PROCURE: Addressing the Programming Concept Understanding Gap for Code Generation in LLMs via Concept-Aware Consistency Learning

Xiaoning Ren[1], **Qiang Hu**[2], **Wei Ma**[3], **Chongyang Liu**[1], **Yan Li**[1], **Yao Zhang**[2], **Lingxiao Jiang**[3], **Yongqiang Lyu**[2,5], **Yinxing Xue**[4,*]
[1]University of Science and Technology of China
[2]Tianjin University
[3]Singapore Management University
[4]Institute of AI for Industries, Chinese Academy of Sciences *
[5] JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.
{hnurxn, liyann}@mail.ustc.edu.cn, {qianghu, zzyy, lyuyq}@tju.edu.cn, {weima, lxjiang}@smu.edu.sg, lcyyy9@gmail.com, yxxue@iaii.ac.cn

Abstract

Although Large Language Models (LLMs) excel at code generation, recent research reveals that they exhibit an insufficient grasp of core programming concepts, such as data flow and control flow. This limitation undermines their robustness when encountering variations in these concepts in practice; however, effective solutions that explicitly target this gap remain limited. To address this challenge, we propose PROCURE, a concept-aware consistency learning framework designed to enhance LLMs' understanding of programming concepts. Specifically, PROCURE first performs automated concept-oriented code augmentation to construct a concept-aligned dataset covering representative programming concepts. It then conducts concept-aware fine-tuning, encouraging the model to capture fine-grained concept variations and learn appropriate generation behaviors under such variations via a novel concept-sensitive consistency loss. To quantify programming concept understanding, we introduce the Concept Consistency Score (CCScore), defined as the proportion of correct generations preserved under concept variations. A higher CCScore indicates a more profound understanding of programming concepts. We evaluate PROCURE on four open-source LLMs across three widely used code generation benchmarks. Experimental results show that PROCURE improves CCScore by an average of 17.9 points, demonstrating its effectiveness in addressing the programming concept understanding gap.

1 Introduction

Large language models (LLMs) have demonstrated promising performance in code generation tasks, marking a new paradigm for programming [Zhang *et al.*, 2023; Schäfer *et al.*, 2023; Abdin *et al.*, 2024]. However, whether LLMs can truly understand code remains an open question. Such understanding is a prerequisite for users to reliably and confidently employ LLMs in code-related tasks, ensuring controllability and trustworthiness. Consequently, in-depth evaluation and enhancement of LLMs' code understanding ability has emerged as a critical research direction for the community.

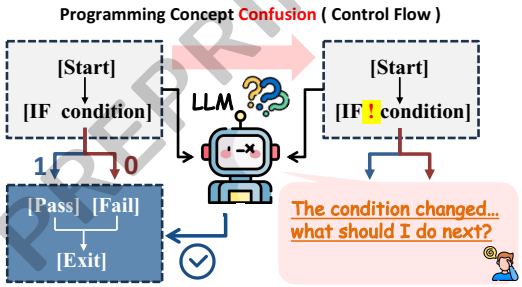

Figure 1: An illustrative example showing the limited understanding of LLMs to programming concept variations.

Existing works [Ma *et al.*, 2023; Orvalho and Kwiatkowska, 2025] revealed that LLMs have the potential to grasp code syntax but do not understand the semantics correctly. Even worse, recent research [Hooda *et al.*, 2024] highlighted a fundamental limitation: LLMs lack sufficient understanding of *Programming Concepts Predicates*[1], including how variables are managed in memory, how execution flows across constructs, and how code segments combine to perform computations. As illustrated in Figure 1, the figure presents a representative failure case in which a model becomes confused when control-flow variations are introduced within the code prefix.

However, effective solutions that explicitly target programming concept understanding are urgently needed, yet largely absent in existing work. Most prior approaches focus on improving robustness through surface-level perturbations and adversarial training [Bielik and Vechev, 2020; Luo *et al.*, 2025], or fine-tune models for other objectives such as security [He *et al.*, 2024], without explicitly model-

*Corresponding author.

[1]Throughout this paper, we use the term *programming concepts* to denote programming concept predicates for brevity.

ing programming concepts or enforcing consistent behavior under concept-level variations. To bridge this gap, we propose a novel framework **PROCURE** (**Pro**gramming **C**oncept **U**nde**R**standing **E**nhancement), which enhances programming concept understanding via concept-aware consistency learning. PROCURE consists of two stages. In the first stage, we construct a concept-aligned dataset through a fully automated, LLM-assisted pipeline, where concept-oriented perturbations are generated using enriched one-shot chain-of-thought prompts augmented with static code analysis signals. In the second stage, we perform concept-aware fine-tuning with a concept-sensitive consistency loss that emphasizes concept-relevant tokens, encouraging the model to respond consistently to fine-grained programming concept variations while preserving functional correctness.

We conduct a comprehensive evaluation on open-source LLMs, including Llama3.1-8B, CodeLlama-13B, StarCoder-7B, and Mistral-7B, using three widely recognized benchmark datasets: HumanEval, MBPP, and CodeContests, which span a diverse range of programming challenges. Experimental results show that PROCURE improves CCScore by an average of 17.9 points, demonstrating its effectiveness in enhancing programming concept understanding and providing new insights for code augmentation and robust training in downstream code-related tasks. In summary, our contributions are outlined as follows:

- We propose PROCURE, a concept-aware consistency learning framework that enhances programming concept understanding in LLMs, which is, to the best of our knowledge, the first to explicitly target this problem.
- Extensive experiments show the effectiveness of PROCURE, improving CCScore by an average of 17.9 points.
- We release *ConceptEval*[2], a new benchmark for evaluating programming concept understanding.

2 Preliminary

2.1 LLM-based Code Generation

LLM-based code generation [Jiang *et al.*, 2025] formulates source code generation as an autoregressive process conditioned on natural language, source code, or both. Given an input token sequence $T = [t_1, \ldots, t_M]$, an LLM generates code by iteratively estimating the conditional probability of the next token:

$$P(Y \mid T) = \prod_{i=1}^{N} P(y_i \mid T, y_1, \ldots, y_{i-1}),$$

where $Y = [y_1, \ldots, y_N]$ denotes the generated token sequence. Among various code generation tasks, code completion is one of the most widely studied settings, where T consists of a functional instruction and a partial program. Following [Hooda *et al.*, 2024], we adopt code completion to evaluate LLMs' understanding of programming concepts, as it directly tests whether models can appropriately adapt their generation to concept-level variations in the input.

2.2 Programming Concepts

Programming Concepts refer to the properties of specific program elements such as variables, data values, data paths, and execution paths, considered from a holistic, program-wide perspective [Hooda *et al.*, 2024; Hoare, 1969]. These concepts may describe, for example, the value range of a variable at a specific location, whether control flow can reach one function location from another, or whether a value assigned to a variable may later be used elsewhere (i.e., data-flow or control-flow). Prior work identifies four major themes —control flow [Allen, 1970; Yang *et al.*, 2015], data flow [Nilsson-Nyman *et al.*, 2009; Fosdick and Osterweil, 1976], identifier names [Lin and Wu, 2008], and data types [Allamanis *et al.*, 2020; Dart and Zobel, 1992]—and proposes a representative set of programming concepts that cover these themes:

- **Control Flow** describes the execution order of statements governed by control structures. Perturbing a control condition (e.g., in an if statement) requires adjusting the corresponding execution path.
- **Data Flow** ensures variable references are in scope and live at their use sites, preserving correct data dependencies and value propagation.
- **Data Types** constrain the set of valid values and operations for variables; type correctness must be maintained to ensure semantic soundness.
- **Identifier Naming** refers to the symbolic names of variables. While not affecting semantics, consistent renaming is necessary to preserve referential correctness.
- **Statement Order** captures cases where independent statements without mutual data or control dependencies can be reordered without affecting program behavior.

In this work, we follow the problem formulation of prior study [Hooda *et al.*, 2024] that identifies the programming concept understanding gap in LLMs. Specifically, we focus on weakly typed programming languages (e.g., Python), where type information is either implicit or dynamically resolved at runtime. Consequently, our analysis centers on the four programming concepts introduced above, excluding data types.

What It Means to Understand Programming Concepts? Understanding a programming concept means that when a concept-oriented change occurs at a specific program location, an appropriate response can be made based on the underlying programming logic to preserve the program's intended functionality. Let $P = \langle l_1, l_2, \ldots, l_n \rangle$ be a program of n lines, with its overall functionality denoted by $[\![P]\!] = F$. Let $\mathcal{E} = \{e_1, \ldots, e_m\}$ be the set of m program elements. Suppose a program element $e_j \in \mathcal{E}$ is modified at line k ($1 \leq k \leq n$), resulting in an intermediate program $P \oplus \Delta(e_j)$. Understanding a programming concept implies that an appropriate transformation $T_{\mathcal{C}(e_j)}$ can be applied, which rewrites only the lines within the impact region $\mathcal{C}(e_j) \subseteq \{l_{k+1}, \ldots, l_n\}$, such that:

$$[\![T_{\mathcal{C}(e_j)}\big(P \oplus \Delta(e_j)\big)]\!] = [\![P]\!] \tag{1}$$

In Equation (1), the transformation $T_{\mathcal{C}(e_j)}$, guided by the semantics of the corresponding programming concept, compensates for the effect of $\Delta(e_j)$ by refactoring only the affected lines, thereby preserving the original functionality F.

[2]https://github.com/hnurxn/Counterfactual_Benchmark

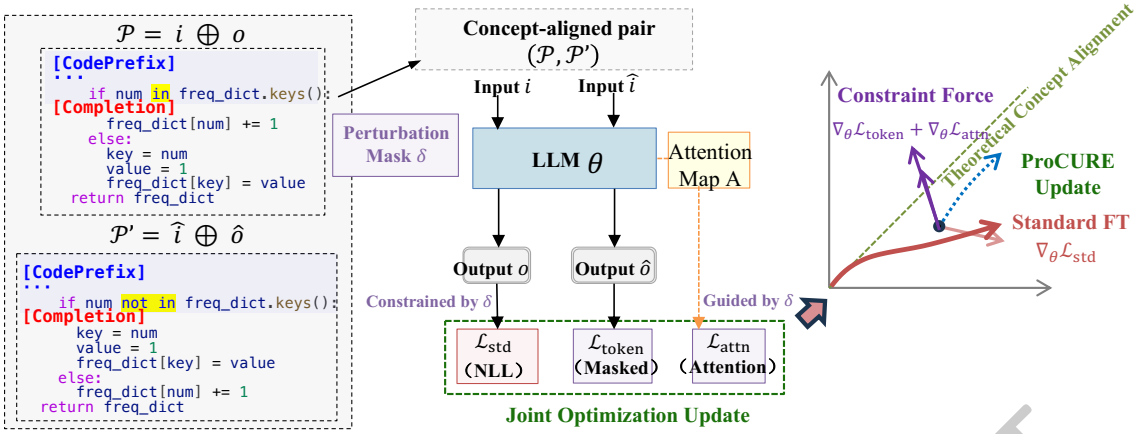

Figure 2: Overview of PROCURE. It first constructs a concept-aligned dataset through automated concept-oriented data augmentation, and then conducts joint optimization with a concept-sensitive consistency objective to enable concept-aware consistency learning.

When applied to LLMs, this definition implies that given a functional instruction and a code prefix, the model should adapt its completion to concept-related changes in the prefix while preserving the intended functionality. However, existing LLMs exhibit limited understanding of such programming concepts, and effective solutions remain lacking. Addressing this gap constitutes the primary motivation of our work.

3 Framework: PROCURE

3.1 Overview

As illustrated in Figure 2, we first develop an automated concept-oriented perturbation engine that systematically applies transformations to input programs with respect to specific programming concepts. Each transformation generates a perturbed variant paired with the original program, and all such pairs collectively form a concept-aligned dataset. Built upon this dataset, we perform joint optimization during fine-tuning, where the standard training objective is augmented with a concept-sensitive consistency objective that emphasizes concept-relevant regions. This optimization encourages the model to preserve functional correctness while appropriately responding to concept-level variations, thereby enabling LLMs to internalize programming concepts and exhibit consistent behavior under concept-oriented changes.

3.2 Concept-oriented Perturbation Engine

As illustrated in Figure 3, we present the pipeline of LLM-assisted Concept-Oriented Perturbation Engine. It consists of two core components: (1) efficient and effective prompt construction for concept-oriented perturbation generation, and (2) automatic validation of the generated perturbations to ensure semantic correctness. By repeatedly applying this pipeline to the original programs, we construct a high-quality concept-aligned dataset for subsequent concept-aware fine-tuning.

Concept-Guided Prompt Construction. The perturbation engine is driven by a set of concept-specific perturbation strategies that modify targeted structural aspects of a program while preserving function equivalence. Following prior work [Hooda *et al.*, 2024; Ma *et al.*, 2024], we consider five representative strategies: If-Else Flip, Def-Use Break, Independent Swap, Name Random, and Name Shuffle. To guide the LLM toward generating valid and concept-aligned perturbations, we perform concept-specific static analysis on the input program and incorporate the extracted signals into the prompt. We further augment the prompt with chain-of-thought reasoning and a one-shot example to improve generation reliability. Details of the perturbation rules, illustrative examples, and the complete prompt template are provided in the Supplementary Material.

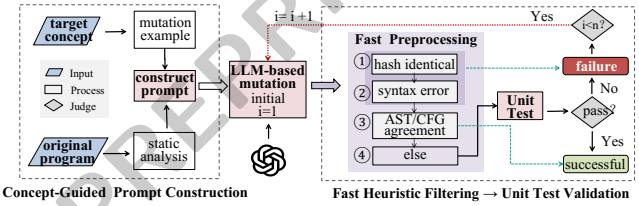

Figure 3: The overall pipeline for concept-oriented perturbation engine, consisting of prompt construction and result validation.

Results Validation. After the LLM generates a perturbed candidate, we apply an automated validation procedure to determine whether it constitutes a valid function-preserving transformation. The procedure follows a two-stage design, consisting of a fast, execution-free filtering stage followed by unit test validation.

Fast Heuristic Filtering. We first apply lightweight checks to quickly eliminate invalid or redundant candidates without executing the program. Specifically, we discard candidates that are identical to the original program based on hash comparison, or that contain syntax errors detected via AST parsing. For the remaining candidates, we examine structural differences: if the abstract syntax trees are identical, or if the control flow graphs remain unchanged despite syntactic variations, the candidate is accepted as a valid perturbation at this stage. *Unit Test Validation.* Candidates that pass heuristic filtering but whose semantic equivalence remains uncertain are further validated through unit test execution. Each original program is associated with a set of functional test cases, and a perturbed

Algorithm 1: Concept-Aware Fine-Tuning

Input: Dataset $\mathcal{D}_{\text{combine}}$, learning rate η, number of epochs E
Output: Fine-tuned model parameters θ_{FT}

Initialize model parameters θ;
for $e = 1$ **to** E **do**
  Shuffle $\mathcal{D}_{\text{combine}}$;
  foreach *mini-batch* $\mathcal{B}$ *sampled from* $\mathcal{D}_{combine}$ **do**
    Partition $\mathcal{B}$ into original samples $\mathcal{B}_{\text{ori}}$ and concept-aligned samples $\mathcal{B}_{\text{con}}$;
    Compute the standard NLL loss $\mathcal{L}_{\text{std}}$ on $\mathcal{B}_{\text{ori}}$;
    foreach *concept-aligned sample* $(\mathbf{i}^c, \mathbf{o}^c) \in \mathcal{B}_{con}$ **do**
      Compute masked token loss $\mathcal{L}_{\text{token}}$;
      Compute attention-guided loss $\mathcal{L}_{\text{attn}}$;
    end
    Compute the overall batch loss: $\mathcal{L}_{\text{batch}} = \mathcal{L}_{\text{std}} + \mathcal{L}_{\text{token}} + \mathcal{L}_{\text{attn}}$;
    Update model parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}_{\text{batch}}$;
  end
end
return θ_{FT}

variant is accepted only if it passes all tests. Candidates failing any test are rejected.

Concept-Aligned Dataset Construction. All validated perturbations are paired with their corresponding original programs to form concept-aligned program pairs. For each pair, we annotate the transformation locations at the token level on the perturbed variant using *difflib*[3], explicitly identifying tokens modified by the concept-oriented perturbation. The resulting collection of annotated pairs constitutes our concept-aligned dataset, which serves as training data for concept-aware fine-tuning and as the basis for constructing an evaluation benchmark.

3.3 Concept-Aware Fine-Tuning

We propose a concept-aware fine-tuning strategy that jointly optimizes standard task learning and programming concept understanding. The core idea is to integrate concept-aware data sampling with a concept-sensitive loss, enabling the model to learn task-specific generation behavior while consistently responding to programming concept variations.

Concept-Aware Data Sampling. Given an original dataset $\mathcal{D}^{\text{ori}} = \{(\mathbf{i}_j, \mathbf{o}_j)\}$, where each input $\mathbf{i}_j = (h, x)$ consists of a functional instruction h and a code prefix x, and $\mathbf{o}_j = y$ denotes the corresponding target completion, and a set of programming concepts $\mathbb{C} = \{c_1, c_2, \ldots, c_n\}$, we construct a concept-aligned dataset $\mathcal{D}^{\text{con}} = \{(\mathbf{i}_j^{c_k}, \mathbf{o}_j^{c_k})\}$ by applying concept-oriented perturbations with respect to each $c_k \in \mathbb{C}$. Both the input and output are transformed in a concept-consistent manner to preserve the program functionality.

We define the combined training dataset as:

$$\mathcal{D}_{\text{combine}} = \left\{ \left((\mathbf{i}_j, \mathbf{o}_j), (\mathbf{i}_j^{c_1}, \mathbf{o}_j^{c_1}), \ldots, (\mathbf{i}_j^{c_n}, \mathbf{o}_j^{c_n}) \right) \;\middle|\; (\mathbf{i}_j, \mathbf{o}_j) \in \mathcal{D}^{\text{ori}} \right\}.$$

[3]https://docs.python.org/3/library/difflib.html

Each entry in $\mathcal{D}_{\text{combine}}$ consists of one original example and its associated concept-aligned variants, all sharing the same underlying functionality. To ensure joint exposure to both sample types during training, we adopt a structured batch construction strategy. Specifically, each mini-batch $\mathcal{B}$ is formed as:

$$\mathcal{B} = \bigcup_{j=1}^{|B|/(n+1)} \left\{ (\mathbf{i}_j, \mathbf{o}_j), (\mathbf{i}_j^{c_1}, \mathbf{o}_j^{c_1}), \ldots, (\mathbf{i}_j^{c_n}, \mathbf{o}_j^{c_n}) \right\},$$

where $|B|$ denotes the total batch size. This design ensures that original and concept-aligned samples are jointly optimized within each parameter update.

Concept-Sensitive Loss Function. Given the mixed nature of the training data, we employ a joint optimization objective composed of two complementary loss components. Original samples focus on learning correct task behavior, while concept-aligned samples explicitly encourage sensitivity to programming concept variations.

Standard Loss. For each original sample $(\mathbf{i}, \mathbf{o}) \in \mathcal{D}^{\text{ori}}$, we apply the standard negative log-likelihood (NLL) loss:

$$\mathcal{L}_{\text{std}}(\mathbf{i}, \mathbf{o}) = -\sum_{t=1}^{|\mathbf{o}|} \log P(\mathbf{o}_t \mid \mathbf{o}_{<t}, \mathbf{i}).$$

Concept-Sensitive Loss. Let $\mathbf{o}$ and $\mathbf{o}^c$ denote the original and concept-aligned outputs, respectively. To localize the effect of concept-oriented perturbations, we construct a binary mask $\boldsymbol{\delta} \in \{0, 1\}^{|\mathbf{o}^c|}$, where $\delta_j = 1$ indicates that token j is affected by the perturbation. We design two complementary loss terms to guide concept-aware learning:

- Masked Token Loss. This term emphasizes prediction accuracy on concept-altered regions:

$$\mathcal{L}_{\text{token}}(\mathbf{i}^c, \mathbf{o}^c) = -\sum_{j=1}^{|\mathbf{o}^c|} \delta_j \log P(\mathbf{o}_j^c \mid \mathbf{o}_{<j}^c, \mathbf{i}^c).$$

- Attention-Guided Loss. To further encourage structural awareness, we regularize the decoder attention so that tokens generated after a perturbation attend more strongly to the altered regions. Given the decoder attention matrix $\mathbf{M}^{\text{attn}} \in \mathbb{R}^{|\mathbf{o}^c| \times |\mathbf{o}^c|}$, we define:

$$\mathcal{L}_{\text{attn}}(\mathbf{i}^c, \mathbf{o}^c) = -\lambda \sum_{j=1}^{|\mathbf{o}^c|} \sum_{k<j} \delta_k \cdot \left| M_{jk}^{\text{attn}} \right|,$$

  where $\lambda > 0$ controls the strength of the attention regularization.

Training Summary. During training, each mini-batch contains both original and concept-aligned samples to enable joint optimization. Specifically, original samples are optimized using the standard negative log-likelihood loss, while concept-aligned samples are trained with a concept-sensitive objective composed of the masked token loss and the attention-guided consistency loss. The overall batch loss is computed as:

$$\mathcal{L}_{\text{batch}} = \sum_{(\mathbf{i}, \mathbf{o}) \in \mathcal{B}_{\text{ori}}} \mathcal{L}_{\text{std}} + \sum_{(\mathbf{i}^c, \mathbf{o}^c) \in \mathcal{B}_{\text{con}}} \left(\mathcal{L}_{\text{token}} + \mathcal{L}_{\text{attn}} \right),$$

and model parameters are updated via gradient descent. The complete training procedure is summarized in Algorithm 1. At each epoch, the combined dataset is shuffled and processed in mixed mini-batches containing both original and concept-aligned samples. Original samples are optimized using the standard negative log-likelihood loss to preserve functional correctness, while concept-aligned samples are trained with a concept-sensitive objective composed of the masked token loss and the attention-guided loss. These loss components are jointly optimized within each parameter update via gradient descent, enabling the model to balance task-level performance and programming concept sensitivity. After all training epochs, the fine-tuned model parameters θ_{FT} are obtained.

4 Experiment Evaluation

4.1 Experimental Setup

Models. We evaluate four state-of-the-art open-source LLMs, representing both general-purpose and code-specialized paradigms. For general-purpose LMs, we select LLaMA3.1-8B [Touvron *et al.*, 2024] and Mistral-7B [Jiang *et al.*, 2023], both trained on large-scale web and code corpora. For coding LMs, we adopt StarCoder-7B [Li *et al.*, 2023] and CodeLLaMA-13B [Roziere *et al.*, 2023], which are pretrained on permissively licensed GitHub code and optimized for program synthesis tasks.

Benchmarks. We evaluate models on three widely-used code generation benchmarks: HumanEval [Chen *et al.*, 2021], MBPP [Austin *et al.*, 2021], and CodeContests [Li *et al.*, 2022]. HumanEval consists of 164 Python problems with unit tests for functional correctness. MBPP includes 1,000 beginner-level programming tasks, and CodeContests provides a challenging set of real-world algorithmic problems from competitive programming platforms. Together, these benchmarks span a broad range of task difficulties, algorithmic patterns, and prompt formulations. Building upon these benchmarks, we further construct a concept-variation benchmark for evaluating programming concept understanding under controlled concept-level perturbations.

Evaluation Metrics. We define a binary attribution function $A : \mathcal{H} \times \mathcal{X} \times \mathcal{Y} \to \{0, 1\}$, where $\mathcal{H}$ denotes the set of natural language instructions, $\mathcal{X}$ the input space, and $\mathcal{Y}$ the output space. $A(h, x, y)$ is 1 if the model's output satisfies the instruction, and 0 otherwise. This attribution function is used to compute the following metrics:

- **Pass@k.** Pass@k [Abdin *et al.*, 2024; Luo *et al.*, 2023; Wei *et al.*, 2024] measures the fraction of problems where at least one of k generated outputs satisfies the instruction:

$$\text{Pass@}k = \frac{1}{n}\sum_{i=1}^{n}\left(1 - \prod_{j=1}^{k}(1 - A(h_i, x_i, y_i^j))\right)$$

  where n is the number of problems, and y_i^j is the j-th output for problem i.
- **Concept Consistency Score (CCScore).** Inspired by prior work [Hooda *et al.*, 2024], we quantify programming concept understanding using the Concept Consistency Score (CCScore), which measures the proportion of correct generations preserved under concept-level variations. Given an input x and its set of concept-perturbed variants $\{x^{c_k}\}_{k=1}^{K}$ ($K \leq 5$), the model is considered consistent on x if it produces identical attribution outcomes across the original input and all its variants, i.e., $A(h, x, M(h, x)) = A(h, x^{c_k}, M(h, x^{c_k}))$ for all k.
  The CCScore is computed as:

$$\frac{\sum_{(h,x)} \mathbb{I}[\forall k,\ A(h,x,M(h,x)) = A(h,x^{c_k},M(h,x^{c_k}))]}{\sum_{(h,x)} \mathbb{I}\left[(A(h,x,M(h,x)) + \sum_{k=1}^{K} A(h,x^{c_k},M(h,x^{c_k})) > 0\right]}.$$

  Inputs are excluded only when the original output and all corresponding variant outputs are invalid, since such failures primarily indicate general model capacity rather than concept understanding. A higher CCScore indicates stronger robustness to concept-level variations.

Environment. All experiments are conducted on a single NVIDIA L40 GPU with 46GB of memory using PyTorch and the HuggingFace Transformers library.

Hyperparameters. For automated concept-oriented perturbation, we use GPT-5 with temperature 1, generating up to 5 attempts per sample for valid outputs. For concept-aware fine-tuning, we optimize models using AdamW with a learning rate of 5×10^{-6}, sequence length of 2048, batch size of 16, and a concept-sensitive loss with $\lambda = 0.1$ over 5 epochs. During evaluation, we use top-p = 0.95 and max_tokens = 512, with temperature set to 0.2 for Pass@1 and CCScore (averaged over 10 runs) and 0.6 for Pass@5, following prior work [Chen *et al.*, 2021; Nijkamp *et al.*, 2022].

Baselines. Since there is currently no fine-tuning method explicitly designed to enhance *programming concept understanding*, we adopt standard supervised fine-tuning (*std-SFT*) and several representative code augmentation–based fine-tuning approaches as baselines. Specifically, we include: (1) a diversity-oriented code transformation baseline *SA-AFT* [Chen *et al.*, 2023] that applies syntax- and structure-level transformations to increase data diversity; (2) a counterfactual instruction-tuning baseline *CTF-Code* [Luo *et al.*, 2025], which augments instructions via fine-grained counterfactual variations in problem descriptions; and (3) a robustness-oriented adversarial fine-tuning baseline *CodeFort* [Zhang *et al.*, 2024], which incorporates semantic-preserving adversarial perturbations with robust training objectives.

4.2 Dataset Experimental Results

Effectiveness and Efficiency of Dataset Construction. Table 1 summarizes the success rates of concept-oriented perturbation generation across datasets and perturbation types. Overall, the proposed pipeline achieves a high success rate of 97.51%, with dataset-level averages of 98.99%, 97.15%, and 96.39% on HumanEval, MBPP, and CodeContests, respectively. Among perturbation strategies, *If-Else Flip* yields the highest success rate (99.33%), while *Name Shuffle* performs worst, particularly on CodeContests (95.06%). Table 2 reports the efficiency of the perturbation process. Across all settings, valid perturbations are obtained with fewer than two attempts on average (1.24 overall), and the average token cost is 1,041 tokens per sample. CodeContests incurs higher costs due to its greater program complexity.

Dataset	If-else Flip	Def-use Break	Independent Swap	Name Random	Name Shuffle	Avg
HumanEval	24 / 24	37 / 37	144 / 145	141 / 145	145 / 145	98.99%
MBPP	198 / 200	179 / 182	957 / 972	924 / 972	946 / 972	97.15%
CodeContests	3796 / 3821	4895 / 5564	6980 / 7004	7183 / 7221	6864 / 7221	96.39%

Table 1: Perturbation success rates across datasets and perturbation types.

Dataset	If-else Flip		Def-use Break		Independent Swap		Name Random		Name Shuffle		Avg (All)	
HumanEval	1.10	673	1.22	803	1.11	625	1.44	841	1.35	704	1.24	729
MBPP	1.13	833	1.30	1001	1.05	657	1.14	718	1.09	637	1.14	769
CodeContests	1.15	1347	1.98	2384	1.06	1177	1.08	1250	1.59	1969	1.37	1625

Table 2: Average attempts and token costs across datasets and perturbation types.

	Perplexity (PPL)		Human Annotation		
Dataset	Original	Perturbed	Succ.	Fail.	Cohen's κ
HumanEval	2457.73	1410.81	99.5%	100%	0.995
MBPP	3117.99	1071.92	99.0%	100%	0.990
CodeContests	1170.14	784.87	98.5%	99.0%	0.975

Table 3: Quality Analysis of the Constructed Dataset.

	Exact Match			Near Duplicate		
Source	HumanEval	MBPP	CodeContests	HumanEval	MBPP	CodeContests
HumanEval	0	0	0	2	0	0
MBPP	0	7	0	0	45	0
CodeContests	0	0	64	0	0	101

Table 4: Exact and Near-Duplicate Detection Results (pair counts).

Dataset Quality. Before fine-tuning, we systematically assess the quality of the constructed concept-aligned dataset $\mathcal{D}^{\text{con}}$ from both automatic statistics and human evaluation, ensuring that the generated perturbations are natural and semantically reliable. Code naturalness reflects the readability and predictability of generated programs. We measure naturalness using perplexity (PPL), computed by a pre-trained CodeGPT model (*microsoft/CodeGPT-small-py*), where lower values indicate more natural code. As shown in Table 3, the perturbed programs consistently achieve lower PPL than the original programs across all benchmarks (e.g., decreasing from 2457.73 to 1410.81 on HumanEval). This trend suggests that concept-oriented perturbations preserve, and in some cases improve, the linguistic regularity of code while introducing structural variations. To further verify semantic validity, we conduct a human evaluation on 400 randomly sampled program pairs, evenly split between valid and invalid perturbations. Two PhD-level annotators independently assess whether each perturbed program correctly reflects the intended programming concept while preserving functionality. Table 3 reports high inter-annotator agreement across all datasets, with Cohen's κ exceeding 0.975, indicating near-perfect consistency. These results confirm that our automated perturbation pipeline produces high-quality, semantically valid samples suitable for concept-aware fine-tuning.

Data Decontamination. Data leakage poses a significant threat to the validity of LLM evaluation. To mitigate this risk, we conduct a comprehensive decontamination analysis at two granularities: (i) *Exact Match* detection via hash comparison, and (ii) *Near-Duplicate Detection* using MinHash with locality-sensitive hashing (LSH) over 5-gram token shingles, with a similarity threshold of $\tau = 0.8$ [Kocetkov *et al.*, 2022]. The detection results are summarized in Table 4. No cross-dataset leakage is observed among HumanEval, MBPP, and CodeContests. However, we identify limited internal redundancy within individual datasets. Specifically, CodeContests contains 64 exact duplicates and 101 near-duplicate pairs, while MBPP exhibits minor redundancy (7 exact and 45 near duplicates), and HumanEval shows only 2 near-duplicate pairs. To ensure a fair and uncontaminated evaluation, we apply a strict sanitization procedure, retaining only a single representative from each detected duplicate cluster and removing all remaining instances.

Dataset Summary. Through the above process, we construct a concept-aligned dataset comprising 33,203 perturbed programs of varying complexity, derived from 8,285 original samples, with each sample associated with 1–5 concept-oriented variants. This dataset is directly applicable to concept-aware fine-tuning. Moreover, by integrating the CCScore evaluation protocol into the data construction pipeline, we further build and release the first open benchmark, *ConceptEval*, for evaluating programming concept understanding in LLMs.

4.3 Fine-tuning Experimental Results

Based on the validated concept-aligned dataset, we evaluate the effectiveness of different fine-tuning strategies in improving both task performance and programming concept understanding. We randomly split the dataset into two equal parts, using 50% for fine-tuning and 50% for testing. For all baseline methods, data augmentation is applied only to the original training samples, and the number of augmented instances is strictly controlled to ensure that all methods are trained with the same total number of training samples, resulting in comparable training cost. During evaluation, Pass@1 and Pass@5 are computed on the original test samples to measure task-solving capability, while the CCScore is computed on paired original and concept-aligned test samples to assess robustness under programming concept variations.

Overall Results. Table 5 summarizes the performance of all methods across four code LLMs. Overall, ProCURE consistently achieves the highest CCScore across all evaluated models, with an average improvement of 17.9 percentage points, demonstrating its effectiveness in strengthening programming concept understanding. In terms of task perfor-

Method	Mistral-7B			Llama3.1-8B			StarCoder-7B			CodeLlama-13B		
	pass@1	pass@5	CCScore	pass@1	pass@5	CCScore	pass@1	pass@5	CCScore	pass@1	pass@5	CCScore
Base (w/o FT)	43.1	49.8	31.1	51.2	58.5	32.4	44.6	48.2	38.1	41.5	45.3	24.8
std-SFT	49.2	56.1	33.2	55.3	62.4	34.7	48.4	53.9	38.9	44.8	50.9	26.6
SA-AFT	61.2	**74.6**	39.6	64.3	74.8	44.1	62.1	**72.9**	43.1	52.9	69.1	34.2
CTF-Code	56.3	66.9	36.4	59.2	69.5	40.3	55.1	65.8	39.1	53.6	68.2	32.4
CodeFort	59.8	70.4	42.0	61.6	72.6	46.2	58.9	68.7	47.3	51.7	64.9	36.1
Ours	**65.4**	73.7	**46.4**	**67.8**	**76.5**	**50.9**	**65.0**	72.5	**59.2**	**55.4**	**73.0**	**41.5**

Table 5: Performance comparison of various fine-tuning techniques across four models, where Ours refers to ProCURE. The best results are bolded, and second-best are underlined.

Method	pass@1	pass@5	CCScore
w/o $\mathcal{L}_{\text{token}}$	57.4	65.9	39.9
w/o $\mathcal{L}_{\text{attn}}$	61.5	71.1	47.6
w/o $\mathcal{L}_{\text{token}} + \mathcal{L}_{\text{attn}}$	57.2	65.3	38.3
Ours	**63.4**	**73.9**	**49.6**

Table 6: Ablation results averaged over four models.

mance, ProCURE achieves consistently strong Pass@1 and Pass@5 results, ranking as the best or second-best across all settings. In contrast, standard supervised fine-tuning (std-SFT) yields noticeable gains in Pass@k but only marginal improvements in CCScore, indicating that while conventional fine-tuning improves task accuracy, it provides limited supervision for learning programming concepts. Compared with augmentation-based baselines (SA-AFT, CTF-Code, and CodeFort), ProCURE maintains a clear advantage in CCScore while preserving competitive task accuracy. SA-AFT achieves the highest Pass@5 on Mistral-7B and StarCoder-7B, indicating that diversity-oriented code transformations mainly improve sampling-based success rates rather than concept-level generalization. Among the baselines, CodeFort yields the strongest CCScore improvements, reflecting the benefit of robustness-oriented adversarial perturbations that expose vulnerabilities related to insufficient concept understanding. In contrast, CTF-Code attains the lowest CCScore gains, as its instruction-level counterfactual augmentation provides limited supervision for learning fine-grained programming concepts.

Ablation Analysis. Table 6 reports ablation results averaged over four models. Removing both concept-sensitive loss terms, which reduces training to standard fine-tuning on the constructed dataset, leads to the largest performance degradation, with clear drops in pass@1 and pass@5 and a substantial decrease in CCScore. This indicates that data augmentation alone is insufficient to capture programming concept variations. When removing individual components, excluding the token-level loss $\mathcal{L}_{\text{token}}$ causes a more pronounced decline than excluding the attention-guided loss $\mathcal{L}_{\text{attn}}$, particularly in CCScore. This suggests that explicit supervision on concept-altered tokens is the primary driver of concept understanding, while attention regularization provides complementary gains. Overall, both components are necessary, and their combination yields the most consistent improvements.

5 Related Work

Perturbation-based code augmentation applies diverse code-specific transformations to existing programs to construct augmented training data for robust fine-tuning. Early studies indicate that neural code models are highly sensitive to minor perturbations, such as variable renaming or syntactic reordering, despite preserved program semantics [Yefet *et al.*, 2020; Bielik and Vechev, 2020]. These observations have motivated various augmentation approaches based on standard transformations or adversarial code samples. A primary line of work focuses on rule-based or heuristic code transformations. ReCode [Wang *et al.*, 2023] systematically investigates over thirty semantic-preserving transformations, revealing substantial performance degradation in code generation models. More recently, CodeFort [Zhang *et al.*, 2024] introduces a comprehensive robustness training framework combining diverse perturbations with consistency-based objectives. Another line of work explores adversarial perturbations. Prior studies design attacks, ranging from identifier substitution to structure-aware perturbations, to expose model vulnerabilities [Zhou *et al.*, 2022; Zhang *et al.*, 2020; Tian *et al.*, 2021]. CodeAttack [Jha and Reddy, 2023] further demonstrates the efficacy of adversarial fine-tuning as a defense mechanism. Subsequent research continues to refine these methods by leveraging syntactic or semantic structures to guide adversarial sample generation [Chen *et al.*, 2023], with recent work extending this to adversarial perturbations specifically targeting code task descriptions [Luo *et al.*, 2025].

Crucially, Hooda et al. [Hooda *et al.*, 2024] identify a "programming concept understanding gap" in LLMs, highlighting the need for effective augmentation and fine-tuning strategies. However, existing methods are not tailored to programming concepts, limiting their ability to capture deep semantic logic. To address this, we propose a novel fine-tuning approach to enhance programming concept understanding.

6 Conclusion

We propose ProCURE, a concept-aware consistency learning framework that enhances LLMs' understanding of programming concepts through concept-oriented code augmentation and concept-sensitive consistency learning. Extensive experiments on four open-source LLMs across three code generation benchmarks show that ProCURE improves CCScore by an average of 17.9 points, demonstrating its effectiveness in addressing the programming concept understanding gap.

Acknowledgments

This work was supported by the 2025 Purple Mountain Talent Program for High-Level Innovative and Entrepreneurial Talents under the project "Theoretical Research on Trustworthy Evaluation of Intelligent Software" (Grant No. E6430219G8). This work was also supported by the National Natural Science Foundation of China (Grant Nos. U23B2041 and U24A6009), and the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace)–Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[Abdin *et al.*, 2024] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024. *URL https://arxiv. org/abs/2404.14219*, 2024.

[Allamanis *et al.*, 2020] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

[Allen, 1970] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[Austin *et al.*, 2021] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, et al. Program synthesis with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[Bielik and Vechev, 2020] Pavol Bielik and Martin Vechev. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR, 2020.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[Chen *et al.*, 2023] Nuo Chen, Qiushi Sun, Jianing Wang, Ming Gao, Xiaoli Li, and Xiang Li. Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14857–14873, 2023.

[Dart and Zobel, 1992] Philip W Dart and Justin Zobel. Efficient run-time type checking of typed logic programs. *The Journal of Logic Programming*, 14(1-2):31–69, 1992.

[Fosdick and Osterweil, 1976] Lloyd D Fosdick and Leon J Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330, 1976.

[He *et al.*, 2024] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. In *International Conference on Machine Learning*, pages 18043–18062. PMLR, 2024.

[Hoare, 1969] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hooda *et al.*, 2024] Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Do large code models understand programming concepts? counterfactual analysis for code predicates. ICML'24. JMLR.org, 2024.

[Jha and Reddy, 2023] Akshita Jha and Chandan K Reddy. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 14892–14900, 2023.

[Jiang *et al.*, 2023] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[Jiang *et al.*, 2025] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, July 2025.

[Kocetkov *et al.*, 2022] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

[Li *et al.*, 2022] Yujia Li, David Choi, Junyoung Chung, Sharan Narang, et al. Competition-level code generation with alphacode. In *International Conference on Learning Representations (ICLR)*, 2022.

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

[Lin and Wu, 2008] Jin-Cherng Lin and Kuo-Chiang Wu. Evaluation of software understandability based on fuzzy matrix. In *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*, pages 887–892. IEEE, 2008.

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

[Luo *et al.*, 2025] Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Mingzheng Xu, Tianhao Cheng, Yixuan Wang, Zheng Chu, Shijie Xuyang, Zhiyuan Ma, YuanTao Fan, et al. Success is in the details: Evaluate and enhance details sensitivity of code llms through counterfactuals. *arXiv preprint arXiv:2505.14597*, 2025.

[Ma *et al.*, 2023] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*, 2023.

[Ma *et al.*, 2024] Lezhi Ma, Shangqing Liu, Lei Bu, Shangru Li, Yida Wang, and Yang Liu. Speceval: Evaluating code comprehension in large language models via program specifications. *arXiv preprint arXiv:2409.12866*, 2024.

[Nijkamp *et al.*, 2022] Erik Nijkamp, Nora Kassner, Richard Pang, Lifu Tu, Caiming Xiong, Weijia Shi, Tommaso Soru, Livio Baldini Soares, Alexander Ratner, et al. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[Nilsson-Nyman *et al.*, 2009] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. Declarative intraprocedural flow analysis of java source code. *Electronic Notes in Theoretical Computer Science*, 238(5):155–171, 2009.

[Orvalho and Kwiatkowska, 2025] Pedro Orvalho and Marta Kwiatkowska. Are large language models robust in understanding code against semantics-preserving mutations? *arXiv preprint arXiv:2505.10443*, 2025.

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

[Schäfer *et al.*, 2023] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2023.

[Tian *et al.*, 2021] Junfeng Tian, Chenxin Wang, Zhen Li, and Yu Wen. Generating adversarial examples of source code classification models via q-learning-based markov decision process. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 807–818. IEEE, 2021.

[Touvron *et al.*, 2024] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, et al. LLaMA 3: Open foundation and instruction models, 2024.

[Wang *et al.*, 2023] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness evaluation of code generation models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada, July 2023. Association for Computational Linguistics.

[Wei *et al.*, 2024] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[Yang *et al.*, 2015] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.

[Yefet *et al.*, 2020] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[Zhang *et al.*, 2020] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020.

[Zhang *et al.*, 2023] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

[Zhang *et al.*, 2024] Yuhao Zhang, Shiqi Wang, Haifeng Qian, Zijian Wang, Mingyue Shang, Linbo Liu, Sanjay Krishna Gouda, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. CodeFort: Robust training for code generation models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 5262–5277, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[Zhou *et al.*, 2022] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–30, 2022.