# Success is in the Details: Evaluate and Enhance Details Sensitivity of Code LLMs through Counterfactuals

Anonymous ACL submission

## Abstract

Code Sensitivity refers to the ability of Code LLMs to recognize and respond to details changes in problem descriptions. While current code benchmarks and instruction data focus on difficulty and diversity, sensitivity is overlooked. We first introduce the CTF-Code benchmark, constructed using counterfactual perturbations, minimizing input changes while maximizing output changes. The evaluation shows that many LLMs have a more than 10% performance drop compared to the original problems. To fully utilize sensitivity, CTF-Instruct, an incremental instruction fine-tuning framework, extends on existing data and uses a selection mechanism to meet the three dimensions of difficulty, diversity, and sensitivity. Experiments show that LLMs fine-tuned with CTF-Instruct data achieve over a 2% improvement on CTF-Code, and more than a 10% performance boost on LiveCodeBench, validating the feasibility of enhancing LLMs' sensitivity to improve performance.

## 1 Introduction

Code generation is essential for enhancing software engineering efficiency (Zhu et al., 2024b), and also a crucial measure of intelligence (OpenAI, 2024). To increase code capabilities, Code Large Language Models (Code LLMs) are developed by pre-training on large-scale code corpora (Hui et al., 2024; DeepSeek-AI et al., 2024). Successful generation requires Code LLMs to accurately map between requirements and algorithmic logic (MacLennan, 1986; Pressman, 2005). A small mismatch will cause the whole task to fail. In Figure 1, changing the description from '**add** one' to '**double** one' alters the underlying algorithmic logic entirely (see detailed explanation in the figure caption). As such, the model's sensitivity to detail becomes a crucial measure of its ability.

However, the ability of Code LLMs to capture and address such fine-grained differences remains unclear. Existing code generation benchmarks primarily emphasize difficulty and diversity. For difficulty, tasks range from basic functions to competition-level algorithms (Liu et al., 2023; Du et al., 2024; Jain et al., 2024). For diversity, benchmarks cover data science, system development, and interdisciplinary applications and so many domains (Zhuo et al., 2024; Lai et al., 2023a; Hu et al., 2024; Liu et al., 2024). However, these benchmarks evaluate LLMs on isolated tasks without assessing their sensitivity to subtle differences in requirement details. This limitation extends to code instruction fine-tuning. Most approaches augment training data along the axes of difficulty and diversity: (1) by incrementally introducing constraints to synthesize harder tasks (Luo et al., 2024b; Wang et al.,

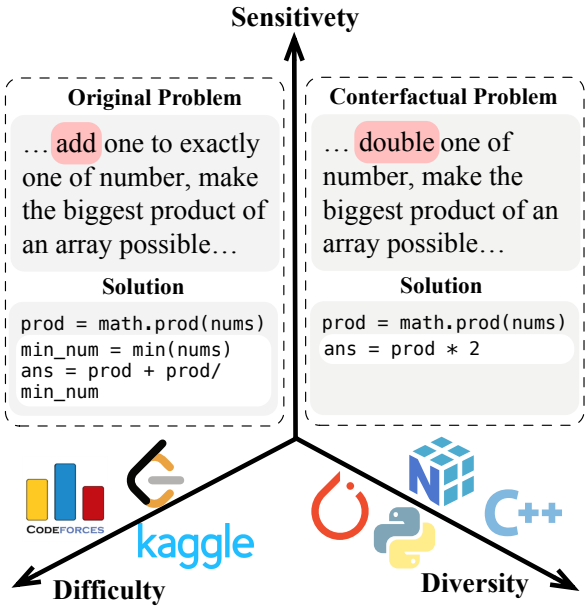


Figure 1: While diversity and difficulty have been explored, sensitivity to problem details remains underexplored. In the original problem, the smallest number should be increased, whereas in the counterfactual version, no matter which number is modified, the result remains the same—double the cumulative sum.

2024a),(2) by rewriting or drawing inspiration from real-world code to produce more diverse samples and broaden domain coverage (Wei et al., 2024; Luo et al., 2024a; Wei et al.; Yu et al., 2024). In contrast, constructing datasets that exploit models' sensitivity remains underexplored.

In response to these gaps, we first introduce the CTF-Code benchmark. The inspiration is from counterfactual studies in NLP (Chen et al., 2023; Sachdeva et al., 2024; Wang et al., 2024c), which make minimal changes to inputs to produce outputs that differ substantially. Concretely, some variations of original problems are sampled at first. Then, algorithm experts write solutions of solvable variations and select CTF problems which most preserve superficial task similarity and alter the algorithmic logic. After, the test inputs of original problems are executed on CTF solutions to generate new outputs to construct CTF test cases. Last, mainstream LLMs are evaluated on the complete CTF-Code benchmark. Experiments reveal that state-of-the-art models like GPT-4o and Qwen2.5-Coder (Hurst et al., 2024; Hui et al., 2024) experience performance drops exceeding 10% on CTF-Code compared to original problems, highlighting significant 'blind spots' in detail sensitivity.

Furthermore, we introduce the CTF-Instruct pipeline for three-dimensional data construction. Starting from an existing dimension (such as difficulty), CTF data are generate to cover the sensitivity dimension. Then, a selection mechanism is applied to the sensitivity-enhanced data to complete the third dimension (e.g., diversity). Finally, the original base-dimension data are merged with the selected sensitivity data to obtain a dataset that is complete across all three dimensions. Experiments show that LLMs fine-tuned with CTF-Instruct data achieve a 2.6% improvement on CTF-Code, and gains on other benchmarks such as HumanEval+ (+4.2%), BigCodeBench-hard (+5.2%), and LiveCodeBench (+11.6%) (Liu et al., 2023; Zhuo et al., 2024; Jain et al., 2024), confirming the help of sensitivity to code abilities.

Our contribution is summarized below:

- We propose CTF-Code, the first benchmark focused on sensitivity, and the evaluation results expose the shortcomings of mainstream Code LLMs in understanding requirement details.
- We design a three-dimensional-completed data generation framework, starting from one dimension, completing sensitivity by generation and the last dimension by selection.
- LLMs trained with CTF-Instruct data achieve substantial performance improvements across CTF-Code and other benchmarks compared to existing methods.

## 2 Related Work

**Code Benchmark** Existing code generation benchmarks primarily include two dimensions: (1) Difficulty: from function-level (Austin et al., 2021; Chen et al., 2021), to class-level (Du et al., 2023), and to contest-level (Jain et al., 2024); (2) Diversity: BigCodeBench (Zhuo et al., 2024) focuses on Python package usage, DS-1000 (Lai et al., 2023b) targets data science, while MultiPLE (Cassano et al., 2023) evaluates multilingual code generation. However, these benchmarks do not address sensitivity, which evaluates a model's ability to handle subtle but critical changes in task requirements. This differs from robustness, which measures the model's ability to produce stable outputs under non-critical changes (e.g., noise or rephrasing) in input (Li et al., 2025; Lin et al., 2025; Wang et al., 2023a) In this work, the first sensitivity benchmark, CTF-Code is introduced.

**Code Instruction Tuning Datasets** Most methods on code instruction tuning data augmentation (Wang et al., 2023b) mainly focus on difficulty enhancement and diversity expansion. Luo et al. (2024b); Xu et al. (2023) increase the difficulty of data by adding constraints to seed data (Chaudhary, 2023). Considering that the seed data may limit the diversity of generated data, Wei et al. (2024); Yu et al. (2024) rewrite real-world data to better align real distributions, thereby avoiding model bias and enhancing the diversity. To combine both dimensions, existing approaches typically adopt multi-stage training (Wei et al., 2024; Wang et al., 2024a) or data mixing strategies (Zheng et al., 2024; Yu et al., 2024; Wu et al., 2024b). While these methods have achieved significant success, the usage and combination of sensitivity is overlooked.

**Counterfactual in NLP** Counterfactuals in NLP aim to explore the model's output variation patterns through minimal input perturbations (Robeer et al., 2021; Nguyen et al., 2024; Sachdeva et al., 2024; Wang et al., 2024c). Unlike adversarial attacks, which introduce subtle, malicious inputs to

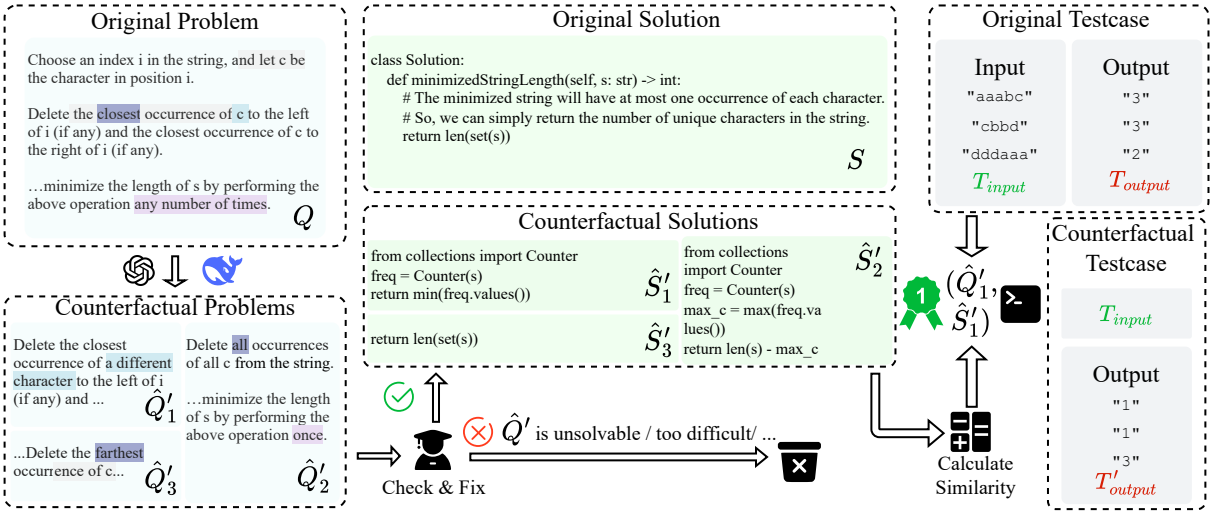


Figure 2: The pipeline of CTF-Code benchmark construction. First, original problems are sent to LLMs to sample semantic permutations on the problem description. Algorithmic experts will carefully check the CTF problems and decide to drop or fix them to generate CTF solutions. After selecting the most suitable CTF problem, its testcases are constructed by executing its solution on the inputs from the original testcases.

mislead the model into producing incorrect or unsafe outputs (Jenko et al., 2025), counterfactuals refer to make small but critical semantic changes in the input and prompt the model to detect and adjust output accordingly. Existing studies in the code domain mainly focus on local modifications to code (Hooda et al., 2024), such as flipping conditional statements or modifying logical operators, testing the model's ability to differentiate and understand counterfactual code (Gu et al., 2024; Cito et al., 2022). To the best of our knowledge, no prior work has leveraged counterfactual perturbations at the problem level to study models' sensitivity to requirement details.

## 3 CTF-Code Benchmark

### 3.1 Formal Definition

Evaluation for code tasks is test-driven, with its basic unit formalized as a tuple $\mathcal{P} = (Q, T, S)$, where $Q$ is the problem description, $T = \{t_i\}_{i=1}^{n}$ is a set of test cases, with each $t_i = (\text{input}_i, \text{output}_i)$, and $S$ is a solution that satisfies $\forall t_i \in T, S(\text{input}_i) = \text{output}_i$. Based on this, the goal of generating counterfactual problem can be formulated as an optimization problem. Given the original problem $\mathcal{P}$, generate $\mathcal{P}' = (Q', T', S')$ such that

$$\begin{aligned} &\underset{Q', S'}{\text{maximize}} \quad \mathcal{D}_S(S, S') \\ &\text{subject to} \quad \mathcal{D}_Q(Q, Q') \leq \epsilon \end{aligned} \tag{1}$$

where $\mathcal{D}_Q$ is the description similarity function, $\mathcal{D}_S$ is the solution difference function, and $\epsilon$ is the similarity threshold. We use the normalized Levenshtein distance (Levenshtein et al., 1966) as $\mathcal{D}_Q$, and define $\mathcal{D}_S$ as one minus the cosine similarity between code embeddings. This optimization objective ensures that $Q'$ is highly similar to $Q$, while $S'$ and $S$ differ significantly. After obtaining $Q'$ and $S'$, the new test cases $T'$ are constructed.

| | Acc. | Problems | Length |
|---|---|---|---|
| Humaneval | **96.3** | 164 | 71.6 |
| LCB-Easy | 95.6 | **215** | **210.5** |

Table 1: Comparison between HumanEval and LiveCodeBench (LCB) -Easy. **Acc.** represents the Pass@1 score of o1-mini on both benchmarks. **Problems** indicates the number of problems, **Length** represents the average word count per problem description.

### 3.2 Benchmark Construction

As shown in Figure 2, the construction of CTF-Code follows a three-phase paradigm: First, select problems that have a large semantic space as the original problem $\mathcal{P}$. Then, apply semantic perturbations to generate CTF description $Q'$ and derive the CTF pairs $Q', S'$ based on the optimization objective. Finally, construct the new test cases $T'$ while ensuring no data bias.

**Original Data Selection** The easy subset of LiveCodeBench (LCB) (Jain et al., 2024) is selected as $\mathcal{P}$. Table 1 shows that LLMs can solve nearly all problems in this subset, minimizing the impact

of difficulty. Additionally, the problem length is 210.5 words, could give more semantic space for perturbations. Furthermore, algorithmic competition problems require participants to carefully consider every detail and boundary condition, where even small deviations can lead to wrong answers. This aligns perfectly with the goal of sensitivity.

**CTF Pair Generation** This step aims to generate $Q'$ and $S'$. Given the complexity, a heuristic generation-selection strategy is proposed to approximate the solution of Equation 1. Based on $Q$, several LLMs sample $K$ candidates $\{\hat{Q}'_k\}_{k=1}^{K}$ using the prompt in Figure 15 in Appendix C. Specifically, after comparing to other LLMs, the best-performing LLMs, including `gpt-4o`, `gpt-4turbo`, and `o1-mini`, each generate five samples, as further sampling primarily yielded duplicates. After reviewing all $\hat{Q}'_k$, we empirically set $\epsilon = 0.13$, which balances formal similarity with allowance for semantic divergence. Only $\hat{Q}'_k$ satisfied $\mathcal{D}_Q(Q, \hat{Q}'_k) \leq \epsilon$ are retained.

However, retained $\hat{Q}'_k$ may be unsolvable or too difficult. Four competition programmers are invited to annotate, each of whom has at least a bronze medal in ICPC [1]. The detailed annotation process is in Appendix B. Annotators are required to read $\hat{Q}'_k$ and judge:(1)solvability, (2) if it is a CTF problem (filter problems which different descriptions yielding identical solutions), and (3) if its difficulty changed from $Q'$. Prior to the annotation, a 10-problem trial is conducted to ensure annotator consistency. Each problem is then independently annotated by two programmers. Where annotations disagree, a third annotator provides a new judgment, and the outcome is determined by majority vote. For the passed $\hat{Q}'_k$, annotators then write $\hat{S}'_k$. The pair $(\hat{Q}'_k, \hat{S}'_k)$ that maximizes Equation 2 is selected as $(Q', S')$.

$$\arg\max_{(\hat{Q}'_k, \hat{S}'_k)} \left[ \mathcal{D}_S(S, \hat{S}'_k) - \lambda \mathcal{D}_Q(Q, \hat{Q}'_k) \right]. \quad (2)$$

$\lambda$ is a scaling factor that ensures $\mathcal{D}_S$ and $\mathcal{D}_Q$ can compute. It is set as 1.2. Through this heuristic rule, we obtain an approximate optimal $Q'$ and $S'$.

**CTF Testcase Completion** To ensure the performance change of LLMs latter only from details change between $(Q, Q')$, a dual-constraint test case generation mechanism is designed to avoid the influence from $(T, T')$. **Input Space Inheritance**: We retain the original testcases' input distribution, i.e., $T'_{\text{input}} = T_{\text{input}} = \{input_i\}_{i=1}^{n}$. **Output Space Reconstruction**: The expected output is generated based on the new solution $S'$, i.e., for each $\text{input}_i \in T_{\text{input}}$, $\text{output}'_i = S'(\text{input}_i)$. Finally, $T' = \{(\text{input}_i, \text{output}'_i)\}_{i=1}^{n}$ is constructed. The data distribution interference is eliminated by fixing the input variables, and the correctness of the test case is ensured by the correctness of $S'$. Additionally, fixed inputs enable backtracking when the LLM behavior differs between $Q, Q'$.

Compared to the traditional code benchmarks that evaluate isolated problems, CTF-Code introduces paired data with only details differences to enable analysis of sensitivity for the first time, as shown in Figure 11 and Figure 13. Ultimately, CTF-Code curated a set of 186 problems.

## 4 CTF-Instruct

Unlike difficulty and diversity, detail sensitivity has not been explored in existing instruction datasets. To address the gap, an incremental data construction approach, CTF-Instruct, is proposed. Starting with datasets that satisfy a single dimension (e.g., difficulty), sensitivity data are generated through counterfactual perturbations. Then, a selection algorithm based on the third dimension (diversity) is applied, ultimately constructing a dataset that cover all three dimensions.

### 4.1 Generation

We first tried generating paired sensitivity data from scratch, but we found that the generated problems are too easy and repetitive. As the existence of high-quality data like Evol-Instruct (110k), which satisfies the difficulty dimension, incrementally expanding the data is more effective and efficient. Prompt 16 with `gpt-4-turbo` is applied to generate sensitivity pair $\mathcal{D}_{sens}$ based on the difficulty data $\mathcal{D}_{diff}$. After generation, duplicates found in existing benchmarks are removed to avoid data leakage following Luo et al. (2024b).

The 102k generated $\mathcal{D}_{sens}$ are evaluated on difficulty and diversity. For difficulty, we follow Wang et al. (2024b) and use their trained scorer to assign 1–5 scores to $Q \in \mathcal{D}_{diff}$, $Q' \in \mathcal{D}_{sens}$. 99% of the absolute score difference of $Q'$ and its seed $Q$ is less than 1 in Table 2, indicating minimal difficulty shift. For diversity, we compute the cosine similarity between embeddings of $(Q, S)$ and $(Q', S')$, extracted by DeepSeek-Coder 1.3B (Guo et al., 2024).

[1] International Collegiate Programming Contest

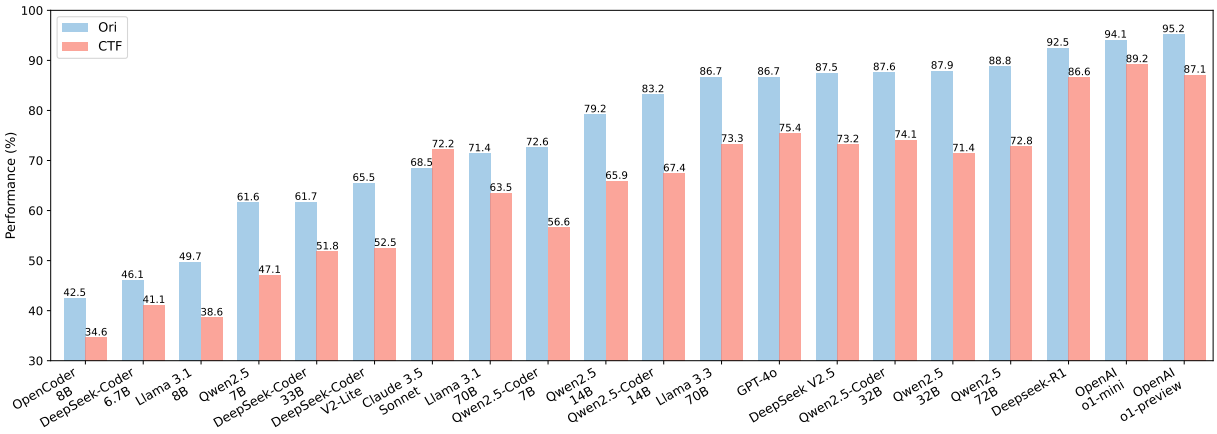


Figure 3: The evaluation results of Code LLMs on CTF-Code.

| Percentile | Similarity | Difficulty Difference |
|---|---|---|
| 25% | 0.99 | 0.00 |
| 50% | 0.97 | 0.01 |
| 75% | 0.92 | 0.14 |
| 95% | 0.64 | 0.89 |

Table 2: Percentile values for the semantic embedding similarity and difficulty difference distributions of original and sensitive data. The Similarity column indicates the percentage of values greater than the given threshold, while the Difficulty Difference column represents the percentage of values less than the given threshold.

The model is tuned on Code-Feedback (Zheng et al., 2024) for one epoch for alignment. Each embedding is the average of the final-layer token hidden states of $S$. As shown in Table 2, 75% of samples have similarity above 0.92. These results suggest that counterfactual generation preserves difficulty and diversity, and more importantly, that the sensitivity dimension is relatively independent of the other two dimensions.

### 4.2 Selection

Since $\mathcal{D}_{sens}$ is generated based on $\mathcal{D}_{diff}$, they naturally share similar diversity limitations. Directly merging them would amplify this bias. However, we can select a subset of $\mathcal{D}_{sens}$ that maximizes diversity relative to $\mathcal{D}_{diff}$. This introduces a distribution shift, partially mitigating the diversity deficiency. We use the semantic embeddings computed in Section 4.1, and apply the k-center greedy (Algorithm 1) to select the most diverse subset of $\mathcal{D}_{sens}$. Notably, some outlier samples (meaningless or corrupted) exhibit extremely large semantic distances. These are excluded by sorting based on distance and removing the tail. We empirically set the subset size $|\mathcal{D}^{sub}_{sens}| = 30$k. By merging $\mathcal{D}^{sub}_{sens}$ with $\mathcal{D}_{diff}$, we obtain the dataset CTF-Instruct that satisfies all three dimensions. An example of this distributional adjustment is shown in Figure 8 in Appendix A.

A similar process is applied when starting from diversity data, Oss-Instruct (75k). After 73k sensitivity data generated by `gpt-3.5-turbo`, we apply the difficulty scorer in Section 4.1 and retain only the 10k-size subset with the highest difficulty scores. After combined the subset with the Oss-Instruct, we get CTF-Instruct$_{\text{oss}}$. In both cases, sensitive data is first generated from the existing dimension, and a selection algorithm is used to fill the remaining missing dimension.

## 5 Experiment

### 5.1 CTF-Code Benchmark

**Models** We evaluate Qwen 2.5 Coder, Deepseek Coder v1 & 2, OpenCoder, Qwen 2.5, Llama 3.1 & 3.3, GPT-4o (`gpt-4o-2024-08-06`), Claude 3.5 Sonnet (`claude-3.5-sonnet-20240620`), o1-mini, o1-preview and Deepseek-R1 (Hui et al., 2024; Guo et al., 2024; Qwen et al., 2025; Zhu et al., 2024a; Huang et al., 2024; Grattafiori et al., 2024; Guo et al., 2025).

**Evaluation** As shown in Figure 3, most LLMs exhibit a significant performance drop on CTF-Code, often exceeding 15%. Reasoning-oriented LLMs such as R1 and O1 experience notably smaller drops, suggesting a stronger ability to capture fine-grained variations in problem requirements. This gap is especially obvious in problems that can be simplified or transformed. Reasoning LLMs tend to abstract key properties to reformulate the problem, whereas other LLMs are more

| Base | Model | EvalPlus | LiveCodeBench | | BigCodeBench | | CTF-Code | |
|---|---|---|---|---|---|---|---|---|
| | | HumanEval (+) | All | Easy | Full | Hard | Ori | CTF |
| DeepSeek Coder 6.7B | DC-6.7B-Instruct | 74.4 (71.3) | 18.9 | 45.3 | 35.5 | 10.1 | 45.8 | 38.1 |
| | Wavecoder | 75.0 (69.5) | 18.9 | 46.0 | 33.9 | 12.8 | 47.7 | 39.2 |
| | Inversecoder | 76.2 (72.0) | 18.1 | 43.1 | 35.9 | 10.8 | 47.8 | 39.1 |
| | Magicoder | 76.8 (71.3) | 19.2 | 46.6 | 36.2 | 13.5 | 48.8 | 43.4 |
| | CTFCoder | **78.7** (**75.0**) | **21.4** | **53.3** | **37.6** | **14.2** | **52.8** | **44.5** |
| | $\text{CTFCoder}_{\text{oss}}$ | 71.3 (65.9) | 18.3 | 46.6 | 37.0 | 12.2 | 51.4 | 43.1 |
| Qwen2.5 Coder 14B | Evol | 85.4 (79.3) | 23.9 | 71.5 | 43.7 | 14.2 | 76.3 | 59.9 |
| | CTF | **88.4** (**80.5**) | **24.6** | **74.1** | 44.1 | **17.6** | **79.5** | **60.8** |
| | w/o select | 85.4 (78.0) | 24.1 | 72.8 | **44.2** | 16.2 | 76.4 | 60.2 |
| | Oss | 84.1 (77.4) | 20.6 | 61.8 | 42.0 | 12.2 | 75.5 | 58.6 |
| | $\text{CTF}_{\text{oss}}$ | 86.0 (79.9) | **22.3** | **67.9** | **42.5** | **18.9** | **78.2** | **60.0** |
| | w/o select | **86.6** (**80.5**) | 20.8 | 67.2 | **42.5** | 14.9 | 76.8 | 59.2 |

Table 3: Performance comparison of CTFCoder with other models. To avoid environmental discrepancies, the official leaderboard results are presented. Only when results are missing, local testing are conducted. 'w/o select' means original data mix random selected sensitive data, without methods in Section 4.2.

likely to mimic the problem description, often becoming misled by the counterfactual phrasing. For LLMs families such as Qwen2.5-Coder, we observe that the sensitivity gap narrows with increasing model size, indicating a positive correlation between model scale and the sensitive ability. Interestingly, Claude-3.5-Sonnet even outperforms its original performance on CTF-Code, highlighting its strong generalization capabilities and practical robustness in code-related scenarios.

To further understand these results, we analyze common failure cases. The most frequent error is that models fail to recognize the semantic change in the CTF variant and instead solve it as if it were the original problem. This may be due to that the original or similar problems exist in the LLM's training data. Even when LLMs capture the details change, their performance often degrades on solvable yet uncommon CTF variants. Common issues include incorrect ordering of logical operators in `if` statements, confusion between data structures (e.g., lists and sets), and failure to handle boundary conditions. These problems are especially frequent in tasks that require case-by-case reasoning or involve numerous conditional branches.

Overall, our findings suggest that current LLMs still have substantial room for improvement in sensitivity to details. Misinterpreting such details not only leads to incorrect solutions but also disrupts the generation process itself. Enhancing sensitivity remains a crucial direction for advancing the performance and reliability of code LLMs.

### 5.2 Instruction Tuning

**Setup** CTFCoder and $\text{CTFCoder}_{\text{oss}}$ are obtained from using CTF-Instruct, $\text{CTF-Instruct}_{\text{oss}}$, respectively, finetuned on Deepseek Coder 6.7B base for 3 epochs. Qwen 2.5 Coder 14B Base (Hui et al., 2024) is also tuned. During training, the batch size is 512 and the sequence length is 2048. The initial learning rate is 2e-5 with 10 warmup steps, and the learning rate scheduler is cosine.

**Baseline & Benchmark** Other LLMs tuned on Deepseek Coder 6.7B Base are compared, including Deepseek Coder 6.7B Instruct (Guo et al., 2024), Magicoder (Wei et al., 2024), Wavecoder (Yu et al., 2024), and Inversecoder (Wu et al., 2024a). Qwen 2.5 Coder finetuned on the original Evol-Instruct and Oss-Instruct are the baselines.

The benchmarks cover a range of difficulty levels, including Humaneval(+) (Chen et al., 2021; Liu et al., 2023), and LiveCodeBench (Jain et al., 2024). Humaneval+ adds a lot of test cases to Humaneval to cover corner cases. LiveCodeBench collects algorithm problems from Online Judges and includes three difficulty levels: easy, medium, and hard. Since GPT-4-turbo's training data ends in December 2023, we test LiveCodeBench questions after January 2024. For diversity, BigCodeBench (Zhuo et al., 2024) is selected for Python package usage, and MultiPLE is for multilingual generation, in-

| Model | C# | C++ | Java | PHP | TypeScript | Bash | JavaScript | Avg |
|---|---|---|---|---|---|---|---|---|
| DC-6.7b-Instruct | 67.7 | 66.5 | **69.0** | 46.6 | 70.4 | 41.8 | 73.9 | 62.3 |
| Wavecoder | 69.0 | 57.8 | **69.0** | 52.2 | **74.2** | 39.9 | 70.8 | 61.8 |
| Inversecoder | 69.6 | 68.3 | 63.9 | 41.6 | 72.3 | 43.3 | 73.3 | 61.8 |
| Magicoder | 67.7 | **69.6** | 65.8 | 44.7 | 69.2 | 41.1 | 72.0 | 61.4 |
| CTFCoder | **72.2** | 67.1 | 65.2 | **53.4** | 73.0 | **43.7** | **74.5** | **64.2** |

Table 4: Performance comparison of various LLMs on different programming languages in MultiPLE.

cluding C#, C++, Java, PHP, TypeScript, Bash, and JavaScript. Additionally, BigCodeBench selects high-difficulty sub-data to form a Hard subset.

### 5.3 Results

Table 3 shows the performance comparison between CTFCoder and other LLMs. CTFCoder demonstrates consistent performance improvements across all benchmarks. Although previous models already cover difficulty and diversity and achieve strong performance, the addition of sensitivity acts like a further "activation". CTFCoder shows significant improvements across all three dimensions. On sensitivity, it has a nearly 3% improvement on CTF-Code, indicating that CTF indeed helps the model pay more attention to details. On difficulty, Humaneval+, BigCodeBench-Hard, and LiveCodeBench, CTFCoder achieves over 4%, 5%, and 11% performance improvements, respectively. On diversity, although CTF-Instruct is not explicitly designed for multilingual programming, it exhibits strong cross-language generalization. CTFCoder achieves the best performance on C#, PHP, Bash, and JavaScript in MultiPL-E, with a notable improvement of nearly 4% on C# and an average gain of 3% across languages in Table 4. Combined with results on BigCodeBench, these demonstrate that CTFCoder generalizes well across diverse domains and programming languages.

CTF-Instruct, building upon the difficulty dimension of Evol-Instruct, results in comprehensive enhancement. This illustrates that generating sensitivity data using existing data as seeds not only preserves the original data dimensions but can even trigger further improvements.

Even though $\text{CTFCoder}_{\text{oss}}$ has a relatively small amount of SFT data, $\text{CTF}_{\text{oss}}$ helps it outperform other models on LiveCodeBench-Easy and BigCodeBench-Full, reflecting the 'activation' effect on diversity works, too. On Qwen 2.5 Coder 14B, compare the baseline, random selection of CTF-Instruct data ('w/o select') and CTF generally

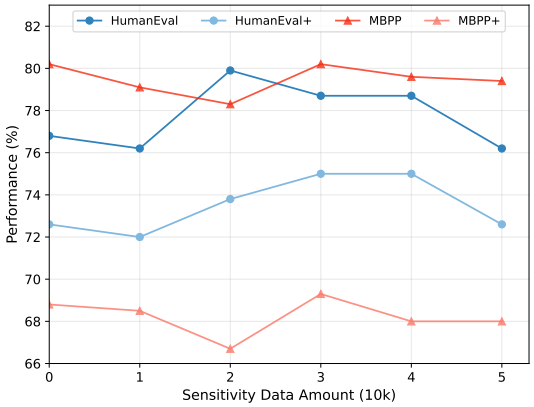


Figure 4: The change in model performance as sensitivity data joined into Evol-Instruct.

shows a progressive performance improvement, highlighting the effectiveness of sensitivity data and the importance of data selection.

## 6 Discussion

**There exists an optimal range for the amount of sensitivity data.** Figure 4 shows the performance trend when sensitivity data is gradually mixed into Evol-Instruct (110K), with the performance evolving in three stages: an initial decline, a mid-stage increase, and a final decline. The initial drop indicates that a certain amount of sensitive data is required to have an effect. The subsequent rise followed by a decline suggests that there is an upper limit for sensitivity data, confirming our observation that directly merging sensitivity and original data dimensions exacerbates the lack of the third dimension. Figure 7 in Appendix A also shows the results for Oss-Instruct (75k). However, it does not exhibit an initial performance drop. This may be because the diversity-oriented data is relatively easy to learn, and thus the addition of sensitive data does not introduce huge interference. However, when too much sensitive data is added, a decline similar to that observed before emerges.

**The effectiveness of the selection strategy is**

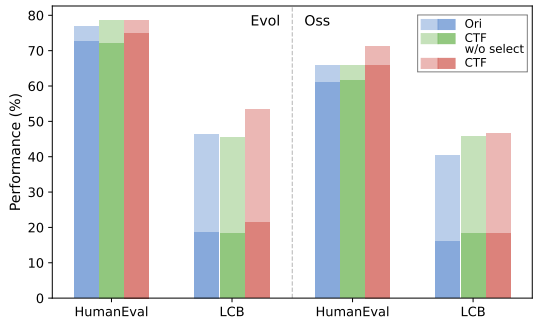


Figure 5: The performance change brought by the selection strategy on Evol-Instruct and Oss-Instruct. The darker shade in Humaneval represents Humaneval+, while in LCB, the darker shade represents LCB-All, and the lighter shade represents LCB-Easy.

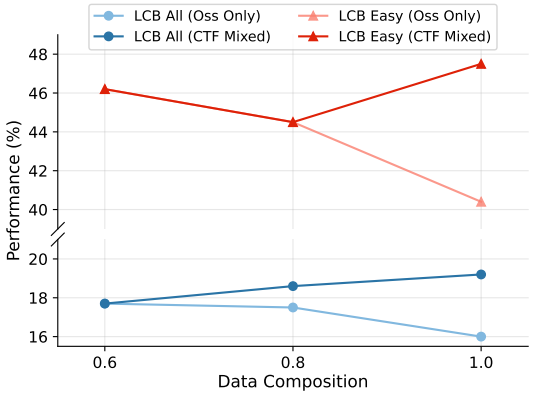


Figure 6: Performance changes with the increase in diversity data (Oss only) and the gradual injection of sensitivity into the diversity data (CTF Mixed).

| Epoch | Strategy | HE (+) | LCB (Easy) |
|---|---|---|---|
| 2 | 2+0 | 74.4 (69.5) | 18.5 (46.8) |
| | 1+1 | **79.9** (**75.0**) | **21.0** (**51.8**) |
| | 2+0 | 65.9 (61.0) | 16.0 (40.4) |
| | 1+1 | **66.5** (**61.6**) | **18.8** (**46.9**) |
| 3 | 3+0 | **76.8** (72.6) | 18.6 (46.4) |
| | 2+1 | **76.8** (**73.2**) | **20.7** (**51.3**) |
| | 3+0 | 65.2 (59.8) | 17.0 (42.5) |
| | 2+1 | **68.3** (**62.2**) | **19.0** (**47.2**) |

Table 5: The results of continual training with CTF. Epoch is the total number of training epochs, and 'x+y' indicates that the model is first trained for $x$ epochs on the original data, followed by $y$ epochs on CTF-Instruct. HE represents Humaneval, and LCB refers to LiveCodeBench-All, with 'Easy' inside the parentheses.

**universal.** Table 3 and Figure 5 compare the performance of different models and data using the selection strategy versus not using it ('w/o select') with the same amount of data. Regardless of the original data or base model, the strategy generally leads to performance improvement. Figure 5 shows that, with Evol-Instruct, performance on LiveCodeBench improved by over 17%, while for OSS-Instruct, performance on Humaneval increased by more than 7% compared to 'w/o select'. This validates our hypothesis that data offset can effectively address the third dimension.

**Under a fixed data amount, incorporating sensitive data still brings improvements.** Since the amount of data used for training the open-source models in the main experiment differs, we designed a controlled experiment to verify the independent gain from the sensitivity dimension. In Figure 6, when the total training data volume is fixed, replacing 40% of the original Oss-instruct data with randomly selected CTF data led to a 20% improvement on LCB-All, whereas simply increasing the Oss-instruct data volume caused a 9.5% performance drop.

**Sensitivity can be directly used for continual training.** Inspired by Magicoder (Wei et al., 2024), in Table 5, after training on the original data for 1 or 2 epochs, an additional epoch of CTF-Instruct is added. Compared to continuing training with the original data alone, this approach shows a significant performance improvement. Particularly on LiveCodeBench, every setup achieves a 10% gain. This further demonstrates the orthogonality of the sensitivity dimension with the other two dimensions, as its benefit does not depend on joint training, allowing for efficient and convenient continual training to achieve gains.

## 7 Conclusion

Beyond diversity and difficulty, we introduced sensitivity as a key dimension for evaluating and improving Code LLMs. By constructing the CTF-Code benchmark, we revealed the shortcomings of existing Code LLMs in understanding details. To futher utilize sensitivity, we propose the CTF-Instruct framework, which generates sensitivity data based on existing dimensions to cover sensitivity and employs a filtering algorithm to shift towards the third dimension. Experiments show that CTF-Instruct data fine-tuned LLMs improves performance on CTF-Code and outperform existing open-source models on general code generation benchmarks, validating the universal benefits of sensitivity optimization for Code LLMs.

## Limitation

Due to constraints in training resources and manpower, our work was limited to constructing a relatively modest set of CTF-Code problems, without exploring the potential for more complex or challenging examples. Additionally, the CTF-Instruct framework was not tested with multi-round generation, nor was it evaluated on larger, more advanced LLMs. While our experiments demonstrate the effectiveness of the proposed approach on the models tested, we acknowledge that the full potential of CTF-Instruct could be realized by scaling up the dataset and conducting more extensive fine-tuning experiments, particularly on models with greater capacity. Furthermore, the impact of training on larger models with more rounds of fine-tuning remains an open question and is a promising direction for future work.

## Ethical Considerations

The data for the proposed methods is drawn solely from publicly accessible project resources on reputable websites, ensuring that no sensitive information is included. Moreover, all datasets and baseline models used in our experiments are also available to the public. We have taken care to acknowledge the original authors by properly citing their work.

## References


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

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

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

Zeming Chen, Qiyue Gao, Antoine Bosselut, Ashish Sabharwal, and Kyle Richardson. 2023. DISCO: Distilling counterfactuals with large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5514–5528, Toronto, Canada. Association for Computational Linguistics.

Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2022. Counterfactual explanations for models of code. In *Proceedings of the 44th international conference on software engineering: software engineering in practice*, pages 125–134.

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. *Preprint*, arXiv:2406.11931.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The counterfeit conundrum:

Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024. Do large code models understand programming concepts? counterfactual analysis for code predicates. In *Forty-first International Conference on Machine Learning*.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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. *Preprint*, arXiv:2403.07974.

Slobodan Jenko, Niels Mündler, Jingxuan He, Mark Vero, and Martin Vechev. 2025. Black-box adversarial attacks on LLM-based code completion. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023a. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023b. 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.

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Zike Li, Mingwei Liu, Anji Li, Kaifeng He, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. Enhancing the robustness of llm-generated code: Empirical study and framework. *arXiv preprint arXiv:2503.20197*.

Feng Lin, Dong Jae Kim, Zhenhao Li, Jinqiu Yang, et al. 2025. Robunfr: Evaluating the robustness of large language models on non-functional requirements aware code generation. *arXiv preprint arXiv:2503.22851*.

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*, volume 36, pages 21558–21572. Curran Associates, Inc.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Xu Wang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024a. Semi-instruct: Bridging natural-instruct and self-instruct for code large language models. *Preprint*, arXiv:2403.00338.

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

Bruce J. MacLennan. 1986. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, USA.

Van Bach Nguyen, Christin Seifert, and Jörg Schlötterer. 2024. CEval: A benchmark for evaluating counterfactual text generation. In *Proceedings of the 17th International Natural Language Generation Conference*, pages 55–69, Tokyo, Japan. Association for Computational Linguistics.

OpenAI. 2024. Openai o1 system card.

Roger S Pressman. 2005. Software engineering: a practitioner's approach. *Pressman and Associates*.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Marcel Robeer, Floris Bex, and Ad Feelders. 2021. Generating realistic natural language counterfactuals. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3611–3625, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Rachneet Sachdeva, Martin Tutek, and Iryna Gurevych. 2024. CATfOOD: Counterfactual augmented training for improving out-of-domain performance and calibration. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1876–1898, St. Julian's, Malta. Association for Computational Linguistics.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2023a. Recode: Robustness evaluation of code generation models. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Yutao Mou, Mengdi Zhang, Jingang Wang, Xunliang Cai, et al. 2024a. Dolphcoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. *arXiv preprint arXiv:2402.09136*.

Yejie Wang, Keqing He, Dayuan Fu, Zhuoma Gongque, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, et al. 2024b. How do your code llms perform? empowering code instruction tuning with really good data. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14027–14043.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

Yongjie Wang, Xiaoqi Qiu, Yu Yue, Xu Guo, Zhiwei Zeng, Yuhong Feng, and Zhiqi Shen. 2024c. A survey on natural language counterfactual generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4798–4818, Miami, Florida, USA. Association for Computational Linguistics.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and LINGMING ZHANG. Selfcodealign: Self-alignment for code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and LINGMING ZHANG. 2024. Magicoder: Empowering code generation with OSS-instruct. In *Forty-first International Conference on Machine Learning*.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024a. Inversecoder: Self-improving instruction-tuned code llms with inverse-instruct. *Preprint*, arXiv:2407.05700.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, et al. 2024b. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct. *arXiv preprint arXiv:2407.05700*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *Preprint*, arXiv:2304.12244.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. 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 5140–5153. Association for Computational Linguistics.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12834–12859. Association for Computational Linguistics.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024a. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Qingfu Zhu, Xianzhen Luo, Fang Liu, Cuiyun Gao, and Wanxiang Che. 2024b. A survey on natural language processing for programming. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 1690–1704, Torino, Italia. ELRA and ICCL.

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. *Preprint*, arXiv:2406.15877.

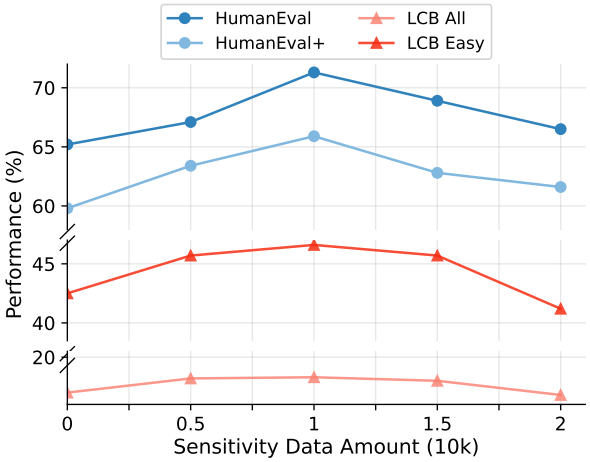


Figure 7: The performance varies with the amount of sensitivity data mixed into OSS-Instruct.

# Appendix

## A Supplement For CTF-Instruct

**Algorithm 1** K-Center Greedy Selection

**Input:** Sensitivity data $\mathcal{D}_{sens}$, needed data amount $\tau$, original data $\mathcal{D}_{base}$
**Output:** Set $\mathcal{D}_{sub} \subseteq \mathcal{D}_{sens}$ of $\tau$ data
$C \leftarrow \mathcal{D}_{base}$ ▷ Initialize centers
**for** $i = 1$ to $k$ **do**
    $\text{dist}_x \leftarrow \min_{y \in \mathcal{D}_{base} \cup \mathcal{D}_{sub}} ||\phi(x) - \phi(y)||_2$
    $x \leftarrow \arg\max_{x \in \mathcal{D}_{sens}} \text{dist}_x$ ▷ Select the farthest data $x$
    $\mathcal{D}_{sub} \leftarrow \mathcal{D}_{sub} \cup \{x\}$ ▷ Update centers
    $\mathcal{D}_{sens} \leftarrow \mathcal{D}_{sens} - \{x\}$ ▷ Update data
**end for**
**Return** $\mathcal{D}_{sub}$ ▷ Return the set of $\tau$ data

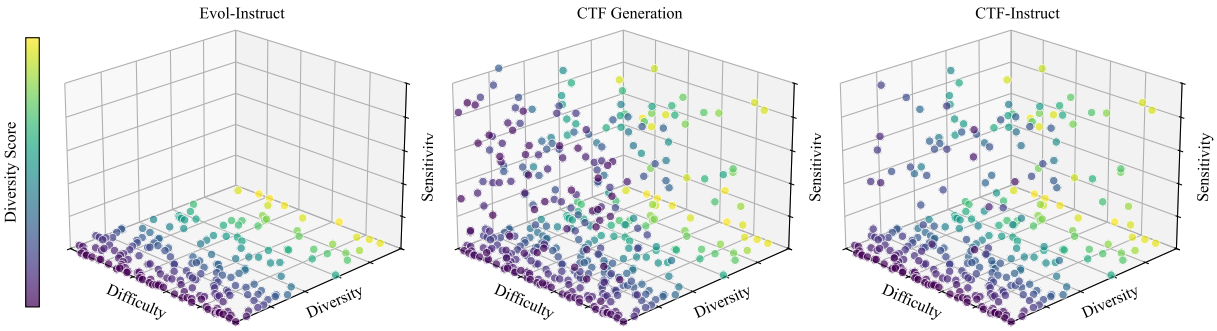


Figure 8: The data distribution change trace during the CTF-Instruct pipeline.

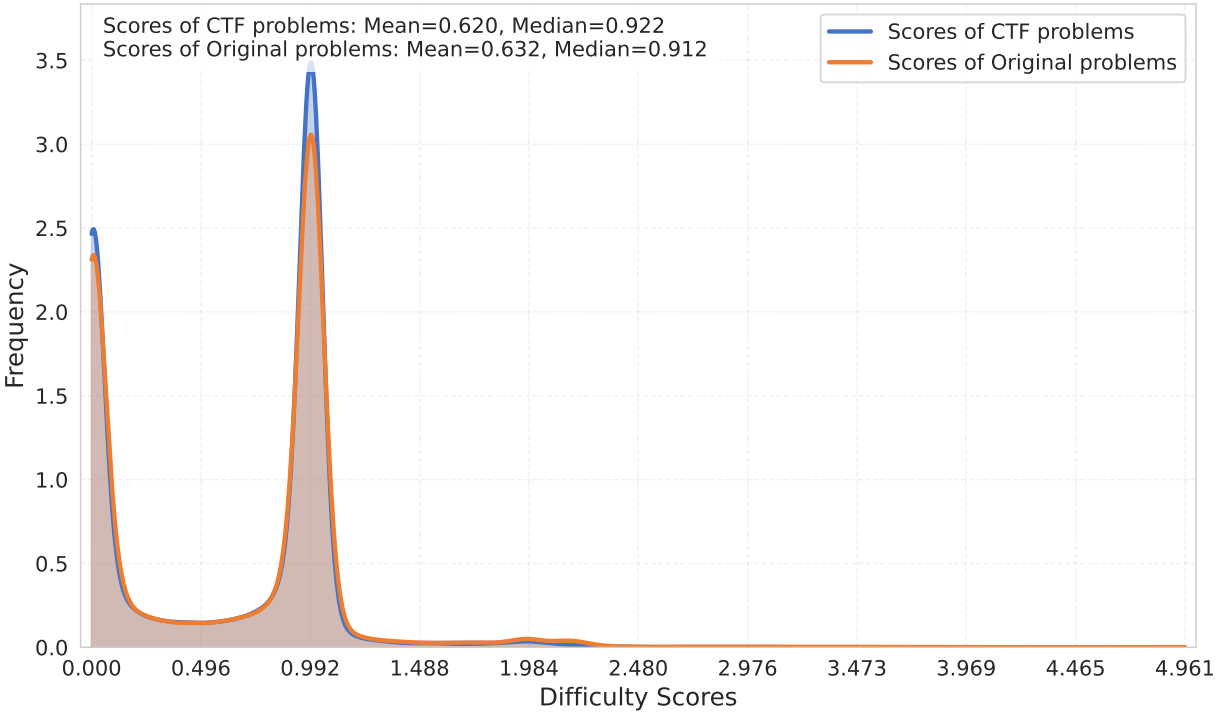


Figure 9: The distribution of difficulty scores of sensitive data and its original data.

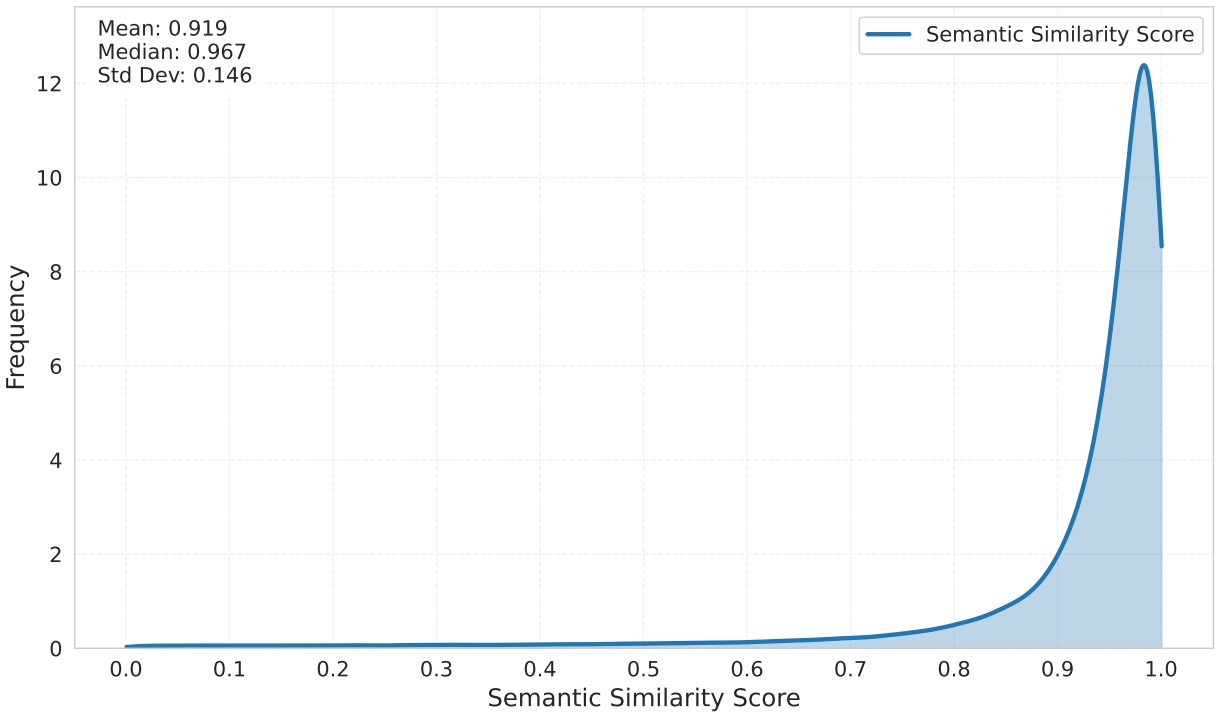


Figure 10: The distribution of the semantic embedding similarity scores of sensitive data and its original data.

## B Annotation Process

### B.1 Mission Background

Today, large language models (LLMs) demonstrate remarkable capabilities in code generation. However, it remains unclear how well LLMs can capture the nuances of programming problem details, such as the distinction between "swapping any two characters" and "swapping two adjacent characters". Can LLMs accurately capture the differences between these two concepts? To investigate this, we propose to modify a set of original problems (LeetCode Easy Level) to construct a new set of **counterfactual** (**CTF**) problems. These CTF problems are designed to have minor textual differences from the original problems while yielding significantly different solutions. To avoid the bias from test cases, we aim to ensure that CTF problems can utilize the test cases of the original problems without the need for reconstruction.

### B.2 Annotation Content

The annotation process does not involve modifying the problem itself, as this task has already been done by the LLMs. Instead, the annotator's role is simply to evaluate whether the modified problem is correct and aligns with our requirements.

### B.3 Construction Workflow

To illustrate the construction workflow in detail, we will supplement it with an example.

1. Read the original problem and briefly explain the meaning of the original problem. As shown in Figure 11, the meaning of the original problem is: "Given a string consisting of three letters `'abc'` in any order, can `'abc'` appear after swapping any two characters at most once?"

2. Read and understand the newly automatically generated problem. If there are errors in the `Sample Input/Output` or in the `Test Cases`, correct them.

3. In comparison with the original problem, classify the new problem into three types (Bad, Robust, CTF) and explain what changes have been made.

   - Bad. The new problem has a significant vulnerability (logical vulnerability or conflict) and can not be a complete problem.
   - Robust. The new problem has only a different wording from the original question, i.e. the algorithm used by the new problem and the answer is exactly the same. As shown in Figure 12, this is a robust version of the original problem. After understand the meaning of the new problem, we can tell that the change is "any substring can be reversed".
     For the new problem, the total length of cards is 3. Reversing a substring of length 3 is equivalent to swapping the letters in positions `1` and `3`, and position `2` will not be changed during the reversing process; reversing a substring of length 2 is equivalent to swapping adjacent letters in the original question. The operation of the original problem and the operation of the new problem are exactly the same. Therefore, the answers are completely consistent and do not need to be modified.
   - CTF. The new problem has only a small difference from the original problem, but it changes the meaning of the original problem, making the answers not exactly the same as the original problem (With not too much variation in difficulty, the more variation in answers the better).
     Figure 13 and Figure 14 are two example of CTF problems. The change of the former problem is "only two adjacent characters can be exchanged", and the change of the latter problem is "cards become `abcd`".

4. Determine whether new test cases need to be added to the CTF problem. For example, the annotator should determine whether the range of data of the new problem is fully consistent with the original problem, and whether the input of test cases of the original problem can be directly executed by the CTF problem.
   For the first CTF problem, there is no need to add new test cases, while for the second CTF problem, some new test cases should be added.

### B.4 Annotation Tabular

As shown in Table 6, we provide an example of the annotation table that the annotator should fill in.

```
## Question Content:

There are three cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$ placed in a row in
    some order. You can do the following operation at most once:


-  Pick two cards, and swap them.  Is it possible that the row becomes $\texttt{abc}$ after the
    operation? Output "YES" if it is possible, and "NO" otherwise.

Input

The first line contains a single integer $t$ ($1 \leq t \leq 6$)  the number of test cases.

The only line of each test case contains a single string consisting of each of the three
    characters $\texttt{a}$, $\texttt{b}$, and $\texttt{c}$ exactly once, representing the
    cards.

Output

For each test case, output "YES" if you can make the row $\texttt{abc}$ with at most one
    operation, or "NO" otherwise.

You can output the answer in any case (for example, the strings "yEs", "yes", "Yes" and "YES"
    will be recognized as a positive answer).Sample Input 1:
6

abc

acb

bac

bca

cab

cba

Sample Output 1:

YES
YES
YES
NO
NO
YES

Note

In the first test case, we don't need to do any operations, since the row is already
    $\texttt{abc}$.

In the second test case, we can swap $\texttt{c}$ and $\texttt{b}$: $\texttt{acb} \to
    \texttt{abc}$.

In the third test case, we can swap $\texttt{b}$ and $\texttt{a}$: $\texttt{bac} \to
    \texttt{abc}$.

In the fourth test case, it is impossible to make $\texttt{abc}$ using at most one operation.

## Starter Code:

## Test Cases:

"[{\"input\": \"6\\nabc\\nacb\\nbac\\nbca\\ncab\\ncba\\n\", \"output\":
    \"YES\\nYES\\nYES\\nNO\\nNO\\nYES\\n\", \"testtype\": \"stdin\"}]"
```

Figure 11: An example of the original problem.

```
## Question Content:

There are three cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$ placed in a row in
    some order. You can perform the following operation at most once:

- Choose any substring of the cards and reverse it.

Is it possible that the row becomes $\texttt{abc}$ after the operation? Output "YES" if it is
    possible, and "NO" otherwise.
...
```

Figure 12: An example of the robust version of the original problem.

```
## Question Content:
There are three cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$ placed in a row in
    some order. You can perform the following operation at most once:

- Pick two **adjacent** cards and swap them.

Is it possible that the row becomes $\texttt{abc}$ after the operation? Output "YES" if it is
    possible, and "NO" otherwise.
...
```

Figure 13: The first example of the CTF version of the original problem.

```
## Question Content:

There are four cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$, $\texttt{d}$ placed
    in a row in some order. You can do the following operation at most once:

- Pick two cards, and swap them.  Is it possible that the row becomes $\texttt{abcd}$ after the
    operation? Output "YES" if it is possible, and "NO" otherwise.
...
```

Figure 14: The second example of the CTF version of the original problem.

| Original Problem Index | Original Problem Meaning | Model | New Problem Index | New Problem Statement Error | New Problem Type | Modification | Add New Test Cases |
|---|---|---|---|---|---|---|---|
| 0 | Given a string composed of letters 'abc' in any order, exchange any two characters to see if string 'abc' can occur. | o1-mini | 0-0 | | Robust | | No |
| 0 | Given a string composed of letters 'abc' in any order, exchange any two characters to see if string 'abc' can occur. | o1-mini | 0-1 | | CTF | Only two adjacent characters can be exchanged | No |
| 1 | Add 1 to a number in an array of positive numbers, how to maximise the array product | o1-mini | 1-1 | | CTF | Replace a number in an array with a number from 0-9, how to make the array product maximum | No |
| 4 | A string with a phone number in front and 2 digits in the middle indicating age. Find those over 60 years old | o1-mini | 4-0 | | CTF | How many people are over 60 years old and have unique phone numbers? | Yes |
| 4 | A string with a phone number in front and 2 digits in the middle indicating age. Find those over 60 years old | o1-mini | 4-1 | | CTF | Age is hexadecimal | No |

Table 6: An example of the annotation table.

## C Prompt

This section shows the prompt used to instruct LLMs to generate desired counterfactual question and instruction tuning data.

```
Please create a **counterfactual** version of the given original python programming problem.
    Your goal is to **make a minimal change to the problem that leads to a significant change
    in the solution**. Follow these detailed steps:
1. Carefully read and comprehend the original problem's context, conditions, constraints, and
    requirements.
2. Identify a critical point in the original problem and think about a modification. **The
    modification should be slight but cause a substantial change in the solution approach**.
3. Consider the influence of the modification. Ask yourself: Would it change data structures or
    algorithms? Explain the influence before output the counterfactual problem. If the
    influence does not impact the solution approach significantly, rethink another critical
    point to modify. Repeat Step 2 and Step 3 until you find a point that satisfies the
    requirement.
4. Modify the original problem based on the most influential point. The modified problem must
    be consistent, clear, and requires a significantly different solution approach. Update the
    sample inputs and outputs to match the new problem condition.
5. Output the counterfactual problem, ensuring the following format:
    - Before the JSON format, include a section marker "###Counterfactual Problem".
    - After the section marker, provide the counterfactual problem in the same JSON format as
    the original, including "question_content", "starter_code", "public_test_cases", and
    "metadata".
### Original Problem
```

Figure 15: The prompt used to generate CTF-Code Problem.

```
Refine a code generation task, initially presented as #Original_Sample#, which is a JSON dict
    including three keys: a task instruction, and the output generated from  the instruction.
Your task is to produce a #Modified_Sample# by altering the original task instruction in a way
    that significantly changes the output, yet with minimal adjustments to the instruction
    itself.

## Requirements:
1. **Minimal Instruction Change**: Achieve the code change with minimal alterations to the
    instruction. The difference will be assessed through evaluated by the Rouge score,
    indicating the high similarity in wording, sentence structure, and length to the original.
2. **No Trival Changes to Instruction**: Ensure the modification to the instruction is
    semantic-relevant. Do not make trivial changes like adding or removing a word, changing the
    order of words, or replacing synonyms.
3. **Maximal Code Change**: Your adjustments should lead to considerable changes in the output,
    impacting aspects like algorithms, data structures, data and control flows, or boundary
    conditions. The difference will be assessed through both the Rouge score and AST score,
    indicating the output's functionality, implementation, and naming should substantially
    diverge from the original.
4. **Encourage Trival Code Change**: The code output should be significantly different. Change
    every aspect of the code, including the function name, variable names.

## Format:
1. Your output should be a #Modified_Sample# dict in **JSON format** as the #Original_Sample#
    is.
2. Using **markdown code snippet syntax** in the instruction and the output.
3. Ensure all characters are **properly escaped** in the JSON string.

## Examples:
{seeds}

## Question:
- Original_Sample:
```

Figure 16: The prompt used to generate CTF-Instruct data.