed by Cohn-Gordon et al. (2019) and Schuster et al. (2024). The core innovation in CodeRSA arises from the pragmatic listener, which is responsible for selecting and reranking code candidates. It does so by imagining how a *pragmatic speaker* would choose an instruction that best distinguishes the intended code among various potential instructions.

Literal Listener. A literal listener (denoted L_0) represents the simplest level of reasoning in the RSA framework. It interprets utterances solely according to their literal meaning, without any higher-level pragmatic inference. Let c denote a candidate program and i a user instruction. Then:

$$\mathrm{P}_{L_0}(c \mid i) = \mathrm{P}_{\mathrm{LLM}}(c \mid i), \quad (229)$$

where $\mathrm{P}_{\mathrm{LLM}}(c \mid i)$ is the probability assigned by the LLM to candidate c given instruction i. In an idealized RSA setting, the literal listener would evaluate all possible programs, but since the space of programs is unbounded, we approximate it by sampling a finite set of candidate codes from the LLM. We additionally define a candidate prior distribution obtained by querying the LLM without any instruction context:

$$\mathrm{P}_{\mathrm{prior}}(c) = \mathrm{P}_{\mathrm{LLM}}(c \mid \varnothing). \quad (239)$$

This prior reflects how plausible a candidate program is in general, independent of the specific user instruction.

Pragmatic Speaker. In the RSA framework, the pragmatic speaker (denoted S_1) is primarily responsible for determining whether an instruction i effectively conveys the intended meaning of a candidate c to the literal listener. Formally, a pragmatic speaker can be defined as:

$$\mathrm{P}_{S_1}(i \mid c) = \frac{\exp\left(\log \mathrm{P}_{L_0}(c \mid i) - C(i)\right)}{\sum_{i'} \exp\left(\log \mathrm{P}_{L_0}\left(c \mid i'\right) - C\left(i'\right)\right)}.$$

Here, $C(i)$ denotes a cost function for using instruction i. In an ideal RSA setting, the normalization spans every possible instruction i', which is intractable for code generation. To approximate this space in practice, we take the sampled candidate codes as anchors and derive m alternative instructions from each of the n code candidates, together with the original instruction i_0, yielding a finite task-relevant instruction set $I = \{i_0, i_1, \ldots, i_{mn}\}$. This construction provides a principled approximation of the otherwise infinite instruction space while keeping RSA's normalization meaningful.

To simplify the model and focus on core pragmatic reasoning, we assume a uniform cost for all instructions, which effectively cancels out during normalization. A detailed modeling of the cost function may provide additional insights, a point we further discuss in Section 6. A pragmatic speaker then can be defined in a simplified form as:

$$\mathrm{P}_{S_1}(i \mid c) = \frac{\mathrm{P}_{L_0}(c \mid i)}{\sum_{i' \in I} \mathrm{P}_{L_0}(c \mid i')}.$$

Pragmatic Listener. The pragmatic listener (denoted L_1) re-examines the original instruction i_0 across all candidates, completing the backward reasoning guided by the pragmatic speaker's preferences. In the standard RSA formulation (Degen, 2023), a pragmatic listener is defined as:

$$\mathrm{P}_{L_1}(c \mid i) \;\propto\; \mathrm{P}_{S_1}(i \mid c) \cdot \mathrm{P}(c),$$

where $\mathrm{P}(c)$ denotes the prior probability of candidate c.

In practice, directly multiplying a normalized distribution by $\mathrm{P}(c)$ can distort the allocation of probability mass, as the prior may dominate post hoc. Instead, CodeRSA incorporates priors via a candidate-specific *temperature* applied before normalization at the speaker stage. Let z_c be the within-task standardized log prior of candidate c (estimated from the LLM without conditioning context), and define a candidate-specific temperature as

$$\tau_c \;=\; e^{-\alpha z_c}, \qquad \alpha \geq 0, \quad \tau_c > 0,$$

where α controls how strongly the prior influences the temperature scaling. A higher prior (larger z_c) yields a smaller temperature ($\tau_c < 1$) and thus a sharper distribution over alternatives. Candidates with higher priors therefore emphasize their most confident clusters more strongly, typically those that align best with the original instruction (e.g., the "main cluster"), giving them a comparative advantage during reranking.

With this calibration, the pragmatic speaker used by the listener is

$$\mathrm{P}_{S_1}(i \mid c; \tau_c) \;=\; \frac{\left(\mathrm{P}_{L_0}(c \mid i)\right)^{1/\tau_c}}{\sum_{i' \in I} \left(\mathrm{P}_{L_0}(c \mid i')\right)^{1/\tau_c}},$$

which reduces to the standard RSA speaker when $\alpha = 0$ (thus $\tau_c = 1$). Finally, the pragmatic listener ranks candidates with respect to the original instruction:

$$\mathrm{P}_{L_1}(c \mid i_0) \;\propto\; \mathrm{P}_{S_1}(i_0 \mid c; \tau_c).$$

This formulation preserves the spirit of RSA while integrating priors in a stable and interpretable manner: Rather than post-hoc reweighting, priors act as adaptive temperatures that shape the pragmatic reasoning process upstream of normalization. In our experiments, we treat α as a tunable hyperparameter and find that performance is stable across a broad range of values (see Section 5).

Clustering Paraphrases. While the basic RSA formulation operates directly over the instruction set I, it can suffer from over-interpreting superficial variations in wording when many instructions are semantically equivalent and differ only in surface form. In such cases, RSA allocates probability mass across paraphrases as if they were meaningful distinct alternatives, diluting the signal and reducing accuracy.

Semantically equivalent instructions:

"return the sum of a list of integers"
"compute the total of all integers in a list"

Non-equivalent instruction:

"return the product of a list of integers"

Table 1: Examples of equivalent and non-equivalent instructions from MBPP. The first group expresses the same semantics, while the second differs in meaning.

To mitigate this, CodeRSA employs semantic clustering. Candidate instructions are grouped into semantic clusters $\mathcal{C} = \{C_1, \ldots, C_K\}$ using an LLM-based equivalence test (implementation details in Section 4), so that pragmatic reasoning operates over clusters rather than individual instructions. This ensures that comparisons emphasize genuine differences in meaning rather than superficial variation.

For a candidate c and cluster C_k, the literal listener probability is aggregated as:

$$\mathrm{P}_{L_0}(c \mid C_k) = \begin{cases} \mathrm{P}_{L_0}(c \mid i_0), & \text{if } i_0 \in C_k, \\ \frac{1}{|C_k|} \sum_{i \in C_k} \mathrm{P}_{L_0}(c \mid i), & \text{otherwise.} \end{cases}$$

The pragmatic speaker distribution over clusters then becomes:

$$\mathrm{P}_{S_1}(C_k \mid c; \tau_c) \;=\; \frac{\big(\mathrm{P}_{L_0}(c \mid C_k)\big)^{1/\tau_c}}{\sum_{C_{k'} \in \mathcal{C}} \big(\mathrm{P}_{L_0}(c \mid C_{k'})\big)^{1/\tau_c}},$$

where the candidate-specific temperature $\tau_c = e^{-\alpha z_c}$ incorporates priors.

Finally, the pragmatic listener reranks candidates with respect to the cluster C^* containing the original instruction i_0:

$$\mathrm{P}_{L_1}(c \mid i_0) \;\propto\; \mathrm{P}_{S_1}(C^* \mid c; \tau_c).$$

This extension preserves the primacy of the original instruction while preventing RSA from over-differentiating among paraphrases. Moreover, the integration of priors through adaptive temperatures ensures that the reranking remains calibrated against candidate plausibility.

4 Experiment Setup

To understand the strengths and weaknesses of CodeRSA, we evaluate the performance of three reranking methods (Coder, CoderReviewer, and CodeRSA) on widely used benchmarks for code generation. Since the advantage of content-driven methods lies in their generality, we rely on commonly adopted default settings and perform only minimal sensitivity checks on key parameters.

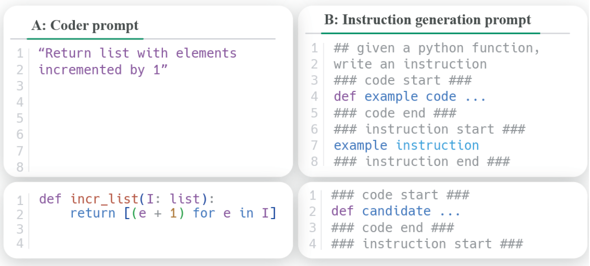
4.1 Dataset and Base Models

We evaluate on two widely used code generation benchmarks. HumanEval (Chen et al., 2021) contains 164 Python programming problems, each presented as an unfinished function with a natural language instruction. MBPP (Austin et al., 2021) includes 974 short programming tasks with natural language prompts. HumanEval offers balanced difficulty, while MBPP introduces greater lexical variety. Note that simpler datasets such as CoNaLa (Yin et al., 2018) already yield near-perfect results, leaving little room for reranking, whereas more challenging datasets such as BigCodeBench (Zhuo et al., 2024) contain many instances that cannot yet be solved by today's state-of-the-art models, which makes it difficult to obtain meaningful comparisons of reranking methods and may obscure the performance differences we aim to study.

We use the following setup: for each problem in HumanEval and MBPP, we sample $n = 10$ candidate codes at a temperature of 1.0. We then evaluate reranking methods on this shared candidate set. A sensitivity check with varying numbers of sampled candidates is provided in Appendix A.4.

For our experiments, we use Llama-3-8B-Instruct (Grattafiori et al., 2024) and Qwen-2.5-7B-Instruct (Yang et al., 2024), two instruction-tuned LLMs of comparable scale. Llama-3-8B-Instruct balances efficiency with strong generation quality, while Qwen-2.5-7B-Instruct provides a competitive open-source alternative. Both achieve competitive performance on HumanEval and MBPP, making them suitable for assessing reranking in our setting. We do not include specialized coder models in this study, since our framework requires both code generation and instruction-level reasoning. Future work could explore hybrid setups, for example, using coder models for program synthesis combined with general-purpose instruction-tuned models for reasoning about instructions.

Our goal is to examine whether RSA-style pragmatic reasoning can be effectively applied to realistic code generation through appropriate methodological design. We evaluate our approach on two established benchmarks (MBPP and HumanEval) using mid-sized instruction-tuned models, which provide sufficient headroom for reranking. For example, Llama-3-8B-Instruct improves from 51.3% pass@1 to 78.2% pass@10 on MBPP, indicating that correct solutions are often generated but not reliably selected in a single attempt. In such settings, the effect of reranking is most clearly observable. Appendix A.3 further reports a sanity check with `Llama-3-70B-Instruct` (Grattafiori et al., 2024), confirming that CodeRSA behaves consistently when applied to a substantially larger model.



A: Coder prompt

```
“Return list with elements
incremented by 1”
```

```
def incr_list(I: list):
    return [(e + 1) for e in I]
```

B: Instruction generation prompt

```
## given a python function,
write an instruction
### code start ###
def example code ...
### code end ###
### instruction start ###
example instruction
### instruction end ###
```

```
### code start ###
def candidate ...
### code end ###
### instruction start ###
```

Figure 2: The prompts used to calculate Coder score and generate additional instructions.

4.2 Implementation of Reranking Methods

Baselines. The Coder Reranking method provides a straightforward way to compare the probability of a code candidate c given the original instruction i. Specifically, it concatenates the instruction and code candidate in order (see Fig. 2, part A), prompting the language model to output token probabilities for the candidate sequentially. The product of these token probabilities then yields the cumulative probability of the entire code snippet. As mentioned in Section 3, Coder Reranking can also be considered a literal listener-level approximation to $\mathrm{P}(c \mid i)$; therefore, we use it as a baseline.

State-of-the-art Method. Zhang et al. (2023a) showed that CoderReviewer Reranking (see Section 2 for details) outperforms Coder Reranking and rivals execution-driven methods such as CodeT. In practice, we use the same prompt format as in Coder Reranking to compute $\mathrm{P}(c \mid i)$. To compute $\mathrm{P}(i \mid c)$, the order of the instruction and the generated code snippet is reversed in the prompt (see Appendix A.6.2).

CodeRSA. To balance runtime and computational constraints, we limit the process to $n = 10$ candidate programs per problem. For CodeRSA, we further generate one additional instruction ($m = 1$) for each candidate using a one-shot prompt (temperature = 0.7; see Fig. 2, part B).

Rather than treating each instruction independently, we next use the same LLM that generated the candidates to perform pairwise semantic equivalence judgments among these instructions. Following prior work on LLM-based clustering (Zhang et al., 2023b), we cluster instructions that express the same functionality (see Section 3). Concretely, we query the LLM with a 3-shot prompt containing examples of both positive and negative semantic equivalence pairs, asking it to judge whether two instructions express the same functionality. We then build a pairwise equivalence graph where each node represents an instruction and an edge indicates semantic equivalence according to the LLM. The connected components of this graph are treated as clusters of mutually equivalent instructions (see Appendix A.5).

For each candidate c, we compute literal listener scores with respect to every instruction, and then aggregate them at the cluster level: non-main clusters take the mean across their members, while the cluster containing the original instruction i_0 retains its direct probability. We also incorporate candidate priors through a candidate-specific temperature parameter τ_c, controlled by a coefficient α (see Section 3 for the definition). Finally, we apply softmax normalization with these temperatures over clusters to obtain cluster-level pragmatic speaker scores. The pragmatic listener then reranks candidates by selecting the one with the highest speaker score with respect to the i_0-cluster, which represents the original user intent.

5 Results

5.1 Quantitative Analysis

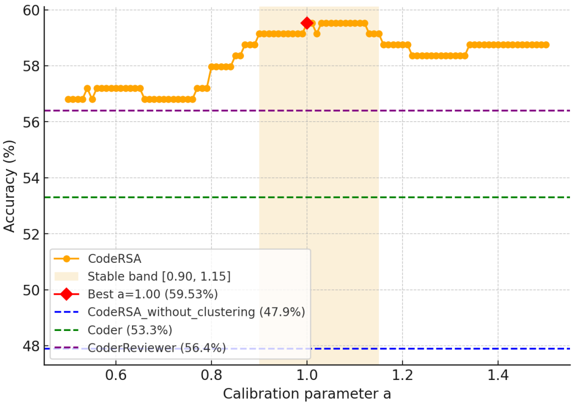
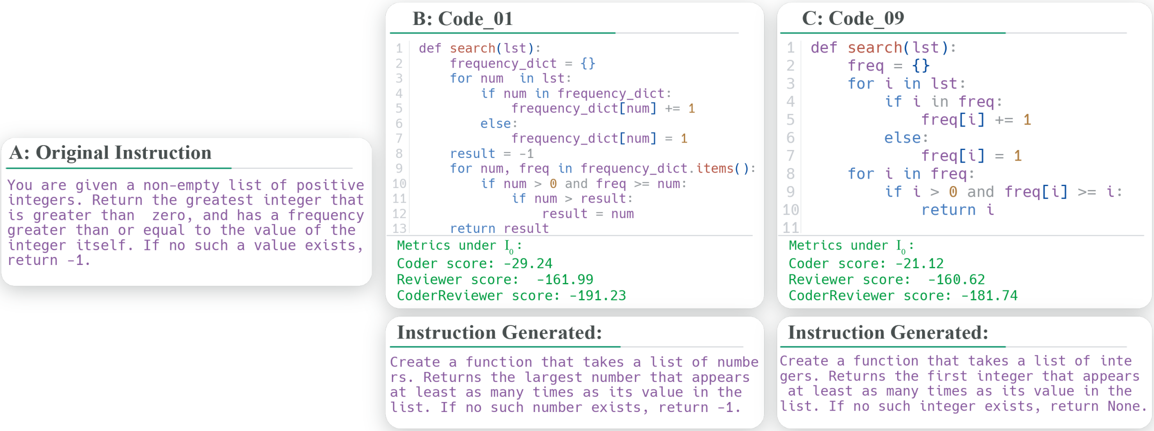


Figure 3: Accuracy of CodeRSA across different values of the calibration parameter α. The shaded region indicates a stable performance band.

In this section, we analyze the quantitative performance of CodeRSA with respect to the calibration parameter α (defined in Section 3), which controls the influence of the prior through temperature scaling, using the MBPP dataset and Llama-3-8B-Instruct model.

Here, *accuracy* denotes the proportion of test instances where the reranking method selects a candidate that passes all test cases provided by the



A: Original Instruction

```
You are given a non-empty list of positive
integers. Return the greatest integer that
is greater than  zero, and has a frequency
greater than or equal to the value of the
integer itself. If no such a value exists,
return -1.
```

B: Code_01

```python
def search(lst):
    frequency_dict = {}
    for num  in lst:
        if num in frequency_dict:
            frequency_dict[num] += 1
        else:
            frequency_dict[num] = 1
    result = -1
    for num, freq in frequency_dict.items():
        if num > 0 and freq >= num:
            if num > result:
                result = num
    return result
```

Metrics under I_0:
Coder score: -29.24
Reviewer score: -161.99
CoderReviewer score: -191.23

Instruction Generated:

```
Create a function that takes a list of numbe
rs. Returns the largest number that appears
at least as many times as its value in the
list. If no such number exists, return -1.
```

C: Code_09

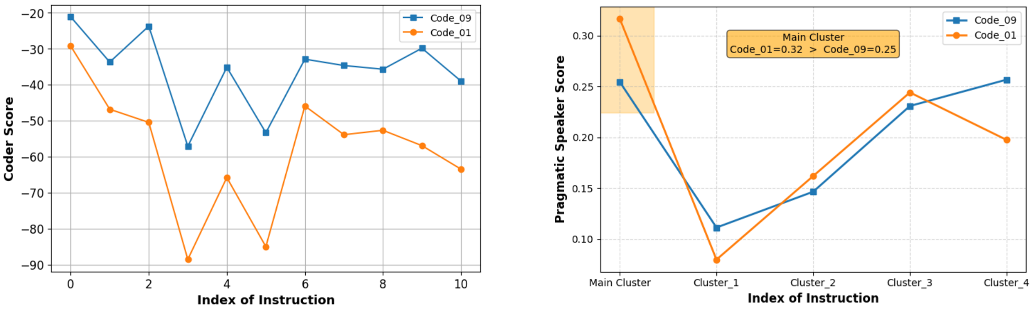
```python
def search(lst):
    freq = {}
    for i in lst:
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
    for i in freq:
        if i > 0 and freq[i] >= i:
            return i

```

Metrics under I_0:
Coder score: -21.12
Reviewer score: -160.62
CoderReviewer score: -181.74

Instruction Generated:

```
Create a function that takes a list of inte
gers. Returns the first integer that appears
 at least as many times as its value in the
list. If no such integer exists, return None.
```

(a) Details of question and two generated examples



(b) Coder Score Comparison

(c) Pragmatic Score Comparison

Figure 4: Qualitative Example: Bias in Coder vs. CodeRSA Correction

benchmark. Figure 3 shows that CodeRSA consistently outperforms baseline reranking methods, with clustering playing a crucial role: removing the clustering step yields a substantial drop in accuracy. The figure also highlights the robustness of CodeRSA to the choice of α: within the stable band of $[0.90, 1.15]$, performance remains consistently above both Coder and CoderReviewer. At $\alpha = 1.0$, CodeRSA achieves the best accuracy of 59.53%, clearly surpassing the baselines. These results demonstrate that CodeRSA's pragmatic reasoning, enhanced by clustering, is not overly sensitive to calibration.

We report further results covering different models (`Qwen2.5-7B-Instruct`) and datasets (HumanEval) in Appendix A.2, which confirm the same pattern of CodeRSA consistently improving performance over baseline methods.

5.2 Qualitative Analysis

Although our experiments show that CodeRSA achieves stable performance, it relies on certain idealized assumptions and an abstract reasoning process. To provide a more intuitive perspective, we include a qualitative analysis that examines how CodeRSA aligns with core RSA intuitions, thereby enhancing reranking quality.

Zhang et al. (2023a) note that reranking based on cumulative token likelihood tends to prefer shorter candidates, since each token probability is < 1 and longer sequences accumulate lower overall scores. This bias makes the Coder approach prone to favoring incomplete or generic programs. In Fig. 4a, the instruction requires returning the greatest integer above zero whose frequency is at least its own value, or -1 if none exists. However, `code_09` omits both the "greatest" requirement and the -1 fallback, making it incomplete but shorter. As shown in Fig. 4b, Coder assigns `code_09` a higher score (-21.12) than the correct `code_01` (-29.24), and thus prefers the degenerate solution. CoderReviewer inherits this issue, as Reviewer alone cannot offset `code_09`'s inflated Coder score.

Fig. 4c reports CodeRSA's cluster-level pragmatic speaker scores after softmax normalization ($\alpha = 1$). Instructions with equivalent semantics are grouped, and the cluster containing the original instruction i_0 is treated as the *main cluster*. Here, `code_01` achieves a score of 0.32 on the main cluster, compared to 0.25 for `code_09`. Notably, `code_09` also receives relatively high confidence on Cluster_4, which dilutes its probability on the main cluster due to RSA normalization. In RSA terms, `code_09` is not strongly aligned with either the main cluster or Cluster_4, indicating that it fits the intended instruction less well than other candidates. By contrast, the probability of `code_01` is concentrated on the main cluster, which better aligns with the original instruction and is therefore favored under pragmatic reasoning.

Taken together, this case study shows how CodeRSA operationalizes RSA reasoning: By normalizing over alternative clusters, it penalizes candidates that spread probability mass across multiple interpretations and favors those that focus on the main cluster, thereby improving robustness and faithfulness in reranking.

6 Discussion

Our proposed CodeRSA approach contains a number of simplifications compared to the original RSA model, which has been developed for describing human–human communication: It assumes a uniform speaker cost for the instructions. While this simplification makes the analysis more tractable, it means that our model does not currently take into account effects related to how "costly" an instruction would be to produce for the human speaker. Future work should investigate variable cost structures to better capture these nuances.

In Section 4, we argued that CodeRSA is most beneficial in situations where the dataset is not too easy (when a simple Coder model already achieves ceiling performance) and not too difficult, i.e., good candidates need to be contained in the set of sampled candidates which is then re-ranked. That is, an important pre-condition for the applicability of the CodeRSA approach is the ability to obtain a high quality probability distribution over instructions and over code candidates. We argue that in complex tasks like language instructions for generating code, a situation where the correct candidate is in the set of sampled candidates, but where it is not ranked first, is not just due to bad model performance that will become better with improved models, but is expected to be a systematic phenomenon that is due to the inherent ambiguity and underspecification in human natural language instructions. Dealing with this type of ambiguity is not a matter of better language modelling, but inherently requires the counterfactual reasoning process implemented in RSA models. Research on human communication has demonstrated the importance of pragmatic reasoning, even though it introduces additional computational overhead. At the same time, studies suggest that humans may switch between different strategies where they sometimes rely on the expensive reasoning process, and sometimes rely on simple heuristics or amortized estimates (Lieder and Griffiths, 2020; Tscshantz et al., 2023; Dasgupta et al., 2018; Pu et al., 2024), avoiding iterative reasoning in easy cases while still engaging in full pragmatic reasoning when tasks are more complex. Such strategies can also be explored in future CodeRSA research.

7 Conclusion

This work introduces CodeRSA, a pragmatics-inspired approach to dealing with ambiguity and underspecification in human instructions for code generation. By modeling the iterative reasoning of a pragmatic listener about a pragmatic speaker, CodeRSA consistently outperforms the Coder Reranking baseline and surpasses the state-of-the-art CoderReviewer approach. Our experiments showed highly consistent results for two models from different model families and two different benchmarks. A qualitative analysis further reveals that, even when incorporating certain idealized assumptions and variations, CodeRSA remains faithful to the core principles and goals of the RSA framework. These results highlight the effectiveness of applying well-established linguistic frameworks to enhance reasoning in language models, opening new avenues for research and development in code-related tasks. Some of our insights related to practical and naturalistic application domains, such as the need for clustering of very similar instructions or the need to re-weight probability estimates for instructions vs. meaning representations may also be applicable to human-computer interaction tasks beyond language-to-code.

8 Limitations

A known limitation of RSA approaches is their computational complexity and associated resource consumption. For example, on a single NVIDIA Tesla A100 (PCIe 4.0, 80GB HBM2e, 300W), performing complete CodeRSA inference on 500 instances takes nearly 6 hours. Our approach compares each potential instruction with every candidate, leading to a quadratic increase in complexity as the number of candidates grows. Although CodeRSA can theoretically handle many candidates, we limited our experiments to ten candidates per question to keep runtime and hardware usage manageable. This restriction inevitably narrows the variety of solutions and may affect how well the approach generalizes to larger-scale scenarios.

Reducing the computational overhead is a major goal for our future work. One promising direction is to design more lightweight scoring mechanisms or to adopt a multi-stage pipeline. For instance, a coarse filtering step could quickly discard low-probability solutions before applying CodeRSA's full RSA-based reasoning to a smaller top-ranked subset. Alternatively, approximate models could reduce the number of token-level evaluations required, thereby preserving much of CodeRSA's pragmatic reasoning benefits at a fraction of the computational cost. Such improvements would allow CodeRSA to scale more effectively and broaden its applicability to larger code generation tasks.

Another limitation concerns the scope of models and benchmarks considered. In the current study, we deliberately focus on mid-sized models and well-established benchmarks as a proof-of-concept design choice, enabling fair and clearly observable comparisons with existing reranking baselines, rather than aiming for exhaustive coverage across model scales or benchmark families. As a result, although our experiments already cover two models (Llama-3-8B-Instruct and Qwen-2.5-7B-Instruct) and two datasets (HumanEval and MBPP), the overall scope remains relatively narrow. In future work, we plan to incorporate additional balanced-difficulty datasets such as DS-1000 (Lai et al., 2023), along with further open-source models like Mistral (Jiang et al., 2023) and newer Qwen releases beyond Qwen-2.5. This expansion will allow us to evaluate reranking methods across a wider range of scenarios, ultimately leading to a more comprehensive assessment of our approach. At the same time, ensuring an appropriate match between model capability and benchmark difficulty remains an important challenge for designing informative evaluations.

References

Md Ahnaf Akib, Md Muktadir Mazumder, and Salman Ahsan. 2024. Analysis on llms performance for code summarization. *arXiv preprint arXiv:2412.17094*.

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

Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *Preprint*, arXiv:2407.21787.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *Preprint*, arXiv:2207.10397.

Mark Chen et al. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Reuben Cohn-Gordon, Noah Goodman, and Christopher Potts. 2019. An incremental iterated response model of pragmatics. *Society for Computation in Linguistics*, 2(1).

Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 79–89.

Ishita Dasgupta, Eric Schulz, Noah D Goodman, and Samuel J Gershman. 2018. Remembrance of inferences past: Amortization in human hypothesis generation. *Cognition*, 178:67–81.

Judith Degen. 2023. The rational speech act framework. *Annual Review of Linguistics*, 9(1):519–540.

Michael C Frank and Noah D Goodman. 2012. Predicting pragmatic reasoning in language games. *Science*, 336(6084):998–998.

Noah D Goodman and Michael C Frank. 2016. Pragmatic language interpretation as probabilistic inference. *Trends in cognitive sciences*, 20(11):818–829.

Aaron Grattafiori et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

HP Grice. 1975. Logic and conversation. *Syntax and semantics*, 3.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2445–2451. IEEE.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Jiwei Li and Dan Jurafsky. 2016. Mutual information and diverse decoding improve neural machine translation. *Preprint*, arXiv:1601.00372.

Falk Lieder and Thomas L Griffiths. 2020. Resource-rational analysis: Understanding human cognition as the optimal use of limited computational resources. *Behavioral and brain sciences*, 43:e1.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024a. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2024. L2ceval: Evaluating language-to-code generation capabilities of large language models. *Transactions of the Association for Computational Linguistics*, 12:1311–1329.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. 2020. Program synthesis with pragmatic communication. *Advances in neural information processing systems*, 33:13249–13259.

Yewen Pu, Saujas Vaduguru, Priyan Vaithilingam, Elena Glassman, and Daniel Fried. 2024. Amortizing pragmatic program synthesis with rankings. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 41221–41234. PMLR.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149.

Sebastian Schuster, Ayesha Ansar, Om Agarwal, and Vera Demberg. 2024. Spreadnala: A naturalistic code generation evaluation dataset of spreadsheet formulas. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 15216–15225.

Alexander Tscshantz, Beren Millidge, Anil K Seth, and Christopher L Buckley. 2023. Hybrid predictive coding: Inferring, fast and slow. *PLoS computational biology*, 19(8):e1011280.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

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

Yuwei Zhang, Zihan Wang, and Jingbo Shang. 2023b. ClusterLLM: Large language models as a guide for text clustering. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13903–13920, Singapore. 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, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Appendix

A.1 A Conjecture: Viewing CoderReviewer through an RSA Lens

CoderReviewer reranks candidates using the bidirectional criterion

$$\mathrm{P}_{\mathrm{CR}}(c \mid i_0) \ \propto\ \mathrm{P}_{\mathrm{LLM}}(c \mid i_0) \cdot \mathrm{P}_{\mathrm{LLM}}(i_0 \mid c),$$

which rewards candidates that are (i) likely under the instruction (Coder) and (ii) able to “explain” the instruction when the code is given as context (Reviewer).

In RSA, a pragmatic listener instead ranks candidates according to

$$\mathrm{P}_{L_1}(c \mid i_0) \ \propto\ \mathrm{P}_{S_1}(i_0 \mid c) \cdot \mathrm{P}(c),$$

combining (a) a speaker term that captures how well the instruction identifies the intended candidate among alternatives, and (b) a prior over candidates.

We conjecture that CoderReviewer can be interpreted as a lightweight, heuristic approximation to this RSA listener objective on the restricted candidate set used for reranking. Intuitively, producing a faithful natural-language description from a concrete piece of code is comparatively constrained, so the reverse model $\mathrm{P}_{\mathrm{LLM}}(i_0 \mid c)$ may behave like an amortized proxy for a speaker preference $\mathrm{P}_{S_1}(i_0 \mid c)$ (without explicitly normalizing over an alternative-instruction set or modeling instruction costs). Meanwhile, when i_0 is underspecified, the forward term $\mathrm{P}_{\mathrm{LLM}}(c \mid i_0)$ can be strongly influenced by instruction-independent code preferences (well-formedness, idiomatic patterns, and length/-format biases), and thus acts as a task-local proxy for the candidate prior $\mathrm{P}(c)$ on the finite set of sampled candidates. Under this view, multiplying the two terms resembles combining a speaker-like compatibility signal with a prior-like plausibility signal, as in RSA.

A.2 More Details of Results

In this subsection, we provide further results on different datasets and models to further validate the robustness of CodeRSA. Specifically, we evaluate on the HumanEval dataset as well as on MBPP, using both `Llama-3-8B-Instruct` and `Qwen2.5-7B-Instruct`.

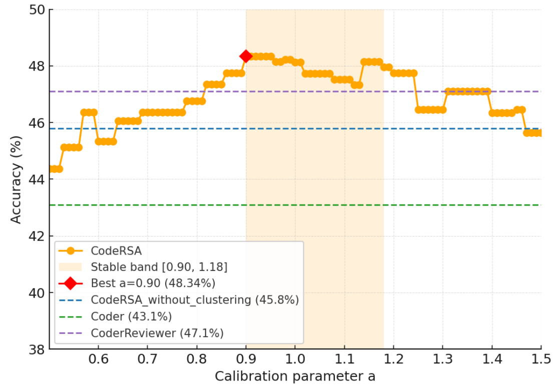


Figure 5: Accuracy of CodeRSA on `HumanEval` with `Qwen2.5-7B-Instruct`. The shaded region shows the stable band.

Across all settings, several consistent trends can be observed.

Clustering effectiveness. Removing the clustering step leads to a noticeable drop in accuracy, highlighting its role in reducing redundancy and stabilizing pragmatic reasoning.

Calibration robustness. CodeRSA is not strongly sensitive to the calibration parameter α; performance remains stable across a relatively wide range rather than relying on a finely tuned value.

Superior accuracy. CodeRSA consistently achieves higher accuracy than both baselines. On the HumanEval dataset, we observe some fluctuations in performance, and the overall accuracy

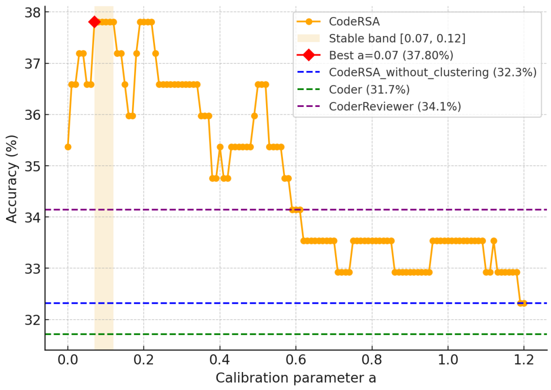


Figure 6: Accuracy of CodeRSA on `HumanEval` with `Llama-3-8B-Instruct`. The shaded region shows the stable band.

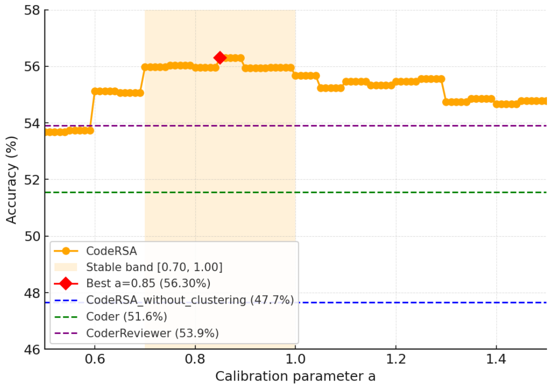


Figure 7: Accuracy of CodeRSA on `MBPP` with `Qwen2.5-7B-Instruct`. The shaded region shows the stable band.

is relatively low. This may be partly due to randomness or parameter settings in the experiments. However, since all reranking methods are evaluated under the same conditions, the relative comparison between them remains fair and informative.

These findings confirm that the improvements achieved by CodeRSA are reliable across different models and datasets. The calibration parameter α is shown to be both interpretable and stable, further supporting the practicality of the approach.

A.3 Additional Experiment: Larger-Model Sanity Check on MBPP

As an additional sanity check, we evaluate CodeRSA with a larger base model. Since RSA-style reranking is computationally expensive, we keep this experiment limited in scope. We use `Llama-3-70B-Instruct` (Grattafiori et al., 2024) on MBPP and restrict evaluation to instances that are unsolved at pass@1 but solved at pass@10, resulting in 84 problems where reranking is non-trivial.

Method	Solved	Acc.
Coder	33/84	39.3%
CoderReviewer	41/84	48.8%
CodeRSA (best $\alpha = 0.9$)	**43/84**	**51.2%**
CodeRSA (avg. $\alpha \in [0.5, 1.5]$)	35.5/84	42.3%

Table 2: Sanity check on `Llama-3-70B-Instruct` for MBPP, evaluated on 84 instances that are unsolved at pass@1 but solved at pass@10. For CodeRSA, "best" is the best value in a sweep with step size 0.1; "avg." is the mean over $\alpha \in \{0.5, 0.6, \dots, 1.5\}$.

Table 2 reports the number of solved problems (out of 84) and the corresponding accuracy after reranking. CodeRSA achieves the best result at $\alpha = 0.9$ in our sweep (step size 0.1), outperforming Coder and slightly improving over CoderReviewer on this subset. Averaged over $\alpha \in [0.5, 1.5]$, CodeRSA remains above Coder, indicating that scaling to a larger model does not lead to abnormal behavior, while calibration still matters for peak performance.

A.4 Accuracy vs. Number of Sampled Candidates (MBPP)

To further examine the impact of candidate diversity on reranking performance, we conducted a controlled study varying the number of sampled code candidates per MBPP problem ($n = 1 \dots 10$). For each value of n, we randomly sampled n candidates from the pool of ten generated solutions and applied three reranking strategies: Coder, CoderReviewer, and our proposed CodeRSA. The experiment was repeated ten times with different random seeds for each value of n, and the figure reports the mean accuracy and its standard deviation across runs.

When $n = 1$, all methods yield identical results, as no reranking can occur. As n increases, performance improves for all methods due to a broader candidate set, but the gain plateaus after approximately $n = 7$. Across all sampling levels, CodeRSA achieves the highest accuracy, maintaining a margin of roughly 2–3 percentage points over CoderReviewer and up to 5 points over Coder. This shows that pragmatic reasoning allows CodeRSA to better leverage candidate diversity while remaining robust to sampling variability. The narrow confidence bands further indicate stable performance even under random candidate selection, confirming

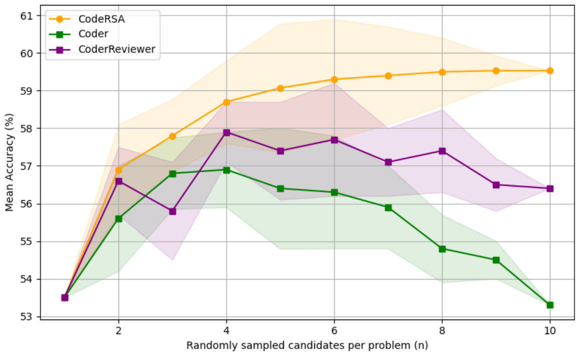


Figure 8: **MBPP: Mean accuracy vs. number of randomly sampled candidates per problem** (n)**.** Curves show CodeRSA (orange), Coder (green), and CoderReviewer (purple). Shaded regions indicate the standard deviation across multiple random samplings.

its reliability when generation stochasticity varies across runs or models.

In our main experiments, we fixed $n = 10$ candidates per problem as a practical balance between computational cost and runtime. The results here further suggest that model performance is not strongly dependent on candidate set size. Future work could explore larger candidate pools when computational resources permit.

A.5 An example of clustering

In the following presentation, each item is denoted in the format:

code_X : Instruction generated from this code

This means that the left-hand side (`code_X`) represents the identifier of the function implementation, and the right-hand side is the instruction generated based on it.

Main Cluster: Maximum value with frequency condition

- **code_1**: Create a function that takes a list of numbers. Returns the largest number that appears at least as many times as its value in the list. If no such number exists, return -1.
- **code_6**: Create a function that takes a list of integers and returns the maximum value that appears at least as many times as its value. If no such value exists, return -1.
- **code_8**: Create a function that takes a list of numbers and returns the maximum integer that occurs at least as many times as its value. If multiple such numbers exist, return the largest one. If no such number exists, return -1.

Cluster 2: Most frequent element

- **code_5**: Create a function that takes a list of integers. Returns the number that appears most frequently in the list. If there are multiple such numbers with the same frequency, return the largest one.
- **code_10**: Create a function that takes a list of integers. Returns the most frequent integer greater than 0. If multiple integers have the same highest frequency, return the smallest one. If the list is empty, return -1.

Cluster 3: Repeated integers

- **code_4**: Create a function that takes a list of integers and returns the smallest positive integer that appears more than once. If no such integer exists, return -1.
- **code_7**: Create a function that takes a list of integers and returns the maximum value that appears more than once. If no such value exists, return -1.

Cluster 4: Missing positive integer

- **code_3**: Create a function that takes a list of integers and returns the first missing positive integer. If the list is empty, return -1.

Cluster 5: First/last integer with frequency condition

- **code_2**: Create a function that takes a list of integers and finds the first integer that occurs at least as many times as its value. If no such integer is found, return None.
- **code_9**: Create a function that takes a list of integers. Returns the first integer that appears at least as many times as its value in the list. If no such integer exists, return None.

A.6 Prompt Used

A.6.1 For Generating the Additional Instruction:

```
##Write an instruction for given python function##
### Function start ###
def any_int(x, y, z):
    if isinstance(x,int) and isinstance(y,int) and isinstance(z,int):
        if (x+y==z) or (x+z==y) or (y+z==x):
            return True
        return False
    return False
### Function end ###

### instruction start ###
Create a function that takes 3 numbers. Returns true if one of the numbers is equal to the
sum of the other two, and all numbers are integers. Returns false in any other cases.
### instruction end ###

### Function start ###
any function
### Function end ###

###instruction start###
```

A.6.2 For Calculating the Reviewer Score (An Example):

```
def any_int(x, y, z):
    if isinstance(x,int) and isinstance(y,int) and isinstance(z,int):
        if (x+y==z) or (x+z==y) or (y+z==x):
            return True
        return False
    return False

# Write a docstring for the above function
Create a function that takes 3 numbers. Returns true if one of the numbers is equal to the
sum of the other two, and all numbers are integers. Returns false in any other cases.
```