2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

FGIT: Fault-Guided Fine-Tuning for Code Generation

Lishui Fan[†], Zhongxin Liu[†*], Haoye Wang[‡], Lingfeng Bao[†], Xin Xia[†], Shanping Li[†]
[†]*The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China*
[‡]*Hangzhou City University, China*
{flscode, liu_zx, lingfengbao, shan}@zju.edu.cn, wanghaoye@hzcu.edu.cn, xin.xia@acm.org

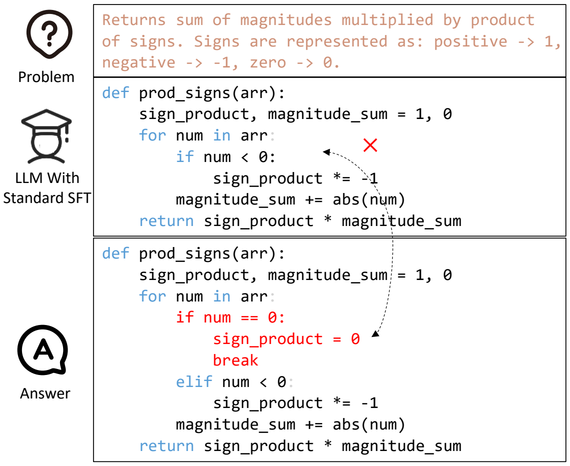
***Abstract*—Modern instruction-tuned large language models (LLMs) have made remarkable progress in code generation. However, these LLMs fine-tuned with standard supervised fine-tuning (SFT) sometimes generate plausible-looking but functionally incorrect code variants. This issue likely stems from the limitation of standard SFT, which treats all tokens equally during optimization and fails to emphasize the error-sensitive segments—specific code differences between correct implementations and similar incorrect variants. To address this problem, we propose Fault-Guided Fine-Tuning (FGIT), a novel fine-tuning technique that enhances LLMs' code generation by (1) extracting multi-granularity (line/token-level) differences between correct and incorrect yet similar implementations to identify error-sensitive segments, and (2) dynamically prioritizing those segments during training via dynamic loss weighting. Through extensive experiments on seven LLMs across three widely-used benchmarks, our method achieves an average relative improvement of 6.9% on pass@1, with some enhanced 6.7B LLMs outperforming closed-source models, e.g., GPT-3.5-Turbo. Furthermore, our fine-tuning technique demonstrates strong generalization with performance improvements ranging from 3.8% to 19.1% across diverse instruction-tuned LLMs, and our ablation studies confirm the contributions of different granularities of differences and hyperparameters.**

***Index Terms*—Large Language Model, Code Generation, Software Engineering**

I. Introduction

Recently, fine-tuning LLMs using synthetic datasets generated by teacher models has emerged as a popular paradigm for improving code generation capabilities [1]–[3]. This paradigm uses teacher models to generate high-quality instruction-response pairs and construct a dataset. These datasets are then used to fine-tune student models with standard SFT method, which uses instructions to guide LLMs to generate outputs matching reference responses by minimizing cross-entropy loss uniformly across all tokens.

Although these LLMs fine-tuned with standard SFT achieve impressive performance on code generation benchmarks [4]–[7], such as HumanEval [8], *they sometimes generate plausible-looking but incorrect code variants* [9], [10]. For example, Llama-3.1-70B-Instruct [11], a model fine-tuned from Llama-3.1-70B using standard SFT, is capable of solving 82.3% of the problems in HumanEval. However, our analysis shows that 34.5% of its failed cases are largely correct and require modifications in only three or fewer locations to be fixed. We conservatively classify these specific instances as failures attributable to deviations in error-prone segments, highlighting the prevalence of such errors in code generation. As shown in Figure 1, in a task to calculate the product of signs and the sum of magnitudes, although the LLM successfully implements logic similar to the correct version—by correctly handling negative numbers and summing magnitudes-it fails to account for zero. We refer to such crucial differences between correct implementations and similar incorrect variants as *error-sensitive segments*. These *error-sensitive segments* act as critical decision points in code generation, where even slight deviations can determine the correctness of the output.



```
Returns sum of magnitudes multiplied by product
of signs. Signs are represented as: positive -> 1,
negative -> -1, zero -> 0.
```

```
def prod_signs(arr):
    sign_product, magnitude_sum = 1, 0
    for num in arr:
        if num < 0:
            sign_product *= -1
        magnitude_sum += abs(num)
    return sign_product * magnitude_sum
```

```
def prod_signs(arr):
    sign_product, magnitude_sum = 1, 0
    for num in arr:
        if num == 0:
            sign_product = 0
            break
        elif num < 0:
            sign_product *= -1
        magnitude_sum += abs(num)
    return sign_product * magnitude_sum
```

Fig. 1: Llama-3.1-70B-Instruct sometimes makes mistakes in *error-sensitive segments* in the outputs.

It is crucial to address these errors for code generation. However, standard SFT [12] often overlooks this issue as it does not specifically focus on these segments essential for correctness. Drawing inspiration from curriculum learning [13], which involves training models by progressively moving from easier to more challenging samples to build learning capabilities, we propose Fault-Guided Fine-Tuning (FGIT). FGIT is a supplementary fine-tuning technique designed to guide instruction-tuned models to further focus their learning on these *error-sensitive segments*, treating these segments as more challenging samples in a targeted curriculum.

Implementing this approach involves two main challenges.

[*] Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00114

The first challenge is to identify these *error-sensitive segments*. Systematically collecting large-scale datasets of fine-grained coding errors from real-world scenarios, along with their corresponding correct versions, is difficult. Furthermore, existing instruction-tuning dataset construction methods primarily focus on generating instruction-response pairs without specifically considering these critical segments [14]. Considering traditional code mutation techniques [15], [16] are often constrained by a limited set of predefined transformation rules and may not generate diverse and semantically plausible incorrect variants, we develop a two-phase segment identification component. First, we leverage a teacher model, which can capture a wide range of diverse and semantically plausible error patterns from its extensive code training data, with a carefully designed prompt to generate multiple functionally incorrect yet similar variants of the correct implementations from the existing instruction tuning dataset. From these generated variants, we select the one with the most similarity to the correct implementation and highlight the differences as *error-sensitive segments* in this paper. We then annotate the tokens in the segments through a multi-granularity method at both line and token levels. Notably, this annotation is applied to both the correct implementations and the selected incorrect variant. The second challenge lies in guiding LLMs to focus on these labeled segments during fine-tuning, as standard SFT treats all tokens with equal importance regardless of their criticality. To address this challenge, we adjust the loss of SFT to prioritize the annotated error-sensitive tokens within correct implementations. Specifically, FGIT processes both correct and incorrect implementations to discriminate the *error-sensitive segments*, and computes loss based on correct code implementations. Unlike standard SFT, which uniformly weights all tokens during loss computation, we dynamically assign relatively higher weights to those tokens in the correct implementation that correspond to the *error-sensitive segments*. This methodology enhances LLMs' capability to discriminate *error-sensitive segments* when solving programming tasks, thereby increasing the likelihood of generating correct implementation details while suppressing error-prone alternatives.

To implement our method, we construct a refined dataset derived from the original instruction-tuning data. Each data point consists of an instruction, its correct implementation from the original dataset, and an LLM-generated similar incorrect variant. We then develop a multi-granularity error-sensitive segment extraction method and combine it with the refined loss function to enhance LLM's code generation capabilities.

We validate the effectiveness of the FGIT through extensive experiments. Notably, through FGIT, the selected LLMs achieve an average relative improvement of 6.9% on pass@1 across three representative code generation benchmarks (HumanEval(+), MBPP(+), and BigCodeBench) [8], [17]–[19]. Among these LLMs, SemCoder-S [20] with 6.7B parameters outperforms closed-source models like GPT-3.5-Turbo [21] on HumanEval(+) and MBPP(+) benchmarks, and MagiCoder$\mathcal{S}$-DS with 6.7B parameters outperforms GPT-3.5-Turbo on HumanEval(+). Our method also demonstrates strong generalization capabilities, showing performance improvements ranging from 3.8% to 19.1% across multiple instruction-tuned LLMs, including those trained on closed-source instruction datasets. Moreover, our ablation experiments on FGIT confirm the contributions of different granularities of differences and hyperparameters.

We summarize our contributions as follows.

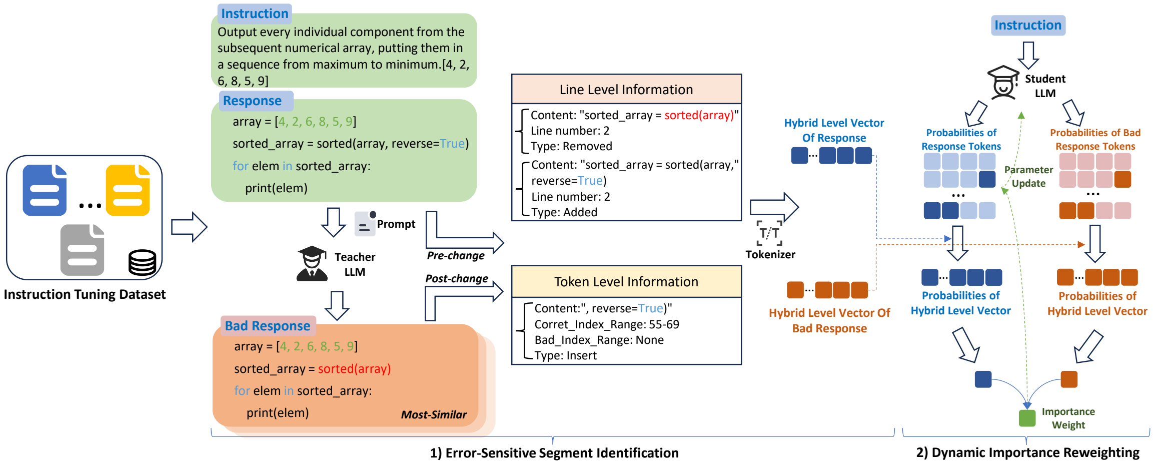
- To the best of our knowledge, we are the first to investigate how to enhance LLMs' understanding of *error-sensitive segments* by refining the SFT process to improve LLMs' code generation capabilities.
- We propose a novel framework, Fault-Guided Fine-Tuning (FGIT), to effectively guide LLMs to focus on error-prone parts in code. This is achieved by (1) extracting multi-granularity code differences (token-/line-level) to identify *error-sensitive segments*, and (2) refining SFT to dynamically assign higher weights to these parts during the training process.
- Through extensive experiments across seven LLMs and three widely-used code generation benchmarks, we demonstrate the effectiveness and generalizability of our approach in effectively boosting LLMs' code generation performance compared to baseline methods.

II. Approach

Figure 2 illustrates the overview of FGIT. This approach takes as input an instruction-tuned LLM and its corresponding instruction-tuning dataset, and outputs an enhanced LLM with improved code generation capabilities that can better discriminate *error-sensitive segments*. It first augments the dataset by generating similar yet incorrect implementations for each correct response. Then, it identifies *error-sensitive segments* between the paired implementations and calculates weights for tokens in the correct implementations that differ from the incorrect variants. During fine-tuning, it only computes loss on the correct implementations, with higher weights assigned to tokens in *error-sensitive segments*, producing an LLM that can better discriminate these *error-sensitive segments*. The methodology consists of two key components:

1) *Error-Sensitive Segments Identification*: This component creates a refined dataset of paired correct and similar yet wrong code samples from the original dataset, and processes code differences at multiple granularities to identify *error-sensitive segments*.
2) *Dynamic Importance Reweighting*: This component strategically reweights token weights in the loss function to prioritize discriminative elements in correct implementations, building upon the identified *error-sensitive segments*. This dynamic weighting method enhances the LLM's attention to the key implementation details in correct code, effectively teaching it to distinguish between valid solutions and their similar yet incorrect counterparts, ultimately improving code generation capabilities.

The two components work together to fine-tune LLMs to distinguish between correct implementations and similar yet incorrect alternatives, thereby improving performance.


Instruction Tuning Dataset
Instruction
Output every individual component from the subsequent numerical array, putting them in a sequence from maximum to minimum.[4, 2, 6, 8, 5, 9]
Response

```
array = [4, 2, 6, 8, 5, 9]
sorted_array = sorted(array, reverse=True)
for elem in sorted_array:
    print(elem)
```

Prompt
Teacher LLM
Bad Response

```
array = [4, 2, 6, 8, 5, 9]
sorted_array = sorted(array)
for elem in sorted_array:
    print(elem)
```

Most-Similar
Pre-change
Post-change
Line Level Information

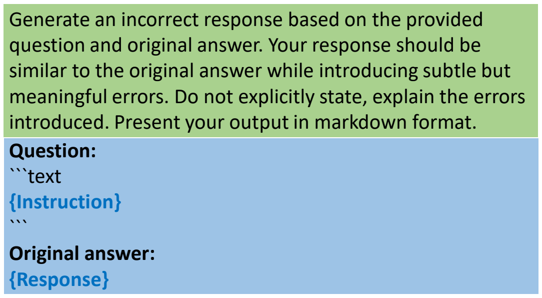
Content: "sorted_array = sorted(array)"
Line number: 2
Type: Removed

Content: "sorted_array = sorted(array," reverse=True)
Line number: 2
Type: Added

Content:", reverse=True)"
Corret_Index_Range: 55-69
Bad_Index_Range: None
Type: Insert

T/T
Tokenizer
Hybrid Level Vector Of Response
Hybrid Level Vector Of Bad Response
Instruction
Student LLM
Probabilities of Response Tokens
Probabilities of Bad Response Tokens
Parameter Update
Probabilities of Hybrid Level Vector
Probabilities of Hybrid Level Vector
Importance Weight
1) Error-Sensitive Segment Identification
2) Dynamic Importance Reweighting

Fig. 2: The overview of FGIT, taking one sample for explanation.



Generate an incorrect response based on the provided question and original answer. Your response should be similar to the original answer while introducing subtle but meaningful errors. Do not explicitly state, explain the errors introduced. Present your output in markdown format.

Question:
```text
{Instruction}
```
Original answer:
{Response}

Fig. 3: The prompt for generating similar yet incorrect response.

A. Error-Sensitive Segments Identification

The input to this component is the instruction-tuning dataset $\mathcal{D} = (c_i^{\text{correct}}, p_i^{\text{target}})_{i=1}^{N}$, where p_i^{target} represents the target problem description and c_i^{correct} denotes the correct implementation. The output is an enhanced dataset $\mathcal{D} = (c_i^{\text{correct}}, c_i^{\text{incorrect}}, p_i^{\text{target}})_{i=1}^{N}$ with error-sensitive segment information, where $c_i^{\text{incorrect}}$ represents the corresponding similar but incorrect implementation. To generate incorrect code variants, we utilize a teacher LLM with a carefully designed prompt. We choose an LLM-based approach over code mutation techniques because teacher models, by leveraging error commonalities learned from their extensive code corpora, offer better flexibility in producing a diverse range of incorrect variants. The prompt template is shown in Figure 3, which consists of two parts. The first part defines the task for producing incorrect responses corresponding to the target problem and answer, specifying that outputs should be similar to correct solutions, with responses constrained to markdown formatting for consistent post-processing. The second part provides contextual references to the target problem description and solution.

We generate multiple incorrect variants for each correct implementation and select the one that is most similar to the correct solution. To identify the most similar one, we employ an embedding-based approach because embeddings can capture semantic similarities that purely lexical comparisons might overlook. Specifically, we generate embeddings with UnixCoder [22] for its deep understanding of code structures and semantics derived from its diverse, large-scale code corpora. We then extract the differences to identify *error-sensitive segments* and process them at different granularity levels to capture both line-level and token-level information. Specifically, we designate $c^{\text{incorrect}}$ as the *pre-change* version and c^{correct} as the *post-change* version.

Line-Level Differences. We align c_i^{correct} and $c_i^{\text{incorrect}}$ line-by-line using Python's `difflib` library. For each line, it assigns a flag indicating whether it should be deleted ($-$), added ($+$), or remain unchanged. We extract the lines marked for deletion from $c^{\text{incorrect}}$ and those marked for addition from c^{correct}.

Let L_c and L_a denote the number of code lines in the correct code c^{correct} and incorrect code $c^{\text{incorrect}}$, respectively. Based on these extracted lines, we construct the line-level boolean mask vectors V_{line}^{c} and V_{line}^{a} for $c^{correct}$ and $c^{incorrect}$ as follows:

$$V_{\text{line}}^{c} = [v_1^c, v_2^c, \ldots, v_{L_c}^c], \text{where } v_i^c = I(\text{line}_i^c \text{ is added})$$
$$V_{\text{line}}^{a} = [v_1^a, v_2^a, \ldots, v_{L_a}^a], \text{where } v_j^a = I(\text{line}_j^a \text{ is deleted})$$

where $I(\cdot)$ is the indicator function that outputs 1 if the condition is true and 0 otherwise.

Token-Level Differences. We utilize the Levenshtein distance algorithm [23] to identify character-level change information between $c^{\text{incorrect}}$ and c^{correct}. The Levenshtein distance algorithm, also known as the edit distance algorithm, quantifies the minimum number of single-character operations (insertions, deletions, or substitutions) required to transform one string into another. We identify the characters that need to be edited to transform the original string ($c_i^{\text{incorrect}}$) into the modified version (c^{correct}), and record their positions accordingly. For instance, if a character operation is an insertion, we record its position in c^{correct}, as shown in Figure 2. Given that the LLM's embedding layer is tightly coupled with the LLM's tokenizer vocabulary, we map these character-level differences to tokens using the LLM's tokenizer. When character modifications span multiple tokens, all affected tokens are marked.

Let T_c and T_a denote the number of tokens in c^{correct} and $c^{\text{incorrect}}$. We construct token-level boolean mask vectors V^c_{token} and V^a_{token} for $c^{correct}$ and $c^{incorrect}$ as follows:

$$V^c_{\text{token}} = [w^c_1, \dots, w^c_{T_c}],\ w^c_k = I(\text{token}^c_k \text{ is added})$$
$$V^a_{\text{token}} = [w^a_1, \dots, w^a_{T_a}],\ w^a_\ell = I(\text{token}^a_\ell \text{ is deleted})$$

Hybrid Level Vectors. To create comprehensive representations of *error-sensitive segments*, we combine line-level and token-level masks. However, these two types of masks operate at different granularities and cannot be directly combined. To get line-level masks with token-level granularity, we first initialize a new token-level mask vector with all zeros, corresponding to the total number of tokens in the code. Then, for each line in the original code, all tokens belonging to that line are assigned the line's mask value (1 if the line is marked, 0 if it is unmarked) in this new token-level mask. This process yields the token-level representations $V^c_{\text{line-to-token}}$ and $V^a_{\text{line-to-token}}$ when applied to V^c_{line} and V^a_{line} respectively. We then use an element-wise addition operation to combine $V_{\text{line-to-token}}$ and V_{token}, as follows:

$$V^c_{\text{hybrid}} = V^c_{\text{line-to-token}} + V^c_{\text{token}}$$
$$V^a_{\text{hybrid}} = V^a_{\text{line-to-token}} + V^a_{\text{token}}$$

These hybrid vectors precisely identify *error-sensitive segments* at multiple granularities, highlighting critical differences between correct and incorrect implementations. Noted that changed tokens must appear in changed lines, our hybrid representation naturally creates a priority system: 1) tokens that are both in changed lines and are themselves changed will have a value of 2 in the hybrid vector; 2) tokens that are only in changed lines but not directly changed will have a value of 1. This provides a more comprehensive view than either granularity alone, with higher values indicating more critical tokens.

B. Dynamic Importance Reweighting

With the identified *error-sensitive segments*, we now refine the SFT process to prioritize these critical differences. Based on the constructed dataset $\mathcal{D} = \{(c^{\text{correct}}_i, c^{\text{incorrect}}_i, p^{target}_i)\}^N_{i=1}$, the standard SFT loss is computed as:

$$\mathcal{L}_{SFT} = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{T_c} logP(c^{\text{correct}}_{i,j}|p^{target}_i, c^{\text{correct}}_{i,1:j-1}) \qquad (1)$$

where N denotes the number of samples in a batch. Notably, the standard SFT loss function treats all tokens equally.

In contrast, FGIT introduces dynamic token-level weights $W = w_1, w_2, ..., w_j$ emphasize *error-sensitive segments*:

$$\mathcal{L}_{Fault} = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{T_c} w_j \cdot logP(c^{\text{correct}}_{i,j}|p^{target}_i, c^{\text{correct}}_{i,1:j-1}) \quad (2)$$

The weight W is computed as follows: Given input $x = p^{\text{target}}$, outputs $y^c = c^{correct}$ and $y^a = c^{incorrect}$, we first obtain the LLM's prediction probabilities for both correct and incorrect implementations given the same instruction:

$$P^c = f_\theta(y^c_k \mid y^c_{1:k-1}, x) \qquad (3)$$
$$P^a = f_\theta(y^a_l \mid y^a_{1:l-1}, x) \qquad (4)$$

where f_θ represents the conditional probability function of the LLM that computes the probability of the next token given the input x and previous tokens. We then apply the hybrid-level vectors to isolate probabilities for *error-sensitive segments*:

$$H^c = P^c \odot V^c_{\text{hybrid}} \qquad (5)$$
$$H^a = P^a \odot V^a_{\text{hybrid}} \qquad (6)$$

Where $\odot$ denotes element-wise multiplication. Inspired by the Bradley–Terry model [24], a pairwise comparison framework widely used in ranking systems [25]–[27], we compute dynamic token weights W for differentiating tokens in *error-sensitive segments*:

$$W = \alpha\ -|\frac{\overline{H^c} - \overline{H^a}}{\overline{H^c} + \overline{H^a}}|$$

where α is a hyperparameter controlling the weight range, $\overline{H^c}$ denotes the mean probability of tokens in *error-sensitive segments* in c^{correct}_i, and $\overline{H^a}$ represents the corresponding value for $c^{\text{incorrect}}$. This formulation ensures that: (1) When the mean probabilities $\overline{H^c}$ and $\overline{H^a}$ are close (indicating the LLM struggles to distinguish the tokens between c^{correct} and $c^{\text{incorrect}}$), the weights for differentiating tokens in c^{correct} approach α, thereby maximizing emphasis on *error-sensitive segments*. (2) Conversely, when $\overline{H^c}$ and $\overline{H^a}$ diverge significantly (demonstrating the LLM can discriminate the differentiating tokens in c^{correct} and $c^{\text{incorrect}}$), the weights diminish toward $\alpha - 1$, reducing emphasis. For tokens shared between c^{correct} and $c^{\text{incorrect}}$, we assign fixed weights $\alpha - 1$, ensuring the LLM maintains baseline attention to shared elements while prioritizing discriminative features. To be noted that this dynamic weight W is differentiable, as $\overline{H^c}$ and $\overline{H^a}$ are derived from the model's output probabilities for c^{correct} and $c^{\text{incorrect}}$ within the *error-sensitive segments*. Consequently, through gradient updates, the optimization process reinforces c^{correct}, also implicitly steering the model away from generating $c^{\text{incorrect}}$.

This dynamic weighting mechanism guides the LLM to focus on challenging discriminative aspects of correct implementations, which can improve its code generation capability.

III. Experiments Setup

A. Benchmarks and Metrics

We conduct experiments on three widely used code generation benchmarks to demonstrate the superiority and generality of FGIT. We use HumanEval(+), MBPP(+) [8], [17], [18] and Bigcodebench [19]. Specifically, our study uses both the full set and hard set with complete configuration of Bigcodebench, namely, BigCodeBench-Full and BigCodeBench-Hard.

To evaluate performance, we use the Pass@K metric, which is widely used in prior studies [8], [28], [29]. Following prior studies [30]–[32], our experimental design adopts K=1, focusing exclusively on first-attempt success rates. This metric also aligns with real-world scenarios where developers aim to produce accurate code on the first attempt [31].

B. Implementation Detail

1) Data generation: We use Qwen2.5-Coder-32B-Instruct [33] as the teacher model with temperature=0.8 to generate three incorrect code implementations for each sample. Our rationale for selecting this model is threefold. First, generating an incorrect solution by referencing a correct one is comparatively less demanding on a model's capabilities than generating a correct solution from scratch. Secondly, our task benefits from a model with strong coding abilities, and Qwen2.5-Coder-32B-Instruct possesses high coding proficiency; for instance, its performance on HumanEval(+) and MBPP(+) even surpasses that of GPT-4o. Lastly, using open-source models also facilitates reproducibility.

To investigate whether the teacher model can generate implementations that are incorrect and similar to the correct solutions, we employ both manual inspection and automated quantitative validation. First, two authors independently examine a sample of 50 generated outputs, which are randomly sampled from the Evol-instruct dataset, assessing them based on two criteria: functional correctness and similarity to the original correct code. A discussion is held to resolve the disagreements. We do not invite others because all the disagreements are resolved during discussion. The Cohen's Kappa coefficient [34] is 0.78. Our analysis shows that the LLM could produce similar yet incorrect samples as expected: 94% of the generated samples are similar to the correct ones yet incorrect, while the remaining 6% introduce changes that do not affect correctness. To quantitatively analyze the correctness and similarity of the generated responses, we select benchmarks with test cases for further validation. Specifically, we choose HumanEval and MBPP to generate incorrect yet similar responses and use Unixcoder [22] to calculate cosine similarity. We find that 94.1% of generated responses are incorrect using the test cases in benchmarks, and the average similarity score is 0.95, aligning with our needs. Due to space limitations, the manually checked samples and generated responses are in our replication package.

2) Settings: All experiments are conducted on a machine with eight Tesla A800 GPUs, each with 80 GB of memory. α is set to 2, which means the weight range of W is (1,2). Given that FGIT is designed for already instruction-tuned LLMs, where the objective is not to train them extensively from a base state but rather to refine their existing capabilities to better handle *error-sensitive segments* present, we train all models for one epoch with a relatively low learning rate of 5e-6, aiming to gently adapt the model and prevent catastrophic forgetting of previously learned knowledge, while efficiently instilling the new focus on these critical segments [35], [36]. The max sequence length is 1024. For inference evaluation, we use greedy decoding to ensure deterministic outputs, which also aligns with prior studies [8], [28].

IV. Results

In this section, we report and analyze the experimental results to answer the following research questions (RQs):

- RQ1: How effective is our approach in improving code generation across different benchmarks?
- RQ2: How do different components of the FGIT method contribute to LLMs' performance?
- RQ3: Does FGIT demonstrate generalizability across different LLMs and their corresponding instruction-tuning datasets?
- RQ4: Does FGIT work for instruction-tuned LLMs whose instruction-tuning dataset is closed-source?

A. RQ1: Overall Effectiveness

In this RQ, we evaluate the effectiveness of our approach by applying it to several Instruction-tuned LLMs using their corresponding instruction-tuning datasets and assess their performance against four baselines:

Base Models: We use the original instruction-tuned LLMs without any additional FGIT as our base models. This comparison demonstrates the absolute improvement achieved through FGIT. Specifically, we select three representative instruction-tuned LLMs: MagiCoder$\mathcal{S}$-CL [7], MagiCoder$\mathcal{S}$-DS [7] and SemCoder-S [20] as our base models.

Closed-Source Model: We include GPT-3.5-Turbo [21] as the closed-source baseline to illustrate the performance gap between our fault-guided fine-tuned LLMs and the advanced closed-source LLM.

Standard-SFT Models: We apply standard SFT on the same base models to create this baseline. This comparison serves two purposes: (1) to examine whether further fine-tuning on coarse-grained instruction-response mappings on their existing dataset can improve performance over the original models, and (2) to highlight the superior performance of our approach in learning fine-grained *error-sensitive segments*.

Other Open Source Instruction-Tuned Code LLMs: We include other instruction-tuned versions of the same base models, which are trained on additional data generated by stronger LLMs (e.g., GPT-4) in addition to Evol-Instruct. Specifically, we include WaveCoder-Ultra [3], OpenCodeInterpreter [2], and AlchemistCoder [37]. This comparison aims to illustrate how the performance of the selected LLMs with FGIT compares to that of current popular instruction-tuned models based on the same foundational model.

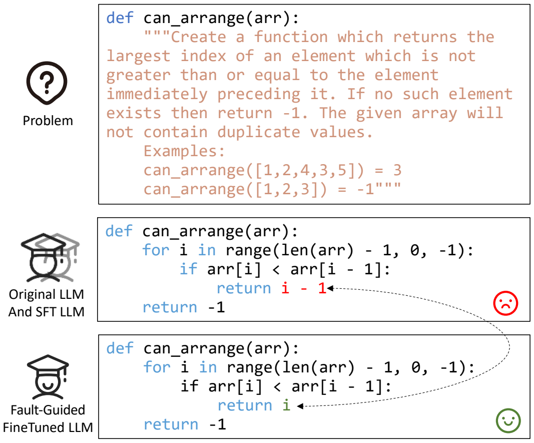
We choose Evol-instruct [7], a decontaminated version of evol-codealpaca-v1 [38], which contains numerous high-quality instruction-following data, as our training dataset. This dataset is decontaminated by removing data that contain docstrings or solutions from multiple benchmarks [8], [17], [39], [40]. We conduct an additional decontamination of this dataset for docstrings or solutions from BigCodeBench following [7], and find no overlap. For MagiCoder$\mathcal{S}$-CL and MagiCoder$\mathcal{S}$-DS, this dataset is their original instruction-tuning dataset. For SemCoder-S, this dataset is a subset of its original instruction-tuned dataset, which is not fully open-sourced.

TABLE I: Performance of different LLMs using FGIT method compared with Standard-SFT on HumanEval(+), MBPP(+) and BigCodeBench, where BCB stands for BigCodeBench.

Model	HumanEval(+)	MBPP(+)	BCB Full	BCB Hard
Closed-Source Models				
GPT-3.5-Turbo (Nov 2023)	76.8 (70.7)	82.5 (69.7)	50.6	21.6
Base Model: CodeLlama-Python-7B				
OpenCodeInterpreter-CL	72.6 (67.7)	66.4 (55.4)	33.2	6.1
AlchemistCoder-CL-7B	74.4 (68.3)	68.5 (55.1)	33.1	6.7
MagiCoder$\mathcal{S}$-CL	70.7 (66.5)	68.4 (56.6)	39.7	12.8
+Standard-SFT	69.5 (64.0)	69.3 (58.7)	39.3	13.5
+FGIT	**73.2 (68.9)**	**71.7 (59.5)**	**42.2**	**15.5**
Base Model: DeepseekCoder-6.7B-Base				
WaveCoder-Ultra-DS-6.7B	75.0 (69.5)	74.9 (63.5)	43.7	16.9
OpenCodeInterpreter-DS	76.2 (72.0)	76.2 (72.0)	44.6	16.9
AlchemistCoder-DS-6.7B	79.9 (75.6)	77.0 (60.2)	42.5	12.2
MagiCoder$\mathcal{S}$-DS	76.8 (71.3)	75.7 (64.4)	47.6	12.8
+Standard-SFT	75.6 (70.7)	79.1 (66.4)	46.9	10.8
+FGIT	**77.4 (74.3)**	**79.6 (69.0)**	**48.2**	**15.5**
SemCoder-S	79.3 (74.4)	79.6 (68.5)	48.5	16.9
+Standard-SFT	79.9 (75.0)	80.7 (67.2)	47.1	16.2
+FGIT	**83.5 (78.7)**	**83.1 (70.6)**	**48.9**	**20.3**

We report results consistently from the EvalPlus leaderboard[1] and BigCodeBench leaderboard[2]. Table I presents the performance of LLMs with FGIT and the baselines across HumanEval(+), MBPP(+), and BigCodeBench. Overall, LLMs with FGIT demonstrate substantial improvements in code generation. We observe that LLMs with FGIT show average relative performance improvements of 4.8% over the base model and 4.9% over LLMs with Standard-SFT on HumanEval(+), MBPP(+), and BigCodeBench. Notably, with FGIT, SemCoder-S with only 7B parameters outperforms GPT-3.5-Turbo on HumanEval(+) and MBPP(+), and MagiCoder$\mathcal{S}$-DS outperforms GPT-3.5-Turbo on HumanEval(+). Both LLMs achieve comparable performance to GPT-3.5-Turbo on BigCodeBench, further validating the exceptional effectiveness of FGIT in enhancing code generation capabilities. While the improvement on BigCodeBench-Full is modest, our approach shows more gains on BigCodeBench-Hard (e.g., 20.8% relative improvement for SemCoder-S). This is likely because more challenging problems contain more *error-sensitive segments*, and FGIT is designed to guide LLMs to handle these *error-sensitive segments*, thus showing greater effectiveness on difficult programming tasks.

When comparing the effects of FGIT versus Standard-SFT on base models, we observe that Standard-SFT provides limited improvements and sometimes even weakens the base models. For example, SemCoder-S with Standard-SFT achieves only 0.8% relative performance improvement on HumanEval(+) and suffers 1.9% relative performance decline on MBPP+. This suggests that simply reinforcing the coarse-grained instruction-response mappings on their existing dataset provides minimal benefits, as these models have already captured these general mappings well during their initial instruction tuning. When compared with other instruction-tuned models, we observe that by enabling models to further learn the *error-sensitive segments* within the original dataset through FGIT, their performance can achieve comparable or even superior results.



```
def can_arrange(arr):
    """Create a function which returns the
largest index of an element which is not
greater than or equal to the element
immediately preceding it. If no such element
exists then return -1. The given array will
not contain duplicate values.
    Examples:
    can_arrange([1,2,4,3,5]) = 3
    can_arrange([1,2,3]) = -1"""
```

```
def can_arrange(arr):
    for i in range(len(arr) - 1, 0, -1):
        if arr[i] < arr[i - 1]:
            return i - 1
    return -1
```

```
def can_arrange(arr):
    for i in range(len(arr) - 1, 0, -1):
        if arr[i] < arr[i - 1]:
            return i
    return -1
```

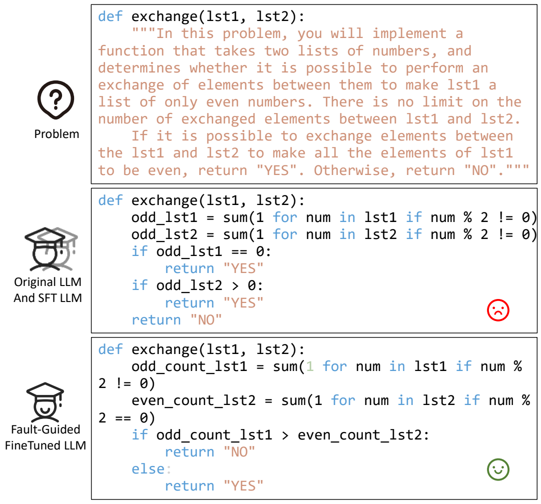
Fig. 4: A case demonstrating how LLMs after FGIT can better focus on *error-sensitive segments* to generate the correct solution.

To further figure out the reasons for FGIT improving LLMs' ability to generate functionally correct code, we manually inspect the results. Based on our analysis, FGIT demonstrates two main advantages over both the original model and Standard-SFT:

First, FGIT can learn diverse *error-sensitive segments* to better recognize and focus on implementation details that are prone to errors, while the original model and Standard-SFT only learn the overall mapping from problem to solution. This method improves the LLM's attention to key implementation choices. We manually examine the tasks from HumanEval(+) that are correctly solved by applying FGIT but are incorrect when applying Standard-SFT. We find that for SemCoder-S, FGIT correct 11 tasks where Standard-SFT failed. In 81.8% of these instances, Standard-SFT's failure stemmed from incorrectly handling *error-sensitive segments*; these errors typically required modifications of three lines of code or fewer to be rectified. FGIT, in contrast, provides a correct implementation, with similar rates observed in MagiCoder$\mathcal{S}$-DS (71.4%) and MagiCoder$\mathcal{S}$-CL (66.7%). For example, Figure 4 presents a comparison of the results of three versions of SemCoder-S on the HumanEval/135 task. In this example, an error-sensitive segment involves deciding whether to return the index of the target element itself *i* or the index of its previous element *i-1*. This distinction directly impacts the functional correctness of the implementation. Both the original model and the model with Standard-SFT incorrectly return the index of the previous element, while the model with FGIT correctly returns the index of the target element. These results further confirm our method's effectiveness in guiding LLMs to recognize *error-sensitive segments*.

Second, by developing a deeper understanding of critical

[1] https://evalplus.github.io/leaderboard.html
[2] https://bigcode-bench.github.io



```python
def exchange(lst1, lst2):
    """In this problem, you will implement a
function that takes two lists of numbers, and
determines whether it is possible to perform an
exchange of elements between them to make lst1 a
list of only even numbers. There is no limit on the
number of exchanged elements between lst1 and lst2.
    If it is possible to exchange elements between
the lst1 and lst2 to make all the elements of lst1
to be even, return "YES". Otherwise, return "NO"."""
```

```python
def exchange(lst1, lst2):
    odd_lst1 = sum(1 for num in lst1 if num % 2 != 0)
    odd_lst2 = sum(1 for num in lst2 if num % 2 != 0)
    if odd_lst1 == 0:
        return "YES"
    if odd_lst2 > 0:
        return "YES"
    return "NO"
```

```python
def exchange(lst1, lst2):
    odd_count_lst1 = sum(1 for num in lst1 if num %
2 != 0)
    even_count_lst2 = sum(1 for num in lst2 if num %
2 == 0)
    if odd_count_lst1 > even_count_lst2:
        return "NO"
    else:
        return "YES"
```

Fig. 5: A case demonstrating how FGIT can improve overall code generation performance.

code segments, FGIT also enhances overall code generation capabilities. By strategically emphasizing *error-sensitive segments* while maintaining appropriate weight for contextual elements, the LLM learns to identify and handle the crucial parts of implementations that determine correctness. Figure 5 demonstrates this using an example from SemCoder-S on HumanEval/110, which requires determining whether swapping elements between two lists can make all elements in *lst1* even. This case highlights improvements that go beyond addressing specific *error-sensitive segments*. Both the original model and Standard-SFT fail to implement the correct verification logic to determine whether *lst2* contains enough even numbers to replace odd numbers in *lst1*. In contrast, the Fault-Guided Fine-Tuned model correctly implements this logic, demonstrating enhanced general coding abilities rather than just handling error-sensitive parts.

> **RQ1 Summary:** FGIT delivers consistent and substantial performance improvements across all three benchmarks, with enhanced LLMs even outperforming GPT-3.5-Turbo on certain benchmarks. The results confirm that explicitly learning fine-grained error-sensitive segment mappings is more effective than simply retraining on coarse-grained instruction-response pairs.

B. RQ2: Component Analysis

To understand how different components contribute to the effectiveness of FGIT, we conduct ablation studies focusing on the impacts of multi-granularity and the loss function.

Impact of Difference Granularity. We explore the impact of code difference granularity, which involves synthesizing line-level and token-level code differences to identify *error-sensitive segments*. Specifically, we conduct ablation experiments using MagiCoder$\mathcal{S}$-DS and SemCoder-S as base models and perform evaluation on three selected benchmarks. Table II shows the impact of different granularities of differences on FGIT. We can observe that the combination of line-level granularity and token-level granularity yields maximum performance gains. For example, when applied to SemCoder-S, this approach achieves a relative average improvement of 4.9% across all benchmarks compared to the base model, compared to just 2.9% for line-level only and 2.4% for token-level only.

TABLE II: Performance ablation of different granularities of differences on HumanEval(+), MBPP(+) and BigCodeBench based on MagiCoder$\mathcal{S}$-DS and SemCoder-S, where BCB stands for BigCodeBench.

Model	HumanEval(+)	MBPP(+)	BCB Full	BCB Hard
MagiCoderS-DS	76.8 (71.3)	75.7 (64.4)	47.6	12.8
+FGIT (Line Level)	75.6 (72.0)	78.8 (68.3)	47.1	14.2
+FGIT (Token Level)	76.2 (72.6)	79.1 (68.5)	47.3	14.2
+FGIT	**77.4 (74.3)**	**79.6 (69.0)**	**48.2**	**15.5**
SemCoder-S	79.3 (74.4)	79.6 (68.5)	48.5	16.9
+FGIT (Line Level)	80.5 (76.8)	**83.1** (70.1)	**49.1**	18.2
+FGIT (Token Level)	81.1 (76.8)	82.8 (70.1)	48.3	16.9
+FGIT	**83.5 (78.7)**	**83.1 (70.6)**	48.9	**20.3**

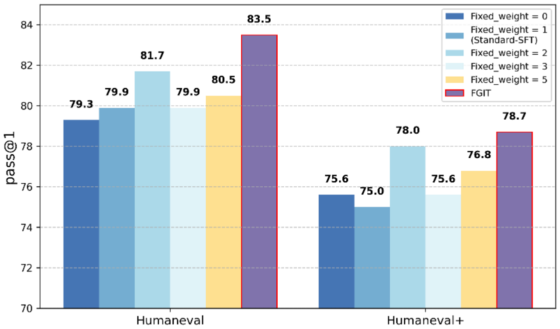


Fig. 6: Performance of different weights based on SemCoder-S on HumanEval(+)

We further investigate models' ability to handle *error-sensitive segments* after combining both granularities, compared to the base models. We select the tasks from HumanEval(+) that are correctly solved after applying FGIT but initially incorrect with the base models. We find that among these tasks, 63.6% of SemCoder-S's improvements result from properly handling *error-sensitive segments*, with these specific *error-sensitive segments*-related corrections involving modifications of three lines or fewer. Similar rates are observed in MagiCoder$\mathcal{S}$-DS (71.4%). This demonstrates that multi-granularity differences enable better *error-sensitive segments*, thereby enhancing LLMs' code generation capabilities.

Impact of the Loss Function. To validate the effectiveness of our dynamic loss weighting design, we compare our dynamic weighting approach against a fixed weighting strategy where all error-sensitive tokens receive the same constant weight during training. We experiment with fixed weights in the range of $[0, 1, 2, 3, 5]$ on HumanEval(+). Due to the evaluation time constraints, we select SemCoder-S as the representative LLM for this ablation study, as it demonstrates the best overall per-

TABLE III: Performance of FGit on other instruction-tuned LLMs with their corresponding datasets on HumanEval(+), MBPP(+) and BigCodeBench, where BCB stands for BigCodeBench.

Model	HumanEval(+)	MBPP(+)	BCB Full	BCB Hard
(Corresponding Dataset OSS-INSTRUCT)				
MagiCoder-DS	66.5 (60.4)	75.4 (61.9)	43.4	12.2
+Standard-SFT	64.6 (58.5)	79.1 (66.1)	43.9	12.2
+FGit	**67.1 (62.2)**	**79.4 (66.4)**	**46.2**	**15.5**
(Corresponding Dataset PYX)				
SemCoder	73.2 (68.9)	79.9 (65.3)	43.5	16.9
+Standard-SFT	71.3 (65.2)	79.9 (66.4)	43.4	14.2
+FGit	**73.7 (69.5)**	**81.0 (67.2)**	**47.9**	**21.6**

formance with FGit. Notably, when the fixed weight equals 1, this configuration is equivalent to standard SFT. Figure 6 shows the performance of SemCoder-S with different fixed weights compared to our dynamic weighting approach. Our experimental results reveal two important findings: (1) A fixed weight of 2 yields better performance than other fixed weight values, suggesting that an appropriate constant weight can help the LLM recognize *error-sensitive segments* and enhance code generation capabilities. (2) Our dynamic weighting approach still outperforms the best fixed weighting configuration. This confirms that dynamically adjusting weights based on the LLM's current discrimination ability provides better guidance for the LLM to focus on critical implementation details that differentiate correct solutions from their erroneous variants.

> **RQ2 Summary:** All components in FGit contribute to the performance. Combining different levels of granularity of code differences (line + token level) is critical to performance. The loss function with dynamic weighting strategies outperforms that with fixed weighting strategies, highlighting the effectiveness of our weighting method.

C. RQ3: The Generalization Capabilities

In this RQ, we aim to explore the generalizability of FGit across different instruction-tuned LLMs when using their own instruction-tuning datasets. Specifically, we select two representative LLMs and their corresponding instruction datasets. 1) We select MagiCoder-DS and its corresponding dataset OSS-Instruct. This dataset is generated from open-sourced code by GPT-3.5-Turbo, and contains 75K samples. 2) SemCoder and its corresponding dataset PYX. This dataset consists of 95K samples, including comprehensive reasoning texts with executable code samples. The dataset is constructed with problem descriptions generated by GPT-3.5-Turbo and corresponding responses generated by GPT-4o-mini [41], creating high-quality instruction-response pairs with detailed reasoning. For each LLM, we process its corresponding dataset through our pipeline and evaluate performance on the same benchmarks used in RQ1. To be noted, we also decontaminate the datasets, adhering to [7], and find no data overlap with our selected evaluation benchmarks. We select Base Models and Standard-SFT Models as our baselines. By using LLMs with different training paradigms and datasets with varying characteristics (open-sourced code versus detailed reasoning with executable samples), we can verify that our approach is not tied to specific LLM series or dataset properties, but rather provides universal benefits.

TABLE IV: Performance of LLMs trained on closed-source instruction-tuning datasets after using FGit on HumanEval(+), MBPP(+) and BigCodeBench, where BCB stands for BigCodeBench.

Model	HumanEval(+)	MBPP(+)	BCB Full	BCB Hard
Base Model: CodeLlama-Python-7B				
CodeLlama-Instruct	36.0 (31.1)	56.1 (46.6)	25.7	4.1
+Standard-SFT	39.0 (34.1)	61.1 (49.7)	26.5	4.1
+FGit	**47.0 (43.9)**	**61.9 (51.3)**	**29.0**	**4.7**
Base Model: DeepseekCoder-6.7B-Base				
DeepseekCoder-Instruct	73.8 (70.7)	74.9 (65.6)	43.8	15.5
+Standard-SFT	75.6 (70.1)	77.8 (66.9)	42.0	13.5
+FGit	**81.7 (76.2)**	**78.6 (66.9)**	**44.3**	**16.9**

Table III shows the performance of LLMs on HumanEval(+), MBPP(+), and BigCodeBench after FGit and Standard-SFT. We can observe that FGit demonstrates robust generalization capabilities to different instruction-tuning LLMs and their corresponding datasets. For MagiCoder-DS and SemCoder, after FGit, the average performances across all benchmarks show relative improvements of 5.3% and 3.8% compared to the base models. In contrast, standard SFT yielded modest relative improvements of 1.4% for MagiCoder-DS and decreased performance by 2.1% for SemCoder. These results show that FGit's benefits are not tied to specific dataset characteristics or model series. Instead, the approach effectively enhances diverse instruction-tuned LLMs by teaching them to focus on *error-sensitive segments* in correct solutions.

> **RQ3 Summary:** FGit exhibits strong generalizability across different instruction-tuned LLMs and their corresponding datasets, consistently outperforming standard SFT.

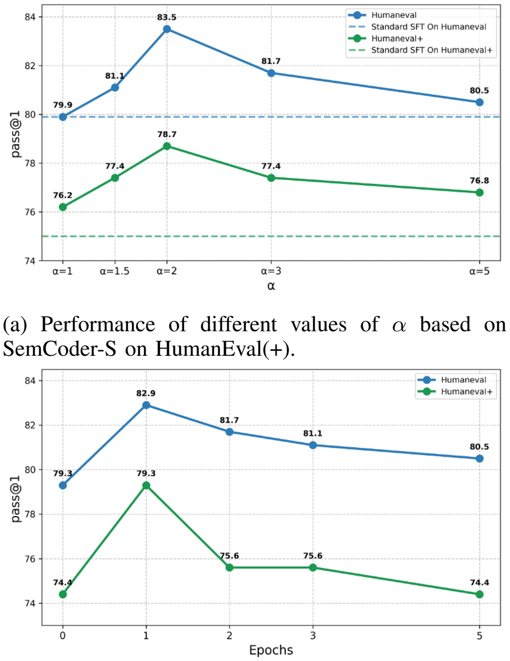
D. RQ4: Effectiveness on LLMs with Closed-Source Instruction Data

A key question for the broader adoption is whether FGit can enhance LLMs whose original instruction-tuning datasets are not publicly available. To investigate this, we applied our method to two widely-used LLMs with closed-source training data in this RQ. Specifically, we choose CodeLlama-7B-Instruct [42] and DeepSeek-Coder-6.7B-Instruct [43] as base models. These LLMs are instruction-tuned on substantial but proprietary datasets - CodeLlama-7B-Instruct underwent instruction tuning on approximately 5B tokens of instruction data, while DeepSeekCoder-6.7B-Instruct is tuned on around 2B tokens. To test our approach without access to these original datasets, we select Evol-Instruct, used in the main experiment, as the training dataset. We select Base Models, and Base Models with Standard SFT as our baselines and evaluate on HumanEval(+), MBPP(+) and BigCodeBench(+).

Table IV shows the performance of these two LLMs using Standard-SFT and fault-fine-tuning on HumanEval(+), MBPP(+), and BigCodeBench. We can find that FGIT is also applicable to LLMs with closed-source datasets. For CodeLlama-Instruct and DeepseekCoder-Instruct, after FGIT, the average relative improvements across all benchmarks are 19.1% and 5.9%. By comparison, the standard SFT yield gains of 7.5% and 0.5%, respectively. This further demonstrates that fault-fine-tuning is also applicable to LLMs with closed-source datasets, showing strong applicability.

RQ4 Summary: FGIT demonstrates strong applicability to instruction-tuned LLMs with closed-source datasets, delivering particularly dramatic improvements for initially weaker models. This expands the method's application scope to broader scenarios where original training datasets are inaccessible, offering a path to enhance LLMs without requiring access to their proprietary training data.

V. Discussion



(a) Performance of different values of α based on SemCoder-S on HumanEval(+).

(b) Performance of different epochs based on SemCoder-S on HumanEval(+).

Fig. 7: Performance analysis of SemCoder-S on HumanEval(+): varying hyperparameter α (top), and varying training epochs (bottom).

Impact of hyperparameters. We first analyse the impact of the parameter α in our loss function, which controls the emphasis placed on error-sensitive tokens, by changing α. Specifically, we conduct experiments with $\alpha \in [1, 1.5, 2, 3, 5]$ on HumanEval(+) and observe the performance changes of LLMs. Due to the evaluation time constraints, we select SemCoder-S as the representative LLM for this ablation study, as it demonstrates the best overall performance with FGIT. Figure 7a illustrates the performance trends as α varies. We can observe that $\alpha = 2$ provides an optimal balance between emphasizing error-sensitive tokens and maintaining attention to shared tokens. Additionally, it can be observed that in all cases, after FGIT, the LLM's performance matches or exceeds that of Standard-SFT, demonstrating the robustness to hyperparameter choices.

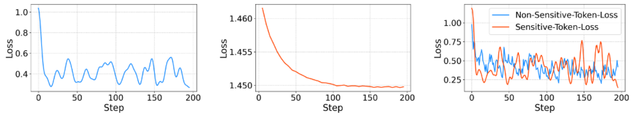
TABLE V: Performance of different LLMs using FGIT method compared with DPO on HumanEval(+), MBPP(+) and BigCodeBench, where BCB stands for BigCodeBench.

Model	HumanEval(+)	MBPP(+)	BCB Full	BCB Hard
Base Model: CodeLlama-Python-7B				
MagiCoder$\mathcal{S}$-CL	70.7 (66.5)	68.4 (56.6)	39.7	12.8
+DPO	66.5 (61.6)	68.8 (58.7)	39.8	14.2
+FGIT	**73.2 (68.9)**	**71.7 (59.5)**	**42.2**	**15.5**
Base Model: DeepseekCoder-6.7B-Base				
MagiCoder$\mathcal{S}$-DS	76.8 (71.3)	75.7 (64.4)	47.6	12.8
+DPO	76.2 (71.9)	79.1 (68.3)	47.8	13.5
+FGIT	**77.4 (74.3)**	**79.6 (69.0)**	**48.2**	**15.5**
SemCoder-S	79.3 (74.4)	79.6 (68.5)	48.5	16.9
+DPO	81.7 (76.2)	81.0 (67.7)	47.9	16.2
+FGIT	**82.9 (79.3)**	**83.1 (70.6)**	**48.9**	**20.3**

Then we investigate the impact of the number of training epochs, which also influences the learning emphasis on *error-sensitive segments*. Specifically, we conduct experiments with epochs in $[1, 2, 3, 5]$ using SemCoder-S and observe the performance changes on HumanEval(+). Figure 7b illustrates the performance trends as the number of training epochs varies. We find that an excessive number of training epochs can lead to a decline in performance. This might be because the model over-focuses on *error-sensitive segments*, neglecting the importance of surrounding tokens.

Compared to Reinforcement Learning Method. Given the increasing popularity of reinforcement learning (RL) methods for improving code generation [44], [45], we believe it's important to compare our approach with these established techniques. As RL methods in code generation typically aim to align model outputs with desired code solutions by increasing the probability of correct implementations while reducing the likelihood of erroneous ones, they share similarities with our work principle of enhancing LLMs' ability to identify *error-sensitive segments* to improve code generation capabilities. Specifically, we compare our approach with the representative DPO method [46], which is widely used and has demonstrated significant advantages in code generation [47]–[51]. This method works by training models to directly maximize the likelihood of preferred outputs over non-preferred ones without requiring explicit reward modeling, learning from paired examples of more and less desirable code implementations.

To ensure a fair comparison, we select the same LLMs and use identical experimental settings as stated in RQ1 for DPO training. The training dataset remains consistent across both DPO and FGIT, and we evaluate and compare the performance of DPO and FGIT on HumanEval(+), MBPP(+),



(a) (b) (c)

Fig. 8: Learning curves for SemCoder-S with FGIT on Evol-Instruct. (a) training loss, (b) validation loss, and (c) training loss categorized by token type.

and BigCodeBench. Table V shows the performance of LLMs trained with different methods. Overall, LLMs with FGIT consistently outperform those trained with DPO. We can observe that FGIT outperforms DPO by a relative average of 4.2%, across three selected benchmarks. This advantage stems from fundamental methodological differences: while DPO relies on coarse-grained preference signals that cannot precisely target *error-sensitive segments*, FGIT specifically maintains learning across all tokens while strategically emphasizing *error-sensitive segments* within code implementations. This approach ensures the LLM retains general coding knowledge while becoming more attentive to critical details that often determine functional correctness.

Additionally, we note that DPO's ability to differentially reward correct implementations and penalize incorrect ones could be leveraged to enhance the learning of error-sensitive segments. Specifically, a tailored reward function could be designed to strengthen the model's focus on these critical segments, potentially combining the strengths of both approaches. We leave this promising direction for future exploration.

Analysis of Overfitting Dynamics. To investigate the possibility of potential overfitting, we analyze the learning dynamics. Specifically, we randomly partition Evol-Instruct into a 9:1 training-validation split and reduce the batch size to obtain a finer-grained view of the loss curves. Figure 8a and Figure 8b illustrate the learning curves for the SemCoder-S model. Notably, the validation loss for FGIT shows a consistent decrease, suggesting FGIT mitigates the overfitting issue. We hypothesise that by assigning higher weights to more challenging, error-sensitive tokens, FGIT guides the model to focus on harder-to-learn patterns, thus mitigating overfitting. Furthermore, to ensure comprehensive learning across the entire code structure, we assign a non-zero base weight $(\alpha - 1)$ to non-sensitive tokens. Figure 8c confirms that the loss for both sensitive and non-sensitive tokens decreases during training.

Comparison with Mutation-Based Method. Traditional mutation-based approaches provide a natural baseline for our method, as they are also designed to introduce errors into code. We replace the LLM-based method with the rule-based universalmutator [52] to generate faulty versions of the code in Evol-Instruct. It is a state-of-the-art, language-agnostic mutation tool. We train SemCoder-S using these mutated samples, keeping all hyperparameters consistent. As shown in Table VI, FGIT outperforms the mutation-based approach. We attribute this to the fact that LLMs, trained on vast code corpora, generate a more diverse and realistic spectrum of plausible errors than a fixed set of mutation rules, leading to more effective learning.

TABLE VI: Performance comparison between FGIT and the traditional mutation-based method on HumanEval(+), MBPP(+) and BigCodeBench, where BCB stands for BigCodeBench.

Method	Humaneval (+)	MBPP(+)	BCB Full	BCB Hard	Avg
Original	79.3 (74.4)	79.6 (68.5)	48.5	16.9	61.2
Mutation-Based	81.1 (77.4)	80.9 (67.7)	**49.2**	17.5	62.3
FGIT	**83.5 (78.7)**	**83.1 (70.6)**	48.9	**20.3**	**64.2**

TABLE VII: Performance Impact of Standard-SFT on seen and unseen datasets on HumanEval(+), MBPP(+) and BigCodeBench, where BCB stands for BigCodeBench.

Model	Dataset	Humaneval (+)	MBPP (+)	BCB Full	BCB Hard	Avg
MagiCoder-DS	None	66.5 (60.4)	75.4 (61.9)	43.4	12.2	53.3
	OSS-Instruct (Original)	64.6 (58.5)	**79.1 (66.1)**	43.9	12.2	54.1
	PYX (New)	**66.5 (60.9)**	77.5 (65.8)	45.7	**15.5**	**55.3**
	Evol-Instruct (New)	64.6 (60.9)	77.2 (65.6)	**47.3**	**15.5**	55.2
SemCoder	None	73.2 (68.9)	79.9 (65.3)	43.5	16.9	58
	PYX (Original)	71.3 (65.2)	**79.9 (66.5)**	43.4	14.2	56.8
	OSS-Instruct (New)	**73.2 (69.5)**	78.0 (64.3)	46.4	**18.9**	**58.4**
	Evol-Instruct (New)	71.3 (67.6)	78.6 (65.6)	**47.6**	**19.6**	**58.4**

Potential Overfitting of Standard-SFT. We observe that Standard-SFT occasionally lead to performance degradation compared to the base model. For instance, as detailed in Table III, SemCoder exhibit a decline in performance across multiple benchmarks after undergoing Standard-SFT. We hypothesize that this can be attributable to potential overfitting. The model is trained on PYX whose data distribution is already familiar from its initial fine-tuning phase. To validate this, we fine-tune models on datasets they have not previously been exposed to. Specifically, we fine-tune MagiCoder-DS on PYX and Evol-Instruct, and SemCoder on OSS-Instruct and Evol-Instruct. As shown in Table VII, the models achieve more gains when trained on unseen datasets. For example, SemCoder, when fine-tuned on either OSS-Instruct or Evol-Instruct, outperforms the version trained on its original dataset, PYX.

VI. Threats To Validity

Threats to external validity relate to the generalizability of our approach. While we evaluate our approach on multiple instruction-tuned models, there may be concerns about generalization to other LLMs. However, this threat is mitigated by our diverse selection of models with different series. Furthermore, the cross-dataset experiments (RQ3) and closed-source dataset experiments (RQ4) demonstrate robust generalization capabilities across different settings. In addition, due to computational resource constraints, our experiments primarily focus on 7B parameter LLMs rather than larger LLMs. In future work, we plan to explore a broader range of model series to further validate our approach's generalizability.

Threats to internal validity involve the impact of the quality of incorrect code and choices of hyperparameters. The effectiveness of our approach depends on the quality of incorrect code variants, the weighting factor α and training epochs. To mitigate threats related to code quality, we prompt a strong teacher model to generate plausible incorrect variants. Subsequently, we manually and quantitatively analyze the portion of generated data which is incorrect yet similar to the correct solutions. While a small portion of noise data remains present, we argue these instances may actually enhance model robustness by preventing overfitting to specific *error-sensitive segments* [53], [54]. For hyperparameter-related threats, we conduct an extensive sensitivity analysis as shown in Figure 7. In future work, we intend to investigate the use of stronger teacher models, such as GPT-4-Turbo, to generate similar incorrect code and examine their impact.

VII. Related Work

A. LLMs for Code Generation

LLMs are increasingly being leveraged to automate various tasks in software engineering [55]–[59]. Among these, code generation has emerged as a particularly prominent area of research and application [58], [60]–[62]. As a momentous milestone, Codex [8] boasting a 12-billion-parameter model demonstrates the extraordinary capability to tackle up to 72% of Python programming problems. After that, a new wave of code generation models, such as AlphaCode [60], CodeGen [58], InCoder [61] and StarCoder [62] are proposed and have shown promising results in the code generation task. Building upon these foundations, more code-focused LLMs emerged, such as Magicoder [7], SemCoder [20], WaveCoder [3] and WizardCoder [1]. These specialized LLMs are typically based on general LLMs in solving domain-specific coding tasks through instruction tuning.

B. Fine-tuning on Code LLM

Fine-tuning pre-trained language models has emerged as a dominant paradigm for optimizing performance in code generation. Instruction tuning [63], [64], as a form of supervised fine-tuning, aims to align LLMs with instruction through high-quality instruction corpora. For instance, Magicoder [7] introduce OSS-Instruct, a dataset generated by a teacher LLM drawing inspiration from open-source code snippets, which effectively enhances code generation capabilities. Similarly, SemCoder [20] propose PYX, a dataset created by a teacher LLM simulating human debugging processes. By incorporating data that simulates execution reasoning and captures code execution nuances, LLMs finetuned with PYX understand and articulate the execution process step-by-step, enhancing their reasoning capabilities. To address limitations in preventing untruthful and unexpected outputs, researchers explore reinforcement learning [63]. To address limitations in undesired outputs, researchers have explored reinforcement learning approaches using preference optimization [47], [65], like DPO [46], to refine outputs. However, these often treat tokens uniformly in their loss calculations, making it difficult for models to distinguish semantically correct implementations from syntactically similar but incorrect ones. In this paper, we aim to address this challenge in code LLMs.

We find recent work Focused-DPO [66] enhances code generation capability by concentrating preference optimization on error-prone points through an improved DPO [46] methodology. Our method is complementary to this approach - while Focused-DPO operates during post-training reinforcement learning stages, our approach operates during the supervised fine-tuning stage. However, as Focused-DPO is under peer review and its implementation is not yet publicly available, we are unable to experimentally validate the potential synergies between our approaches in this work.

VIII. Conclusion

In this paper, we introduce Fault-Guided Fine-Tuning (FGit), a novel fine-tuning technique that enhances code generation capabilities in instruction-tuned LLMs by refining their ability to distinguish between correct implementations and subtly incorrect variants. Through extensive experiments across seven LLMs and three widely-used benchmarks, we demonstrate that our method achieves an average relative improvement of 6.9% on pass@1, with certain enhanced 6.7B LLMs even outperforming GPT-3.5-Turbo on selected benchmarks. The technique also exhibits strong generalization capabilities across diverse instruction-tuned LLMs and maintains effectiveness even when applied to LLMs with closed-source instruction datasets.

Acknowledgment

This research/project was supported by the National Natural Science Foundation of China (No.62202420 and No. 62302430), Zhejiang Provincial Natural Science Foundation of China (No.LZ25F020003 and No.LQ24F020017).

Data Availability

Our code is available: https://github.com/ZJU-CTAG/FGIT.

References

[1] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*.

[2] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, "Opencodeinterpreter: Integrating code generation with execution and refinement," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 12 834–12 859.

[3] Z. Yu, X. Zhang, N. Shang, Y. Huang, C. Xu, Y. Zhao, W. Hu, and Q. Yin, "Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 5140–5153.

[4] M. Chen, Z. Liu, H. Tao, Y. Hong, D. Lo, X. Xia, and J. Sun, "B4: Towards optimal assessment of plausible code solutions with plausible tests," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1693–1705.

[5] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.

[6] M. Chen, H. Tian, Z. Liu, X. Ren, and J. Sun, "Jumpcoder: Go beyond autoregressive coder via online modification," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 11 500–11 520.

[7] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," in *Forty-first International Conference on Machine Learning*, 2024.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[9] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[10] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[11] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[12] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language models with self-generated instructions," in *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

[13] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 41–48.

[14] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu *et al.*, "Instruction tuning for large language models: A survey," *arXiv preprint arXiv:2308.10792*, 2023.

[15] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and computing*, vol. 4, pp. 87–112, 1994.

[16] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, 2016, pp. 849–863.

[17] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[18] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[19] T. Y. Zhuo, V. M. Chien, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," in *The Thirteenth International Conference on Learning Representations*.

[20] Y. Ding, J. Peng, M. J. Min, G. Kaiser, J. Yang, and B. Ray, "Semcoder: Training code language models with comprehensive semantics reasoning," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[21] OpenAI, "ChatGPT," 2022. [Online]. Available: https://openai.com/blog/chatgpt/

[22] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[23] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[24] D. R. Hunter, "Mm algorithms for generalized bradley-terry models," *The annals of statistics*, vol. 32, no. 1, pp. 384–406, 2004.

[25] R. Baker and P. Scarf, "Modifying bradley–terry and other ranking models to allow ties," *IMA Journal of Management Mathematics*, vol. 32, no. 4, pp. 451–463, 2021.

[26] T. Yan, "Ranking in the generalized bradley–terry models when the strong connection condition fails," *Communications in Statistics-Theory and Methods*, vol. 45, no. 2, pp. 340–353, 2016.

[27] J. E. Menke and T. R. Martinez, "A bradley–terry artificial neural network model for individual ratings in group competitions," *Neural computing and Applications*, vol. 17, pp. 175–186, 2008.

[28] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 2332–2354, 2024.

[29] Y. Huang, Z. Lin, X. Liu, Y. Gong, S. Lu, F. Lei, Y. Liang, Y. Shen, C. Lin, N. Duan *et al.*, "Competition-level problems are effective llm evaluators," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 13 526–13 544.

[30] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2254–2268, 2024.

[31] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[32] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–30, 2024.

[33] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[34] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[35] Y. Choi, M. A. Asif, Z. Han, J. Willes, and R. G. Krishnan, "Teaching LLMs how to learn with contextual fine-tuning," in *The Thirteenth International Conference on Learning Representations*, 2025.

[36] Á. Ortiz Villa, "Gender-balanced corpus generation for neural machine translation," B.S. thesis, Universitat Politècnica de Catalunya, 2025.

[37] Z. Song, Y. Wang, W. Zhang, K. Liu, C. Lyu, D. Song, Q. Guo, H. Yan, D. Lin, K. Chen *et al.*, "Alchemistcoder: Harmonizing and eliciting code capability by hindsight tuning on multi-source data," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[38] theblackcat102, "The evolved code alpaca dataset." 2023. [Online]. Available: https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1

[39] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: a scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3675–3691, 2023.

[40] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.

[41] OpenAI, "GPT-4o-mini," 2024. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[42] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[43] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[44] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[45] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, "Compilable neural code generation with compiler feedback," in *Findings of the Association for Computational Linguistics*, 2022, pp. 9–19.

[46] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, pp. 53 728–53 741, 2023.

[47] J. Yang, B. Hui, M. Yang, J. Yang, J. Lin, and C. Zhou, "Synthesizing text-to-sql data from weak and strong llms," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 7864–7875.

[48] Y. Miao, B. Gao, S. Quan, J. Lin, D. Zan, J. Liu, J. Yang, T. Liu, and Z. Deng, "Aligning codellms with direct preference optimization," *arXiv preprint arXiv:2410.18585*, 2024.

[49] L. Gee, M. Gritta, G. Lampouras, and I. Iacobacci, "Code-optimise: Self-generated preference data for correctness and efficiency," in *Findings of the Association for Computational Linguistics*, 2025, pp. 79–94.

[50] K. Zhang, G. Li, Y. Dong, J. Xu, J. Zhang, J. Su, Y. Liu, and Z. Jin, "Codedpo: Aligning code models with self generated and verified source code," *arXiv preprint arXiv:2410.05605*, 2024.

[51] V. Gallego, "Refined direct preference optimization with synthetic data for behavioral alignment of llms," in *International Conference on Machine Learning, Optimization, and Data Science*, 2024, pp. 92–105.

[52] S. Deb, K. Jain, R. Van Tonder, C. Le Goues, and A. Groce, "Syntax is all you need: A universal-language approach to mutant generation," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 654–674, 2024.

[53] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[54] Q. Xie, Z. Dai, E. Hovy, T. Luong, and Q. Le, "Unsupervised data augmentation for consistency training," *Advances in neural information processing systems*, vol. 33, pp. 6256–6268, 2020.

[55] A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, M. S. Aktar, and T. Mikkonen, "Towards human-bot collaborative software architecting with chatgpt," in *Proceedings of the 27th international conference on evaluation and assessment in software engineering*, 2023, pp. 279–285.

[56] S. Pan, Y. Wang, Z. Liu, X. Hu, X. Xia, and S. Li, "Automating zero-shot patch porting for hard forks," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 363–375.

[57] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code-change-related tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–36, 2025.

[58] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*.

[59] L. Fan, M. Chen, and Z. Liu, "Sek: Self-explained keywords empower large language models for code generation," in *Findings of the Association for Computational Linguistics*, 2025, pp. 6249–6278.

[60] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[61] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*.

[62] R. Li, L. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: May the source be with you!" *Transactions on machine learning research*, 2023.

[63] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[64] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[65] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy, "Execution-based code generation using deep reinforcement learning," *Transactions on Machine Learning Research*.

[66] K. Zhang, G. Li, J. Li, Y. Dong, and Z. Jin, "Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points," *arXiv preprint arXiv:2502.11475*, 2025.