Enhancing Large Language Models in Coding Through Multi-Perspective Self-Consistency

Anonymous ACL submission

Abstract

Large language models (LLMs) have exhibited remarkable ability in code generation. However, generating the correct solution in a single attempt still remains a challenge. Prior works utilize *verification properties* in software engineering to verify and re-rank solutions in a majority voting manner. But the assumption behind them that generated verification properties have better qualities than solutions may not always hold. In this paper, we treat them equally as different *perspectives* of LLMs' reasoning processes. We propose the **Multi-Perspective Self-Consistency (MPSC)** framework incorporating both inter- and intra-consistency across outputs from multiple perspectives. Specifically, we prompt LLMs to generate diverse outputs from three perspectives, *Solution*, *Specification* and *Test case*, constructing a 3-partite graph. With two measure functions of consistency, we embed both inter- and intra-consistency information into the graph. The optimal choice of solutions is then determined based on analysis in the graph. MPSC significantly boosts performance of foundation models (ChatGPT in this paper) on various benchmarks, including HumanEval (+**15.91**%), MBPP (+**6.43**%) and CodeContests (+**9.37**%), even surpassing GPT-4.

1 Introduction

In recent years, pre-trained large language models (LLMs) have demonstrated unprecedented proficiency in understanding, generating, and reasoning with human language (Brown et al., 2020; Chowdhery et al., 2022; OpenAI, 2023; Touvron et al., 2023). Among the diverse applications of LLMs, code generation stands out as pivotal task and has been acknowledged as a fundamental task for benchmarking (Liang et al., 2023). This task entails models to generate source codes from provided natural language intents. Many foundation models have exhibited remarkable zero-shot performance in code generation, such as ChatGPT and GPT4 (OpenAI, 2023), with successful deployments in real-world applications like Github Copilot.

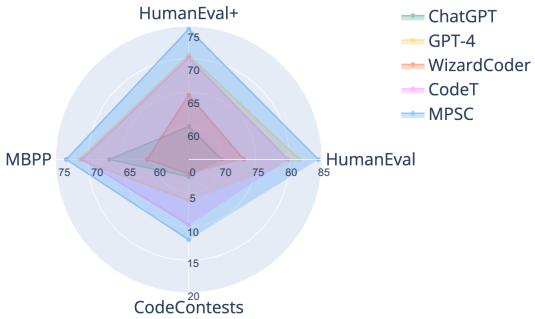

Figure 1: Pass@1 of MPSC. With ChatGPT as the foundation model, MPSC even surpasses GPT-4 and achieves SOTA performance on all four benchmarks.

Despite the remarkable abilities, LLMs often struggle to generate the correct code in a single attempt. Therefore, previous works sample diverse codes from LLMs and re-rank them by introducing *verification properties* from software engineering. For example, CodeT (Chen et al., 2022) generates test cases as a verification property, while ALGO (Zhang et al., 2023) generates oracles, the brute-force version of desired algorithms, as a verification property. These methods are essentially variants of majority voting, making the assumption that the correctness of experts (i.e. the verification properties) is better than that of choices (i.e. the desired code outputs). However, both verification properties and desired code outputs are usually generated by the identical model with respect to the same question, and hence the preference over verification properties is not always correct.

Instead, we believe that both desired code outputs and verification properties should be treated equally, since they are different *perspectives* of LLM's deliberate thinking process in face of identical questions. Aggregating various outputs from different perspectives can lead to a more credible result. To achieve this, we propose the **Multi-**

Perspective Self-Consistency (MPSC) framework that incorporates both inter-consistency across multiple perspectives and intra-consistency within a single perspective. In this way, MPSC can fully leverage the consistency information within LLMs and select the model output with the most consistent functionality as the final answer.

In our framework, various verification techniques from software engineering can be included as extended perspectives for better reasoning. Specifically, we prompt the LLM to simultaneously generate diverse outputs from three well-established perspectives in software engineering, namely *Solution*, *Specification* and *Test case* (Abrahamsson et al., 2017). Solutions implement the desired functionality, specifications demonstrate the intended properties in formal language, while test cases outline the expected behavior for some specific inputs. Then, we treat these model outputs as vertices in a graph, and establish connections (i.e. edges) based on the pairwise agreement of vertices from different perspectives. Our goal is to identify the most reliable output using a score function, which evaluates all vertices by considering both intra- and inter-consistency information encoded in the graph. Specifically, the intra-consistency information guides the function to favor the most internally consistent output within a single perspective, while inter-consistency ensures that the scores for two outputs from different perspectives are similar if they reach a consensus. We formalize the learning process of the score function as an optimization problem adhering to these two consistency criteria and leverage an iterative algorithm proposed by Zhou et al. (2003b) to achieve this goal.

We evaluate MPSC on four widely used code generation benchmarks, including HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023), MBPP (Austin et al., 2021) and CodeContests (Li et al., 2022). Experimental results show that our method boosts the performance of ChatGPT by a large margin, 15.91% in HumanEval, 15.64% in HumanEval+, 6.43% in MBPP and 9.37% in CodeContests[1]. Our framework even surpasses GPT-4 (OpenAI, 2023) as shown in Figure 1.

2 Multi-Perspective Self-Consistency

A single perspective can often lead to an incomplete understanding of a problem, akin to the parable of "blind men describing an elephant". The reasoning process of LLMs follows a similar pattern. LLMs generally cannot guarantee the correctness of generated output in a single attempt, especially in code generation, which necessitates proficient natural language understanding, deliberate reasoning and rigorous controllability.

However, a key aspect of human intelligence is the ability to think from multiple perspectives, resulting in a more comprehensive understanding of situations and more accurate solutions to problems. Inspired by the human cognition process, we propose a novel code generation method by reasoning from three well-established perspectives, solutions, specifications and test cases. Although noisy outputs may inevitably be included in the generated outputs of every perspective, we can leverage both intra-consistency and inter-consistency among the diverse outputs to distinguish the best ones from the noise. An overview of our proposed MPSC method is illustrated in Figure 2.

2.1 Solution, Specification and Test Case

Given a user intent in natural language, we introduce solution, specification and test case as three perspectives to describe the desired functionality. A *solution* is the source code implementing the functionality denoted as $g : \mathbb{X} \rightarrow \mathbb{Y}$, which is also the target of code generation. A *test case* is a pair of valid input and output satisfying the required functionality denoted as $(x, y) \in \mathbb{X} \times \mathbb{Y}$. *Specification* draws inspiration from *Formal Verification* in software engineering, which mathematically proves the correctness of one program by ensuring its satisfaction of some certain formal specifications. In the context of software engineering, formal verification is usually written in formal programming languages, e.g. Coq (Team, 2023) and Dafny (Leino, 2010), and conducted by accompanying verification tools. For the generalization of the proposed method in different program language scenarios, we adopt the idea of formal verification and limit the specifications within pre-conditions and post-conditions, which can be written as functions in the same programming language like solutions, without struggling with formal languages. Specifically, a pre-condition constrains the requirements that a valid input should satisfy, while a post-condition constrains the relationships that a pair of valid inputs and outputs should satisfy. We denote them as $h^{pre} : \mathbb{X} \rightarrow \{False, True\}$ and $h^{post} : \mathbb{X} \times \mathbb{Y} \rightarrow \{False, True\}$. Detailed examples of outputs are shown in Figure 2.

[1]We will release our code on GitHub.

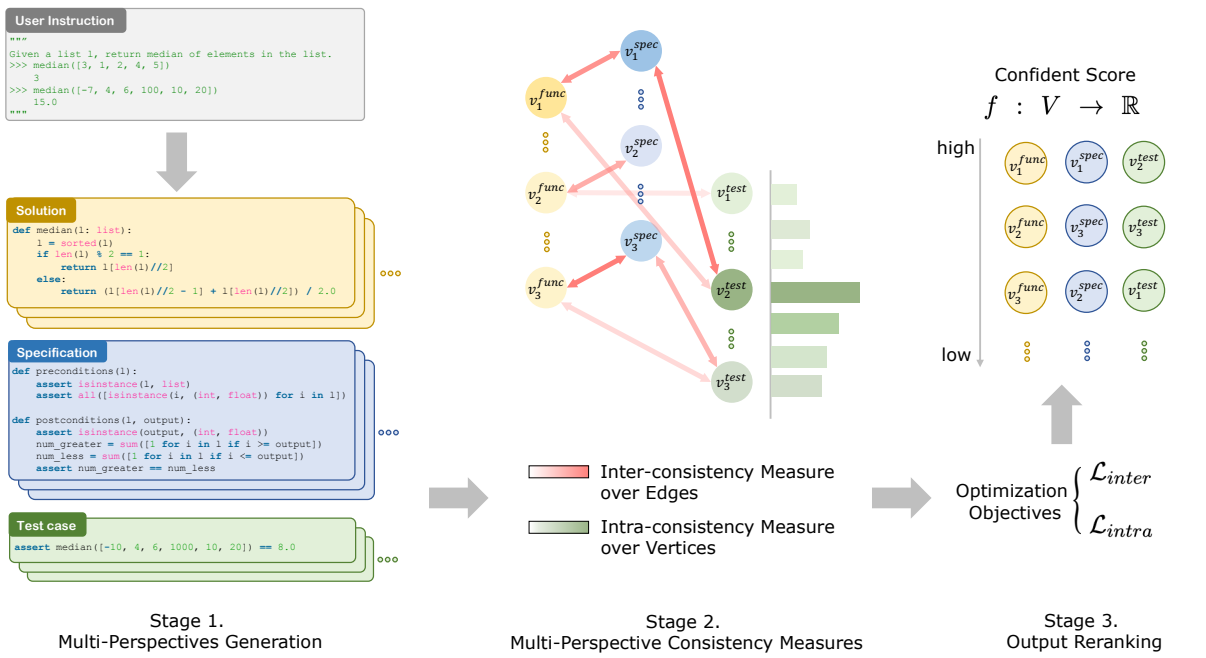

Figure 2: Overview of our MPSC code generation method. (a) Stage 1: we require a LLM to generate diverse solutions, specifications and test cases. A detailed example of the three perspectives of function `median(l)` from HumanEval is presented. (b) Stage 2: we construct a 3-partite graph based on the generated outputs and then calculate both inter- and intra-consistency measures over edges and vertices respectively. The magnitudes of measurements are demonstrated by the shade of colors. (c) Stage 3: Incorporating the multi-perspective consistency information, we then learn a score function to re-rank outputs within each perspective.

2.2 Graph Construction

We require LLMs to generate diverse outputs from all three perspectives. We employ an 3-partite graph representation to capture the relationships among these generated outputs. Specifically, we represent the generated solutions $\{g_1, g_2, ..., g_I\}$ with a vertex set V^{func}, the specification set $\{(h_1^{pre}, h_1^{post}), ..., (h_J^{pre}, h_J^{post})\}$ with V^{spec}, the test case set $\{(x_1, y_1), ..., (x_K, y_K)\}$ with V^{test}, and hence construct a vertex set $V = V^{func} \cup V^{spec} \cup V^{test}$. With edges connecting each pair of vertices from two distinct sets, we construct an undirected 3-partite graph $\mathcal{G} = (V, E)$. Our goal is to leverage the graph to encode the multi-perspective consistency information, and then learn a score function $f : V \rightarrow \mathbb{R}$ (also a vector $\boldsymbol{f}$, $f_i = f(v_i)$) from it to choose the most reliable output among all.

2.3 Inter-Consistency between Different Perspectives

We distinguish between two kinds of consistency based on the perspectives involved. Intra-consistency is defined as the degree to which a given output aligns with others within the same perspective, following the original definition in Wang et al. (2022). Conversely, inter-consistency is defined as the degree of consensus between a pair of outputs from two different perspectives.

With the well-established definitions of these three perspectives in software engineering, each output implicitly describes a latent functionality regardless of whether it is a solution, a specification or a test case. Consequently, we define the inter-consistency between two vertices from different perspectives as the alignment of their latent functionalities. And the most appealing aspect is that we can quantify the alignments with a code interpreter in a deterministic manner[2]. We formalize the inter-consistency as a measure function $\omega(\cdot, \cdot) : V \times V \rightarrow \mathbb{R}$ (also the adjacency matrix $\boldsymbol{W}$, where $W_{i,j} = \omega(v_i, v_j)$) to weight different edges in different ways as shown in Table 1.

We then derive an optimization objective based on inter-consistency measurements,

$$\mathcal{L}_{inter} = \sum_{(v_i, v_j) \in E} W_{i,j} (f(v_i) - f(v_j))^2 = \boldsymbol{f}^T \boldsymbol{L} \boldsymbol{f} \quad (1)$$

[2]We provide the Python code snippets implementing the verification in Appendix B.

Vertex Types	Expression of $\omega(v_i, v_j)$
$v_i \in V^{func}$, $v_j \in V^{spec}$	$\mathbb{E}_{x \in \mathbb{X}}[\mathbf{1}_{h_j^{pre}(x) \to h_j^{post}(x, g_i(x))}]$
$v_i \in V^{func}$, $v_j \in V^{test}$	$\mathbf{1}_{g_i(x_j)=y_j}$
$v_i \in V^{spec}$, $v_j \in V^{test}$	$\mathbf{1}_{h_i^{pre}(x_j) \wedge h_i^{post}(x_j, y_j)}$
otherwise	0

Table 1: Mathematical expressions of different inter-consistency measures $\omega(\cdot, \cdot)$.

, where $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$ is the laplacian matrix of the graph $\mathcal{G}$[3]. The loss function is the weighted sum of the local variation of each edge on the graph. An underlying assumption is that *a pair of outputs exhibiting consistency are either both correct or both incorrect*. Therefore, the difference between scores of two connected vertices should be constrained by the penalty corresponding to the degree of consistency, i.e. edge weight.

2.4 Intra-Consistency within the Same Perspective

Following Wang et al. (2022), we define the intra-consistency of one generated output as its similarity to others within the same perspective, which is denoted as a function $\varphi(\cdot) : V \to \mathbb{R}$ (also a vector $\boldsymbol{y}$, $y_i = \varphi(v_i)$).

Wang et al. (2022) limits the consistency to mere equalities in final answers, thereby lacking efficacy when applied to open-form tasks. In the scenario of code generation, we extend the scope of intra-consistency to lexical and semantic similarities.

Lexical intra-consistency by Bayes risk. Minimum Bayes risk decoding (Kumar and Byrne, 2004) selects the hypothesis $\mathrm{h} \in \mathbb{H}$ that minimizes the expected loss $R(\mathrm{h}) = \mathbb{E}_{\mathrm{y} \sim P(\mathrm{y})}[L(\mathrm{y}, \mathrm{h})]$ over the distribution of label y. Because of the unavailability of $P(\mathrm{y})$, $P(\mathrm{h})$ is usually used as a proxy distribution in practice. Then the Bayes risk can be rewritten as $R(\mathrm{h}) = \sum_{\mathrm{h}' \in \mathbb{H}} L(\mathrm{h}', \mathrm{h}) \cdot P(\mathrm{h}')$, which is in fact measure the consistency of h over the hypothesis space. Specifically, we utilize negative BLEU metrics (Papineni et al., 2002) as the loss function L aiming at lexical similarity and assume the hypothesis distribution is uniform, i.e.

$$\varphi(v_i) = C \cdot \sum_{v_j \in K(v_i)} \mathrm{BLEU}(v_i, v_j)$$

, where C is the normalizing constant so that measures of outputs in one perspective sum up to 1.

[3]In our experiment, we use the symmetric normalized Laplacian $\boldsymbol{L}^{sym} = \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{L} \boldsymbol{D}^{-\frac{1}{2}}$ for more robust performance.

Semantic intra-consistency by structural equivalence. In the realm of graph theory, two vertices are deemed structurally equivalent if they share identical weighted connections with the same third-party vertices. Utilizing this equivalence relation, we delineate V^{func}, V^{spec}, and V^{test} into their respective structural equivalence classes. Noted that the weights assigned to edges reflect the alignments of latent functionalities associated with the vertices, and hence outputs within each equivalence class can be regarded as exhibiting consistent functional behavior. Therefore, we define the intra-consistency measure based on the structural equivalence classes within each perspective. Suppose v_i belongs to the solution set V^{func}. The structural equivalence class of v_i is denoted as $S(v_i) \subset V^{func}$, and neighbor sets of v_i can be partitioned into two subsets $\{N_t(v_i) | t \in \{spec, test\}\}$ depending on the perspective they belong to. Overall, the lexical intra-consistency measure is defined as the multiplication of these three sets, i.e.

$$\varphi(v_i) = C \cdot |S(v_i)| \cdot \prod_t |N_t(v_i)|$$

, where C is the normalizing constant. The notation is similar when v_i belongs to other perspectives.

Intra-consistency is in fact an estimate of the LLM's uncertainty (Kuhn et al., 2023; Xiong et al., 2023b) and reflects the confidence of the model on one specific output. Therefore, we can utilize the intra-consistency information as a supervision signal by ensuring the closeness between the score function f and the intra-consistency measure φ with Mean Squared Error (MSE),

$$\mathcal{L}_{intra} = \frac{1}{2} \sum_{v_i \in V} |f(v_i) - \varphi(v_i)|^2 = \frac{1}{2} ||\boldsymbol{f} - \boldsymbol{y}||^2 \quad (2)$$

2.5 Optimization Formulation

After all, following the criteria of inter- and intra-consistency, we can then formulate the learning process of f as an optimization problem that combines both $\mathcal{L}_{inter}$ (Eq. 1) and $\mathcal{L}_{intra}$ (Eq. 2):

$$\min_{f: V \to \mathbb{R}} \{\alpha \cdot \mathcal{L}_{inter} + (1 - \alpha) \cdot \mathcal{L}_{intra}\} \quad (3)$$

To solve this optimization problem on the graph, we adopt the iterative algorithm proposed by Zhou et al. (2003b). The details of the algorithm can be found in Appendix A.

3 Experiment

3.1 Experiment Settings

Dataset and metrics. We conduct experiments on four widely used Python code generation benchmarks, including HumanEval, HumanEval+, MBPP and CodeContests. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are two hand-written Python programming problems. HumanEval+ (Liu et al., 2023) adds more unit tests based on HumanEval. CodeContests (Li et al., 2022) is a much more challenging dataset consisting of competition problems from the Codeforces platform. The evaluation metric is Pass@k (Chen et al., 2021), which is an unbiased estimator of the probability that at least one out of the k solutions generated by the model passes unit tests. Details about the metric can be found in Appendix C.

Implementation and baselines. We compare several baselines from different LLMs for code like ChatGPT[4] (i.e. GPT-3.5-Turbo), GPT-4 (OpenAI, 2023), Code Llama (Rozière et al., 2024), WizardCoder (Luo et al., 2023) and Deepseek Coder (Guo et al., 2024), to other post-hoc approaches enhancing LLMs during inference, including Self-consistency (Wang et al., 2022), MBR-EXEC (Shi et al., 2022), CodeT (Chen et al., 2022) and Self-collaboration (Dong et al., 2023). For both MPSC and other post-hoc augmentation approaches, we employ GPT-3.5-Turbo as the foundation model to generate 200 solutions. MPSC additionally generates 50 specifications and 100 test cases for each problem. Following the original setting in Chen et al. (2022), we additionally generate 500 test cases for other baselines.

Variants of MPSC. As shown in Eq.2, the intra-consistency measure $\varphi(\cdot)$ is essentially used as a supervision signal without leveraging the semantics of "consistency". Therefore, we include two variants in our experiments: (1) MPSC-Uniform is the baseline without any prior intra-consistency information and treats every vertex equally, i.e. $\varphi(v_i) = C$. (2) MPSC-Label includes the public example test cases in docstrings as silver labels, i.e. $\varphi(v_i) = C \cdot 1_{v_i \text{ is label}}$[5]. Further details regarding the implementation of our method and baselines are provided in Appendix E.

[4]https://chat.openai.com/

[5]The number of example test cases is two in average. Noted that MBPP doesn't provide test cases in docstrings.

3.2 Main Results

The experimental results on the four benchmarks are presented in Table 2. We observe that MPSC consistently enhances the code generation capabilities of the foundation model (i.e. GPT-3.5-Turbo) across all benchmarks with a remarkable margin of improvement. Particularly, when k is set to 1, which is the most prevalent scenario in real-world applications, the performance improvement is notably significant (+15.91% on HumanEval, +15.64% on HumanEval+, +6.43% on MBPP and +9.37% on CodeContests). With the foundation model GPT-3.5-Turbo, our MPSC can even outperform GPT-4 in Pass@1 across all benchmarks. Compared to other post-hoc augmentation approaches, even though they utilize more test cases, our MPSC still shows consistent advantages in all benchmarks, excluding the Pass@5 score in MBPP benchmark. MPSC-Uniform serves as the bottom line of MPSC framework and still achieves great gains for the foundation model, which demonstrates that relying on inter-consistency proves to be entirely effective. Moreover, incorporating various types of intra-consistency information leads to further improvements. Specifically, MPSC-Label and MPSC-Semantic exhibit particularly strong results. They are two representative approaches leveraging the external supervision signals or the internal consistency information respectively. Surprisingly, MPSC-Semantic can match or even surpass MPSC-Label in some benchmarks, which further highlights the significance of consistency information in LLMs. Besides, we also note that the performance of MPSC-Semantic and MPSC-Lexical remains largely unchanged as k increases. This phenomenon aligns with the nature of MPSC, which assesses solutions based on their consistency within the foundation model. It implies that top-ranked solutions exhibit semantic similarity and are consistently either correct or incorrect. This reaffirms the capability of our proposed MPSC to effectively aggregate consistency information within LLMs, thereby selecting the most consistent answers. We have a more detailed discussion about this phenomenon in Appendix C.

3.3 Further Analysis

Ablation study. We conduct an ablation study to examine the impact of different perspectives on MPSC. The results are presented in Table 4. Evidently, both the specification and test case perspec-

Benchmark	HumanEval			HumanEval+		
Metric	Pass@1	Pass@2	Pass@5	Pass@1	Pass@2	Pass@5
GPT4	81.48	86.31	**90.46**	70.52	75.48	**79.54**
GPT-3.5-Turbo	68.38	76.24	83.15	58.75	66.58	73.96
DeepSeekCoder	79.30	-	-	-	-	-
WizardCoder	73.20	-	-	-	-	-
Code Llama	62.20	-	-	-	-	-
Self-consistency	$73.86_{+5.48}$	$73.93_{-2.31}$	$74.10_{-9.05}$	$63.50_{+4.75}$	$64.70_{-1.88}$	$65.67_{-8.29}$
MBR-EXEC	$72.96_{+4.58}$	$76.47_{+0.23}$	$79.00_{-4.15}$	$62.12_{+3.37}$	$67.08_{+0.50}$	$71.38_{-2.58}$
CodeT	$78.05_{+9.67}$	$78.05_{+1.81}$	$78.30_{-4.85}$	$67.87_{+9.12}$	$68.75_{+2.17}$	$69.65_{-4.31}$
Self-collaboration	$74.40_{+6.02}$	-	-	-	-	-
MPSC-Uniform	$74.17_{+5.79}$	$77.02_{+0.78}$	$78.53_{-4.62}$	$65.05_{+6.30}$	$69.76_{+3.18}$	$71.72_{-2.24}$
MPSC-Lexical	$82.32_{+13.94}$	$84.76_{+8.52}$	$86.48_{+3.33}$	$\mathbf{74.39_{+15.64}}$	$75.00_{+8.42}$	$77.24_{+3.28}$
MPSC-Semantic	$83.38_{+15.00}$	$84.25_{+8.01}$	$84.45_{+1.30}$	$73.54_{+14.79}$	$74.46_{+7.88}$	$75.26_{+1.30}$
MPSC-Label	$\mathbf{84.29_{+15.91}}$	$\mathbf{86.79_{+10.55}}$	$87.13_{+3.98}$	$73.47_{+14.72}$	$\mathbf{76.66_{+10.08}}$	$77.25_{+3.29}$

Benchmark	MBPP			CodeContests		
Metric	Pass@1	Pass@2	Pass@5	Pass@1	Pass@2	Pass@5
GPT-4	71.26	**74.27**	**76.99**	6.1	8.28	11.72
GPT-3.5-Turbo	66.80	72.34	76.60	2.57	4.22	7.16
DeepseekCoder	70.00	-	-	-	-	-
WizardCoder	61.20	-	-	2.15	3.40	5.37
Code Llama	61.20	-	-	-	-	-
Self-consistency	$71.70_{+4.90}$	$71.73_{-0.61}$	$71.82_{-4.78}$	$8.10_{+5.53}$	$8.42_{+4.20}$	$8.48_{+1.32}$
MBR-EXEC	$70.79_{+3.99}$	$73.14_{+0.80}$	$74.85_{-1.75}$	$8.25_{+5.68}$	$8.87_{+4.65}$	$9.08_{+1.92}$
CodeT	$71.90_{+5.10}$	$71.95_{-0.39}$	$72.02_{-4.58}$	$9.92_{+7.35}$	$10.18_{+5.96}$	$10.30_{+3.14}$
Self-collaboration	$68.20_{+1.40}$	-	-	-	-	-
MPSC-Uniform	$69.34_{+2.54}$	$70.06_{-2.28}$	$71.85_{-4.75}$	$4.71_{+2.14}$	$6.65_{+2.43}$	$8.31_{+1.15}$
MPSC-Lexical	$68.38_{+1.58}$	$70.26_{-2.08}$	$71.43_{-5.17}$	$5.45_{+2.88}$	$5.45_{+1.23}$	$6.06_{-1.10}$
MPSC-Semantic	$\mathbf{73.23_{+6.43}}$	$73.29_{+0.95}$	$73.55_{-3.05}$	$10.09_{+7.52}$	$10.29_{+6.07}$	$10.30_{+3.14}$
MPSC-Label	-	-	-	$\mathbf{11.94_{+9.37}}$	$\mathbf{15.55_{+11.33}}$	$\mathbf{18.20_{+11.04}}$

Table 2: The results on four code generation benchmarks. The foundation model for MPSC, Self-consistency, MBR-EXEC, CodeT, Self-collaboration are all GPT-3.5-Turbo. The improvements are calculated between methods and GPT-3.5-Turbo. The best and second best performance for each dataset are shown in **bold** and underline.

tives play crucial roles in our framework. Additionally, test cases contribute more to the improvements than specifications. We attribute the observation to the better quality of test cases, as generating an accurate test case is considerably simpler than abstracting a comprehensive and sound specification.

Qualities of three perspectives. We present the accuracy[6] of generated solutions, specifications and test cases in Table 5. The quality of all three perspectives is insufficient individually. Indeed, the generated verification properties (i.e. specifications and test cases) are even poorer than the generated solutions. It implies that prior works (Zhang et al., 2023; Chen et al., 2022) relying on generated verification properties as experts for majority voting on solutions may fail, as these experts perform worse than the choices (i.e. solutions) themselves. Additionally, it indicates that the significant improvements brought by MPSC do not solely depend on the high quality of verification properties. The improvements come from the consistency information within LLMs, which helps to distinguish noise from high-quality solutions.

Generalization over different LLMs. MPSC is designed as a model-agnostic framework that assumes black-box access to the underlying foundation model. In assessing the extent of MPSC's generalization, we employ many other LLMs in addition to ChatGPT as foundation models. In specific, we consider the strongest model, GPT4, and three highly proficient open-source coding LLMs, Code Llama, WizardCoder and DeepSeek Coder in Python. The experimental results presented in Table 3 show that MPSC consistently yields significant improvements across all models, which demonstrates the robust generalization capabilities embedded in our proposed framework.

[6]Accuracy is equal to pass@1.

Benchmark	HumanEval			HumanEval+		
Metric	Pass@1	Pass@2	Pass@5	Pass@1	Pass@2	Pass@5
GPT-4 (gpt4-0614)	81.48	86.31	90.46	70.52	75.48	79.54
+MPSC	**92.15**$_{+10.67}$	**91.62**$_{+5.31}$	**91.80**$_{+1.34}$	**81.72**$_{+11.2}$	**81.77**$_{+6.29}$	**82.12**$_{+2.58}$
WizzardCoder-34B†	67.84	72.12	**75.98**	58.70	62.88	**66.88**
+MPSC	**74.06**$_{+6.22}$	**75.00**$_{+2.88}$	75.07$_{-0.91}$	**65.45**$_{+6.75}$	**65.71**$_{+2.83}$	66.19$_{-0.69}$
Code Llama-34B	51.78	59.24	67.07	41.49	48.30	55.93
+MPSC	**70.97**$_{+19.19}$	**70.55**$_{+11.31}$	**71.36**$_{+4.29}$	**58.44**$_{+16.95}$	**59.00**$_{+10.70}$	**60.02**$_{+4.09}$
WizzardCoder-13B	60.35	66.10	72.01	50.25	56.00	61.98
+MPSC	**73.60**$_{+13.25}$	**74.96**$_{+8.86}$	**74.57**$_{+2.56}$	**61.33**$_{+11.08}$	**62.99**$_{+6.99}$	**62.67**$_{+0.69}$
Code Llama-13B	44.63	50.99	57.86	35.93	41.71	48.19
+MPSC	**62.94**$_{+18.31}$	**64.93**$_{+13.94}$	**64.66**$_{+6.80}$	**50.04**$_{+14.11}$	**51.24**$_{+9.53}$	**51.36**$_{+3.17}$
WizzardCoder-7B	53.81	59.62	66.06	45.06	50.83	57.69
+MPSC	**63.85**$_{+10.04}$	**64.04**$_{+4.42}$	**67.32**$_{+1.26}$	**53.69**$_{+8.63}$	**55.07**$_{+4.24}$	**59.45**$_{+1.76}$
Code Llama-7B	39.38	45.18	52.79	34.33	39.18	45.25
+MPSC	**58.54**$_{+19.16}$	**57.83**$_{+12.65}$	**59.31**$_{+6.52}$	**49.04**$_{+14.71}$	**49.96**$_{+10.78}$	**50.46**$_{+5.21}$
Deepseek Coder-6.7B	71.73	80.92	**86.73**	61.72	71.42	**78.54**
+MPSC	**82.38**$_{+10.65}$	**83.92**$_{+3.00}$	84.71$_{-2.02}$	**70.04**$_{+8.32}$	**72.12**$_{+0.70}$	73.96$_{-4.58}$

Table 3: The performance of MPSC-Semantic with different foundation models. †: We use nucleus sampling with temperature as 0.2 instead of greedy generation in this experiment. The best performance is shown in **bold**.

Benchmark	HumanEval	HumanEval+	MBPP	CodeContests
Ours	83.38	73.54	73.23	10.09
w/o Specification	82.32	73.52	70.18	9.17
w/o Test case	78.30	68.49	72.00	8.71
w/o Both	68.38	58.75	66.80	2.57

Table 4: The ablation study results (Pass@1) on four benchmarks.

Perspective	Solution	Specification	Test Case
HumanEval	68.38	45.93	63.82
MBPP	66.80	53.70	34.64

Table 5: Accuracy of solutions, specifications and test cases generated by GPT-3-Turbo.

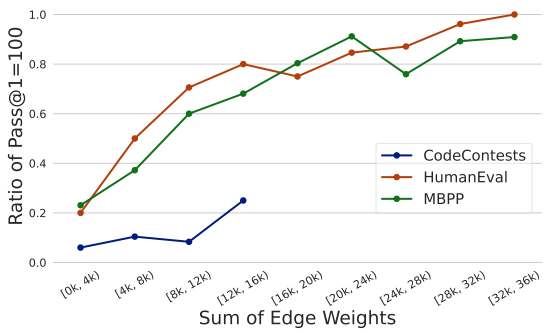

Figure 3: The correlation between the performance of MPSC and the edge density.

Impact of edge sparsity. Our framework significantly depends on the inter-consistency information between model outputs, which is represented as edges within the constructed graph. A critical question arises concerning the impact of edge sparsity on the framework's efficacy. To address this, we categorize all queries in the dataset into distinct bins based on the total edge weights in their corresponding graphs and compute the perfect performance ratio (i.e. Pass@1=100) for each bin. In this experiment, we employ the MPSC-Uniform configuration[7]. Figure 3 illustrates the correlation between edge density and performance. The results clearly demonstrate a positive correlation between the number of edges and the overall performance of our framework.

Impact of sampling number. To examine the effect of the sampling number of different perspectives, we conduct an analysis experiment by varying the numbers of generated specifications and test cases. As shown in Table 6, MPSC constantly brings significant performance gains with varied specifications and test cases. We note that MPSC generally suffers a slight degradation in performance when fewer specifications or test cases are used, which is consistent with our intuition. But the performance decline is relatively small (about 4% with only 10% of specifications and test cases retained). The observation indicts the remarkable performance and efficiency of MPSC, suggesting the

[7]The Uniform configuration is utilized to eliminate the influence of intra-consistency information.

#Specification	#Test Case				
	10	20	50	100	200
10	$78.93_{+10.55}$	$80.08_{+11.7}$	$83.82_{+15.44}$	$83.96_{+15.58}$	$84.17_{+15.79}$
20	$77.17_{+8.79}$	$80.13_{+11.75}$	$83.82_{+15.44}$	$83.42_{+15.04}$	$85.44_{+17.06}$
50	$80.23_{+11.85}$	$80.24_{+11.86}$	$82.24_{+13.86}$	$83.38_{+15.00}$	$83.57_{+15.19}$
100	$80.39_{+12.01}$	$80.9_{+12.52}$	$80.92_{+12.54}$	$81.55_{+13.17}$	$84.17_{+15.79}$

Table 6: Pass@1 of MPSC-Semantic with different sampling numbers on HumanEval.

potential for real-world application with reduced computational costs.

4 Related Work

Prompting techniques on consistency. Based on Chain-of-thought mechanism (Wei et al., 2022), previous works have adopted various prompting techniques and decoding strategies to reveal the consistency of LLM outputs and further enhance the capabilities of LLMs. One line of approaches decodes multiple times from the same perspective and aggregate the results (Wang et al., 2022; Zhou et al., 2022; Jung et al., 2022; Sun et al., 2022). For example, Wang et al. (2022) targets tasks with fixed answer sets and scores each answer based on the output frequency. Building on this, Sun et al. (2022) introduces recitation as context for augmentation. While Jung et al. (2022) focus on the two-value entailment relations (True or False) between statements and explanations. They treat the inference process as a weighted MAX-SAT problem and utilize a logistic solver to solve it. Another line draws inspiration from the "Dual Process" in cognitive science, which posits that human reasoning is dominated by System 1 and System 2 (Daniel, 2017; Sloman, 1996). As a result, these approaches require LLMs to play different roles like generator (i.e. System 1) and verifier (i.e. System 2), and optimize the result iteratively by a conversational way (Madaan et al., 2023; Shinn et al., 2023; Zhu et al., 2023). Xiong et al. (2023a) also proposes the concept of "inter-consistency". Instead of referring to the consistency within the same LLM, they focus to tackle the inter-inconsistency problem between different models with a formal debate framework.

LLM for code. LLMs pretrained on large-scale code data have demonstrated strong capabilities in the field of code generation, such as Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), CodeGen (Nijkamp et al., 2023), InCoder (Fried et al., 2022), StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), WizardCoder (Luo et al., 2023), DeepSeekCoder (Guo et al., 2024). However, they remain unreliable, particularly in scenarios involving complex input-output mappings. Because of the low tolerance of compilers and operating systems for bugs, the instability makes LLMs hard to deploy into real-world applications. Several methods have been proposed to mitigate the phenomenon (Shi et al., 2022; Chen et al., 2022; Zhang et al., 2022; Key et al., 2022; Ni et al., 2023; Dong et al., 2023; Olausson et al., 2023; Chen et al., 2023; Zhang et al., 2023). The line of work with the most direct relevance to ours is to re-rank generated solutions in a post-hoc manner. For example, Shi et al. (2022) matches the execution results of generated solutions for minimum Bayes risk selection. Zhang et al. (2022) prompts another model as a reviewer to check whether generated programs satisfy the given language instruction by measuring $p(\text{instruction}|\text{program})$. CodeT (Chen et al., 2022) additionally generates test cases to verify the generated solutions. Similarly, ALGO (Zhang et al., 2023) additionally generates exhaustive search algorithms as oracle programs to generate high quality test cases for verification.

Ranking on graph. In our framework, the final stage is in fact a ranking problem in graph. There exists some renowned graph ranking algorithms like PageRank (Page et al., 1998) and HITS (Kleinberg, 1999). While our approach is inspired by manifold ranking (Zhou et al., 2003b), which is built upon a regularization framework on discrete spaces (i.e. graphs in this scenario) (Zhou et al., 2003a; Zhou and Schölkopf, 2004, 2005).

5 Conclusion

In this paper, we present a novel code generation method, Multi-Perspective Self-Consistency (MPSC), aimed at enhancing the performance of LLMs in complex code generation tasks where a single attempt may not suffice to ensure the accuracy of the output. Our proposed MPSC strategy capitalizes on both intra- and inter-consistency across three perspectives, solutions, specifications and test cases, to identify the most reliable answer. We systematically validate the effectiveness of MPSC through comprehensive experiments conducted on four widely used code generation benchmarks. The evaluation results demonstrate that MPSC outperforms existing methods and achieves the state-of-the-art performance on all of them.

Limitations

Evaluation in the wild. Even though MPSC has shown remarkable performance on most widely used code generation benchmarks, its effectiveness in real-world scenarios remains largely unexplored. Existing code generation benchmarks often present simplified code generation tasks compared to the intricacies encountered in actual code developments, where user intents are harder to understand and the desired functionalities are more complex.

Generalization to other tasks. Since our proposed MPSC framework is designed to be model-agnostic and task-agnostic. We only conduct experiments in the code generation task in this paper. Actually, MPSC can be applied to other textual generation tasks like math problem solving and question answering. However, unlike code generation, where code interpreter can measure the agreement between outputs in a deterministic way, assessing the agreement between natural language outputs is non-trivial. A general task-agnostic inter-consistency measure is to solely rely on LLMs, whose evaluation ability for arbitrary textual inputs has been demonstrated recently. We leave it for future works to discuss.

References

Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis.

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.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests.

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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways.

Kahneman Daniel. 2017. *Thinking, fast and slow*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

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

Jaehun Jung, Lianhui Qin, Sean Welleck, Faeze Brahman, Chandra Bhagavatula, Ronan Le Bras, and Yejin Choi. 2022. Maieutic prompting: Logically consistent reasoning with recursive explanations. *arXiv preprint arXiv:2205.11822*.

Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I Speak, You Verify: Toward Trustworthy Neural Program Synthesis.

Jon M. Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation.

Shankar Kumar and William Byrne. 2004. Minimum Bayes-Risk Decoding for Statistical Machine Translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 169–176, Boston, Massachusetts, USA. Association for Computational Linguistics.

K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg. Springer Berlin Heidelberg.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science*, 378(6624):1092–1097.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic evaluation of language models.

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. *arXiv preprint arXiv:2305.01210*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback.

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

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation.

R OpenAI. 2023. Gpt-4 technical report. *arXiv*, pages 2303–08774.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural Language to Code Translation with Execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: An autonomous agent with dynamic memory and self-reflection.

Steven A Sloman. 1996. The empirical case for two systems of reasoning. *Psychological bulletin*, 119(1):3.

Zhiqing Sun, Xuezhi Wang, Yi Tay, Yiming Yang, and Denny Zhou. 2022. Recitation-Augmented Language Models. In *The Eleventh International Conference on Learning Representations*.

The Coq Development Team. 2023. The coq proof assistant.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Kai Xiong, Xiao Ding, Yixin Cao, Ting Liu, and Bing Qin. 2023a. Examining the Inter-Consistency of Large Language Models: An In-depth Analysis via Debate.

Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. 2023b. Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers.

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2022. Coder Reviewer Reranking for Code Generation.

D. Zhou and B. Schölkopf. 2004. A Regularization Framework for Learning from Graph Data. In *ICML 2004 Workshop on Statistical Relational Learning and Its Connections to Other Fields (SRL 2004)*, pages 132–137.

Dengyong Zhou, Olivier Bousquet, Thomas Lal, Jason Weston, and Bernhard Schölkopf. 2003a. Learning with Local and Global Consistency. In *Advances in Neural Information Processing Systems*, volume 16. MIT Press.

Dengyong Zhou and Bernhard Schölkopf. 2005. Regularization on Discrete Spaces. In *Pattern Recognition*, Lecture Notes in Computer Science, pages 361–368, Berlin, Heidelberg. Springer.

Dengyong Zhou, Jason Weston, Arthur Gretton, Olivier Bousquet, and Bernhard Schölkopf. 2003b. Ranking on Data Manifolds. In *Advances in Neural Information Processing Systems*, volume 16. MIT Press.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Xinyu Zhu, Junjie Wang, Lin Zhang, Yuxiang Zhang, Yongfeng Huang, Ruyi Gan, Jiaxing Zhang, and Yujiu Yang. 2023. Solving Math Word Problems via Cooperative Reasoning induced Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4471–4485.

A Details of the Iterative Algorithm

Description of algorithm. The iterative algorithm is shown in Algorithm 1.

Algorithm 1: Iterative Optimization

Input: degree matrix $\boldsymbol{D} = \text{diag}(d_1, ..., d_N)$, initialization score vector $\boldsymbol{y}$, weighted adjacency matrix $\boldsymbol{W}$, threshold ϵ

Output: optimal confidence score vector $\boldsymbol{f}^*$

begin

$\boldsymbol{f}^{(0)} \longleftarrow \boldsymbol{y}$

$\boldsymbol{T} \longleftarrow \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{W}\boldsymbol{D}^{-\frac{1}{2}}$

$i \longleftarrow 0$

Do

$\boldsymbol{f}^{(i+1)} \longleftarrow \alpha\boldsymbol{T}\boldsymbol{f}^{(i)} + (1-\alpha)\boldsymbol{y}$

$i \longleftarrow i+1$

While $||\boldsymbol{f}^{(i)} - \boldsymbol{f}^{(i-1)}|| \leq \epsilon$

$\boldsymbol{f}^* \longleftarrow \boldsymbol{f}^{(i)}$

return $\boldsymbol{f}^*$

Proof of Convergence We first expand the expression of $\boldsymbol{f}^{(n)}$ according to the recursive formula

$$\begin{aligned}\boldsymbol{f}^{(n)} &= \alpha\boldsymbol{T}\boldsymbol{f}^{(n-1)} + (1-\alpha)\boldsymbol{y} \\ &= (\alpha\boldsymbol{T})^{n-1}\boldsymbol{f}^{(0)} + (1-\alpha)\sum_{i=0}^{n-1}(\alpha\boldsymbol{T})^i\boldsymbol{y}\end{aligned}$$

Notice that $\boldsymbol{T}$ is similar to a stochastic matrix $\boldsymbol{W}\boldsymbol{D}^{-1} = \boldsymbol{D}^{\frac{1}{2}}(\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{W}\boldsymbol{D}^{-\frac{1}{2}})\boldsymbol{D}^{-\frac{1}{2}} = \boldsymbol{D}^{\frac{1}{2}}\boldsymbol{T}\boldsymbol{D}^{-\frac{1}{2}}$. Therefore the eigenvalues of $\alpha\boldsymbol{T}$ are in $[-\alpha, \alpha]$. With $\alpha \in (0, 1)$, we have

$$\lim_{n\to\infty}(\alpha\boldsymbol{T})^n = 0$$

$$\lim_{n\to\infty}\sum_{i=0}^{n-1}(\alpha\boldsymbol{T})^i = (\boldsymbol{I} - \alpha\boldsymbol{T})^{-1}$$

Therefore

$$\boldsymbol{f}^* = \lim_{n\to\infty} f^{(n)} = (1-\alpha)(1-\alpha\boldsymbol{T})^{-1}\boldsymbol{y}$$

Proof of Equivalence Denote the optimization function as

$$\begin{aligned}\mathcal{F} &= \alpha\boldsymbol{f}^T(\boldsymbol{I} - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{W}\boldsymbol{D}^{\frac{1}{2}})\boldsymbol{f} + \frac{(1-\alpha)}{2}(\boldsymbol{f}-\boldsymbol{y})^2 \\ &= \alpha\boldsymbol{f}^T(\boldsymbol{I} - \boldsymbol{T})\boldsymbol{f} + \frac{(1-\alpha)}{2}(\boldsymbol{f}-\boldsymbol{y})^2\end{aligned}$$

Differentiate $\mathcal{F}$ with respect to $\boldsymbol{f}$, we have

$$\frac{\partial\mathcal{F}}{\partial\boldsymbol{f}} = \alpha(\boldsymbol{I} - \boldsymbol{T})\boldsymbol{f} + (1-\alpha)(\boldsymbol{f} - \boldsymbol{y})$$

Let the derivatives to 0, the solution $\boldsymbol{f}' = (1-\alpha)(\boldsymbol{I} - \alpha\boldsymbol{T})^{-1}\boldsymbol{y} = \boldsymbol{f}^*$. Therefore, the iterative algorithm is actually optimizing the objective function.

Results of closed-form solution. Despite the existence of a closed-form solution for the optimization problem, the required matrix inversion operation is computationally expensive. Conversely, the iterative algorithm exhibits rapid convergence and demonstrates strong empirical performance. Consequently, we employ the iterative algorithm in our experiments. Additionally, we provide the performance of the closed-form solution in Table 7. Our results indicate that the iterative algorithm achieves a performance on par with that of the direct computation of the closed-form solution.

Metric	Pass@1	Pass@2	Pass@5
HumanEval			
MPSC-Lexical	82.32/82.32	84.76/84.76	86.48/86.59
MPSC-Semantic	83.38/83.46	84.25/84.15	84.45/84.45
HumanEval+			
MPSC-Lexical	74.39/74.39	75.00/75.00	77.24/77.34
MPSC-Semantic	73.54/73.97	74.46/75.04	75.26/75.96
CodeContests			
MPSC-Lexical	5.45/5.45	5.45/5.45	6.06/6.06
MPSC-Semantic	10.09/10.26	10.29/10.30	10.30/10.30
MBPP			
MPSC-Lexical	68.38/68.38	70.26/70.26	71.43/71.43
MPSC-Semantic	73.23/73.27	73.29/73.55	73.55/73.56

Table 7: Performance of MPSC optimized by the iterative algorithm or calculating closed-form solution directly. The results are presented in form of (`iterative algorithm / closed-form solution`).

B Implementation of Inter-Consistency

We present the code snippets measuring the inter-consistency between each pair of perspectives in Listing 1, 2, 3. After execution with Python interpreter, the `final_result` is acquired as $\omega(v_i, v_j)$.

```python
"""Generated specifications"""
# Pre-conditions
def preconditions(input):
...

# Post-conditions
def postconditions(input, output):
...

"""Generated test cases"""
test_case = {'input': ...,
             'output': ...}

"""Check inter-consistency"""
def check():
    pass_result = None
    try:
        preconditions(test_case['input'])
        postconditions(test_case['input'], test_case['output'])
        pass_result = True
    except Exception as e:
        pass_result = False
    return pass_result
global final_result
final_result = check()
```

Listing 1: Inter-consistency between specifications and test cases.

```python
"""Generated solutions"""
def entry_point(input):
...

"""Generated specifications"""
# Pre-conditions
def preconditions(input):
...

# Post-conditions
def postconditions(input, output):
...

"""Generated casual inputs"""
casual_inputs = [...]

"""Check inter-consistency"""
def check():
    pass_result = []
    for ci in casual_inputs:
        try:
            output = entry_point(ci)
            postconditions(ci, output)
            pass_result.append(True)
        except Exception as e:
            pass_result.append(False)
    return sum(pass_result) / len(pass_result)
global final_result
final_result = check()
```

Listing 2: Inter-consistency between solutions and specifications.

```python
"""Generated solutions"""
def entry_point(input):
...

"""Generated test cases"""
test_case = {'input': ...,
             'output': ...}

"""Check inter-consistency"""
def check():
    try:
        output = entry_point(test_case['input'])
        pass_result = (output == test_case['output'])
    except Exception as e:
        pass_result = False
    return pass_result
global final_result
final_result = check()
```

Listing 3: Inter-consistency between solutions and test cases.

C Discussion about Pass@k

In this section, we discuss the flaws of Pass@k in Chen et al. (2021) and propose a variant for evaluating methods involved selection and filtering.

Chen et al. (2021) propose an unbiased estimator called Pass@k, which estimates the probability of a model passing unit tests within k attempts. In specific, Chen et al. (2021) first samples a total of n solutions with c of them are correct, randomly samples k solutions for testing, and use the probability of passing tests for estimation,

$$\text{Pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

Although their implementation serves as an effective measure of the code generation ability of different foundation models (referred to as the first category of methods in the following), it is not suitable to evaluate methods involving filtering or selection during the inference stage (Li et al., 2022; Chen et al., 2022) (referred to as the second category of methods in the following), as the n generated solutions are identical.

To address the limitation, we implement a variant of Pass@k. We assume each method provides a score function over the n generated solutions, which provides a unified view for the two method categories and hence enables a fair comparison. The first category can be regarded as assigning an identical score to each solution. Similar to the original definition of Pass@k, we evaluate the method by testing the top-k solutions with the highest scores. As the score function may assign the same score to multiple solutions, the test result of the top-k is not deterministic but an expected value.

Mathematically, let's assume that a method sequentially arranges outputs into an ordered list $\{s_1, ..., s_n\}$, such that $\forall i > j, s_i \preceq s_j$ according to their scores. We define a set of solutions $\mathbb{S}^k = \{s_i | s_k \preceq s_i\}$, which represents the solution set selected by the method. Suppose the cardinalily of $\mathbb{S}^k$ is $\hat{n}$, the number of correct solutions within $\mathbb{S}^k$ is $\hat{c}$. Noted that $\hat{n} \geq k$, and thus we uniformly sample k solutions $\{s'_1, ..., s'_k\}$ from $\mathbb{S}^k$ for estimation,

$$\begin{aligned}
\text{Pass@}k \text{ (Ours)} &= \mathbb{E}_{s'_1,...,s'_k}[\mathbf{1}_{\cup_{i=1}^k s'_i \text{ is correct}}] \\
&= \Pr(\cup_{i=1}^k s'_i \text{ is correct}) \\
&= 1 - \Pr(\cap_{i=1}^k s'_i \text{ is incorrect}) \\
&= 1 - \frac{\binom{\hat{n}-\hat{c}}{k}}{\binom{\hat{n}}{k}}
\end{aligned}$$

For a the first category of methods, $\hat{n}$ equals n since it treats each solution equally. As a result, our implementation of Pass@k is identical to the original implementation in Chen et al. (2021).

Pass@k of MPSC when k is large As shown in Table 2, the performance of MPSC-Semantic and MPSC-Lexical remains largely unchanged as k increases. This phenomenon aligns with the nature of MPSC, which assesses solutions based on their consistency, hence assigning similar scores to solutions with similar semantics. Consequently, all solutions will aggregate into many clusters depending on whether the assigned scores are identical, resulting in a lack of diversity within each cluster. Therefore, the pass rate of solutions selected by MPSC will remains constant even in more attempts (i.e. varying k), if the number of attempts is still less than the top-ranked cluster size. A trivial method to address the problem is to increase the diversity by adopting a round-robin selection from all clusters, rather than selecting solutions according to scores from highest to lowest. The corresponding results are presented in Table 8. We can see the performance of Pass@5 is improved compared with that of Table 2.

Method	Pass@1	Pass@2	Pass@5
HumanEval			
GPT-3.5-Turbo	68.38	76.24	83.15
MPSC-Lexical	82.32	84.76	86.59
MPSC-Semantic	83.35	86.08	89.75
HumanEval+			
GPT-3.5-Turbo	58.75	66.58	73.96
MPSC-Lexical	74.39	75.0	77.44
MPSC-Semantic	73.08	77.89	83.20
MBPP			
GPT-3.5-Turbo	66.80	72.34	76.60
MPSC-Lexical	68.38	70.26	71.43
MPSC-Semantic	73.24	74.46	78.01
CodeContests			
GPT-3.5-Turbo	2.57	4.22	7.16
MPSC-Lexical	5.45	5.45	6.06
MPSC-Semantic	10.07	10.18	11.39

Table 8: Performance of MPSC evaluated by the Pass@k metric in a round-robin way.

D More Analysis

D.1 MPSC with User-provided Test Cases

In practical applications of code generation, users often provide a limited number of test cases to outline the desired functionality, thereby assisting the model in generating code that aligns with the requirements. In Table 2, MPSC-Label has shown remarkable performance. In this study, we investigate the potential performance improvements of the method in such scenarios by incorporating various quantities of golden test cases. These golden test cases are generated and then validated using canonical solutions provided in the benchmarks. We conduct experiments on the HumanEval and MBPP dataset [8] and present the results in Table 9. The substantial performance enhancements achieved with the inclusion of merely five golden test cases underscore the feasibility of implementing MPSC in user-interactive application scenarios.

Method	Pass@1	Pass@2	Pass@5
HumanEval			
MPSC	74.17	77.02	78.53
+ 1 test case	85.37	86.59	85.13
+ 2 test cases	85.98	86.18	85.36
+ 5 test cases	88.41	89.23	88.69
+ 10 test cases	89.02	90.24	88.81
HumanEval+			
MPSC	65.05	69.76	71.72
+ 1 test case	85.37	86.59	85.13
+ 2 test cases	85.98	86.18	85.36
+ 5 test cases	88.41	89.23	88.69
+ 10 test cases	89.02	90.24	88.81
MBPP			
MPSC	69.34	70.06	71.85
+ 1 test case	69.85	71.99	72.69
+ 2 test cases	70.78	72.46	73.04
+ 5 test cases	71.25	72.93	73.47
+ 10 test cases	71.72	73.4	73.27

Table 9: Performance of MPSC with different numbers of golden test cases.

D.2 Analysis of Other Perspectives

MPSC not only selects the optimal output from the target perspective but also chooses outputs from auxiliary perspectives, thereby generating corresponding by-products. In the context of code generation, which is the primary focus of this paper, MPSC can additionally identify more reliable test cases and specifications. We evaluate the quality of these by-products and present the results in Table 11. The experimental results demonstrate that MPSC is proficient in selecting high-quality outputs across all relevant perspectives.

[8] CodeContests doesn't provide canonical solutions. Therefore, we cannot conduct evaluation of test cases.

Metric	Pass@1	Pass@2	Pass@5
HumanEval			
WizzardCoder-34B	$66.04_{\pm 1.27}$	$70.95_{\pm 0.84}$	$75.51_{\pm 0.43}$
+MPSC	$76.32_{\pm 1.82}$	$76.50_{\pm 1.20}$	$75.93_{\pm 0.68}$
WizzardCoder-13B	$58.62_{\pm 1.23}$	$65.05_{\pm 0.75}$	$71.81_{\pm 0.14}$
+MPSC	$74.96_{\pm 0.96}$	$75.51_{\pm 0.44}$	$75.33_{\pm 0.57}$
WizzardCoder-7B	$52.20_{\pm 2.17}$	$58.16_{\pm 1.94}$	$64.73_{\pm 1.68}$
+MPSC	$65.09_{\pm 1.83}$	$65.54_{\pm 1.38}$	$66.51_{\pm 0.83}$
HumanEval+			
WizzardCoder-34B	$56.12_{\pm 1.83}$	$60.54_{\pm 1.66}$	$64.67_{\pm 1.56}$
+MPSC	$65.21_{\pm 0.20}$	$64.85_{\pm 1.10}$	$64.74_{\pm 1.36}$
WizzardCoder-13B	$49.42_{\pm 0.59}$	$54.91_{\pm 0.77}$	$60.81_{\pm 0.83}$
+MPSC	$63.12_{\pm 1.35}$	$63.68_{\pm 0.63}$	$63.77_{\pm 0.94}$
WizzardCoder-7B	$43.74_{\pm 2.34}$	$49.48_{\pm 2.37}$	$56.26_{\pm 2.30}$
+MPSC	$54.64_{\pm 1.21}$	$56.55_{\pm 1.05}$	$58.21_{\pm 0.88}$

Table 10: The average performance of MPSC with three sample sets under different random seeds. We use MPSC-Semantic configuration in this experiment.

D.3 Time Overhead

The additional time overhead imposed by MPSC mainly comes from inter-consistency measurements and the iterative algorithm (Alg. 1). For the former, we need to process a total of $(200 \times 50 + 50 \times 100 + 100 \times 200) = 35000$ edges with a time limit of 0.001 seconds per edge. In contrast, CodeT (Chen et al., 2022), the strongest baseline, requires to process a total of $200 \times 500 = 100000$ edges. Moreover, this process can be fully parallelized. For the latter, the iterative algorithm converges rapidly, typically within an average of about 50 rounds, requiring less than 0.1 second. Overall, the time overhead of MPSC is acceptable in most code development scenarios, laying the groundwork for real-world deployments.

D.4 Stability of MPSC

We explore the stability of MPSC with respect to the sampling process. We conduct the sampling process of WizardCoder with three random seeds and then assess the performance of MPSC on the generated solutions. The average results are shown in Table 10. It is evident that the improvement brought by MPSC is very stable.

Benchmark	HumanEval			MBPP		
Metric	Pass@1	Pass@2	Pass@5	Pass@1	Pass@2	Pass@5
			Specification			
GPT-3.5-Turbo	45.93	58.76	72.26	53.7	62.37	70.60
MPSC	73.58	73.59	74.38	71.86	71.38	73.40
			Test case			
GPT-3.5-Turbo	63.83	80.71	93.23	34.64	44.32	53.19
MPSC	96.95	96.95	96.95	55.72	55.95	57.18

Table 11: The quality of specifications and test cases selected by MPSC. They can also be evaluated in Pass@k. We use MPSC-Semantic configuration in this experiment.

E Experiment Settings and Baselines

We incorporate various baselines in code generation. First of all, we include many strong large language models like ChatGPT (gpt-3.5-turbo 0614 version), GPT-4 (gpt4-0614 version), Code Llama-Instruct-34B, WizardCoder-Python-34B and DeepSeekCoder-7B-Instruct. The specific hyper-parameters for inference of ChatGPT and GPT4 are shown in Table 12. MPSC requires an additional hyper-parameter α, which controls the balance between inter-consistency and intra-consistency in the algorithm. Given that the quality of inter-consistency significantly depends on the edge density of the graph, we utilize the mean edge weight to determine the value of α. Empirically, we assign a relatively small value of α (0.01) when the edges are sparse on the graph, indicated by the mean edge weight less than 0.16. Other, we assign a large value of α (0.95) otherwise to better leverage inter-consistency.

Temperature	0.8
Top P	0.95
Frequency Penalty	0
Presence Penalty	0

Table 12: The Inference hyper-parameters of LLMs.

We also include several baselines like Self-Consistency MBR-EXEC, CodeT and Self-collaboration, which enhance the inference capability of LLMs in a post-hoc manner.

- CodeT: This baseline first uses generated test cases to verify each solution by code execution. Then it utilizes RANSAC algorithm to create consensus sets based on execution results. The size of consensus set is then used to rank solutions. We generate 500 test cases for CodeT following the original implementation in Chen et al. (2022).
- Self-Consistency: We implement this baseline following Chen et al. (2022). If two solution pass the same set of generated test cases and specifications, we regard them "consistent". Then we take a majority voting to rank solutions following Wang et al. (2022).
- MBR-EXEC: This baseline ranks solutions by minimum Bayes risk decoding based on the execution results in the 500 generated test cases.

For a fair comparison between our proposed MPSC and these baselines, we employ the same solutions generated by ChatGPT for them to rerank. In specific, we sample 200 solutions following the conventional setting. Since some methods leverage generated test cases and specifications as well, we use the same set of test cases and specifications generated by ChatGPT for both MPSC and these baselines.

We don't include ALGO (Zhang et al., 2023) as baseline, because it requires to keep generating oracle programs until they pass all public example test cases, whose complexity is unlimited.

F Prompt for MPSC

Prompt for Generating Specifications

I want you to act as a python programmer. Given a docstring about a python method, you need to write its pre-conditions in one test function "def preconditions(input)" and post-conditions in another test function "def postconditions(input, output):". You should ensure invalid input or output of the method will raise error in the two test functions.

```Python
{Demonstration Docstrings 1}
pass
#Pre-conditions
{Demonstration Preconditions 1}
#Post-conditions
{Demonstration Postconditions 1}
```

```Python
{Demonstration Docstrings 2}
pass
#Pre-conditions
{Demonstration Preconditions 2}
#Post-conditions
{Demonstration Postconditions 2}
```

```Python
{Docstrings}
pass
```

Prompt for Generating Solutions

I want you to act like a Python programmer. I will give you the declaration of a function and comments about its property. You need to implement the body of the function in the code block. Do not modify any code I provide. Do not provide any explanations.

Here is the question.

```Python
{Docstrings}
```

Prompt for Generating Test Cases

```Python
# Given a docstring, continue to write the following code with 10 valid assertion statements to check the correctness of the function. Provide diverse test cases.
{Docstrings}
pass

# check the correctness of with 10 different valid assertion statements in the form of "assert {entry point}(...) == ..."
assert
```

G Case Study

We present some cases from HumanEval, MBPP, CodeContests in the following pages.

```python
"""User Intent"""
def get_row(lst, x):
    """
    You are given a 2 dimensional data, as a nested lists,
    which is similar to matrix, however, unlike matrices,
    each row may contain a different number of columns.
    Given lst, and integer x, find integers x in the list,
    and return list of tuples, [(x1, y1), (x2, y2) ...] such that
    each tuple is a coordinate - (row, columns), starting with 0.
    Sort coordinates initially by rows in ascending order.
    Also, sort coordinates of the row by columns in descending order.

    Examples:
    get_row([
      [1,2,3,4,5,6],
      [1,2,3,4,1,6],
      [1,2,3,4,5,1]
    ], 1) == [(0, 0), (1, 4), (1, 0), (2, 5), (2, 0)]
    get_row([], 1) == []
    get_row([[], [1], [1, 2, 3]], 3) == [(2, 2)]
    """


"""Solution selected by MPSC"""
def get_row(lst, x):
    result = []
    for row_index, row in enumerate(lst):
        for col_index, val in reversed(list(enumerate(row))):
            if val == x:
                result.append((row_index, col_index))
    result.sort(key=lambda x: x[0])
    return result


"""Solution selected by CodeT"""
def get_row(lst, x):
    result = []
    for i, row in enumerate(lst):
        if x in row:
            for j, val in sorted(enumerate(row), key=lambda x: -x[1]):
                if val == x:
                    result.append((i, j))
    return result
```

Listing 4: Cases from HumanEval.

```python
"""User Intent"""
def count_first_elements(test_tup):
    '''
    Write a function to find the number of elements that occurs before the tuple element in the given tuple.
    '''


"""Solution selected by MPSC"""
def count_first_elements(test_tup):
    count = 0
    for i in range(len(test_tup)):
        if isinstance(test_tup[i], tuple):
            break
        count += 1
    return count


"""Solution selected by CodeT"""
def count_first_elements(test_tup):
    count = 0
    for i in range(len(test_tup)):
        if test_tup[i] == test_tup[0]:
            break
        count += 1
    return count
```

Listing 5: Cases from MBPP.

```python
"""User Intent"""
def solution(inputs: str) -> str:
    '''
    You are given an array a of length n.
    Let's define the eversion operation. Let x = a_n. Then array a is partitioned into two parts: left and right. The left part contains the elements of a that are not greater than x (    x). The right part contains the elements of a that are strictly greater than x (> x). The order of elements in each part is kept the same as before the operation, i. e. the partition is stable. Then the array is replaced with the concatenation of the left and the right parts.
    For example, if the array a is [2, 4, 1, 5, 3], the eversion goes like this: [2, 4, 1, 5, 3]     [2, 1, 3], [4, 5]     [2, 1, 3, 4, 5].
    We start with the array a and perform eversions on this array. We can prove that after several eversions the array a stops changing. Output the minimum number k such that the array stops changing after k eversions.

    Input:
    Each test contains multiple test cases. The first line contains the number of test cases t (1     t     100). Description of the test cases follows.
    The first line contains a single integer n (1     n     2     10^5).
    The second line contains n integers a_1, a_2, ..., a_n (1     a_i     10^9).
    It is guaranteed that the sum of n over all test cases does not exceed 2 10^5.

    Output:
    For each test case print a single integer k     the number of eversions after which the array stops changing.
    Example
    assert solution('3\n5\n2 4 1 5 3\n5\n5 3 2 4 1\n4\n1 1 1 1') == '1\n2\n0'
    '''
"""Solution selected by MPSC"""
def solution(inputs: str) -> str:
    def eversion(a):
        n = len(a)
        l, r = [], []
        x = a[-1]
        for i in range(n - 1):
            if a[i] <= x:
                l.append(a[i])
            else:
                r.append(a[i])
        return l + [x] + r

    def is_same(a, b):
        return len(a) == len(b) and all(ai == bi for ai, bi in zip(a, b))

    res = []
    inputs = inputs.strip().split('\n')
    t = int(inputs[0])
    idx = 1
    for _ in range(t):
        n = int(inputs[idx])
        a = list(map(int, inputs[idx+1].split()))
        idx += 2
        cnt = 0
        while True:
            new_a = eversion(a)
            cnt += 1
            if is_same(a, new_a):
                break
            a = new_a
        res.append(str(cnt - 1))
    return '\n'.join(res)
```

```python
"""Solution selected by CodeT"""
def solution(inputs: str) -> str:
    # split the input into list of test cases
    test_cases = inputs.strip().split('\n')[1:]
    result = []

    # loop through each test case
    for i in range(0, len(test_cases), 2):
        n = int(test_cases[i])
        a = list(map(int, test_cases[i+1].split()))
        k = 0

        # perform eversion operation until array stops changing
        while True:
            x = a[-1]
            left, right = [], []
            for j in range(n-1):
                if a[j] <= x:
                    left.append(a[j])
                else:
                    right.append(a[j])
            new_a = left + [x] + right
            if new_a == a:
                break
            a = new_a
            k += 1

        result.append(str(k))

    # join the results and return
    return '\n'.join(result)
```

Listing 6: Cases from CodeContests.