Accepted: 04 October 2025
Revised: 21 August 2025
Received: 24 October 2024

RESEARCH-ARTICLE

Exploring Data-Efficient Adaptation of Large Language Models for Code Generation

XUE JIANG, Peking University, Beijing, China
YIHONG DONG, Peking University, Beijing, China
ZHIYUAN FAN, Peking University, Beijing, China
ZHI JIN, Peking University, Beijing, China
WENPIN JIAO, Peking University, Beijing, China
GE LI, Peking University, Beijing, China

Exploring Data-Efficient Adaptation of Large Language Models for Code Generation

XUE JIANG, YIHONG DONG, ZHIYUAN FAN, ZHI JIN, WENPIN JIAO, and GE LI*, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China and School of Computer Science, Northwestern Polytechnical University, Xi'an, China

Although Large Language Models (LLMs) have made significant progress in code generation, they still struggle with code generation tasks in specific scenarios. These scenarios usually necessitate the adaptation of LLMs to fulfill specific needs, but the limited training data available in practice leads to poor code generation performance. Therefore, how to effectively adapt LLMs to new scenarios with few training data is a major challenge for current code generation. In this paper, we propose a novel adaptation approach named DEED, which stands for **D**ata-**E**fficient adaptation with **E**rror-**D**riven learning for code generation. DEED leverages the errors made by LLMs as learning opportunities, using error revision to overcome their own shortcomings, thus achieving efficient learning. Specifically, DEED involves identifying error code generated by LLMs, employing Self-Revise for code revision, optimizing the model with revised code, and iteratively adapting the process for continuous improvement. Experimental results show that, compared to other mainstream fine-tuning approaches, DEED achieves superior performance with few training data, showing an average relative improvement of 46.2% in Pass@1 on multiple code generation benchmarks. We also validate the effectiveness of Self-Revise, which generates revised code that optimizes the model more efficiently compared to the code samples from datasets. Moreover, DEED consistently demonstrates strong performance across various LLMs, underscoring its applicability.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Code Generation, Data-Efficient Adaptation, Large Language Models

1 INTRODUCTION

Code generation is an important technology that can improve the efficiency and quality of software development. Given the human requirement expressed in natural language, code generation allows machines to generate executable programs that satisfy this requirement. Code generation has been a research hot topic in the fields of artificial intelligence, software engineering, and natural language processing. Recently, code generation technologies have made significant advancements in both academia and industry [13, 23, 24, 57, 59]. In particular, large language models (LLMs) demonstrate great potential in code generation tasks [11, 28, 36, 54, 75, 77].

*Corresponding author

Authors' Contact Information: Xue Jiang, jiangxue@stu.pku.edu.cn; Yihong Dong, dongyh@stu.pku.edu.cn; Zhiyuan Fan, fanzhiyuan@mail.nwpu.edu.cn; Zhi Jin, zhijin@pku.edu.cn; Wenpin Jiao, jwp@sei.pku.edu.cn; Ge Li, lige@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China and School of Computer Science, Northwestern Polytechnical University, Xi'an, China.

ACM 1557-7392/2025/10-ART

https://doi.org/10.1145/3772721

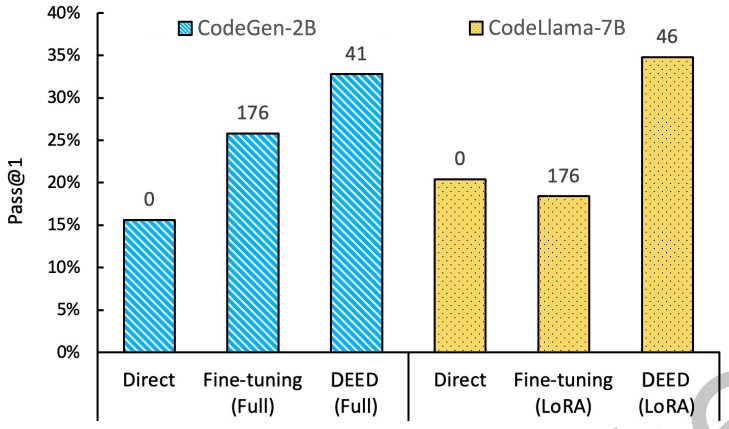

Fig. 1. The performance of direct generation, fine-tuning, and our proposed DEED on MBPP dataset under the circumstance of limited data. The numbers on the bars indicate the training data used by different methods.

However, LLMs still face significant challenges in code generation for adapting to specific scenarios or domains [4, 14].

For specific code generation scenarios, fine-tuning is an essential adaptation method to ensure LLMs fulfill particular needs [9, 16, 45, 60]. However, in these specific scenarios, it is difficult to obtain sufficient training data for fine-tuning LLMs due to common reasons such as industry secrecy and scarcity of resources. For example, the programs in aerospace, medical devices, and transportation, where training samples are inherently scarce due to the scarcity of high-value industry data, high collection costs, and the typical targeting of long-tail demands. Under the circumstance of limited data, mainstream fine-tuning methods might not enable LLMs to achieve the desired code generation performance and may even lead to a degradation in model performance [3, 69, 72], as shown in Figure 1. *In such data-scarce scenarios, only a small amount of data is available, but high code generation performance is still required, which is hard to solve but important in real-world applications.* Consequently, how to effectively adapt LLMs to specific scenarios with limited data available is a major challenge for code generation in practice.

The mainstream fine-tuning methods use a large number of samples gathered under specific scenarios for training [68]. They enable the model to exhaustively learn the features present in these samples and thus adapt to the specific scenarios. However, they have two disadvantages. First, compelling LLMs to relearn the entire code samples of new scenarios is inefficient. Considering that LLMs are pre-trained on large-scale and diverse samples, it's reasonably assumed that they possess a certain level of general knowledge, lacking only particular information for application in specific scenarios. Second, when faced with insufficient data volume or data drift, the model may learn certain undesirable features (such as inaccurate or irrelevant programming knowledge and patterns), thereby affecting its learning efficiency and negatively impacting its final performance.

To overcome the disadvantages of mainstream fine-tuning methods, we take inspiration from the error-driven learning observed in humans. 1) Error-driven learning requires learners to identify their errors through testing. It helps learners to identify what they have mastered and what they still need to learn, allowing them to narrow the scope of learning and avoid wasting efforts on irrelevancies. 2) Through error revision, learners can understand their deficiencies and make targeted improvements, thus enhancing learning efficiency and effectiveness. This

motivates us to explore methods to achieve data-efficient adaptation of LLMs for code generation guided by error-driven learning.

In this paper, we propose DEED, a **D**ata-**E**fficient adaptation based on **E**rror-**D**riven learning for code generation. DEED aims to alleviate the problem of poor code generation performance of fine-tuning LLMs in data-scarce scenarios. Following the error-driven learning, our method proceeds in four steps: ❶ **Error Code Collection.** We identify and collect error codes generated by LLMs, aiming to mine the weaknesses of LLMs. ❷ **Automatic Code Revision.** To obtain revisions of error codes in a low-cost way, we design Self-Revise to realize automatic revision leveraging information in the original dataset and code execution feedback. ❸ **Model Optimization.** We optimize the LLMs using the revised code, making them focus on learning the revision of these critical errors, thereby improving the learning efficiency of LLMs. ❹ **Iterative Adaptation.** We adopt an iterative strategy, which involves repeating the preceding three steps, to continuously optimize and improve the performance of LLMs.

We extensively evaluate our proposed DEED on five public code generation benchmarks using the test pass rate metric. Our results show that under the circumstance of limited data, DEED achieves significantly better performance across various LLMs compared to the mainstream adaptation approaches. Figure 1 illustrates part of this result. Specifically: 1) On five benchmarks, DEED achieves more than 27.1% relative improvement in the Pass@1 metric compared to the best-performing adaptation approaches. 2) The effectiveness of Self-Revise is validated. It produces revised code that better supports the optimization of the code generation model than the code samples provided in training dataset. 3) Our iterative adaptation process facilitates continuous enhancements in performance and maintains training stability. 4) DEED proves to be effective across various LLMs, demonstrating its broad applicability. These results highlight the potential of DEED in addressing the challenges of code generation in data-scarce scenarios.

To summarize, the main contributions of this paper are:

- We propose that error-driven learning for LLMs adaptation is effective, i.e., utilizing revisions of LLMs' erroneous outputs for training has higher learning efficiency than samples from the dataset.
- Based on the principle of error-driven learning, we propose a data-efficient adaptation method of LLMs for code generation, named DEED, which can effectively adapt the model to specific scenarios with few training data.
- DEED outperforms the mainstream fine-tuning and prompting methods on five code generation datasets across various LLMs.

2 PRELIMINARY STUDY

Aligning LLMs with specific scenarios and addressing their unique challenges by learning samples in the dataset is difficult when training data are limited. We conduct a preliminary study to explore the effectiveness of using error-driven learning in the LLMs adaptation process of code generation.

We consider the potential significant discrepancy between the model-generated output and the sample in the dataset. By learning the revisions of the model's erroneous outputs, we can find more effective navigation in the optimization process. This might provide a shorter, smoother path to a good local minimum compared to learning from samples in the dataset, rather than attempting to direct it toward a distant area that may not align well with its existing knowledge or biases. We conduct a statistical analysis of the discrepancies in the model's latent representations. Specifically, on MBPP [5] dataset, we obtain erroneous outputs of CodeGen-2B [54], revisions of the outputs, and samples in MBPP. We concatenate the requirements with their code, input them into CodeGen-2B, and extract the hidden representations from the model's final layer. Then, we compute the Euclidean distances within the model's representational space to quantify the disparities between these three elements. The experimental results show that the average distance between the model's erroneous outputs and

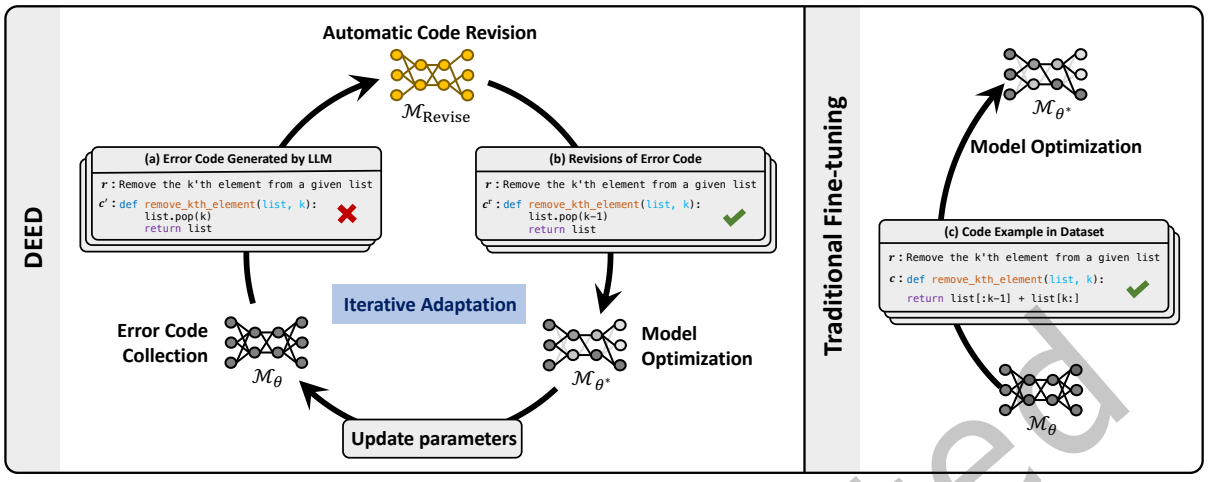

Fig. 2. An overview of the proposed DEED and its differences from traditional fine-tuning methods.

the dataset's samples is 12.35, whereas the average distance between the erroneous outputs and their revisions is significantly lower, at 6.39. **These experimental results suggest that within the model's representation space, revised codes are closer and similar to the erroneous output codes than the original code samples. This evidence supports our hypothesis of why the error-driven learning method is more efficient from the perspective of model optimization.**

Therefore, our work is determined to explore the use of error-driven learning to achieve a data-efficient adaptation method, aimed at enhancing the performance of LLMs in specific code generation scenarios.

3 METHODOLOGY

Given a code generation scenario/domain with a limited-sample training dataset $\mathcal{D}_{train} = \{(r, c)\}$, where each data pair (r, c) consists of an input requirement r and an associated example of desired output code c. For a pre-trained LLM $\mathcal{M}_\theta$ with parameter θ, we aim to adapt $\mathcal{M}_\theta$ to the specific scenario of $\mathcal{D}_{train}$. Inspired by error-driven learning, we propose DEED to achieve data-efficient adaptation of LLMs. DEED consists of the following four steps: Error Code Collection (§3.1), Automatic Code Revision (§3.2), Model Optimization (§3.3), and Iterative Adaptation (§3.4). LLMs are well adapted after applying DEED, and their subsequent use (inference) remains unchanged compared to direct generation with LLMs, while improving the accuracy of one-time predictions without introducing any additional overhead. The overview of DEED and its differences from traditional fine-tuning are shown in Figure 2.

3.1 Error Code Collection

In this step, we systematically identify and collect erroneous output of LLMs using testing as criteria. We employ rejection sampling [8] to draw error code samples from the distribution produced by $\mathcal{M}_\theta$. For each requirement $r \in \mathcal{D}_{train}$, we sample

$$c' \sim \mathcal{M}_\theta(r) \mid \neg f, \tag{1}$$

where we sample multiple times and employ the criterion function f to determine the retention of the code sample. Specifically, the error code sample c' is retained when $f(r, c') = 0$, where $f(r, c') = 0$ if the rejection condition is satisfied, otherwise 1.

We define f as a test evaluation function since testing is the criterion for judging the correctness of code in practice:

$$\textsc{TestEval}(r, c') \coloneqq \begin{cases} 0, & \text{if } c' \text{ fails } S_r, \\ 1, & \text{otherwise,} \end{cases} \tag{2}$$

where S_r is a suit of test cases under the requirement r and is equipped by code generation datasets. When collecting error codes for test failures, we can keep the failed test cases and error messages simultaneously for further error diagnosis.

To gain insights into the propensity of $\mathcal{M}_\theta$ to make certain errors, it is advisable to select error code sample c' for which the model demonstrates relatively high confidence. Therefore, among multiple error codes collected for the same r, we select the one with the highest generation probability, which is determined by the average log-probability of each token in the generated code.

$$\log P(c'|r) = \frac{1}{|c'|} \sum_i \log P(s_i|s_{<i}, r), \tag{3}$$

where s_i is the i-th output token, $s_{<i}$ denotes the set of previous tokens, and $|c'|$ indicates the length of the code.

3.2 Automatic Code Revision

In this step, we perform automatic revision for error codes using our Self-Revise method. Based on the LLM $\mathcal{M}_\theta$ itself, Self-Revise revises the error code by providing the information in the original dataset and pointing out the error with code execution feedback. Our objective is to derive a revised code that fixes the critical bug in the error code. As illustrated by examples (a), (b), and (c) in Figure 2, although there is already a correct code (c) in the dataset, it may differ significantly from the error code (a), leading to the critical bug being unclear. The pipeline of automatic code revision is shown in Figure 3. Specifically, we leverage the following parts as the input of Self-Revise:

- **Requirement (r)**: Clarify the requirement that need to be addressed;
- **Correct Solution (g)**: Provide a type of correct solution as a reference to reduce the difficulty of revision. The correct solution used here is the code sample c in the training dataset;
- **Error Code (c′)**: Give the error code that needs to be revised. The error code is generated by $\mathcal{M}_\theta$ under r;
- **Error Messages (m) and Failed Test Cases (t)**: Point out the error messages received during execution and the specific test cases where the error code fails, allowing for more focused troubleshooting and revision.

These parts are combined as the input of Self-Revise according to the template:

$$T = \text{Template}(r, g, c', m, t) \tag{4}$$

where Template is shown in Figure 3.

Following previous work [21, 73], we use two settings for Self-Revise, i.e., fine-tuning (FT) and few-shot prompting (FSP), to get $\mathcal{M}_{\text{Revise}}$ for revising error codes.

Self-Revise (FT) entails the process of fine-tuning $\mathcal{M}_\theta$ with a small number of data for the purpose of automatic code revision. The training objective is to minimize $L(\theta)$:

$$L(\theta) = \sum_i l_{ce}(\mathcal{M}_\theta(T_i), c_i^*) \tag{5}$$

where l_{ce} represents the standard cross-entropy loss, and we update the parameters initialized with $\mathcal{M}_\theta$ to obtain $\mathcal{M}_{\text{Revise}}$ in Self-Revise (FT).

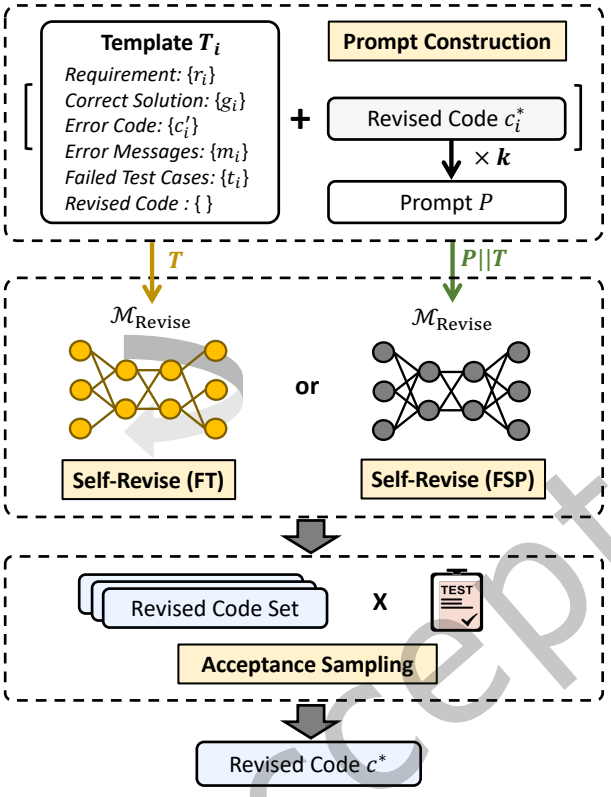

Fig. 3. Illustration of automatic code revision.

Self-Revise (FSP) adopts the few-shot prompting technique and leverages k examples of T_i and c_i^* to construct the prompt P for aligning $\mathcal{M}_\theta$ to automatic code revision. In Self-Revise (FSP), $\mathcal{M}_{\text{Revise}}(\cdot)$ is defined as $\mathcal{M}_\theta(P \,||\, \cdot)$, where $||$ denotes the concatenation operation.

In contrast to the previous error code collection step, for each error code c', we construct T and use acceptance sampling to obtain the revised code c^*:

$$c^* \sim \mathcal{M}_{\text{Revise}}(T) \mid f. \tag{6}$$

where c^* is retained if $\textsc{TestEval}(r, c^*) = 1$ in Eq. (2), i.e., the revised code c^* passes its test cases. We sample multiple times, and it is sufficient if $\mathcal{M}_{\text{Revise}}$ could correctly revise the error code once. To prevent $\mathcal{M}_{\text{Revise}}$ from simply replicating the provided correct solution g, we exclude the output identical to g. Subsequently, for each requirement r, select the version that is most similar to the error code among the remaining code revisions.

3.3 Model Optimization

In this step, we employ pairs of the requirement r and its revised code c^* to further fine-tune the model $\mathcal{M}_\theta$. This process leads to the enhanced version of $\mathcal{M}_\theta$, referred to as $\mathcal{M}_{\theta^*}$, in the specific scenario of dataset $\mathcal{D}_{train}$.

For fine-tuning [17], we update all parameter θ of LLMs as

$$\theta^* = \arg\min_\theta \sum_{(r,c^*)} l_{ce}(\mathcal{M}_\theta(r), c^*), \tag{7}$$

When the computational resources are insufficient, we employ Low-Rank Adaptation (LoRA) [31] to fine-tune LLMs. For a weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA represents its update with a low-rank decomposition:

$$W + \Delta W = W + \Delta\alpha W_{\text{down}} W_{\text{up}}, \tag{8}$$

where α is a tunable scalar hyperparameter, $W_{\text{down}} \in \mathbb{R}^{d\times r}$, $W_{\text{up}} \in \mathbb{R}^{r\times k}$, and $r << min(r, k)$. In LoRA, we update parameter θ^* as

$$\theta^* = \theta + \Delta\theta, \tag{9}$$

$$\Delta\theta = \arg\min_{\Delta\theta} \sum_{(r,c^*)} l_{ce}(\mathcal{M}_{\theta+\Delta\theta}(r), c^*). \tag{10}$$

3.4 Iterative Adaptation

The preceding three steps can go through multiple iterations until a certain number of rounds is reached or the revised code no longer increases.

For l-th iteration that $l > 1$, we initialize its initial model $\mathcal{M}_{\theta_l}$ as the enhanced model of the previous iteration $\mathcal{M}_{\theta^*_{l-1}}$. Based on $\mathcal{M}_{\theta_l}$, we repeat the process in steps of error code collection and automatic code revision to sample error codes $\{c'\}_l$ and revised codes $\{c^*\}_l$, respectively. Inspired by experience replay [51] in reinforcement learning, we use the union of collected data in each previous iteration $\{(r, c^*)\}_{1:l}$ to stabilize the learning process and improve data utilization efficiency, i.e.,

$$\{(r, c^*)\}_1 \cup \cdots \cup \{(r, c^*)\}_i \cdots \cup \{(r, c^*)\}_l, \tag{11}$$

to update parameters in the model optimization step, thereby yielding the enhanced model of the l-th iteration $\mathcal{M}_{\theta^*_l}$. At each iteration, the model is trained to convergence.

This iteration is a step-by-step optimization designed to continuously improve the adaptability of models to the specific scenario. The complete process of DEED is summarized in Algorithm 1.

Algorithm 1 Pseudocode of DEED.

Require: Dataset $\mathcal{D}_{train} = \{(r, c)\}$, initial LLM $\mathcal{M}_\theta$.
Ensure: LLM $\mathcal{M}_{\theta^*}$.

Initial iteration index $l = 0$ and $\mathcal{M}_{\theta_{l+1}} = \mathcal{M}_\theta$.
 # Iterative Adaptation
repeat
  Update $l = l + 1$.
  # Error Code Collection
  Perform rejection sampling to collect error codes $\{c'\}_l$ based on $\mathcal{M}_{\theta_l}$ via Eq. (1) and (2).
  # Automatic Code Revision
  Perform acceptance sampling to collect revised codes $\{c^*\}_l$ based on $\mathcal{M}_{\theta_l}$ and Self-Revise via Eq. (2), (4), and (6).
  Calculate the union of $\{(r, c^*)\}_{1:l}$ via Eq. (11).
  # Model Optimization
  Fine-tune $\mathcal{M}_{\theta_l}$ to yield $\mathcal{M}_{\theta^*_l}$ via Eq. (7) if the computational resources are sufficient, otherwise via Eq. (8), (9), and (10).
  Update $\mathcal{M}_{\theta_{l+1}} = \mathcal{M}_{\theta^*_l}$.
until End condition is satisfied
return $\mathcal{M}_{\theta^*_l}$

4 EVALUATION SETUP

We present extensive experiments that span five representative code generation datasets, two fine-tuning settings, and four different LLMs of varying sizes or series. We aim to investigate six research questions:

- **RQ1: How does DEED perform compared to the mainstream adaptation approaches?** This question aims to investigate the superiority of our method compared to existing approaches in data-scarce scenarios.
- **RQ2: How does DEED perform when applied to various LLMs?** This question explores the usefulness of DEED across different LLMs, assessing its applicability and performance consistency in varying LLMs.
- **RQ3: What kind of training sample has the better training effect?** This question helps us understand the promotion of training efficiency by training with the revision of LLMs' erroneous outputs.
- **RQ4: How does the number of iterations affect the effectiveness of DEED?** This question explores the changes in LLMs' performance during the process of iterative optimization, to demonstrate the necessity of iterative adaptation in our method.
- **RQ5: What is the impact of implementing the automatic code revision component of DEED in conjunction with alternative LLMs?** We choose to use LLMs themselves for revision, termed Self-Revise. This question explores the effects of using other LLMs for revision.
- **RQ6: How does each input component of Self-Revise contribute to the effectiveness?** Since Self-Revise utilizes components such as correct solutions, failed test cases, and error messages, this question aims to analyze the impact of these components on its performance.

4.1 Datasets

We use five public code generation datasets to simulate the specific scenario with limited data and apply DEED to each dataset to evaluate its effectiveness. Specifically,

- **HumanEval** [13] is a widely-used code generation benchmark, containing 164 handwritten programming problems, proposed by OpenAI. Each programming problem includes a function signature, a natural language description, use cases, a correct solution in Python, and several test cases.
- **MBPP** [5] contains crowd-sourced Python programming problems. We use the version in the work [10], which consists of 276 problems and some generated error codes alongside their human-revised counterparts, thus facilitating subsequent experiments.
- **HumanEval-ET** and **MBPP-ET** [20] are extended versions of MBPP and HumanEval, respectively, where each includes over 100 additional test cases per problem. This updated version enhances the soundness of code evaluation compared to the original benchmarks.
- **DataScience** [37] comprises 291 data science problems utilizing Pandas libraries. This dataset can evaluate the ability of LLMs to utilize specific data-analysis libraries for code generation.

We sample $\min(200, 40\% * \mathcal{D})$ problems from the datasets as $\mathcal{D}_{train}$, while the remaining problems serve as $\mathcal{D}_{test}$.

4.2 Implementation Details

We use a single A6000 GPU to conduct all experiments. We select CodeGen-2B [54] as our base model by default, which is a well-known open-source LLM for code and is suitable for full fine-tuning within our computational resource constraints. $\mathcal{M}_\theta$ is initialized to our base model, and $\mathcal{M}_{\text{Revise}}$ is derived from $\mathcal{M}_\theta$ though Self-Revise (§3.2). In fine-tuning setting, $\mathcal{M}_{\text{Revise}}$ only needs to be trained at the beginning and then remains unchanged for subsequent operations.

For full parameter fine-tuning, i.e., Fine-tuning (Full) [17], we use the AdamW optimizer [48], with hyperparameters $\beta_1 = 0.9$ and $\beta_2 = 0.9$, accompanied by a linear learning rate schedule. The initial learning rate is set to 5e-6, with a batch size of 1 and gradient accumulation of 32 steps for training across 10 epochs. For parameter-efficient

fine-tuning, i.e., Fine-tuning (LoRA) [31], the learning rate is set to 2e-4. Additionally, the rank r is adjusted to 128, and the scaling factor α is set at 8. All other hyperparameters remain aligned with Fine-tuning (Full). For few-shot prompting [6], we set the number of examples in prompt to 4. All baselines in the experiments use consistent settings.

In the error code collection step (§3.1) and the automatic code revision step (§3.2), we use temperature [2, 29] sampling to generate multiple samples: 5 samples in the former and 30 in the latter, with the temperature set to 0.8. To obtain the final revised code in the automatic code revision step, we choose one of the revised codes exhibiting the minimum Levenshtein distance [39] to the error code. The number of iterative adaptations is set to 2. The maximum generation length is uniformly limited to 1024 tokens.

4.3 Metrics

Following the practice of real software development which utilizes testing for evaluation [1, 58], we employ the Pass@k [42] metric to measure the functional correctness of the generated code by executing test cases. We use the unbiased version [13] of Pass@k, where $n >= k$ samples are generated for each problem, count the number of correct samples $c <= n$ which pass test cases and calculate the following estimator,

$$\text{Pass@k} = \mathop{\mathbb{E}}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \tag{12}$$

For automatic code revision, we add the pass@any metric which refers to the percentage of tasks for which the model generates at least one correct code that passes all test cases.

In the final evaluation of this paper, we set the temperature to 0.8 and generate $n = 50$ samples, which are then used to calculate unbiased Pass@k [13] via Eq. (12) in all experiments. All evaluation results are averaged over five test runs.

5 RESULTS

5.1 RQ1: Comparison of DEED and the mainstream adaptation approaches

Baselines. In this section, we evaluate the effectiveness of DEED by comparing it against four mainstream approaches used as baselines:

- **Direct Generation**: We evaluate the LLM directly to demonstrate the performance of the original LLM.
- **Fine-tuning (Full)**: We employ full-parameter fine-tuning for the LLM on $\mathcal{D}_{train}$.
- **Fine-tuning (LoRA)**: We fine-tune the LLM on $\mathcal{D}_{train}$ using LoRA [31], which greatly reduces the number of trainable parameters.
- **Few-shot Prompting**: We use 4-shot prompt [6] to align LLMs with the input-output format of $\mathcal{D}_{train}$, where 4 examples in prompt are randomly selected from $\mathcal{D}_{train}$.
- **Self-refine** [50] and **Self-debug** [15] iteratively refine the generated code through prompting techniques. The number of iterations is set to 2.

Considering the baselines involve full-parameter fine-tuning, CodeGen-2B is uniformly selected as the base model in this experiment. For DEED, we use 30% data of $\mathcal{D}_{train}$ for Self-Revise (FT)[1], while the remaining 70% data of $\mathcal{D}_{train}$ is employed for model optimizing, where we use full-parameter fine-tuning.

[1]In addition to MBPP dataset, for the other two datasets (i.e., HumanEval and DataScience), we generate one error code per sample in a subset comprising 30% of the training set, using CodeGen-2B. Subsequently, authors collaboratively apply the minimal necessary revisions to correct these error codes.

Table 1. Pass@k (%) of DEED and baselines on HumanEval, MBPP, and DataScience datasets. The values in brackets represent results on the extended version of the datasets, and the teal number after ↑ denotes the relative improvement of DEED over the second-highest score.

Datasets	Method	Pass@1	Pass@5	Pass@10
HumanEval	Direct Generation	24.8%	44.7%	51.8%
	Fine-tuning (Full)	29.8%	47.9%	56.4%
	Fine-tuning (LoRA)	27.4%	46.9%	53.9%
	Few-shot Prompting	25.2%	45.8%	53.1%
	Self-Refine	25.3%	45.2%	51.9%
	Self-Debug	26.4%	46.4%	54.2%
	DEED	**38.6%** (↑ 29.5%)	**54.7%**	**62.2%**
HumanEval-ET	Direct Generation	17.0%	30.8%	36.3%
	Fine-tuning (Full)	21.5%	34.2%	41.5%
	Fine-tuning (LoRA)	20.3%	33.7%	39.9%
	Few-shot Prompting	19.1%	33.5%	39.1%
	Self-Refine	17.5%	32.6%	38.2%
	Self-Debug	19.2%	33.6%	41.0%
	DEED	**28.6%** (↑ 33.0%)	**39.3%**	**46.5%**
MBPP	Direct Generation	15.6%	31.4%	40.2%
	Fine-tuning (Full)	25.8%	45.2%	57.6%
	Fine-tuning (LoRA)	19.8%	39.8%	55.2%
	Few-shot Prompting	24.4%	38.0%	49.4%
	Self-Refine	25.6%	38.8%	50.2%
	Self-Debug	20.2%	34.5%	40.6%
	DEED	**32.8%** (↑ 27.1%)	**46.8%**	**64.0%**
MBPP-ET	Direct Generation	10.6%	21.7%	28.2%
	Fine-tuning (Full)	18.1%	33.1%	43.7%
	Fine-tuning (LoRA)	15.1%	32.5%	42.4%
	Few-shot Prompting	17.5%	27.3%	36.9%
	Self-Refine	17.8%	27.6%	37.2%
	Self-Debug	14.1%	25.7%	29.6%
	DEED	**24.9%** (↑ 37.6%)	**36.3%**	**49.3%**
DataScience	Direct Generation	0.8%	3.1%	5.6%
	Fine-tuning (Full)	2.6%	6.5%	9.6%
	Fine-tuning (LoRA)	2.2%	6.0%	8.9%
	Few-shot Prompting	1.9%	4.5%	5.7%
	Self-Refine	2.1%	4.7%	5.8%
	Self-Debug	1.2%	2.8%	4.1%
	DEED	**5.3%** (↑ 103.8%)	**9.5%**	**12.3%**

Results. We conducted experiments on five public datasets, i.e., MBPP, HumanEval, and DataScience. The experimental results are summarized in Table 1. This comparison yielded four insightful observations: **1) Significant superiority of DEED**: Our proposed DEED performs significantly better than the other six baselines on

the five datasets. Notably, DEED exhibits significant relative improvements of 29.5%, 33.0%, 27.1%, 37.6%, and 103.8%, respectively, when compared to the best-performing baseline Fine-tuning (Full). Self-refine and Self-debug underperform on small LLMs like codegen-2B. Self-debug excels over Self-refine only on the HumanEval dataset, where public test cases and results are available. On other datasets, Self-debug relies on the LLM's generated code explanations and feedback. Moreover, we find that DEED not only surpasses Self-refine and Self-debug in terms of performance but also in cost. These methods have a significant disadvantage in cost as they require iterative refinements for each sample, leading to increased time and token consumption. Their cost is directly proportional to the number of sampling and the number of iterations, while DEED is free of these two factors during inference. **2) Worst performance of Direct Generation**: The performance of Direct Generation is significantly lower than Fine-tuning (Full), Fine-tuning (LoRA), and Prompt baselines. This result suggests that directly applying LLMs for evaluation may be less suitable for specific scenarios, resulting in performance differences. **3) Fine-tuning (LoRA) is less effective than Fine-tuning (Full)**: Although Fine-tuning (LoRA) offers the advantage of reduced computational resource requirements for fine-tuning LLMs, it trades off the performance. **4) Less improvement of Few-shot Prompting**: Few-shot prompting is the most commonly used prompting technique, but its main limitation lies in its difficulty in imparting new knowledge or developing new capabilities in the model. It primarily assists the model in adjusting its outputs to better align with expected results, therefore its adaptability is limited.

Summary of RQ1: In code generation scenarios with limited training data, DEED exhibits improvements compared to the mainstream adaptation approaches, achieving relative improvements between 27.2% and 103.9%.

5.2 RQ2: DEED with Different LLMs

Baselines. We use the following types of well-known LLMs to perform DEED, including

- **CodeGen-2B and CodeGen-6B** [54] are code LLMs trained on natural language and programming data for conversation-based program synthesis.
- **Llama-7B** [62] is a general-purpose foundational LLM that has been trained on diverse data to handle a wide range of tasks, which is developed by Meta.
- **CodeLlama-7B** [57] is an open foundational LLM for code generation tasks, derived from continuous training and fine-tuning based on Llama architecture [62].

Among them, CodeGen-2B uses full fine-tuning, and the remaining LLMs use parameter-efficient fine-tuning with LoRA. Each LLM has the same baselines as RQ1 (§5.1), i.e., **Direct Generation**, **Fine-tuning**, and **Few-shot Prompting**.

Results. The results of applying DEED to different LLMs are shown in Table 2. The results demonstrate that DEED consistently achieves significant improvements across all these LLMs, outperforming the Direct Generation, Fine-tuning, and Few-shot Prompting baselines. Moreover, we also find that the performances of Code LLMs are better than pure LLMs, and there is a clear trend indicating that as the number of parameters in the LLMs increases, the efficacy of the Direct Generation and Few-shot Prompting approaches increases steadily. In contrast, the effectiveness of DEED on these LLMs is also continuously improving.

Summary of RQ2: DEED consistently enhances performance across all LLMs, outperforming Direct Generation, Fine-tuning, and Few-shot Prompting baselines, showcasing the applicability of DEED.

Table 2. Pass@k (%) of DEED and baselines with different LLMs, and the teal number after ↑ denotes the relative improvement of DEED over Fine-tuning.

Models	Method	Pass@1	Pass@5	Pass@10
CodeGen-2B	Direct Generation	15.6%	31.4%	40.2%
	Fine-tuning (Full)	25.8%	45.2%	57.6%
	Few-shot Prompting	24.4%	38.0%	49.4%
	DEED (Full)	**32.8%** (↑ 27.1%)	**46.8%**	**64.0%**
CodeGen-6B	Direct Generation	19.6%	40.2%	60.8%
	Fine-tuning (LoRA)	26.6%	46.8%	63.0%
	Few-shot Prompting	26.2%	45.2%	60.2%
	DEED (LoRA)	**33.4%** (↑ 25.6%)	**47.4%**	**67.6%**
Llama-7B	Direct Generation	13.4%	29.8%	37.4%
	Fine-tuning (LoRA)	15.2%	27.4%	34.0%
	Few-shot Prompting	16.6%	26.2%	33.8%
	DEED (LoRA)	**22.0%** (↑ 32.5%)	**30.4%**	**40.8%**
CodeLlama-7B	Direct Generation	20.4%	43.8%	52.8%
	Fine-tuning (LoRA)	19.9%	42.4%	53.2%
	Few-shot Prompting	27.8%	46.6%	64.8%
	DEED (LoRA)	**34.8%** (↑ 25.2%)	**49.2%**	**65.8%**

5.3 RQ3: The Effect of Training Sample Variants

Baselines. We investigate the influence of different training data on the final adapted model $\mathcal{M}_{\theta^*}$ to validate the effectiveness of using revisions of model's erroneous output for training. The different variants of training data include:

- **W/o Training**: Direct generation without without any training data.
- **Raw $\mathcal{D}_{train}$**: All samples in $\mathcal{D}_{train}$, which are the raw requirement and code sample pairs in the dataset.
- **$\mathcal{D}_{train} \cap$ DEED**: The samples of the same problem as DEED in $\mathcal{D}_{train}$
- **DEED $\cup$ $\mathcal{D}_{train}$**: Include not only samples of problems obtained through Self-Revise, but also samples of other problems in $\mathcal{D}_{train}$.
- **Human-revised $\mathcal{D}_{train}$**: Samples obtained from human revision, which are the pairs of requirement and revised code of LLM's erroneous output.
- **DEED**: Samples obtained through Self-Revise, which are the pairs of requirement and revised code of LLM's erroneous output.

Results. As shown in Table 3, we discover that: **1) DEED exceeds Raw $\mathcal{D}_{train}$, despite Raw $\mathcal{D}_{train}$ having more training data.** This proves that training using revisions produced by Self-Revise is more efficient compared to using samples in the dataset. **2) The effect of $\mathcal{D}_{train} \cap$ DEED is comparatively weaker**, which reveals that DEED is not simply improved by selecting better problems. **3) DEED $\cup$ $\mathcal{D}_{train}$ is not as effective as DEED**, which shows that some samples in $\mathcal{D}_{train}$ have negative effects for training. **4) The performance of DEED surpasses that of the Human-revised $\mathcal{D}_{train}$.** This finding may be attributed to a disconnect between the revision made by humans and the model's learning expectations. While human revisions are applied to all code samples in $\mathcal{D}_{train}$, some samples may inherently be challenging for the current model. As such, forced learning

Table 3. Comparison of the effect of different training data variants.

Variants	Pass@1	Pass@5	Pass@10
W/o Training	15.6%	31.4%	40.2%
Raw $\mathcal{D}_{train}$	25.8%	45.2%	57.6%
$\mathcal{D}_{train} \cap$ DEED	22.4%	33.8%	42.8%
DEED $\cup$ $\mathcal{D}_{train}$	29.2%	44.2%	58.0%
Human-revised $\mathcal{D}_{train}$	28.0%	46.2%	59.8%
DEED	32.8%	46.8%	64.0%

from these samples may have counterproductive effects, highlighting a potential limitation in human-revised $\mathcal{D}train$.

Summary of RQ3: The revisions of error code through Self-Revise surpass other training sample variants, including both samples in the dataset and the samples revised by humans, in terms of training efficiency and effectiveness.

5.4 RQ4: The Effect of Iterations

Baselines. We study the effect of iterations on DEED. We analyze the progression of DEED's effectiveness across different iterations, starting from 0 iterations (i.e., generated directly with LLMs) and extending to one, and up to four iterations.

Table 4. Performance of DEED with the different number of iterations, where NRC indicates the Number of Revised Codes added in each iteration, and Loss is calculated at the end of training in each iteration.

Iterations	Pass@1	Pass@5	Pass@10	NRC	Loss
0	15.6%	31.4%	40.2%	-	-
1	31.6%	46.3%	60.6%	31 (+31)	0.0891
2	32.8%	46.8%	64.0%	41 (+10)	0.0375
3	33.0%	46.7%	62.6%	43 (+2)	0.0094
4	33.2%	47.1%	64.0%	44 (+1)	0.0056

Results. We conduct this experiment on MBPP dataset, and its results are displayed in Table 4. From the results, we can observe a trend: as the number of iteration rounds increases, the performance of DEED first shows an increasing trend and then gradually stabilizes. The amount of revised code in each iteration is also increasing, indicating that errors are continuously discovered, corrected, and learned. Additionally, the results show a consistent reduction in loss as the number of iterations increases, which affirms the training stability of DEED. Considering that Pass@10 has oscillations from the 2nd iteration to the 4th iteration, we choose to end after the second iteration as the final performance of DEED.

Summary of RQ4: As iteration rounds increase, the performance of DEED initially improves and then stabilizes in Pass@1, and over 98% performance can be achieved in two iteration.

5.5 RQ5: Automatic Code Revision Based on Other Models

Baselines. We evaluate the performance of automatic code revision and the impact on the final model $\mathcal{M}_{\theta^*}$, which is obtained through DEED, when using alternative LLMs to substitute the base model as $\mathcal{M}_{\text{Revise}}$. The base model and alternative LLMs are as follows:

- **Base Model**: We use CodeGen-2B [54] as the base model for automatic code revision, i.e., Self-Revise.
- **CodeGen-6B** [54]: A variant in the same series as our base model but with a larger parameter count.
- **Llama-7B** [62]: A LLM with different training data and architectures of the base model.
- **CodeLlama-7B** [57]: A code LLM with different training data and architectures of the base model.
- **ChatGPT** [55]: A powerful AI chatbot developed based on LLMs. Since ChatGPT is closed-source and cannot be fine-tuned, we employ ChatGPT for Self-Revise (FSP).
- **GPT-3.5-turbo** [56] is an instruction-tuned LLM developed by OpenAI, optimized for dialogue and efficient inference.

In this experiment, we obtain $\mathcal{M}_{\text{Revise}}$ in both fine-tuning and few-shot prompting settings for comparison, and $\mathcal{M}_{\theta^*}$ is consistently fixed as the base model.

Table 5. Comparison of automatic code revisions based on different LLMs and settings as well as their impact on the final model, where $\mathcal{M}_{\text{Revise}}$ represents the Self-Revise model that used during the training and $\mathcal{M}_{\theta^*}$ represents the final model that used during the inference. $\mathcal{M}_{\text{Revise}}$ is reported the raw results on the 70% $*$ $\mathcal{D}_{train}$ part and $\mathcal{M}_{\theta^*}$ is fine-tuned with filtered results as described in §3.2.

Method	$\mathcal{M}_{\text{Revise}}$			$\mathcal{M}_{\theta^*}$	
	Pass@1	Pass@10	Pass@any	Pass@1	Pass@10
Few-shot Prompting					
CodeGen-6B	19.4%	60.1%	70.8%	26.8%	59.0%
Llama-7B	23.5%	67.7%	81.9%	20.8%	54.2%
CodeLlama-7B	20.2%	64.9%	75.0%	25.2%	59.6%
ChatGPT	61.4%	87.3%	92.1%	27.0%	62.4%
GPT-3.5-turbo	59.4%	69.5%	72.7%	29.0%	51.0%
Base Model (Self-Revise (FSP))	18.9%	57.1%	69.4%	26.2%	58.2%
Fine-tuning					
CodeGen-6B	5.0%	20.3%	26.6%	29.4%	64.2%
Llama-7B	2.7%	8.5%	12.6%	23.2%	58.4%
CodeLlama-7B	5.1%	21.0%	34.6%	24.0%	60.2%
Base Model (Self-Revise (FT))	3.9%	18.9%	24.6%	**32.8%**	**64.0%**

Results. Table 5 illustrates the experimental results of automatic code revision based on different models, and we can observe that: **1) Self-Revise (FT) employing the same model as the base model yields the best performance of $\mathcal{M}_{\theta^*}$.** For baselines using other LLMs in fine-tuning, CodeLlama exhibits superior performance in terms of Pass@k in $\mathcal{M}_{\text{Revise}}$, but its final effectiveness is somewhat compromised. This limitation is attributed to the divergence in training data and architectural frameworks between CodeLlama and the base model, leading to inconsistencies in the revised code with the base model's expectations. In contrast, CodeGen-6B, which is the same series of the base model with a large parameter, demonstrates slightly lower Pass@k in $\mathcal{M}_{\text{Revise}}$ than

CodeLlama but still achieves commendable results for $\mathcal{M}_{\theta^*}$. **2) Although the Pass@k of Self-Revise (FSP) is higher than Self-Revise (FT) in $\mathcal{M}_{\text{Revise}}$, it does not perform as well on the ultimate $\mathcal{M}_{\theta^*}$.** We find this discrepancy may be due to the Self-Revise (FSP)'s tendency to learn superficial forms, i.e., it often resorts to directly copying code from the correct solution provided in the prompt, even when explicitly instructed not to in the prompt. Using ChatGPT as $\mathcal{M}_{\text{Revise}}$ results in substantially higher Pass@k compared to using the base model, does not significantly enhance the final model $\mathcal{M}_{\theta^*}$.

To understand why the ChatGPT-based revision underperforms the self-revision model, despite providing more training data, we identify two reasons: 1) We manually check the revised data and find that although the quantity of revisions increased, ChatGPT sometimes disregards the instruction to make "minimal necessary revisions." Instead, it tends to reference the ground-truth correct code, resulting in substantial changes rather than minimal fixes. We conduct a further experiment with OpenAI's GPT-3.5-turbo, and the result is shown in Table 5. While this issue is less severe and its final performance is better than that of the revision using ChatGPT, it still did not surpass the Self-Revise (FT). 2) We believe a second reason is the significant capability gap between ChatGPT and our base model (CodeGen-2B). ChatGPT is a much more powerful model, and some of the revision knowledge it provides may not be effectively understood and absorbed by CodeGen-2B. Instead, the base model may simply memorize the patterns, which does not improve its generalization ability. We conclude that this phenomenon highlights a crucial point: the suitability and quality of data are more important than its quantity.

Qualitative Examples. We use the case study to qualitatively assess the effectiveness of automatic code revision (§3.2), i.e., Self-Revise (FSP) and Self-Revise (FT) employed by DEED, examples of which are presented in Figure 4. Upon manual inspection of the outcomes produced by Self-Revise (FSP), two prevalent modification patterns are identified. First, the removal of redundant code is a common alteration. This includes the deletion of unnecessary blocks such as "if name == 'main' " and other test codes, which are often extraneous in the context of the desired output. Second, Self-Revise (FSP) exhibits a tendency to directly copy correct code samples from the prompt. In contrast, Self-Revise (FT) is capable of making minimal yet effective modifications to the model's initial error code outputs, thereby generating the correct code. Based on the observations, Self-Revise (FT) is recommended as the more preferable approach for automatic code revision within DEED.

Summary of RQ5: The effectiveness of automatic code revision improves with the use of more powerful LLMs. However, using the LLM consistently with the final optimized LLM and under the fine-tuning setting is most beneficial for the final performance.

5.6 RQ6: Ablation Study of DEED

Baselines. We further perform the ablation study to investigate the effectiveness of each input component in DEED. Requirements and error codes are the indispensable basic inputs for performing automatic code revision. Therefore, we perform ablation experiments on the remaining three components, i.e., **correct solution**, **failed test cases**, and **error messages**. By removing these components individually, we observe their specific impact on the performance of automatic code revision and the final model, and thus evaluate the effectiveness of these components.

Results. We conduct the ablation study on MBPP dataset as shown in Table 6. First, we find that removing the failed test cases resulted in the largest drop in performance of all metrics. Failed test cases can demonstrate the inconsistency between the model-generated code output and the desired output, allowing LLMs to reason about and correct erroneous operations. Experimental results show that this point is most helpful for automatic code revision. Second, removing error messages also results in a loss of performance. Error messages directly indicate surface errors in the generated code (such as syntax errors and runtime errors) and the location of the errors, which is also helpful for LLMs to revise the code. The correct code samples in the dataset can provide some

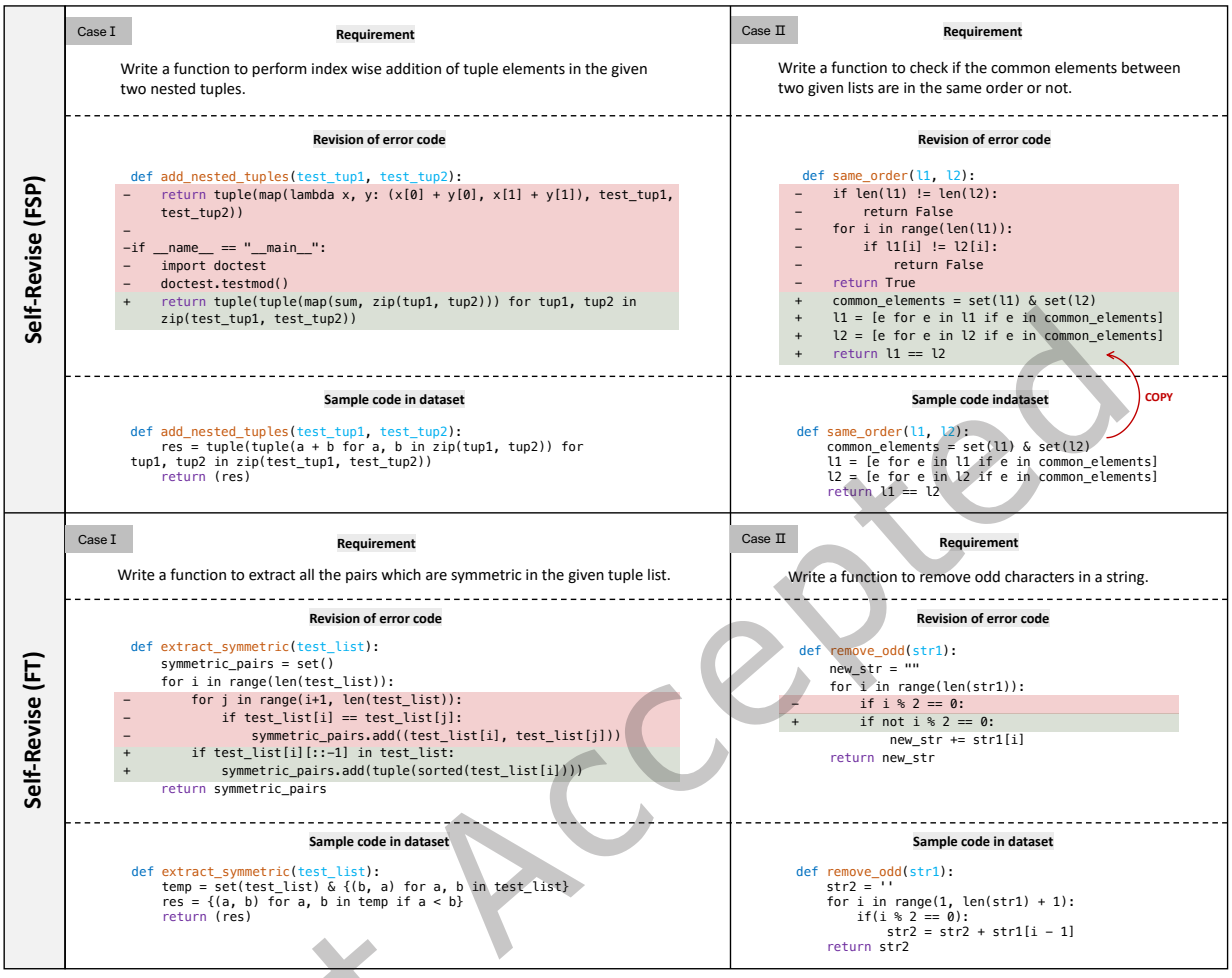

Fig. 4. Cases for two settings of Self-Revise, where "-" and "+" respectively indicate lines of code before and after revision.

Table 6. Effectiveness of each input component in DEED, where $\mathcal{M}_{\text{Revise}}$ represents the Self-Revise model that used during the training and $\mathcal{M}_{\theta^*}$ represents the final model that used during the inference.

Method	$\mathcal{M}_{\text{Revise}}$			$\mathcal{M}_{\theta^*}$	
	Pass@1	Pass@10	Pass@any	Pass@1	Pass@10
DEED	3.9%	18.9%	24.6%	**32.8%**	**64.0%**
- Correct Solution	3.4%	15.4%	19.8%	30.1%	61.9%
- Error Messages	3.1%	14.2%	17.3%	28.6%	58.7%
- Failed Test Cases	2.3%	5.1%	6.3%	26.1%	47.6%

reference for revising errors of LLMs, thus reducing the difficulty of correction. We find that the performance of our method without the correct solution drops a little (less than 4%) compared to the original method. It is still more effective than five existing methods (i.e., Fine-tuning (Full), Fine-tuning (LoRA), Few-shot Prompting, Self-Refine, Self-Debug) as detailed in Table 1.

Summary of RQ6: The analysis of the ablation study indicates that all of the input components in Self-Revise positively contribute to the effectiveness of automatic code revision and the final model.

6 RELATED WORK

6.1 Adaptation of LLMs

Many tasks rely on adapting LLMs to multiple downstream applications. Such adaptation is usually done via fine-tuning, which updates all the parameters of the pre-trained model. Since LLMs contain a large number of model parameters and performing full parameter tuning would be very expensive, a number of parameter-efficient fine-tuning approaches [31, 38, 41] have been developed. Adapter tuning [30, 32] inserts small trainable modules, known as adapters, into pre-trained models. These adapters are usually simple neural networks (e.g., feedforward neural networks) inserted between layers of LLMs. Prompt tuning [38, 47] creates and tunes a small set of artificial tokens, known as a "soft prompt," that is prepended to the input text. These soft prompts are not human-readable text but are instead learned parameters that are optimized during training. Prefix tuning [41, 46] adds a sequence of continuous vectors, known as a "prefix," to the input of each layer of the transformer model. These prefixes are trainable parameters and are optimized during the tuning process. Actually, prefix tuning can also be regarded as deep prompting tuning. Low-rank adaptation [31] modifies the LLMs by adding low-rank matrices to the model's parameters. These lightweight tuning approaches typically achieve near-full fine-tuning performance while reducing the number of parameters required for training. They primarily optimize the efficiency of training the model parameters but are not directly targeted at improving the efficiency of data sample usage.

Another approach of adaptation that does not require training is prompting [44], which depends on in-context learning [7, 19]. By utilizing natural language instructions and a few examples of a task, this approach enables LLMs to recognize and execute new tasks without explicit gradient updates. However, a limitation of this approach is that the models often merely mimic the surface form of the prompt, struggling to deeply understand or adapt to complex and abstract task requirements. Moreover, this approach can be highly sensitive to the specific phrasing of the prompt, leading to variability in the quality and consistency of the generated outputs [26].

Recent advancements in instruction tuning have leveraged LLMs to synthesize extensive data from initial data, enhancing the ability of LLMs to follow instructions. Works such as self-instruction [64], code evol-instruct [49], and oss-instruct [66] approach the problem from the perspective of data augmentation. In contrast, our work focuses on improving the learning efficiency of the models.

Our approach is orthogonal to the aforementioned adaptation techniques, allowing for its concurrent application with these approaches to enhance overall effectiveness.

6.2 Code Generation with LLM

The rise of pre-training techniques has brought new momentum to the field of code generation. Against this backdrop, LLMs, such as Codex [13], CodeGen [53], AlphaCode [42], CodeGeeX [77], and CodeLlama [57] have emerged, greatly enhancing the performance of code generation.

For LLM-based code generation, there are some methods to refine the outputs produced by LLMs. Self-refine [50] enables LLMs to provide feedback on and correct their own generated content. Self-debug [15] allows the LLMs to explain and refine their generated code based on execution results. They belong to prompting methods that are constrained by input length and highly sensitive to prompts [76]. Moreover, Self-edit [73] involves

training an additional editor. This category of methods treats refinement as a post-processing step after code generation, whereas we utilize a self-revise to assist the model in efficient training and thereby enhance the model itself. Compared to these post-processing methods, DEED only requires test cases during training. When training is complete, DEED can be directly used without incurring any additional resource or time costs.

Recently, Chen et al. [10] propose an ILF method focused on using human feedback to refine model results. However, it necessitates continuous human involvement and the provision of feedback throughout the model's training phase, which incurs significant costs in practical applications. Chen et al. [12] propose a distillation method that employs ChatGPT [55] to generate a large amount of refinement to train small models. However, this method presents two primary limitations. Firstly, it necessitates a highly performant "teacher" model, significantly surpassing the capabilities of the "student" model. Secondly, commercial constraints and other factors likely prohibit its implementation. Furthermore, in parallel to our work, Ding et al. [18] propose CYCLE, which enhances the model's self-refinement capability to correct erroneous predictions in previously generated programs according to the test results, rather than focusing on improving the accuracy of a single, one-time prediction, as is the focus of our approach.

In contrast, our work is an adaptation approach, focusing on obtaining a better model adapted to specific domains with limited data. By exploiting the inherent potential of LLMs to achieve error-driven learning, we improve the learning efficiency of LLMs.

7 THREATS TO VALIDITY

There are three major threats to the validity of our work.

1) Threats to external validity concern the quality of experimental datasets and the generalizability of our results. First, we simulate specific code generation scenarios with five public code generation datasets, which are mainstream benchmarks and have been used in many related works [33–35, 65, 74]. Second, DEED can be applied to any LLMs, and we choose four well-known LLMs [43, 67, 70, 71] of different sizes, training data, and architectures for our experiments.

2) Threats to internal validity involve the impact of hyperparameters. Deep learning models are known to be sensitive to hyperparameters.For our proposed DEED, we only do a small-range grid search on hyperparameters, including iterations of DEED, learning rates, and training epochs, leaving other hyperparameters the same as those in previous studies [6, 31, 54, 57, 61, 62], which have explored effective settings of the hyperparameters through extensive experiments. For the baselines, their settings are consistent with our approach.

3) Threats to construct validity pertain to the reliability of evaluation metrics. To address this threat, we employ Pass@k [42] as the evaluation metric, which leverages test cases to gauge the functional correctness of code. Additionally, we employ the unbiased version of Pass@k [13] to diminish evaluation errors that arise from sampling. Pass@k is the mainstream metric for code generation and is widely used in previous studies [20, 25, 52, 63]. On this basis, each experiment is run five times, and its average result is reported.

8 LIMITATIONS

In this section, we analyze two limitations of DEED as follows.

First, our approach requires test cases. The key point to emphasize is that we only need test cases during the preprocessing stage of training data production. At this stage, we have both the requirements and the ground truth code, which is an important premise. Therefore, during the preprocessing stage, obtaining some test cases is not difficult. We can generate a large number of candidate test cases using traditional methods [27] or LLMs [11] and filter out the correct test cases using the ground truth code [43], which we leave as future work. Compared to test cases, what is truly scarce are the pairs of requirements and ground truth code. Our approach focuses on improving the model's performance in this more urgent need.

Second, DEED is only used in low-resource scenarios. Limited training data is a realistic and key problem existing in the real world, which is usually more difficult to solve. In this paper, we try to give an effective approach to address this problem and demonstrate that our approach can achieve significant improvements in the experiments of low-resource settings.

9 CONCLUSION

In this work, we have proposed DEED, a Data-Efficient adaptation with Error-Driven learning for code generation, improving the code generation performance of LLMs in specific scenarios with limited samples. DEED outperforms mainstream adaptation approaches (e.g., full-parameter fine-tuning, LoRA, and few-shot prompting) with limited training data on five code generation benchmarks. Our analysis indicates that LLMs can learn more efficiently by learning from the revisions of their error than by traditionally learning from code examples in datasets. Extensive results show that DEED can largely enhance five different LLMs of varying sizes or series, which proves its applicability. We believe that improving the LLMs' learning efficiency and adapting LLMs with fewer samples has a broad application in real-world scenarios. We leave this area of exploration for future work.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program under Grant No. 2023YFB4503801, the National Natural Science Foundation of China under Grant No. 62192733, 62192730, 62192731, and the Major Program (JD) of Hubei Province (No.2023BAA024)

A EFFECT OF TRAINING DATA SIZE ON PERFORMANCE

We investigate the effect of training data size on performance. This experiment is conducted on MBPP dataset, and the results are presented in Figure 5. As the amount of training data decreases, our approach consistently outperforms both the Direct Generation with base model and the Fine-tuning method. The Fine-tuning method exhibits a much steeper performance decline as data is reduced, quickly underperforming the Direct Generation, which suggests it is more prone to overfitting. However, our method can learn effectively from scarce data.

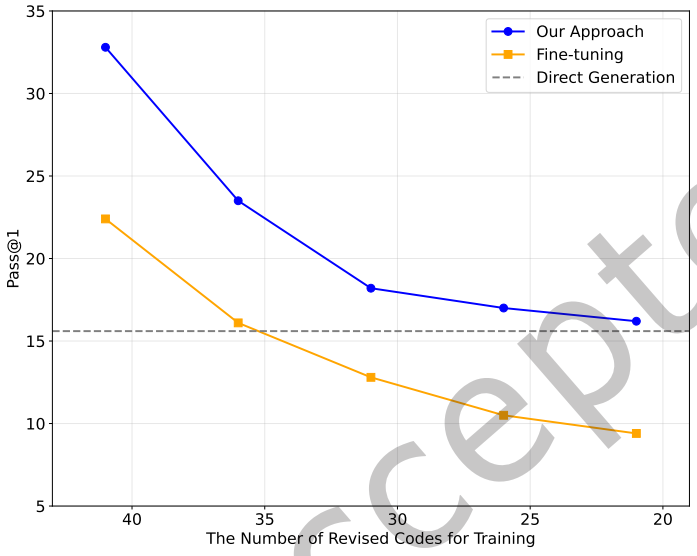

Fig. 5. Performance analysis with varying sizes of training data on MBPP dataset.

Furthermore, we also find a useful case study that helps illustrate the learning capabilities of our approach. One error pattern in the MBPP data is that the model (CodeGen-2B) fails to import Python libraries used in the code. We observe that even after removing all training samples that corrected for a specific missing library, the model can still learn to import it by generalizing from samples involving other libraries. This suggests that the model can learn more generalizable error patterns from existing data.

B PROJECT-LEVEL CODE GENERATION WITH DOMAIN-SPECIFIC EVALUATIONS

We conduct a further evaluation on Evocodebench [40]. Evocodebench is a project-level code generation dataset designed to address the problem of data leakage [22] in evaluation. It achieves this by continuously evolving its data from real-world software projects, while also focusing on domain-specific assessments. Following the original benchmark's methodology, we use the natural language (NL) requirements, function signatures, and surrounding code as prompts. Performance is then measured against the provided test cases using the Pass@k metric. For this evaluation, we use the "camp zifnerf" project of Evocodebench, which contains 49 scientific and engineering samples, and randomly split them into training and test sets at a 2:1 ratio.

The results are shown in Table 7. Our method outperforms the Direct Generation and Fine-tuning baselines across Pass@1, Pass@5, and Pass@10 metrics. This result demonstrates the effectiveness of our approach in real data-scarce scenarios. Furthermore, this performance indicates DEED's broad applicability, proving its efficacy for domain-specific, project-level tasks.

Method	**Pass@1**	**Pass@5**	**Pass@10**
Direct Generation	24.0%	46.7%	50.0%
Fine-tuning	26.7%	44.4%	46.7%
DEED	**28.0%**	**51.1%**	**53.3%**

Table 7. Performance of DEED on Evocodebench dataset.

C THE INSTRUCTION FOR AUTOMATIC CODE REVISION

We use a reasonable instruction at the beginning of the prompt for automatic code revision. All baselines use the same prompt. The specific instruction is as follows:

```
"I've encountered an error code. I will show you the correct code snippet and ask for
your assistance in fixing the error based on that correct code with minimal necessary
revisions."
```


REFERENCES

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2002. Agile software development methods: Review and analysis. (2002).

[2] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. 1985. A Learning Algorithm for Boltzmann Machines. *Cogn. Sci.* 9, 1 (1985), 147–169.

[3] Armen Aghajanyan, Akshat Shrivastava, Anchit Gupta, Naman Goyal, Luke Zettlemoyer, and Sonal Gupta. 2021. Better Fine-Tuning by Reducing Representational Collapse. In *ICLR*. OpenReview.net.

[4] Toufique Ahmed, Christian Bird, Premkumar Devanbu, and Saikat Chakraborty. 2024. Studying LLM Performance on Closed-and Open-source Data. *arXiv preprint arXiv:2402.15100* (2024).

[5] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021).

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[8] George Casella, Christian P Robert, and Martin T Wells. 2004. Generalized accept-reject sampling schemes. *Lecture Notes-Monograph Series* (2004), 342–347.

[9] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *ESEC/SIGSOFT FSE*. ACM, 18–30.

[10] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving Code Generation by Training with Natural Language Feedback. *CoRR* abs/2303.16749 (2023).

[11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *ICLR*.

[12] Hailin Chen, Amrita Saha, Steven C. H. Hoi, and Shafiq Joty. 2023. Personalised Distillation: Empowering Open-Sourced LLMs with Adaptive Learning for Code Generation. *CoRR* abs/2310.18628 (2023).

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen

Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* (2021). https://arxiv.org/abs/2107.03374

[14] Meng Chen, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, Juhong Wang, and Xiaodong Gu. 2023. On the Effectiveness of Large Language Models in Domain-Specific Code Generation. *CoRR* abs/2312.01639 (2023).

[15] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023).

[16] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Trans. Software Eng.* 48, 12 (2022), 4818–4837.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.

[18] Yangruibo Ding, Marcus J. Min, Gail E. Kaiser, and Baishakhi Ray. 2024. CYCLE: Learning to Self-Refine the Code Generation. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 392–418.

[19] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A Survey for In-context Learning. *CoRR* abs/2301.00234 (2023).

[20] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. CodeScore: Evaluating Code Generation by Learning Code Execution. *CoRR* abs/2301.09043 (2023).

[21] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *CoRR* abs/2304.07590 (2023).

[22] Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, and Ge Li. 2024. Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models. *CoRR* abs/2402.15938 (2024).

[23] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A Survey on Code Generation with LLM-based Agents. *CoRR* abs/2508.00083 (2025).

[24] Yihong Dong, Ge Li, and Zhi Jin. 2023. CODEP: Grammatical Seq2Seq Model for General-Purpose Code Generation. In *ISSTA*. ACM, 188–198.

[25] Yihong Dong, Yuchen Liu, Xue Jiang, Bin Gu, Zhi Jin, and Ge Li. 2025. Rethinking Repetition Problems of LLMs in Code Generation. In *ACL (1)*. Association for Computational Linguistics, 965–985.

[26] Yihong Dong, Kangcheng Luo, Xue Jiang, Zhi Jin, and Ge Li. 2023. PACE: Improving Prompt with Actor-Critic Editing for Large Language Model. *CoRR* abs/2308.10088 (2023).

[27] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*. ACM, 416–419.

[28] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022).

[29] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *ICLR*. OpenReview.net.

[30] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. In *ICML (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 2790–2799.

[31] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*. OpenReview.net.

[32] Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. 2023. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. In *EMNLP*. Association for Computational Linguistics, 5254–5276.

[33] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2023. Bias assessment and mitigation in llm-based code generation. *arXiv preprint arXiv:2309.14345* (2023).

[34] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codas, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-Aware Neural Code Rankers. In *NeurIPS*.

[35] Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. 2025. ROCODE: Integrating Backtracking Mechanism and Program Analysis in Large Language Models for Code Generation. In *ICSE*. IEEE, 334–346.

[36] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. *CoRR* abs/2303.06689 (2023).

[37] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *ICML (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 18319–18345.

[38] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *EMNLP (1)*. Association for Computational Linguistics, 3045–3059.
[39] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.
[40] Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. EvoCodeBench: An Evolving Code Generation Benchmark with Domain-Specific Evaluations. In *NeurIPS*.
[41] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *ACL/IJCNLP (1)*. Association for Computational Linguistics, 4582–4597.
[42] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
[43] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *NeurIPS*.
[44] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9 (2023), 195:1–195:35.
[45] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, and Yihan Liao. 2024. Delving into Parameter-Efficient Fine-Tuning in Code Change Learning: An Empirical Study. *CoRR* abs/2402.06247 (2024).
[46] Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021. P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks. *CoRR* abs/2110.07602 (2021).
[47] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. GPT Understands, Too. *CoRR* abs/2103.10385 (2021).
[48] Ilya Loshchilov and Frank Hutter. 2017. Fixing Weight Decay Regularization in Adam. *CoRR* abs/1711.05101 (2017).
[49] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *CoRR* abs/2306.08568 (2023).
[50] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *NeurIPS*.
[51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.* 518, 7540 (2015), 529–533.
[52] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *ICML (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 26106–26128.
[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[54] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *ICLR*. OpenReview.net.
[55] OpenAI. 2022. *ChatGPT*. https://openai.com/blog/chatgpt/
[56] OpenAI. 2024. GPT-3.5-Turbo. https://platform.openai.com/docs/models/gpt-3.5-turbo.
[57] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).
[58] Nayan B. Ruparelia. 2010. Software development lifecycle models. *ACM SIGSOFT Softw. Eng. years* 35, 3 (2010), 8–13.
[59] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *ESEC/SIGSOFT FSE*. ACM, 1533–1543.
[60] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-Tuning of Pre-trained Code Models: An Experimental Study and Beyond. In *ISSTA*. ACM, 39–51.
[61] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).
[62] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog,

Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[63] Xin Wang, Xiao Liu, Pingyi Zhou, Qixia Liu, Jin Liu, Hao Wu, and Xiaohui Cui. 2022. Test-Driven Multi-Task Learning with Functionally Equivalent Code Transformation for Neural Code Generation. In *ASE*. ACM, 188:1–188:6.

[64] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *ACL (1)*. Association for Computational Linguistics, 13484–13508.

[65] Xiaokai Wei, Sujan Kumar Gonugondla, Shiqi Wang, Wasi Uddin Ahmad, Baishakhi Ray, Haifeng Qian, Xiaopeng Li, Varun Kumar, Zijian Wang, Yuchen Tian, Qing Sun, Ben Athiwaratkun, Mingyue Shang, Murali Krishna Ramanathan, Parminder Bhatia, and Bing Xiang. 2023. Towards Greener Yet Powerful Code Generation via Quantization: An Empirical Study. In *ESEC/SIGSOFT FSE*. ACM, 224–236.

[66] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *ICML*. OpenReview.net.

[67] Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. 2024. Grounding Data Science Code Generation with Input-Output Specifications. *CoRR* abs/2402.08073 (2024).

[68] Haoran Xu, Seth Ebner, Mahsa Yarmohammadi, Aaron Steven White, Benjamin Van Durme, and Kenton Murray. 2021. Gradual Fine-Tuning for Low-Resource Domain Adaptation. In *Proceedings of the Second Workshop on Domain Adaptation for NLP*. 214–221.

[69] Runxin Xu, Fuli Luo, Zhiyuan Zhang, Chuanqi Tan, Baobao Chang, Songfang Huang, and Fei Huang. 2021. Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning. In *EMNLP (1)*. Association for Computational Linguistics, 9514–9528.

[70] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When Language Model Meets Private Library. In *EMNLP (Findings)*. Association for Computational Linguistics, 277–288.

[71] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-training on Sketches for Library-oriented Code Generation. In *IJCAI*. ijcai.org, 2369–2375.

[72] Haojie Zhang, Ge Li, Jia Li, Zhongjin Zhang, Yuqi Zhu, and Zhi Jin. 2022. Fine-Tuning Pre-Trained Language Models Effectively by Optimizing Subnetworks Adaptively. In *NeurIPS*.

[73] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *ACL (1)*. Association for Computational Linguistics, 769–787.

[74] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. In *ICLR*. OpenReview.net.

[75] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023. Coder Reviewer Reranking for Code Generation. In *ICML (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 41832–41846.

[76] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *ICML (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 12697–12706.

[77] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *CoRR* abs/2303.17568 (2023).