SimSCOOD: Systematic Analysis of Out-of-Distribution Generalization in Fine-tuned Source Code Models

Anonymous ACL submission

Abstract

Large code datasets have become increasingly accessible for pre-training source code models. However, for the fine-tuning phase, obtaining representative training data that fully covers the code distribution for specific downstream tasks remains challenging due to the task-specific nature and limited labeling resources. These lead to out-of-distribution (OOD) generalization issues with unexpected model inference behaviors that have not been systematically studied yet. In this paper, we contribute the first systematic approach that simulates various OOD scenarios along different dimensions of source code data properties and study the fine-tuned model behaviors in such scenarios. We investigate the behaviors of models under different fine-tuning methodologies, including full fine-tuning and Low-Rank Adaptation (LoRA) fine-tuning methods. Our comprehensive analysis, conducted on four state-of-the-art pretrained models and applied to two code generation tasks, exposes multiple failure modes attributed to OOD generalization issues.

1 Introduction

There has been increasing success in applying Large Language Models (LLMs) to various source code understanding and generation tasks. LLMs for codes such as GraphCodeBERT (Guo et al., 2021), CodeT5+ (Wang et al., 2023), CodeGen (Nijkamp et al., 2023), and Code Llama (Rozière et al., 2023) are pretrained using large-scale code datasets, and serve as universal initialization for a variety of downstream tasks. These tasks include code summarization (Alon et al., 2019; LeClair et al., 2020), text-to-code (Iyer et al., 2018), and program repair (Tufano et al., 2018; Hajipour et al., 2021).

The emerging abilities of LLMs, such as in-context learning, demonstrate their potential to handle a wide range of tasks (Wei et al., 2022; Brown et al., 2020). However, it has been shown that not all tasks can be effectively addressed by relying only on the pretrained LLMs (Anil et al., 2022). To adapt pretrained models for specific tasks, they can be fine-tuned with specific datasets. This fine-tuning process can involve optimizing all parameters or adopting a parameter-efficient approach (Houlsby et al., 2019; Hu et al., 2022), such as Low-Rank Adaptation (LoRA)(Hu et al., 2022). Despite having access to the large code datasets to pre-train these models, it remains challenging in practice to fully cover the code distribution, specifically in fine-tuning datasets, where the availability of labeled data is limited. Furthermore, Kumar et al. (2022) show that, in the image classification tasks, fine-tuning the parameters of the pretrained models can distort the pretrained features.

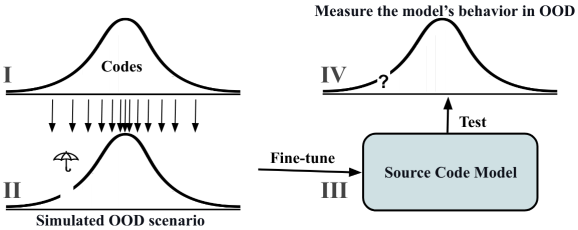


Figure 1: Our approach simulates out-of-distribution (OOD) scenarios and analyzes the corresponding behaviors of models. (I) Original source code distribution along a certain dimension. (II) OOD simulation by masking out a sub-region of the distribution. (III) Model fine-tuning. (IV) Evaluation on OOD data.

Therefore, it is unclear how the fine-tuned code generation models generalize to scenarios not seen or are rare in the fine-tuning distribution (Shen et al., 2021). For example, there is a lack of existing studies to uncover how these models generalize to programs with specific language elements or semantics not seen in fine-tuning datasets. A common way to study model behaviors in OOD scenarios is to collect testing datasets in the complementary domains of the fine-tuning dataset domain (Shen et al., 2021). However, because the underlying distribution of programs is intractable, it is barely feasible to justify whether two raw datasets share a domain or not. Not to mention the substantial costs of constituting a variety of OOD datasets.

Simulating various OOD scenarios by masking out sub-regions of training data distribution is an alternative way to systematically study the model behaviors (Schott et al., 2022; Wiles et al., 2022). There are several distribution dimensions based on data properties. In the source code domain, we can have access to the structural information to model the source code distribution based on the **length**, **syntax**, and **semantics** of programs. For example, in terms of the syntax dimension, we can mask out all the data with *uniray expressions* or specific API to create a syntax-based OOD scenario.

In this work, we propose a systematic approach to analyzing the behaviors of fine-tuned source code models in various OOD and few-data regime scenarios. We achieve this by harnessing the token size, syntax information, and contextual embeddings of programs to simulate the OOD scenarios in terms of length, syntax, and semantics dimensions, as illustrated in Figure 1. By utilizing these data dimensions and control over the data, we can systematically examine the performance of fine-tuned models in OOD scenarios and investigate their generalization capabilities.

To summarize, the main contributions of this paper are as follows: 1. Our work pioneers in investigating the behaviors of the fine-tuned source code models in OOD scenarios. 2. We propose a systematic approach to simulate various OOD scenarios by masking out sub-regions of source code distribution along the length, syntax, and semantics dimensions. At the time of publication, we will publish the implementation of our work. 3. We find that the performance of the fine-tuned models can significantly deteriorate in various OOD scenarios despite the model encountering similar examples during the pre-training phase. In particular, in syntax and length-based OOD scenarios, the drop can be as substantial as 90%. 4. Our systematic analysis shows that, while full fine-tuning and LoRA fine-tuning perform comparably on in-distribution code data, LoRA fine-tuning demonstrates significantly better performance on OOD data. 5. Our analysis of data/model properties provides insights into model finetuning and shapes future datasets/research to focus on OOD of code models, which has the potential to enhance generalization accuracy across various code generation tasks.

2 Related Work

LLMs for Codes. With the availability of large-scale code datasets (Kocetkov et al., 2022), there is growing interest in employing LLMs to develop a pre-training model for source code understanding and generation. CodeBERT extends the RoBERTa-based model (Liu et al., 2019) to understand and generate source codes. Guo et al. (2021) extend CodeBERT by using a semantic-aware objective function. CodeT5 and CodeT5+ (Wang et al., 2021, 2023) are developed based on encoder-decoder architecture, making them versatile models for addressing a wide range of code generation tasks. CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023), and Code Llama (Rozière et al., 2023) employ decoder-only architecture to pre-train code generation models. While these models show remarkable results by following natural language instructions, it has been demonstrated that LLMs still have difficulty in understanding the codes (Austin et al., 2021; Li et al., 2022). In our work, we focus on generation tasks to spot weak and strong points of the fine-tuned LLMs in generating rare and unseen programs.

Out-of-Distribution Analysis in Natural Languages and Programming Languages. Despite the importance of OOD analysis and detection in production (Shen et al., 2021), there are surprisingly much fewer efforts to investigate OOD behaviors of NLP and PL approaches (Arora et al., 2021). Hendrycks et al. (2020); Kong et al. (2020) study the behavior of pretrained LLMs in OOD scenarios. Even though they show pretrained models have higher robustness in OOD scenarios, the provided results indicate that there is still room for improvement. Bui and Yu (2021) propose an energy-bounded-based approach to detect OOD data in source code classification tasks. Their approach defines OOD scenarios by masking out data belonging to the specific class(es) (Bui and Yu, 2021) and does not cover the code generation tasks.

Fine-tuning LLMs. LLMs have demonstrated impressive capabilities in handling various tasks using zero-shot and few-shot learning approaches (Brown et al., 2020; Kojima et al., 2022). However, not all tasks can be effectively handled by relying on pretrained LLMs (Anil et al., 2022; Scialom et al., 2022). For such tasks, we can employ fine-tuning techniques with the datasets for the targeted downstream tasks. Furthermore, recent works indicate that fine-tuning LLMs with instructions can enhance their capabilities (Ouyang et al., 2022; Xu et al., 2023; Dai et al., 2023). Despite the effectiveness of the fine-tuning procedure, Kumar et al. (2022) shows that fine-tuning the models can distort the pretraining features and adversely impact the OOD generalization performance in image classification tasks. In this work, for the first time, we systematically investigate the behavior of the fine-tuned source code models by carefully designing various OOD scenarios.

3 SimSCOOD: Simulation of Source Code Out-of-Distribution Scenarios

In this work, we propose a systematic approach to investigate the fine-tuned code model behaviors on OOD data by simulating the OOD scenarios in multiple dimensions. Our simulation strategy allows us to construct measurable OOD scenarios without the additional costs of accessing another dataset. More importantly, by simulating the OOD scenarios, we have control over different properties of OOD scenarios. We achieve this by masking out specific sub-regions of data distribution.

These OOD scenarios span over three data dimensions, including **length**, **syntax**, and **semantics**. These dimensions cover different aspects of the programs. In length-based OOD scenarios, we can study the length-generalization ability of the fine-tuned models. For example, whether the models can produce longer codes with high quality and how well the models can interpolate over distribution gaps. Syntax-based scenarios enable us to study the models by masking out specific language elements. More interestingly, using syntax-based scenarios, we can analyze to what extent each model can generate unseen language elements. Using semantic-based scenarios, we can investigate how the models behave if we mask out the data with specific functionalities. Benefiting from these scenarios, we can also implicitly quantify how well the models compose different code language elements to achieve unseen or rare functionality.

Modeling the Distribution of Source Code. Here, we experiment with different pretrained models and probe their behaviors in each scenario. We achieve this using our new approach that systematically constructs various scenarios to challenge the OOD performance of each model. As a result, the distribution of source code can be characterized using the aforementioned dimensions that we call properties in the following. We model the joint distribution of the source code as $q(p_1, ..., p_n)$ where each p_i is a specific property of the source code in distribution q. Given this distribution we can sample a dataset $\mathcal{D} = \{x_1, \ldots, x_N | x_i \sim q(p_1, ..., p_n)\}$. To create each OOD scenario we need to sample a new dataset $\hat{\mathcal{D}} = \{x_1, \ldots, x_N | x_i \sim \hat{q}(p_1, ..., p_n)\}$ where $\hat{q}(p_f, ..., p_k) = 0$, meaning the samples with properties $p_f, ..., p_k$ are masked out. Note that we just formulated OOD scenarios with categorical properties, whereas it also holds for continuous properties by $p(a < p_i < b)$ with $a < b$ and $a, b \in \mathbf{R}$.

To sample dataset $\hat{\mathcal{D}}$, we get inspiration from the rejection sampling technique (Casella et al., 2004). Here, $\hat{q}(p_1, ..., p_n)$ is our target distribution and we consider $q(p_1, ..., p_n)$ as our proposal distribution. We reject or accept the sample data $x \sim q(p_1, ..., p_n)$ using the following step function,

$$f(x) = \begin{cases} 1 & \text{if } \mathbf{P}(x) \notin \tilde{\mathcal{P}} \\ 0 & \text{if } \mathbf{P}(x) \in \tilde{\mathcal{P}} \end{cases} \quad (1)$$

Where $\mathbf{P}(x)$ returns the properties of data x, and $\tilde{\mathcal{P}}$ are the properties that we do not want the sampled data x to contain. Using the rejection sampling technique with a hard-decision function (Equation 1) we can construct dataset $\hat{\mathcal{D}} = \{x_1, \ldots, x_N | x \sim \hat{q}(p_1, ..., p_n)\}$ with accepted samples, and also have access to dataset $\tilde{\mathcal{D}} = \{x_1, \ldots, x_N | x \sim \tilde{q}(p_1, ..., p_n)\}$ which are all of the rejected samples. To examine model behaviors in each OOD scenario, we fine-tune models using $\hat{\mathcal{D}}$ data, and test them on test set of $\tilde{\mathcal{D}}$. Figure 2 depicts an overview of the different scenarios. In the following, we provide the details of how we simulate each OOD scenario (subsection 4.1).

3.1 Length-based OOD Scenarios

To simulate length-based scenarios, we use the histogram of program token sizes to represent the distribution of a given dataset. See Figure 2 left as an example. To create each OOD scenario, according to the rejection sampling technique, we draw samples from the distribution and reject only the samples in the histogram's specified sub-region.

As an example, in one of the OOD scenarios, we can consider token size between 120 and 135 as OOD testing data. Then $\hat{\mathcal{D}} = \{x \sim \hat{q}(p_1, ..., p_n)\}$ where $\hat{q}(120 < p_i < 135) = 0$ is the accepted data in the rejection sampling tech-

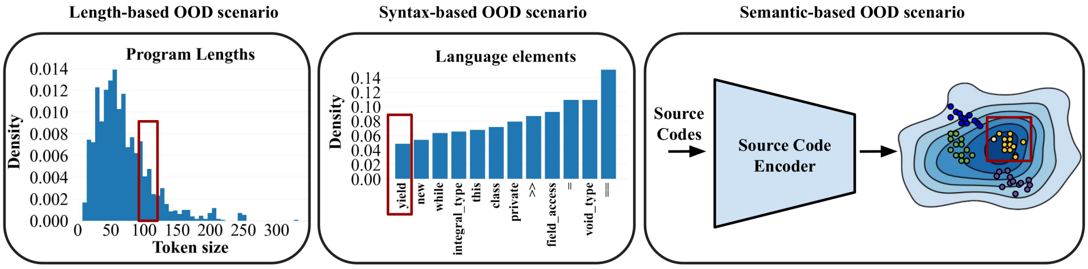


Figure 2: Overview of different out-of-distribution scenarios. Part of the data that needs to be masked out from the training distribution is highlighted by the red rectangles.

nique. Experimenting with the length-based OOD scenarios enables us to analyze how fine-tuned source code models generalize to interpolate and extrapolate over distribution gaps.

3.2 Syntax-based OOD Scenarios

Each programming language has its own grammar, which is a set of rules to define valid program statements. Using the grammar, we can parse each program into an abstract syntax tree (Guo et al., 2021) and have access to all of the elements used in the program. For example, we can identify all the programs with *conditional* or specific APIs in the given dataset. In this work, we leverage the grammatical information of the programming language to create syntax-based OOD scenarios. We use the histogram of language elements to model the syntax distribution of a given source code dataset. Figure 2 middle shows an example of how we construct a syntax-based OOD scenario by masking out specific language elements. To create an OOD scenario, using the rejection sampling technique, we sample testing data $\tilde{\mathcal{D}}$ that contain certain language elements (e.g., *yield*), namely, $\tilde{\mathcal{P}} = \{yield\}$. We then fine-tune our model using $\hat{\mathcal{D}}$ which is the set of data that does not contain *yield*, and test the model using $\tilde{\mathcal{D}}$. In order to set up systematic syntax-based OOD scenarios, we can replace *yield* in $\tilde{\mathcal{P}}$ with other language elements and APIs. Using syntax-based scenarios, in addition to analyzing model behaviors in such OOD scenarios, we can also explore if various fine-tuned LLMs can generate unseen language elements. For example, we can investigate if the pretrained models can generate specific elements not seen during fine-tuning.

3.3 Semantic-based OOD Scenarios

The programs' semantics is another dimension to model the distribution of source code data. However, it is not clear how we can model the semantics of the programs, especially in the cases where we do not have input-output examples or any meta-data. It has been shown that a pretrained model can be used to cluster the data based on their semantics (Aharoni and Goldberg, 2020). Furthermore, recent studies conducted by Troshin and Chirkova (2022) and Ahmed et al. (2023) have demonstrated that pretrained code models represent program semantics within the continuous space. They accomplished this by probing the pretrained models and conducting experiments involving the manipulation of code fragments. Following the success of unsupervised domain clustering and the model's abilities to understand the semantics of programs, we propose to utilize the pretrained source code model to cluster programs within the continuous space. We employ the state-of-the-art CodeT5+ encoder (Wang et al., 2023) in our study to map a dataset of programs to a set of continuous representation vectors. We then cluster the vectors to group programs with similar semantics. As a result, we can create semantic-based OOD scenarios via the rejection sampling procedure to reject all samples that belong to a specific cluster and accept the rest as $\hat{\mathcal{D}}$. Like other scenarios, we can use $\hat{\mathcal{D}}$ as fine-tuning data and $\tilde{\mathcal{D}}$ as test data. Our semantic-based OOD scenarios provide an approximated proxy of real-world OOD scenarios to investigate the OOD generalization capabilities of the fine-tuned models. Furthermore, these OOD scenarios allow us to analyze the model's abilities to deal with unseen or rare program functionalities. We provide implementation details in subsection 4.2.

4 Experiments

In this section, we first articulate the experiment setups, including the pretrained models, downstream tasks, and the OOD data construction. Then, we demonstrate the model performance in OOD scenarios. We also analyze how well the model can perform by revealing 50% of the masked data ($\approx 1.5\%$ of the entire data). In the following, we call the 50% masked-out cases few-data regime.

4.1 Setups

Pretrained Models. We analyze the behavior of four widely-used pretrained models for source codes. These models are designed using a variety of architectures, pre-training objective functions, numbers of parameters, and pre-training datasets. GraphCodeBERT (Guo et al., 2021) is an encoder-only pretrained model with 125M parameters. CodeT5 (Wang et al., 2021) employs T5 (Raffel et al., 2020) encoder-decoder architecture. In our implementations, we use CodeT5-base with 220M parameters. Here, we also investigate the behavior of larger models, including CodeT5+ (Wang et al., 2023) with 770M parameters and Code Llama with 13B parameters. CodeT5+ (Wang et al., 2023) is an extension of CodeT5 (Wang et al., 2021), and Code Llama (Rozière et al., 2023) is a decoder-only build on top of Llama 2 (Touvron et al., 2023) for code-specialized tasks. We provide more details in Appendix A.

Downstream Tasks. We study the behavior of the models on two different downstream tasks, including text-to-code generation and code refinement. These tasks are part of the most challenging tasks in the CodeXGLUE benchmark (Lu et al., 2021). **Text-to-code** is the task of generating a program given a natural language description. In CodeXGLUE benchmark (Lu et al., 2021), CONCODE dataset (Iyer et al., 2018) is proposed for this task. **Code refinement** is the task of resolving the bugs in a given program by automatically generating a corrected program Tufano et al. (2019).

Evaluation Metrics. Exact match (Wang et al., 2021), CodeBLEU (Ren et al., 2020), and BLEU score (Papineni et al., 2002) have been commonly used to evaluate the model performance in the downstream tasks. The exact match metric evaluates if the generated code matches the target code at the token-level. BLEU score measures the n-gram overlap between the output and the target code. CodeBLEU considers syntactic and data-flow matches of the codes in addition to the n-gram overlap. In this work, we focus on the exact match metric to quantify the model behaviors. This is due to the nature of OOD scenarios, where it is desirable to see if the model can generate specific unseen programs correctly. It is important to note that Wang et al. (2021) have demonstrated that for the code refinement task, achieving a high BLEU score can be accomplished with a simple duplication of the input codes, comparable to state-of-the-art models. Furthermore, it has been shown that CodeBLEU and BLEU scores are not necessarily correlated with the correctness of the programs (Evtikhiev et al., 2022; Hendrycks et al., 2021). We report BLEU score results in Appendix G.

4.2 Data Construction and Fine-tuning

In the data construction process, for each scenario, we choose $\tilde{\mathcal{P}}$ in a way that counts for $\approx 3\%$ of the entire fine-tuning data. In OOD scenarios, we mask out all of the data items with properties $\tilde{\mathcal{P}}$. For the few-data regime cases, we mask-out half (50%) of data with properties $\tilde{\mathcal{P}}$ ($\approx 1.5\%$ of the entire fine-tuning data). In all the scenarios, we infer the fine-tuned models on test data with $\tilde{\mathcal{P}}$ properties. Note that, in the text-to-code task, we mask out the data based on the target data (code data rather than text data) properties. For the code refinement tasks, we masked the data based on the input.

Length-based Scenarios. To generate data for length-based scenarios, we characterize the dataset of programs based on the token size. For each scenario, $\tilde{\mathcal{P}}$ specifies a continuous range of program token sizes. We consider five ranges in our experiments: $\tilde{\mathcal{P}}_1 = \{[0\%, 3\%]\}$, $\tilde{\mathcal{P}}_2 = \{[24\%, 27\%]\}$, $\tilde{\mathcal{P}}_3 = \{[48\%, 51\%]\}$, $\tilde{\mathcal{P}}_4 = \{[72\%, 75\%]\}$, and $\tilde{\mathcal{P}}_5 = \{[97\%, 100\%]\}$. Note that $\tilde{\mathcal{P}}_1 = \{[0\%, 3\%]\}$ represents the top 3% smallest programs, in terms of token size. We consider $\tilde{\mathcal{P}}_1$ and $\tilde{\mathcal{P}}_5$ as length-based extrapolation scenarios and $\tilde{\mathcal{P}}_2$, $\tilde{\mathcal{P}}_3$, and $\tilde{\mathcal{P}}_4$ as length-based interpolation scenarios.

Syntax-based Scenarios. In syntax-based scenarios, we characterize program datasets based on the distribution of language elements. For each task, we select five different elements that cover $\approx 3\%$ of the data. For example, in text-to-code task we consider $\tilde{\mathcal{P}}_1 = \{\textit{true}\}$. We provide details of the selected language elements in Appendix E.

Semantic-based Scenarios. In this work, we employ CodeT5+ (770M parameters) (Wang et al., 2023) encoder to characterize the semantics distribution of programs. We feed the tokenized programs to the CodeT5+ encoder and obtain the corresponding feature vectors $\mathbf{V}$ of size $1024 \times t$, where t is the size of the input program. We obtain the continuous representation of the programs by averaging the tokens' embedding following (Koto et al., 2021). We then cluster the programs in continuous space using the K-means algorithm. We set the number of clusters $K = 35$ using the elbow method (Bholowalia and Kumar, 2014). To accelerate the clustering procedure, we perform dimensionality reduction PCA with a target dimension of 50. We determine the dimension in a way that all the components explain at least 80% of the data variance. We provide the average results of five randomly selected clusters. Each cluster can represent a set of $\tilde{\mathcal{P}}_i$ properties. We provide examples of clusters' semantic in Appendix F.

Model Fine-tuning Details. We fine-tune four pretrained models for two different tasks in various scenarios. We stick to their defaults for fair comparisons. For fine-tuning the models with LoRA method, we follow Hu et al. (2022). We provide more details in Appendix C.

4.3 How Do Fine-tuned Models Generalize in OOD Scenarios?

Table 1 and Table 2 shows the overall results of different models in length-, syntax-, and semantic-based scenarios, respectively. These tables show the model performance in the OOD scenarios where the models do not have access to the fine-tuning data with $\tilde{\mathcal{P}}$ properties. Furthermore, Table 1 and Table 2 show how well the models perform when they have access to 50% of the masked data. Note that in Table 1 and Table 2, all of the results are the average of different scenarios and show the relative exact match to the 100% baseline (models with access to the full data distribution). In Table 1 and Table 2, we provide the results of fine-tuning the models using full fine-tuning and LoRA fine-tuning methods. Note that for Code Llama 13B, due to the substantial resource requirements involved in full fine-tuning, we only report the LoRA fine-tuning results. Additionally, in line with GraphCodeBERT (Guo et al., 2021), we only investigate this model on the code refinement task. In these tables, for the length-based scenarios, we have five different scenarios, three for the interpolation cases and two for the extrapolation cases, so we report the average results for each case. In syntax-based and semantic-based scenarios, we report the average results of five different scenarios.

We conclude according to Table 1 and Table 2 that: 1. Interpolation cases in the length-based OOD scenarios, are the easiest OOD scenarios for the models in different tasks. 2. Syntax-based and length-based extrapolation OOD scenarios are the most challenging scenarios for the models. 3. Using LoRA fine-tuning we can achieve significantly better generalization accuracy compared to full fine-tuning. 4. Few-data regime scenarios show that adding a few relevant data to the fine-tuning distribution can gain huge performance improvement. In the following, we describe our key findings in more detail.

Model performance decreases in various OOD scenarios. Table 1 and Table 2 show that all of the models have difficulty in dealing with different OOD scenarios. These include models with different architecture and parameter sizes. For example, in Table 1, we observe that for the Code Llama model with 13B parameters, the performance significantly dropped in the length-based extrapolation scenario. It achieves only 23.57% of the baseline performance.

Table 1 and Table 2 indicate that length-based interpolation scenarios are the least challenging OOD scenarios for various models in both text-to-code and code refinement tasks. While length-based interpolation is the easiest OOD scenario, it is worth noting that CodeT5+ with full fine-tuning only attains 49.67% of the baseline performance (See Table 1). Additionally, Table 1 and Table 2 reveal that the models exhibit the most significant performance reduction in the length-based extrapolation and syntax-based OOD scenarios. This performance drop occurred despite the models being exposed to similar examples during pre-training.

A comparison between the outcomes of the semantic scenarios presented in Table 1 and Table 2 highlights that the text-to-code task is more challenging than the code refinement task. This is mainly due to the multi-modality nature of the task, wherein the models need to learn to map natural languages to unseen or rare programs.

Takeaway: Performance of fine-tuned models, regardless of architectures and sizes, can significantly deteriorate in OOD cases, even when the models have seen similar data during pre-training.

LoRA fine-tuning exhibits better OOD generalization compared to full fine-tuning. In Table 1 and Table 2, we provide the results of fine-tuning the models using two different fine-tuning approaches: full fine-tuning and LoRA fine-tuning. The results presented in these tables indicate that LoRA fine-tuning consistently exhibits superior OOD generalization across various scenarios. For example, Table 1 shows that in the length-based extrapolation scenario, fine-tuning CodeT5 with

Table 1: Overall results of the model performance for different scenarios in **text-to-code** task. The results provide the relative exact match to the 100% baseline for different scenarios. Length Inter and Length Extra refer to length-based interpolation and extrapolation scenarios, respectively. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. OOD and Few refer to OOD and few-data regime scenarios, respectively.

Models		Length Inter		Length Extra		Syntax		Semantic	
		FT	LoRA	FT	LoRA	FT	LoRA	FT	LoRA
CodeT5	OOD	53.92%	66.91%	0.00%	24.99%	16.46%	34.81%	31.90%	51.42%
	Few	86.56%	103.79%	28.56%	55.0%	93.90%	100.0%	37.56%	72.43%
CodeT5+	OOD	49.65%	70.94%	5.0%	26.09%	47.95%	68.97%	39.69%	55.71%
	Few	76.40%	96.36%	77.38%	101.72%	67.21%	78.54%	66.04%	83.68%
Code Llama	OOD	-	71.75%	-	23.57%	-	64.81%	-	56.72%
	Few	-	94.08%	-	63.21%	-	86.08%	-	84.74%

Table 2: Overall results of the model performance for different scenarios in **code refinement** task. The results provide the relative exact match to the 100% baseline for different scenarios. Length Inter and Length Extra refer to length-based interpolation and extrapolation scenarios, respectively. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. OOD and Few refer to OOD and few-data regime scenarios, respectively. GCBERT denotes to the GraphCodeBERT model (Guo et al., 2021).

Models		Length Inter		Length Extra		Syntax		Semantic	
		FT	LoRA	FT	LoRA	FT	LoRA	FT	LoRA
GCBERT	OOD	82.91%	87.89%	37.82%	74.35%	1.30%	2.35%	60.38%	69.05%
	Few	86.52%	94.45%	90.15%	90.46%	75.42%	77.92%	76.45%	84.43%
CodeT5	OOD	84.10%	86.70%	48.95%	61.53%	10.23%	28.78%	77.41%	79.36%
	Few	85.48%	89.97%	57.30%	80.29%	83.08%	85.82%	83.63%	88.73%
CodeT5+	OOD	80.70%	83.39%	73.44%	82.39%	21.41%	37.14%	73.65%	78.67%
	Few	93.28%	94.65%	79.56%	90.77%	72.83%	81.01%	85.30%	93.29%
Code Llama	OOD	-	81.70%	-	57.69%	-	43.70%	-	70.14%
	Few	-	87.68%	-	85.71%	-	87.66%	-	89.23%

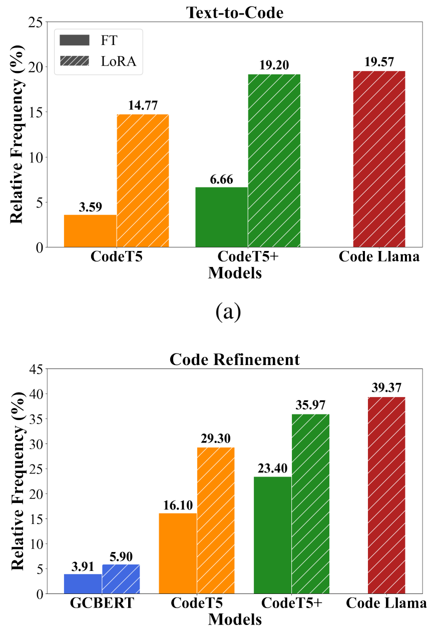
Table 3: Exact match results of the fine-tuned models using the full fine-tuning dataset for text-to-code and code refinement tasks. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. GCBERT refers to (Guo et al., 2021).

Models	Text-to-Code		Refinement	
	FT	LoRA	FT	LoRA
GCBERT	-	-	10.74	11.38
CodeT5	22.15	21.65	14.43	14.53
CodeT5+	24.95	24.70	15.18	15.29
Code Llama	-	27.65	-	19.19

LoRA resulted in a 24.99% relative exact match, whereas the model's relative performance using full fine-tuning was 0.0%. Furthermore, as demonstrated in Table 2, in the syntax-based OOD scenario, the utilization of LoRA for fine-tuning CodeT5 and CodeT5+ results in significantly superior performance compared to employing full fine-tuning for these models. This observation shows that LoRA, effectively leverages the previously acquired knowledge, resulting in improved OOD generalization compared to full fine-tuning.

Table 3 provides in-distribution performance results of the models fine-tuned using both full fine-tuning and LoRA fine-tuning methods. This table displays the exact match accuracy of the models on the complete test set under the condition that the models have access to the entire fine-tuning distribution. Table 3 demonstrates that employing LoRA fine-tuning enables us to achieve performance that is comparable to full fine-tuning. It is important to highlight that in all of the experiments involving LoRA fine-tuning, the pretrained weights are frozen, and we only need to optimize newly injected weights. These LoRA parameters account for less than 1% of the pretrained weights. Note that we provide BLEU score results in Appendix D.

Takeaway: While full and LoRA fine-tuning methods show comparable results over in-distribution data, LoRA fine-tuning outperforms full fine-tuning in OOD scenarios. This suggests that with freezing pretrained weights, LoRA fine-tuned models can effectively utilize their pretraining knowledge in dealing with OOD scenarios.



(a)

(b)

Figure 3: The ratios of frequency of generated unseen language elements over the frequency in ground truth data. Solid and hatched bars show the results of the model fine-tuned with the full fine-tuning and LoRA fine-tuning, respectively.

Models can gain significant improvement by using a few data. Table 1 and Table 2 provide the results for few-data regime scenarios. In these scenarios, we only mask out 50% of the data with $\tilde{\mathcal{P}}$ properties ($\approx$ 1.5% of the fine-tuning data). The Table 1 and Table 2 demonstrate in each scenario that by adding data in size $\approx$ 1.5% of the fine-tuning data, the model can gain significant accuracy performance. For example, Table 1 shows that in syntax-based scenarios, applying LoRA fine-tuning to CodeT5 can lead to a gain of 100% of relative performance by adding a small amount of data. We provide results of revealing 25% and 75% of data in subsection G.2.

Takeaway: By incorporating a small amount of relevant data (representing $\approx$ 1.5% of the fine-tuning data) into the fine-tuning set, models can achieve substantial performance enhancements.

4.4 Can Fine-tuned LLMs Generate Unseen Language Elements?

In the syntax-based OOD scenarios, we can assess the fine-tuned LLMs' ability to leverage their prior knowledge in generating unseen language elements. For instance, can the fine-tuned models generate the *yield* element if they have not been exposed to any code data containing *yield* during fine-tuning? In Figure 3, we present the relative frequencies of generating unseen elements by models fine-tuned using both full and LoRA fine-tuning methods. The results in Figure 3 show the frequencies of generating unseen elements relative to the frequencies in ground truth programs. We report the average results of five different unseen elements during fine-tuning. The list of these elements was reported Appendix E. In Figure 3, the solid bars represent the results for models fine-tuned using full fine-tuning, while the hatched bars depict the results for models fine-tuned using the LoRA method.

Figure 3 shows that the fine-tuned LLMs are able to generate unseen language elements in different tasks. Interestingly, the models fine-tuned using the LoRA fine-tuning exhibit the ability to generate a higher percentage of unseen elements when compared to fully fine-tuned models. This indicates that the models fine-tuned with the LoRA method possess a superior capability to leverage their previously acquired knowledge. We can see this as an advantage. However, in specific scenarios, this advantage can translate into model failures and pose security issues. For example, the model could generate a deprecated API or element, or there can even be cases when the pre-training dataset is poisoned in the first place (Schuster et al., 2021). Furthermore, we observe that generating unseen elements is more challenging in the text-to-code task (Figure 3a) compared to the code refinement task (Figure 3b). The main reason is that in the text-to-code task, the models need to learn the mapping from natural language to the programs.

Takeaway: Models fine-tuned with LoRA generate more unseen elements than those fine-tuned using the full fine-tuning approach, which is advantageous. Nonetheless, in certain scenarios, this capability may result in security issues by generation of deprecated elements and APIs.

5 Conclusion

In this work, we propose a systematic approach to investigate the behaviors of fine-tuned LLMs in OOD scenarios for the program domain. Given the data, we simulate OOD scenarios based on the program's length, syntax, and semantics. Using these scenarios, we shed light on the models' fragility in the OOD scenarios, potential performance drop, and the necessity to improve dataset construction. Furthermore, our results reveal that, although models fine-tuned with full fine-tuning and LoRA exhibit similar in-distribution accuracy, LoRA shows higher OOD generalization accuracy.

Limitations

One of the limitations of our approach is the computational cost. To investigate the model behavior in each dimension, we need to fine-tune individual models. This makes our investigation computationally expensive. Furthermore, in this work, we focus on the code generation tasks as they provide more fine-grained results to investigate the model behavior. It would also be interesting to investigate how the models perform in OOD scenarios in understanding tasks, such as clone detection or defect detection.

In our work, we leverage the contextual embedding of source code to model the semantics of the source codes. We use K-means clustering to group programs based on their semantics. Even though we check if these clusters represent specific meaning (we provide examples of cluster semantics in Appendix F), we do not measure how well these programs are clustered in terms of their semantics. The performance of the clustering algorithm can be measured using datasets with meta-data about the semantics of each data item, which we do not have access to in this study.

Potential Risks. Our research on how models behave in out-of-distribution (OOD) and few-data regime scenarios sheds light on the fine-tuning of models and the development of future datasets. Nonetheless, it is crucial to recognize that malicious actors could exploit these findings to create datasets that intentionally introduce OOD-related issues, with the implicit or explicit goal of targeting specific communities and companies. We recommend that end-users take our findings into consideration when using the source code datasets to train their models.

References

Roee Aharoni and Yoav Goldberg. 2020. Unsupervised domain clusters in pretrained language models. In *ACL*.

Toufique Ahmed, Dian Yu, Chengxuan Huang, Cathy Wang, Prem Devanbu, and Kenji Sagae. 2023. Towards understanding what code language models learned. *arXiv*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *ICLR*.

Cem Anil, Yuhuai Wu, Anders Johan Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Venkatesh Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. 2022. Exploring length generalization in large language models. In *NeurIPS*.

Udit Arora, William Huang, and He He. 2021. Types of out-of-distribution texts and how to detect them. In *EMNLP*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv*.

Purnima Bholowalia and Arvind Kumar. 2014. Ebk-means: A clustering technique based on elbow method and k-means in wsn. *IJCA*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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*.

Nghi D. Q. Bui and Yijun Yu. 2021. Energy-bounded learning for robust models of code. *CoRR*.

George Casella, Christian P Robert, and Martin T Wells. 2004. Generalized accept-reject sampling schemes. *LNMS*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. 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. *arXiv*.

Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Li, Pascale Fung, and Steven Hoi. 2023. Instructblip: Towards general-purpose vision-language models with instruction tuning.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2022. Out of the bleu: how should we assess quality of the code generation models? *arXiv preprint arXiv:2208.03133*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *EMNLP*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *ICLR*.

Hossein Hajipour, Apratim Bhattacharyya, Cristian-Alexandru Staicu, and Mario Fritz. 2021. Samplefix: Learning to generate functionally diverse fixes. In *ECML PKDD*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. 2020. Pretrained transformers improve out-of-distribution robustness. In *ACL*.

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*.

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*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *EMNLP*.

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

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *NeurIPS*.

Lingkai Kong, Haoming Jiang, Yuchen Zhuang, Jie Lyu, Tuo Zhao, and Chao Zhang. 2020. Calibrated language model fine-tuning for in- and out-of-distribution data. In *EMNLP*.

Fajri Koto, Jey Han Lau, and Timothy Baldwin. 2021. Discourse probing of pretrained language models. In *NAACL*.

Ananya Kumar, Aditi Raghunathan, Robbie Matthew Jones, Tengyu Ma, and Percy Liang. 2022. Fine-tuning can distort pretrained features and underperform out-of-distribution. In *ICLR*.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC*.

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

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*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*.

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*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *NeurIPS*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *ACL*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

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

Lukas Schott, Julius Von Kügelgen, Frederik Träuble, Peter Vincent Gehler, Chris Russell, Matthias Bethge, Bernhard Schölkopf, Francesco Locatello, and Wieland Brendel. 2022. Visual representation learning does not generalize strongly within the same domain. In *ICLR*.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security*.

Thomas Scialom, Tuhin Chakrabarty, and Smaranda Muresan. 2022. Fine-tuned language models are continual learners. In *EMNLP*.

Zheyan Shen, Jiashuo Liu, Yue He, Xingxuan Zhang, Renzhe Xu, Han Yu, and Peng Cui. 2021. Towards out-of-distribution generalization: A survey. *CoRR*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv*.

Sergey Troshin and Nadezhda Chirkova. 2022. Probing pretrained models of source code. *arXiv preprint arXiv:2202.08975*.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *ASE*.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *TOSEM*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *TMLR*.

Olivia Wiles, Sven Gowal, Florian Stimberg, Sylvestre-Alvise Rebuffi, Ira Ktena, Krishnamurthy Dj Dvijotham, and Ali Taylan Cemgil. 2022. A fine-grained analysis on distribution shift. In *ICLR*.

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

A Pretrained Models

Here, we provide more detail about the pretrained models we used in our experiments.

A.1 BERT-based Models

CodeBERT (Feng et al., 2020) is an encoder-only transformer-based model that is pretrained using CodeSerchNet dataset (Husain et al., 2019). This dataset consists of 2.1M pairs of individual functions and code documentations with 6.4M code-only data items across multiple programming languages. This model uses a 12-layer RoBERTa-based (Liu et al., 2019) architecture with 125M parameters. It is trained using masked language modeling (MLM) and replaced token detection objective.

Guo et al. (2021) proposed GraphCodeBERT by extending CodeBERT (Feng et al., 2020) using a semantic-aware pre-training objective function. They incorporate data-flow information in the pre-training stage to encode the semantic information of the program.

A.2 CodeT5

CodeT5 (Wang et al., 2021) employ T5 (Raffel et al., 2020) encoder-decoder architecture. The authors use CodeSearchNet (Husain et al., 2019) with 1.2M pairs of functions' code with corresponding documentation, and 0.8M code-only data items. In our experiments, we use CodeT5-base with 220M. This model uses MLM objective and identifier-aware objective functions in the pre-training procedure.

CodeT5+ (Wang et al., 2023) is a family of encoder-decoder LLMs (Wang et al., 2021) that is developed with the flexibility to cover a wide range of downstream tasks. CodeT5+ achieved this flexibility by employing a mixture of pretraining objectives including span denoising, contrastive learning, text-code matching, and causal LM pretraining tasks(Wang et al., 2023). In our experiments we employ CodeT5+ with 770M parameters.

A.3 Code Llama

Code Llama (Rozière et al., 2023) is a family of LLM for code developed based on Llama 2 models (Touvron et al., 2023). The models are designed using decoder-only architectures with 7B, 13B, and 34B parameters. Code Llama encompasses different versions tailored for a wide array of tasks and applications, including the foundational model, specialized models for Python code, and instruction-tuned models. Code Llama outperforms open models on HumanEval (Chen et al., 2021) and MBPP benchmarks (Austin et al., 2021) up to 53% and 55%, respectively. In our experiments, we use the foundation model version of Code Llama with 13B parameters.

B Further Details of Datasets and Computational Resources

To study the behavior of the code generation models in OOD scenarios, we use two datasets of the CodeXGLUE benchmark (Lu et al., 2021) specifically designed for text-to-code and code refinement tasks. The CodeXGLUE benchmark is licensed under Creative Commons Zero v1.0 Universal. The text-to-code task dataset includes 100k training samples, 2k validation samples, and 2k test samples of Java codes. Meanwhile, the code refinement dataset comprises 52,364 training samples, along with 6,545 validation samples and 6,545 test samples of Java codes.

All our experiments are conducted using a single NVIDIA 40GB Ampere A100 GPU. In our study, we fine-tuned more than 350 models, which resulted in 843 GPU hours.

C Hyperparameters for LoRA Fine-tuning

In Table 4, we present the LoRA hyperparameters that were applied in the fine-tuning of various models. We fine-tune these models utilizing AdamW with a linear learning rate decay schedule. During the validation and testing phases, we employed beam search with a beam size of 10, following Wang et al. (2021, 2023); Guo et al. (2021).

For fine-tuning GCBERT, CodeT5, and CodeT5+ in the text-to-code task, we set the maximum input and output sequence length to 320 and 150 tokens, respectively. In the case of fine-tuning Code Llama, we set the maximum sequence length to 470 tokens. In the code refinement task, to fine-tune GCBERT, CodeT5, and CodeT5+, we set the maximum input and output sequence length to 240 and 240 tokens. We fine-tune Code Llama for code refinement tasks by setting the maximum sequence length to 480.

D Comparison of Full Fine-tuning and LoRA fine-tuning Method

In Table 5, you can find the in-distribution performance results of fine-tuned models using the full

Table 4: The LoRA hyperparameters we used to fine-tune the models for text-to-code and code refinement tasks.

Models	Batch Size	#Epoch	Learning Rate	Rank (r_q, r_v)	LoRA α
GCBERT	32	20	$5e^{-4}$	16, 16	32
CodeT5	32	20	$5e^{-4}$	16, 16	32
CodeT5+	16	15	$5e^{-4}$	16, 16	32
Code Llama	4	5	$5e^{-4}$	16, 16	32

and LoRA fine-tuning methods. This table corresponds to a version of Table 3, which additionally includes BLEU score results.

E List of Language Elements

In syntax-based scenarios, we consider one element in each scenario and mask-out the source code with that particular element. Here, we provide the details of five language elements used in our experiments. Note that we pick the element that covers $\approx 3\%$ of the fine-tuning data. We conduct our syntax-based experiments based on the following language elements of each task,

1. **Text-to-Code**: {*else*, *floating_point_type*, *unary_expression*, *array_access*, *true*}

2. **Code Refinement**: {*while_statement*, *long*, *array_creation_expression*, *break*, $\geqslant$}

F Do the clusters represent programs with specific semantics?

Table 6 provides semantics of five random clusters (out of 35) in text-to-code tasks. We randomly check 20 source codes in each cluster to check their semantics.

G More experimental results

G.1 BLEU score Results

In Table 7 Table 8, we provide BLEU score results of different scenarios for the text-to-code and code refinement tasks, respectively. As we mention in subsection 4.1, BLEU scores are not necessarily correlated with the correctness of the programs (Hendrycks et al., 2021) and human judgment (Evtikhiev et al., 2022). For example, a text-to-code model with a high BLEU score could mislead users. Furthermore, Wang et al. (2021) show that in the code refinement task, the BLEU score of a naive copy of the input code can be as good as the state-of-the-art methods. Table 7 shows the performance (BLEU score) dropped for different models in all of the OOD scenarios compared to the 100% baseline. For example, in the length-based extrapolation scenario for the CodeLlama model, the BLEU score dropped over 16 points when compared to the 100% baseline performance. Furthermore, as shown in Table 7, it is evident that across all OOD scenarios, fine-tuning the models using the LoRA approach consistently results in higher BLEU scores. As depicted in Table 8, it is apparent that there are fewer performance drops in comparison to the text-to-code results outlined in Table 7. This distinction can be primarily attributed to the code refinement task's inherent characteristics, wherein naively copying the input tokens to the outputs can yield state-of-the-art BLEU scores.

G.2 Effect of revealing different percentages of the masked data

In Table 9 and Table 10 we show the effect of revealing different percentages of the masked data on the model's performance. Specifically, we showcase CodeT5+ performance in different scenarios by revealing 25%, 50%, and 75% of the masked data (The data was masked for the OOD scenarios). Table 9 presents results for the text-to-code task, while Table 10 displays results for the code refinement task.

Table 9 and Table 10 demonstrate that the model can gain a high performance even by revealing 25% (0.75% of training data). For instance, in Table 9, within length extrapolation scenarios, the full fine-tuned model notably showed relative performance increases from 5.0% (OOD) to 64.63% (Few-25%). Furthermore, both tables indicate that revealing 50% and 75% of the masked data can enhance the model's performance across different scenarios. Nevertheless, the observed performance gains for Few-75% are less apparent compared to the Few-50% and Few-25% cases.

Table 5: Exact match (EM) and BLEU (B) results of the fine-tuned models using the full fine-tuning dataset for text-to-code and code refinement tasks. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. GCBERT refers to (Guo et al., 2021).

Models	Text-to-Code				Refinement			
	FT		LoRA		FT		LoRA	
	EM	B	EM	B	EM	B	EM	B
GCBERT	-	-	-	-	10.74	90.93	11.38	86.45
CodeT5	22.15	39.60	21.65	38.90	14.43	89.33	14.53	89.40
CodeT5+	24.95	44.06	24.70	43.78	15.18	88.19	15.29	89.65
Code Llama	-	-	27.65	45.19	-	-	19.19	90.34

Table 6: Semantics of five clusters in text-to-code task.

Cluster-ID	Semantic
0	Property setter functions
1	Property string getter functions
6	Initialize object
11	Using getter function
17	String concatenation

G.3 Qualitative examples

In Figure 4, Figure 5, and Figure 6, we present qualitative results showcasing instances where the Code Llama model was not able to generate the targeted codes in the OOD scenarios. These examples highlight the challenge that even large fine-tuned LLMs face when handling OOD data. Figure 4 shows an example of the syntax-based OOD scenarios in which the model was unable to generate and use the *else* element. In Figure 5 demonstrates another example from the text-to-code task. Here, we provide an example of the length-based extrapolation OOD scenarios. In these scenarios, our goal is to investigate whether the model is able to extrapolate from shorter programs to longer ones. Figure 5 shows that Code Llama was unable to generate the target program correctly. Note that Figure 5 shows an example of $\tilde{\mathcal{P}}_5 = \{[97\%, 100\%]\}$ OOD scenario, where only 3% of the entire fine-tuning data is masked out. Figure 6 shows an example of the code refinement task. In Figure 6, we provide an example of the syntax-based scenario, in which Code Llama encountered difficulty in generating the *while_statement*. In this syntax-based scenario *while_statement* is the unseen language element.

Input text: Returns true if view's layout direction is right-to-left.

(a) Target Code

```
boolean function (View arg0) {
 if ( Build.VERSION.SDK_INT >=
     VERSION_CODES.JELLY_BEAN_MR1 ) {
  return arg0.getLayoutDirection() ==
     View.LAYOUT_DIRECTION_RTL;
 }
 else {
  return false;
 }
}
```

(b) Generated Code

```

boolean function (View arg0) {
 return arg0.getLayoutDirection() ==
     View.LAYOUT_DIRECTION_RTL;
}
```

Figure 4: An example of generated code by Code Llama in the syntax-based OOD scenario for the text-to-code task. Here *else* is the unseen language element.

Table 7: Overall results of the model performance for different scenarios in **text-to-code** task. The results provide the BLEU score for different scenarios. Length Inter and Length Extra refer to length-based interpolation and extrapolation scenarios, respectively. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. OOD and Few refer to OOD and few-data regime scenarios, respectively. Full refers to 100% baseline (when a model has access to 100% of the fine-tuning set).

Models		Length Inter		Length Extra		Syntax		Semantic	
		FT	LoRA	FT	LoRA	FT	LoRA	FT	LoRA
CodeT5	OOD	40.19	42.03	15.09	15.23	24.08	24.18	44.58	46.21
	Few	48.91	46.47	20.18	18.46	25.20	24.95	45.43	47.97
	Full	47.79	48.34	24.08	23.34	27.01	25.83	48.48	49.65
CodeT5+	OOD	40.58	44.07	15.98	17.48	24.39	26.41	40.52	43.11
	Few	50.07	50.10	19.33	21.67	27.25	27.25	48.93	50.77
	Full	51.80	51.23	23.29	22.63	28.98	28.04	50.89	51.03
Code Llama	OOD	-	54.34	-	21.24	-	25.37	-	47.74
	Few	-	60.35	-	36.73	-	28.06	-	50.76
	Full	-	62.11	-	37.44	-	29.50	-	51.38

Input text: Does this nodetest pass using the specified nodetester instance?

(a) Target Code

```
void function (NodeTester arg0, short [
    ] arg1) {
 NodeIterator loc0 = documentTraversal.
     createNodeIterator(rootNode ,
     NodeFilter.SHOW_ALL, new
     NodeTypeNodeFilter(arg1), true );
 for(Node loc1 = loc0.loc1 ( ); loc1 !=
     null; loc1 = loc0.loc1()){
  arg0.testNode(loc1, this);
 }
 arg0.noMoreNodes(this);
 }
```

(b) Generated Code

```
String function (Node arg0) {
 StringBuilder loc0 = new StringBuilder
     ();
 handleWsRec(arg0, loc0);
 return loc0.toString();
}
```

Figure 5: An example of generated code by Code Llama in the length-based OOD scenario for the text-to-code task. Here we provide an example of length-based extrapolation scenarios ($\tilde{\mathcal{P}}_5 = \{[97\%, 100\%]\}$).

(a) Target Code

```
public java.lang.String toString(){
 java.lang.String VAR_1 = STRING_1;
 int temp = start;
 while ( temp != end){
  VAR_1 += (array[temp]) + STRING_2 ;
  if( temp == (( array.length ) - 1 )){
   temp = 0;
  }else{
   temp ++;}
 }
 VAR_1 += array[temp];
 VAR_1 += STRING_3;
 return VAR_1;
 }
```

(b) Generated Code

```
public java.lang.String toString(){
 java.lang.String VAR_1 = STRING_1;
 if ((METHOD_1()) != null ){
  VAR_1 += METHOD_2();
  if (( METHOD_1 ()) != null){
   VAR_1 += STRING_2;
  }
 }
 VAR_1 += STRING_3;
 return VAR_1;
 }
```

Figure 6: An example of generated code by Code Llama in the syntax-based OOD scenario for the code refinement task. Here *while_statement* is the unseen language element.).

Table 8: Overall results of the model performance for different scenarios in **code refinement** task. The results provide the BLEU score for different scenarios. Length Inter and Length Extra refer to length-based interpolation and extrapolation scenarios, respectively. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. OOD and Few refer to OOD and few-data regime scenarios, respectively. Full refers to 100% baseline (when a model has access to 100% of the fine-tuning set). GCBERT denotes to the GraphCodeBERT model (Guo et al., 2021).

Models		Length Inter		Length Extra		Syntax		Semantic	
		FT	LoRA	FT	LoRA	FT	LoRA	FT	LoRA
GCBERT	OOD	88.22	88.37	83.01	81.45	79.44	81.74	88.36	85.76
	Few	88.59	88.32	85.14	82.75	90.36	87.67	88.95	86.28
	Full	88.32	88.56	84.61	82.99	90.10	87.93	89.73	86.45
CodeT5	OOD	87.37	88.65	80.35	84.11	83.05	87.08	84.68	87.75
	Few	86.67	88.06	81.62	84.22	89.19	90.19	86.54	88.24
	Full	87.39	88.74	83.22	84.22	89.88	88.78	87.69	88.96
CodeT5+	OOD	83.08	86.29	81.26	82.15	84.60	85.48	84.73	85.97
	Few	84.81	87.30	83.03	82.26	88.83	88.96	85.91	86.72
	Full	86.05	87.75	83.17	83.16	89.45	89.01	87.46	86.62
Code Llama	OOD	-	86.40	-	78.30	-	83.29	-	81.32
	Few	-	88.79	-	84.07	-	90.92	-	89.12
	Full	-	89.03	-	84.26	-	91.96	-	89.80

Table 9: Overall CodeT5+ performance results for different scenarios with different amounts of data in **text-to-code** task. Few-XX% show the results of revealing 25%, 50%, and 75% of the masked data to the model. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. OOD and Few refer to OOD and few-data regime scenarios, respectively.

CodeT5+	Length Inter		Length Extra		Syntax		Semantic	
	FT	LoRA	FT	LoRA	FT	LoRA	FT	LoRA
ODD	49.65%	70.94%	5.0%	26.09%	47.95%	68.97%	39.69%	55.71%
Few-25%	69.34%	88.72%	64.63%	86.55%	63.16%	73.75%	59.71%	78.47%
Few-50%	76.40%	96.36%	77.38%	101.72%	67.21%	78.54%	66.04%	83.68%
Few-75%	89.32%	98.82%	93.62%	99.36%	79.50%	88.73%	76.65%	91.28%

Table 10: Overall CodeT5+ performance results for different scenarios with different amounts of data in **code refinement** task. Few-XX% show the results of revealing 25%, 50%, and 75% of the masked data to the model. FT denotes full fine-tuning, and LoRA refers to the LoRA fine-tuning method. OOD and Few refer to OOD and few-data regime scenarios, respectively.

CodeT5+	Length Inter		Length Extra		Syntax		Semantic	
	FT	LoRA	FT	LoRA	FT	LoRA	FT	LoRA
ODD	80.70%	83.39%	73.44%	82.39%	21.41%	37.14%	73.65%	78.67%
Few-25%	89.66%	91.53%	76.82%	87.47%	58.36%	75.44%	81.48%	88.82%
Few-50%	93.28%	94.65%	79.56%	90.77%	72.83%	81.01%	85.30%	93.29%
Few-75%	98.23%	99.51%	86.56%	92.21%	84.24%	89.75%	89.32%	96.52%