Parameter-Efficient Instruction Tuning Code Large Language Models: A Comprehensive Study on OctoCoder Models

Anonymous ACL submission

Abstract

The high cost of full-parameter fine-tuning (FFT) of Large Language Models (LLMs) has led to a series of parameter-efficient fine-tuning (PEFT) methods. However, it remains unclear which methods provide the best cost-performance trade-off at different model scales. We introduce ASTRAIOS, a suite of 28 instruction-tuned OctoCoder models using 7 tuning methods and 4 model sizes up to 16 billion parameters. Through investigations across 5 tasks and 8 different datasets encompassing both code comprehension and code generation tasks, we find that FFT generally leads to the best downstream performance across all scales, and PEFT methods differ significantly in their efficacy based on the model scale. LoRA usually offers the most favorable trade-off between cost and performance. Further investigation into the effects of these methods on both model robustness and code security reveals that larger models tend to demonstrate reduced robustness and less security. At last, we explore the relationships among updated parameters, and task performance. We find that the tuning effectiveness observed in small models generalizes well to larger models, and the validation loss in instruction tuning can be a reliable indicator of overall downstream performance. We believe that our findings of PEFT can generalize to other decoder-only LLMs.[1]

1 Introduction

Large language models (LLMs) (Zhao et al., 2023) trained on Code (Code LLMs) have shown strong performance on various software engineering tasks (Hou et al., 2023). There are three main model paradigms: *(A)* Code LLMs for code completion (Nijkamp et al., 2022; Fried et al., 2022; Li et al., 2023); *(B)* Task-specific fine-tuned Code LLMs for a single task (Hou et al., 2023); and *(C)* Instruction-tuned (Ouyang et al., 2022) Code LLMs that excel at following human instructions and generalizing well on unseen tasks (Wang et al., 2023b; Muennighoff et al., 2023b). Recent instruction-tuned Code LLMs, including WizardCoder (Luo et al., 2023) and OctoCoder (Muennighoff et al., 2024), have achieved state-of-the-art performance on various tasks without task-specific fine-tuning. However, with the increasing parameters of Code LLMs, it becomes more expensive to perform full-parameter fine-tuning (FFT) to obtain instruction-tuned models. In practise, to save computational cost, parameter-efficient fine-tuning (PEFT) have been applied to instruct-tuned LLMs (Liu et al., 2022; Zadouri et al., 2023; Hu et al., 2023a; Gao et al., 2023; Muennighoff et al., 2024). This training strategy aims to achieve comparable performance to FFT by updating fewer parameters. While there are many PEFT methods (Ding et al., 2022), the predominant PEFT method is still LoRA, which is proposed in 2021 (Hu et al., 2021). However, there is no empirical evidence showing LoRA remains the best for instruction-tuned code LLMs. In this paper, we investigate instruction-tuned code LLMs with the following research question: *what are the best PEFT methods for Code LLMs?*

Existing analysis on PEFT methods presents several opportunities for further exploration: *(1)* **Beyond Task-Specific LLMs**. Most prior works (Zhou et al., 2022; Ding et al., 2023) only focus on the model paradigm *(B)*, where the selected base models are fine-tuned on specific downstream tasks. While these studies provide insights into PEFT methods on task-specific LLMs, the transferability of their findings to the instruction tuning paradigm is unclear. *(2)* **Diverse Domains**. Studies on PEFT methods tend to evaluate in the predominant domains like vision (Sung et al., 2022; He et al., 2023; Hu et al., 2023b) and text (Houlsby et al., 2019; He et al., 2021), leaving other do-

---

[1] The codebase (under Apache-2.0 license) and models (under BigCode OpenRAIL-M license) will be publicly available.

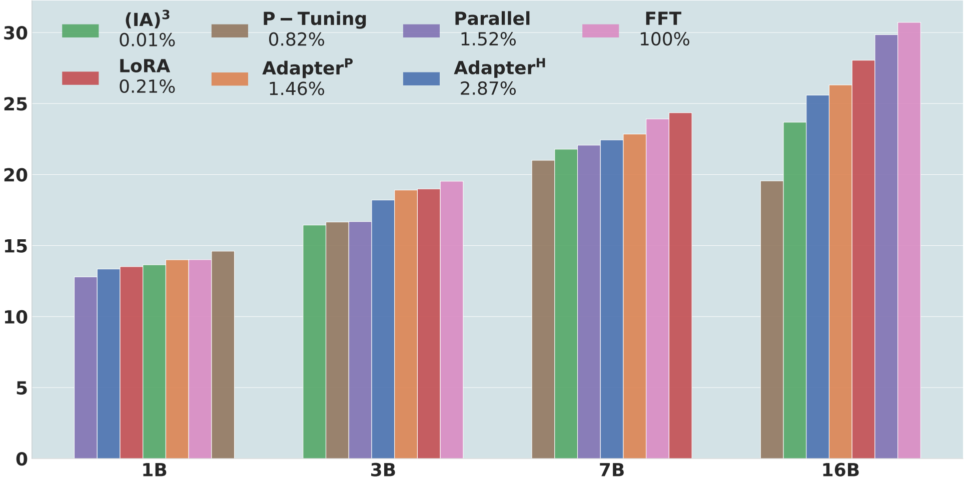


Figure 1: Mean task performance of ASTRAIOS models across 5 representative tasks and 8 datasets. We indicate the average percentage of total parameters updated for each PEFT method.

mains like code underexplored. *(3)* **Inclusive PEFT Methods**. Prior investigations on PEFT mainly consider a limited number of methods, such as adapter-based tuning (Houlsby et al., 2019) or reparametric tuning (Aghajanyan et al., 2021), which does not capture the full breadth of available methods. *(4)* **Multidimensional Evaluation**. Previous works only consider limited evaluation on representative downstream tasks (Chen et al., 2022; Fu et al., 2023; Ding et al., 2023). We argue that other evaluation dimensions like model robustness (Han et al., 2021) and output code safety (Weidinger et al., 2021; Zhuo et al., 2023b; Pearce et al., 2022; Dakhel et al., 2023) are also important, especially in the era of LLM agents (Ouyang et al., 2022; Xie et al., 2023). *(5)* **Scalability**. Most prior PEFT work has only explored LLMs with insufficient scales of model sizes and training time, which makes their scalability questionable (Lester et al., 2021; Chen et al., 2022; Hu et al., 2023a).

To explore these identified opportunities further, we introduce ASTRAIOS, a suite of 28 instruction-tuned Code LLMs, which are fine-tuned with 7 tuning methods based on the StarCoder (Li et al., 2023) base models (1B, 3B, 7B, 16B). We instruction-tune the models based on the open-source dataset, CommitPackFT from OctoPack (Muennighoff et al., 2024), to balance their downstream capabilities. We utilize PEFT configurations with Hugging Face's best practices (Mangrulkar et al., 2022) and integrate a few PEFT methods from recent frameworks (Hu et al., 2023a). We first inspect the scalability of different tuning methods through the lens of cross-entropy loss during instruction tuning. Specifically, we assess the scales of model size and training time. Our main evaluation focuses on 5 representative code tasks, including clone detection (Svajlenko and Roy, 2021), defect detection (Zhou et al., 2019), code synthesis (Muennighoff et al., 2024), code repair (Muennighoff et al., 2024), and code explain (Muennighoff et al., 2024). We further study the tuning methods from two aspects: *model robustness* (Wang et al., 2023a) and *code security* (Pearce et al., 2022). We assess how well models can generate code based on the perturbed examples and how vulnerable the generated code can be.

The main experimental results can be found in Figure 1, where we observe that FFT generally leads to the best downstream performance across all scales. In addition, we find that PEFT methods differ significantly in their efficacy depending on the model scale. At 16B parameters, Parallel Adapter (He et al., 2021) and LoRA (Hu et al., 2021) are the most competitive methods with FFT. Meanwhile, at 1B parameters, they are both slightly outperformed by P-Tuning and (IA)^3. Thus, the choice of the PEFT method should be considered along with the model scale at hand. Nevertheless, LoRA usually offers the most favourable trade-off between cost and performance.

Meanwhile, we also observe that larger PEFT Code LLMs perform better on code generation tasks while they do not show such patterns on code comprehension tasks like clone detection and defect detection. In addition, increasing model size improves generation task performance but exhibits vulnerabilities to adversarial examples and biases

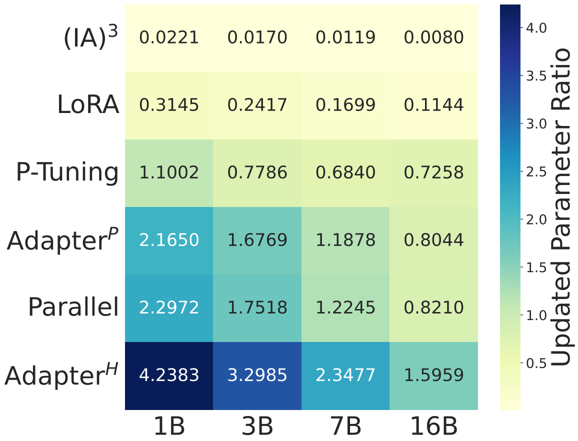


Figure 2: Percentage (%) of total parameters updated for each PEFT method in ASTRAIOS models.

towards insecure code. Additionally, we investigate the relationships among updated parameters, cross-entropy loss, and task performance. We find that the final loss of small PEFT models can be extrapolated to the larger ones. We also observe strong correlations between final loss and overall downstream task performance. Although the instruction dataset we choose is general and is not directly correlated with the benchmark downstream tasks, we suggest that the performance on such general data can serve as a proxy for the downstream performance.

2 The ASTRAIOS Suite and Benchmark

In this section, we document our model choices, training configurations, and evaluations in detail for easy reproducing our experimental results in this paper.

2.1 Model

Base Model There are many Code LLMs available that could be a suitable base model. However, some of them are not fully open such as Code-Llama (Roziere et al., 2023), where the training data is not discussed. To maximize transparency, we select the StarCoder series as our base models. Concretely, four model scales including 1B, 3B, 7B and 16B parameters are selected.

PEFT Model We focus on three kinds of PEFT methods (Ding et al., 2022): *(1)* **Adapter-based Tuning** (Houlsby et al., 2019): An early approach, which injects small-scale neural modules as adapters to LLMs and only tune these adapters for model adaptation. *(2)* **Prompt-based Tuning** (Li and Liang, 2021): It wraps the original input with additional context introducing virtual task-specific tokens without adding layers of modules like adapters. *(3)* **Intrinsic-rank-based Tuning** (Aghajanyan et al., 2021): A representative method is LoRA, which assumes that the change of weights during model tuning has a low rank and thus low-rank changes to the matrices suffice. For all methods, we utilize the implementations in the open-source PEFT library[2] (Mangrulkar et al., 2022) and the LLM-Adapters work (Hu et al., 2023a) built on top of it. We benchmark 6 PEFT methods, including 4 adapter-based, 1 prompt-based, and 1 intrinsic-rank-based tuning methods as shown in Figure 2.

2.2 Instruction Tuning

Dataset Following previous work, we select the dataset CommitPackFT+OASST from OctoPack (Muennighoff et al., 2024) as the instruction tuning dataset, which helps StarCoder to achieve superior performance. We note that there could be other choices by utilizing other datasets (e.g., the publicly available dataset CodeAlpaca (Chaudhary, 2023)) . However, they usually focus on a certain aspect of code-related tasks and lack generality to different tasks.

Configuration We train all models with a sequence length of 2048 tokens, with the batch size as 1, the warm-up step as 12, and the global steps as 200. We set the learning rate as 1×10^{-4} for PEFT models and 1×10^{-6} FFT models with a cosine scheduler in both cases. For PEFT methods, we use 8-bit-quantized models during training (Dettmers et al., 2022). The training details and cross-entropy loss are documented in Appendix D.

2.3 Evaluation

Code Comprehension To evaluate code comprehension, we select two representative tasks: clone detection and defect detection. Clone detection aims to identify segments of code that are either exact duplicates or structurally similar with variations in identifiers, literals, types, layout, and comments, or even more broadly similar in terms of functionality. Defect detection targets for identifying bugs, vulnerabilities, or antipatterns in code. We select two widely-used datasets from CodeXGLUE benchmark (Lu et al., 2021): BigCloneBench (Svajlenko and Roy, 2021) and Devign (Zhou et al.,

---

[2] `https://github.com/huggingface/peft`

2019). As the original BigCloneBench and Devign are designed to evaluate classification models, we prepend additional instructions to prompt the instruction-tuned models to complete such tasks. We follow the evaluation settings of CodeXGLUE and use F1 and Accuracy for BigClone and Devign, respectively. Due to the non-trivial number of test examples in these two datasets, we sample 2,000 from each to save costs. As BigCloneBench and Devign are in the binary classification tasks, we use temperature 0 for model inference to get deterministic outputs.

Code Generation We use HumanEvalPack (Muennighoff et al., 2024), a benchmark recently proposed that enables easy evaluation of instruction-tuned Code LLMs. The benchmark is structured around three core tasks in code generation, each designed to test different capabilities of the model. The first task, Code Synthesis, involves the model in synthesizing functional code given a function with a docstring detailing the desired code behavior. The second task, Code Repair, challenges the model to identify and fix a subtle bug in an otherwise correct code function, using provided unit tests as a guide. The third and final task, Code Explanation, requires the model to generate a clear and concise explanation for a correctly written code function. For the evaluation on HumanEvalPack, we use its Python and Java splits and compute Pass@1 for each task. We use temperature 0.2 and sample 20 outputs per test example.

Model Robustness Evaluating the robustness of code generation models is crucial in understanding their real-world applicability and reliability. Models that can maintain high-performance levels despite variations and perturbations in input data are more likely to be effective in diverse and dynamic coding environments (Bielik and Vechev, 2020; Henkel et al., 2022; Wang et al., 2023a). Motivated by such model behaviors, we utilize ReCode (Wang et al., 2023a), a benchmark framework designed to assess the robustness of Code LLMs. We use HumanEval (Chen et al., 2021) as the base dataset and curated it to mimic natural variations while preserving the semantic integrity of the original inputs. The perturbations cover a range of transformations (Zhuo et al., 2023c) on code format, function, variable names, code syntax, and docstrings. These transformations are not arbitrary but represent changes occurring naturally in coding practices. The quality of the perturbed data in ReCode is verified through human evaluation and objective similarity scores, ensuring the relevance and reliability of the dataset for robustness assessment. We use temperature 0.2 and 20 samples per test example for the generation. To compute the level of model robustness, we adopt Robust Pass@k (RP@k) from ReCode and also compute Robust Change@k (RC@k) as follows:

$$RP@k := \mathbb{E}_x \left[1 - \frac{n - r_c s(x)}{\binom{n}{k}} \right] \quad (1)$$

$$RC@k := |Pass@k - Robust\ Pass@k| \quad (2)$$

Code Security One limitation of Code LLMs is their tendency to generate code with potential security vulnerabilities, as various studies have highlighted (Dakhel et al., 2023; Asare et al., 2023). In our work, we aim to empirically examine how PEFT methods can influence the security aspects of Code LLM outputs. We utilize the “Asleep at the Keyboard” (AATK) benchmark (Pearce et al., 2022), which includes 89 security-centric scenarios, to provide a comprehensive evaluation across three distinct dimensions: Diversity of Weakness (DoW), encompassing 18 unique vulnerability classes from the MITRE Common Weakness Enumeration (CWE) taxonomy, sourced from the 2021 CWE Top 25 Most Dangerous Software Weaknesses; Diversity of Prompt (DoP), assessing responses to different prompts within the SQL injection vulnerability class; and Diversity of Domain (DoD), involving scenarios in Verilog, a hardware description language. Our analysis predominantly focuses on the DoW axis, comprising 54 scenarios–25 in C and 29 in Python–covering 18 CWEs. This focus is due to the automatic evaluation challenges associated with the other two dimensions. After filtering out scenarios that lack an automated test, we thoroughly examine 40 scenarios, including 23 in C and 17 in Python. We use temperature 0.2 and 20 samples per test example for the generation.

3 Main Results: Task Performance

We seek to examine how well selective PEFT methods contribute to task performance in this section. To benchmark the performance, we leverage the representative code downstream tasks: *(1)* Defect Detection, *(2)* Code Clone, *(3)* Code Synthesis, *(4)* Code Repair and *(5)* Code Explanation. For the first two code comprehension tasks, there is no

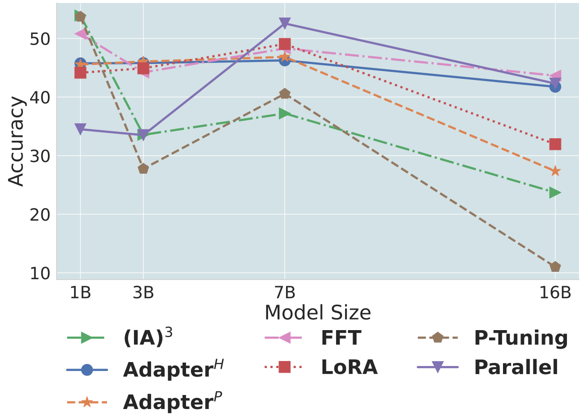


Figure 3: Accuracy results of Astraios models on Defect Detection.

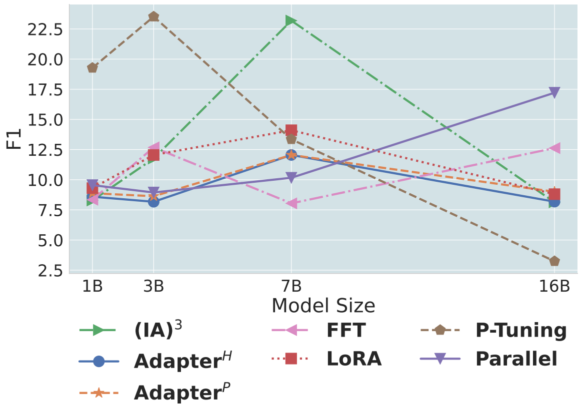


Figure 4: F1 results of Astraios models on Clone Detection.

existing study stating that the larger code LLMs result in a better understanding of code. We are the first to study this aspect when varying the model sizes. Regarding the latter three code generation tasks, previous power-law studies (Kaplan et al., 2020; Hoffmann et al., 2022) have shown that increasing model sizes can also lead to better task performance on generation tasks. We further validate this finding on the PEFT settings.

Code Comprehension Surprisingly, as shown in Figures 3 and 4, the results of both tasks are not well aligned with the patterns we observe on code generation tasks. All tuning methods consistently behave like the inverse scaling, which has been discussed in (McKenzie et al., 2023). We hypothesize that Code LLMs have not seen enough task-specific training data and cannot generalize to those unseen tasks (Yadlowsky et al., 2023). As Astraios models are pre-trained on various source code from GitHub repositories for next token prediction and fine-tuned on GitHub commits for code refinement, they may not have a profound understanding of defects and cloned code. We also show the results of the two code comprehension tasks when varying the model sizes in Appendix G.

Code Generation Table 1 demonstrates the performance on three different code generation tasks on the Python and Java splits in HumanEvalPack. Over the six benchmarks, we first observe that FFT results in consistent gains when the model parameters increase. When examining the PEFT methods, We find they can also provide reasonable performance scalability similar to FFT. Therefore, the lower test loss may lead to better performance across various downstream generation tasks for Code LLMs. However, we notice that the benefit of base model sizes may also differ from tasks and languages. For instance, 1B and 3B models typically underperform in code repair compared to code synthesis. When the model parameters expand to 7B and 16B, their performance across these tasks becomes more comparable.

Overall Performance To compare the overall task performance of different tuning methods, we compute the mean cumulative scores for each tuning method per model size. We present the rankings in Figure 1. We show that FFT remains the best regarding overall task performance, while LoRA and Parallel Adapter are comparable to FFT. However, there is still a huge performance gap between most PEFT methods and FFT, suggesting that they cannot guarantee optimal performance. Regarding the tuning efficiency, we use updated parameters as the metric to summarize two more findings. Firstly, (IA)^3 is efficient enough to perform reasonably by updating much fewer parameters than the other PEFT methods. Secondly, we notice that Adapter^P always performs better than Adapter^H, even though Adapter^H updates more model parameters. The counter-intuitive observation indicates that Adapter^H may not be worth deploying in real-world practice.

4 Further Analysis

In this section, we further study two aspects of Code LLMs beyond task performance. Specifically, we highlight the importance of model robustness and generated code security, which indicate real-world practicality. We tend to understand the trend of model behavior across tuning methods and

Table 1: Pass@1 results of ASTRAIOS models on HumanEvalPack Python and Java splits. The best performance is highlighted in **bold**. The second best performance is underlined.

	Method	Code Synthesis				Code Repair				Code Explanation			
		1B	3B	7B	16B	1B	3B	7B	16B	1B	3B	7B	16B
Python	LoRA	**17.26**	25.37	32.01	38.08	3.29	11.16	21.74	27.50	**20.49**	22.53	**25.34**	30.52
	P-Tuning	15.79	24.33	29.39	35.58	1.86	13.69	20.34	18.72	9.48	11.92	14.60	15.43
	Adapter^H	15.70	23.87	28.26	33.29	3.14	**15.55**	**22.50**	22.28	17.77	22.35	24.24	26.07
	Adapter^P	17.04	24.76	30.67	34.97	3.69	12.87	19.54	26.46	16.07	**24.05**	22.87	30.67
	Parallel	15.98	**26.65**	28.81	35.88	**4.91**	8.11	16.13	26.43	19.70	23.14	23.93	**31.10**
	(IA)^3	16.13	25.34	30.52	36.80	2.01	14.05	17.07	23.60	9.51	11.86	14.30	16.19
	FFT	16.95	25.21	**32.38**	**38.47**	3.26	14.45	21.40	**29.88**	15.37	23.45	26.13	30.85
Java	LoRA	2.84	16.52	**24.27**	40.33	**3.72**	5.06	13.60	30.35	7.07	**14.33**	14.70	16.86
	P-Tuning	**10.67**	14.73	20.73	37.19	0.00	**7.53**	11.74	22.25	6.07	9.79	**17.32**	13.02
	Adapter^H	8.99	13.45	17.53	33.41	0.12	6.89	14.70	24.91	6.74	9.57	13.99	14.85
	Adapter^P	10.46	16.77	21.28	33.68	3.66	6.52	**15.40**	32.07	6.65	11.62	14.15	16.28
	Parallel	9.60	15.91	21.59	38.56	0.49	5.09	8.87	29.39	**7.62**	12.16	14.51	**17.93**
	(IA)^3	10.34	16.46	21.95	39.91	2.87	4.54	13.02	25.30	6.13	13.99	17.04	15.85
	FFT	10.18	**17.04**	23.87	**41.16**	0.00	5.61	16.10	**32.47**	7.16	13.60	15.12	16.62

model sizes.

4.1 Model Robustness

While the performance on downstream tasks is essential, we argue that the evaluation of model robustness is also necessary to characterize different tuning methods systematically. We therefore consider benchmarking the robustness of code synthesis, one of the most representative downstream tasks of source code.

We compute each tuning method's worst-case RP@1 and RC@1 of each perturbation category. Among the four types of perturbation, all models perform the worst on syntax transformation, confirming the findings in (Wang et al., 2023a). Furthermore, RP@1 per tuning method increases when the model size is scaled up, indicating the generation capability is consistently improved. We noticed that FFT may not perform better than other PEFT methods on smaller models, such as 1B and 3B. However, it results in the best RP@1 on larger models like 16B. By comparing different model sizes, we observe that RC@1 consistently increases when the model gets bigger, indicating that larger models will be less robust. To rank among the tuning methods through the lens of robustness, we compute the mean RC@1 similar to Section 3 and illustrate in Figure 5. We observe that FFT and LoRA do not show strong robustness. Instead, adapter-based tuning seems more robust while having comparable performance to FFT, which is similar to what Han et al. (2021) have found in NLP tasks. We reports all RP@1 and RC@1 of each perturbation category in Appendix J.

4.2 Code Security

Table 2: Valid and Insecure rates of ASTRAIOS models on AATK benchmark. We note that the insecure rate is calculated based on the valid programs. The best performance is highlighted in **bold**. The second best performance is underlined.

Method	Valid% (↑)				Insecure% (↓)			
	1B	3B	7B	16B	1B	3B	7B	16B
LoRA	85.9	89.1	75.9	**87.1**	23.1	26.2	**20.9**	35.0
P-Tuning	70.1	68.6	86.8	82.0	32.8	25.9	28.1	34.5
Adapter^H	84.5	90.9	**87.5**	86.8	29.0	26.0	31.9	34.1
Adapter^P	83.9	92.1	82.8	86.3	31.7	25.2	26.6	37.8
Parallel	**88.9**	**94.1**	70.0	86.0	30.2	**19.3**	22.3	32.6
(IA)^3	78.0	62.1	77.4	86.6	34.8	25.2	23.1	**30.4**
FFT	82.9	93.6	80.1	84.1	**22.6**	27.4	21.2	38.3

Previous studies (Dakhel et al., 2023; Asare et al., 2023). have shown that Code LLMs can generate code with security vulnerabilities, which can be exploited by malicious users. However, few studies have studied different tuning methods from the output security perspective. In this experiment, we intend to understand how tuning methods affect the capability to generate secure code on AATK benchmark.

We follow the original setting in (Pearce et al., 2022) and compute the valid and insecure rates, which are illustrated in Table 2. When comparing the valid rate of PEFT methods, it does not show better performance when the model size increases, indicating that current models may not learn the program validity intrinsically. However, we ob-

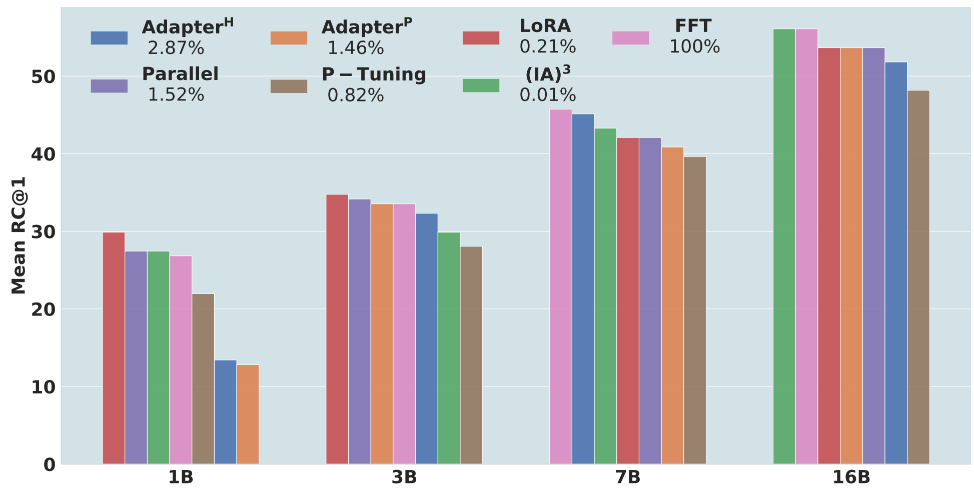


Figure 5: Mean RC@1 of ASTRAIOS on ReCode. Lower RC@1 indicates better robustness. We indicate the percentage of total parameters updated for each PEFT method.

serve that the changes in the insecure rate show that larger models are more likely to generate insecure code. This observation suggests that the growth of learning capability can result in learning more data, including insecure programs. The study on the insecure rate among tuning methods further shows that FFT and LoRA are still better than the other tuning methods regarding the security level. While the other methods have a similar insecure rate, P-Tuning may have more chances to generate less secure programs, which may not be suitable for deploying in security-sensitive scenarios.

5 Discussion

In this section, we seek to conduct a preliminary analysis of the performance of Code LLMs through the lens of updated parameters. Specifically, we ask two questions: *(1) What is the relationship between the updated parameters and cross-entropy loss?*; and *(2) Can we utilize the performance of loss to predict the task performance of Code LLMs?*.

Loss of small models can be projected to larger ones. The relationship between the updated parameters of ASTRAIOS models and their final loss is analyzed in Figure 6. Our analysis does not reveal a consistent pattern across different model sizes when it comes to the correlation between model loss and updated parameters. However, an interesting finding is the consistency in relative loss performance across different model sizes when comparing various tuning methods. This consistency suggests that the improvements achieved by each tuning method are likely to be similar regardless of the model's size. Therefore, the loss observed in smaller models, when tuned with different methods, can be a useful predictor for the performance of the larger models.

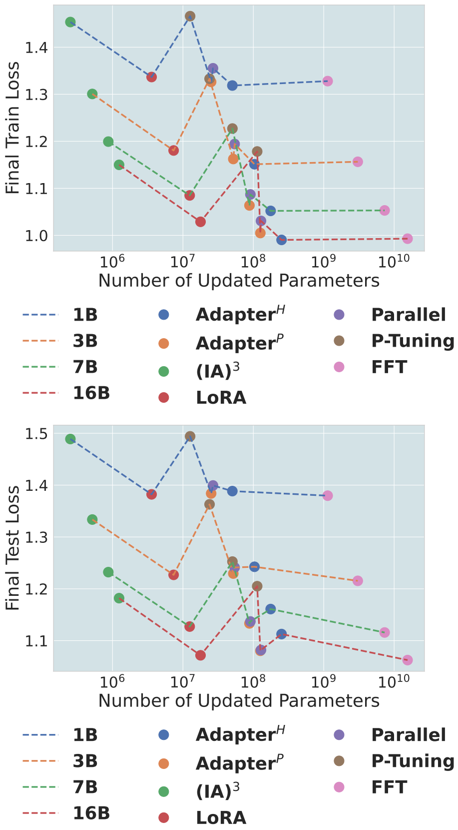


Figure 6: Relationships between cross-entropy loss and the number of updated parameters. Lower loss indicates the bigger models, as shown in Appendix D.

Instruct-tuning loss is a strong predictor of downstream performance. Assuming that the

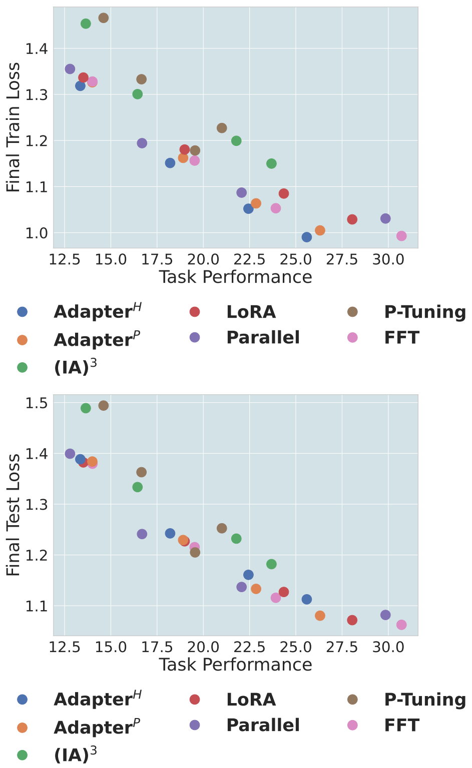


Figure 7: Relationships between cross-entropy loss and overall task performance.

model has been instruction-tuned already but not yet done for the evaluation, we seek to understand if we can utilize such loss to predict its performance on downstream tasks. Despite our instruction data being derived from general sources like GitHub commits and broad NLP domains, which are not directly aligned with the downstream tasks discussed in Section 3, we find some strong correlations. Motivated by the aforementioned scenario, we aggregate all the data points of mean task performance and their corresponding final loss in Figure 7. We observe that the models with lower loss generally have better overall performance on downstream tasks. Specifically, the pattern is stronger on test loss than on train loss. We explain by the fact that the models do not learn to fit the test split and can present a more accurate determination of their actual performance. Our observation suggests that general instruction data can work as a good proxy of downstream tasks in Code LLMs, similar to the prior findings in NLP (Anil et al., 2023; Wei et al., 2023).

6 Related Work

Code Large Language Models Many base Code LLMs have been proposed recently (Chen et al., 2021; Nijkamp et al., 2022; Fried et al., 2022; Allal et al., 2023; Zheng et al., 2023; Li et al., 2023; Roziere et al., 2023) mostly targeting code completion. With the help of these base Code LLMs, there have been extensive studies fine-tuning task-specific Code LLMs to perform software engineering tasks (Hou et al., 2023). Later, a series of works has been proposed for instruction-tuning the base Code LLMs (Luo et al., 2023; Shen et al., 2023; Muennighoff et al., 2024; Bai et al., 2023), aiming to enhance the generalization capabilities of these models on diverse tasks. As fine-tuning Code LLMs with full parameters is costly, most models have been tuned with LoRA (Hu et al., 2021), a parameter-efficient tuning method. In this work, we seek to answer how good LoRA is and if there are other comparable tuning methods.

Model Analysis Across Scales Understanding why and how neural models behave is crucial for developing more advanced ones. Existing studies have investigated predictable patterns in the behavior of trained language models across scales (Kaplan et al., 2020; Henighan et al., 2020; Hernandez et al., 2021; Hoffmann et al., 2022; Wei et al., 2022; Muennighoff et al., 2023a; Xia et al., 2023) and their learning dynamics (McGrath et al., 2022; Tirumala et al., 2022; Biderman et al., 2023). However, they either focus on pre-training or task-specific full-parameter fine-tuning. There is no attempt to understand the mechanism of parameter-efficient instruction tuning. In this paper, we work on this perspective and analyze Code LLMs (Wan et al., 2022; Troshin and Chirkova, 2022; Zhuo et al., 2023a).

7 Conclusion

This work empirically studies the parameter-efficient instruction-tuning of Code LLMs. We introduce a model suite consisting of 28 instruction-tuned OctoCoder across scales and PEFT methods. We characterize the tuning methods on representative downstream tasks, model robustness, and output security, highlighting the importance of understanding these models via comprehensive evaluation. We also discuss the relationships between updated parameters and task performance. We hope these analyses will inspire further follow-up work on understanding the mechanism of tuning methods and developing new approaches. We share more detailed analysis in the Appendix.

Limitations

We discuss a few limitations of our works to motivate future studies in this direction:

Experiment Noise We observe that our empirical results are based solely on a single run of each task, due to budget constraints that prevent us from tuning and evaluating the same Code LLMs multiple times. Although the single evaluation approach limits the breadth of our results and may introduce unexpected experiment noise, it provides a preliminary insight into the performance and potential of PEFT in different scenarios. Future investigations with multiple runs are necessary to establish more robust conclusions and understand the variance and reliability of our results.

Fair Evaluation To compare different PEFT strategies fairly, we have used the same training configurations described in Section 2.2. However, as we find that some PEFT strategies like Prompt Tuning may be sensitive to the training hyperparameters in Section D, the consistent configurations can be unfair. On the other hand, finding the optimal hyperparameters for each PEFT strategy is impractical and can cost more than training with FFT. A more efficient approach is to reuse the hyperparameters in previous work, which motivates us to adopt the default settings in the PEFT library and LLM-Adapter framework. Meanwhile, we believe there may be other practical approaches to benchmark PEFT strategies, encouraging the community to investigate further.

PEFT Strategy We notice that there are many more PEFT strategies (Karimi Mahabadi et al., 2021; Zaken et al., 2022; Wang et al., 2022; Edalati et al., 2022) have been proposed recently. Due to the limited computation budget, we do not include them all in our ASTRAIOS model suite. However, we have publicly made all our source code, data, and models available. We encourage future development in analyzing PEFT strategies on Code LLMs, which helps design more efficient PEFT strategies.

Data Scaling One limitation of our work is that we do not verify the validity of data scaling on PEFT strategies. However, this factor has been well-studied in various works (Kaplan et al., 2020; Hoffmann et al., 2022; Muennighoff et al., 2023a) for model pre-training and fine-tuning. As we find that the performance of PEFT on Code LLMs monotonically increases when scaling up the model size and training time, these selected PEFT strategies are likely aligned with the previous findings of data scaling. We recommend further verification on this aspect.

References

Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. 2021. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 7319–7328.

Armen Aghajanyan, Lili Yu, Alexis Conneau, Wei-Ning Hsu, Karen Hambardzumyan, Susan Zhang, Stephen Roller, Naman Goyal, Omer Levy, and Luke Zettlemoyer. 2023. Scaling laws for generative mixed-modal language models. *arXiv preprint arXiv:2301.03728*.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Owura Asare, Meiyappan Nagappan, and N Asokan. 2023. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering*, 28(6):1–24.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

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

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

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Guanzheng Chen, Fangyu Liu, Zaiqiao Meng, and Shangsong Liang. 2022. Revisiting parameter-efficient tuning: Are we really there yet? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2612–2626.

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

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2022. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235.

Ali Edalati, Marzieh Tahaei, Ivan Kobyzev, Vahid Partovi Nia, James J Clark, and Mehdi Rezagholizadeh. 2022. Krona: Parameter efficient tuning with kronecker adapter. *arXiv preprint arXiv:2212.10650*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Zihao Fu, Haoran Yang, Anthony Man-Cho So, Wai Lam, Lidong Bing, and Nigel Collier. 2023. On the effectiveness of parameter-efficient fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 12799–12807.

Peng Gao, Jiaming Han, Renrui Zhang, Ziyi Lin, Shijie Geng, Aojun Zhou, Wei Zhang, Pan Lu, Conghui He, Xiangyu Yue, et al. 2023. Llama-adapter v2: Parameter-efficient visual instruction model. *arXiv preprint arXiv:2304.15010*.

Wenjuan Han, Bo Pang, and Ying Nian Wu. 2021. Robust transfer learning with pretrained language models through adapters. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 854–861.

Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2021. Towards a unified view of parameter-efficient transfer learning. In *International Conference on Learning Representations*.

Xuehai He, Chunyuan Li, Pengchuan Zhang, Jianwei Yang, and Xin Eric Wang. 2023. Parameter-efficient model adaptation for vision transformers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 817–825.

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B Brown, Prafulla Dhariwal, Scott Gray, et al. 2020. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*.

Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 526–537. IEEE.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. 2021. Scaling laws for transfer. *arXiv preprint arXiv:2102.01293*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

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 *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2021. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. 2023a. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Zi-Yuan Hu, Yanyang Li, Michael R Lyu, and Liwei Wang. 2023b. Vl-pet: Vision-and-language parameter-efficient tuning via granularity control. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3010–3020.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. 2021. Compacter: Efficient low-rank hypercomplex adapter layers. *Advances in Neural Information Processing Systems*, 34:1022–1035.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059.

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

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2023. Gpt understands, too. *AI Open*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

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

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`.

Thomas McGrath, Andrei Kapishnikov, Nenad Tomašev, Adam Pearce, Martin Wattenberg, Demis Hassabis, Been Kim, Ulrich Paquet, and Vladimir Kramnik. 2022. Acquisition of chess knowledge in alphazero. *Proceedings of the National Academy of Sciences*, 119(47):e2206625119.

Ian R McKenzie, Alexander Lyzhov, Michael Pieler, Alicia Parrish, Aaron Mueller, Ameya Prabhu, Euan McLean, Aaron Kirtland, Alexis Ross, Alisa Liu, et al. 2023. Inverse scaling: When bigger isn't better. *arXiv preprint arXiv:2306.09479*.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*.

Niklas Muennighoff, Alexander M Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. 2023a. Scaling data-constrained language models. *arXiv preprint arXiv:2305.16264*.

Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng Xin Yong, Hailey Schoelkopf, Xiangru Tang, Dragomir Radev, Alham Fikri Aji, Khalid Almubarak, Samuel Albanie, Zaid Alyafeai, Albert Webson, Edward Raff, and Colin Raffel. 2023b. Crosslingual generalization through multitask finetuning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15991–16111, Toronto, Canada. Association for Computational Linguistics.

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. In *The Eleventh International Conference on Learning Representations*.

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. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE.

Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True few-shot learning with language models. *Advances in Neural Information Processing Systems*, 34:11054–11070.

Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. 2020. Mad-x: An adapter-based framework for multi-task cross-lingual transfer. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7654–7673.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners.

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

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

Yi-Lin Sung, Jaemin Cho, and Mohit Bansal. 2022. Vl-adapter: Parameter-efficient transfer learning for vision-and-language tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5227–5237.

Jeffrey Svajlenko and Chanchal K Roy. 2021. Bigclonebench. *Code Clone Analysis: Research, Tools, and Practices*, pages 93–105.

Kushal Tirumala, Aram Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. 2022. Memorization without overfitting: Analyzing the training dynamics of large language models. *Advances in Neural Information Processing Systems*, 35:38274–38290.

Sergey Troshin and Nadezhda Chirkova. 2022. Probing pretrained models of source codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 371–383.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2377–2388.

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

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

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Zhen Wang, Rameswar Panda, Leonid Karlinsky, Rogerio Feris, Huan Sun, and Yoon Kim. 2022. Multitask prompt tuning enables parameter-efficient transfer learning. In *The Eleventh International Conference on Learning Representations*.

Jason Wei, Najoung Kim, Yi Tay, and Quoc V Le. 2022. Inverse scaling can become u-shaped. *arXiv preprint arXiv:2211.02011*.

Tianwen Wei, Liang Zhao, Lichang Zhang, Bo Zhu, Lijie Wang, Haihua Yang, Biye Li, Cheng Cheng, Weiwei Lü, Rui Hu, et al. 2023. Skywork: A more open bilingual foundation model. *arXiv preprint arXiv:2310.19341*.

Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, et al. 2021. Ethical and social risks of harm from language models. *arXiv preprint arXiv:2112.04359*.

Mengzhou Xia, Mikel Artetxe, Chunting Zhou, Xi Victoria Lin, Ramakanth Pasunuru, Danqi Chen, Luke Zettlemoyer, and Veselin Stoyanov. 2023. Training trajectories of language models across scales. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13711–13738, Toronto, Canada. Association for Computational Linguistics.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, Leo Z. Liu, Yiheng Xu, Hongjin Su, Dongchan Shin, Caiming Xiong, and Tao Yu. 2023. Openagents: An open platform for language agents in the wild. *CoRR*, abs/2310.10634.

Steve Yadlowsky, Lyric Doshi, and Nilesh Tripuraneni. 2023. Pretraining data mixtures enable narrow model selection capabilities in transformer models. *arXiv preprint arXiv:2311.00871*.

Ted Zadouri, Ahmet Üstün, Arash Ahmadian, Beyza Ermis, Acyr Locatelli, and Sara Hooker. 2023. Pushing mixture of experts to the limit: Extremely parameter efficient moe for instruction tuning. In *The Twelfth International Conference on Learning Representations*.

Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. 2022. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1–9.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2022a. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022b. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Xin Zhou, Ruotian Ma, Yicheng Zou, Xuanting Chen, Tao Gui, Qi Zhang, Xuan-Jing Huang, Rui Xie, and Wei Wu. 2022. Making parameter-efficient tuning more efficient: A unified framework for classification tasks. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 7053–7064.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

Terry Yue Zhuo, Xiaoning Du, Zhenchang Xing, Jiamou Sun, Haowei Quan, Li Li, and Liming Zhu. 2023a. Pop quiz! do pre-trained code models possess knowledge of correct api names? *arXiv preprint arXiv:2309.07804*.

Terry Yue Zhuo, Yujin Huang, Chunyang Chen, and Zhenchang Xing. 2023b. Red teaming chatgpt via jailbreaking: Bias, robustness, reliability and toxicity. *arXiv preprint arXiv:2301.12867*, pages 12–2.

Terry Yue Zhuo, Zhou Yang, Zhensu Sun, Yufei Wang, Li Li, Xiaoning Du, Zhenchang Xing, and David Lo. 2023c. Data augmentation approaches for source code models: A survey. *arXiv preprint arXiv:2305.19915*.

Part I

Appendix

Contents

A What is ASTRAIOS?

ASTRAIOS is a suite of 28 instruction-tuned StarCoder models, employing 7 different PEFT methods across 4 model sizes, with up to 16B parameters. Named after the Greek Titan god of the stars, ASTRAIOS, this model collection represents a vast array of "stars", each model illuminating a path to understanding the cost-performance trade-offs in Code LLMs. Through extensive testing across various tasks and datasets, ASTRAIOS evaluates the efficacy of fine-tuning methods with an emphasis on understanding their performance implications at different model scales, robustness, and security aspects. The suite serves as a celestial guide in the Code LLM universe, helping to chart the most efficient and effective methods for model fine-tuning.

B Artifacts

Name	Public Link
Base Models	
StarCoderBase 1B	`https://huggingface.co/bigcode/starcoderbase-1b`
StarCoderBase 3B	`https://huggingface.co/bigcode/starcoderbase-3b`
StarCoderBase 7B	`https://huggingface.co/bigcode/starcoderbase-7b`
StarCoderBase	`https://huggingface.co/bigcode/starcoderbase`
Instruction Tuning Data	
CommitPackFT + OASST	`https://huggingface.co/datasets/bigcode/guanaco-commits`
Original PEFT Implementation	
LoRA	`https://github.com/huggingface/peft`
P-Tuning	`https://github.com/huggingface/peft`
Adapter^H	`https://github.com/AGI-Edgerunners/LLM-Adapters`
Adapter^P	`https://github.com/AGI-Edgerunners/LLM-Adapters`
Parallel	`https://github.com/AGI-Edgerunners/LLM-Adapters`
(IA)^3	`https://github.com/huggingface/peft`
Prompt	`https://github.com/huggingface/peft`
AdaLoRA	`https://github.com/huggingface/peft`
Evaluation Framework	
Code Generation LM Evaluation Harness	`https://github.com/bigcode-project/bigcode-evaluation-harness`
Astraios Models	
Astraios LoRA 1B	REDACTED
Astraios P-Tuning 1B	REDACTED
Astraios Adapter^H 1B	REDACTED
Astraios Adapter^P 1B	REDACTED
Astraios Parallel 1B	REDACTED
Astraios (IA)^3 1B	REDACTED
Astraios LoRA 3B	REDACTED
Astraios P-Tuning 3B	REDACTED
Astraios Adapter^H 3B	REDACTED
Astraios Adapter^P 3B	REDACTED
Astraios Parallel 3B	REDACTED
Astraios (IA)^3 3B	REDACTED
Astraios LoRA 7B	REDACTED
Astraios P-Tuning 7B	REDACTED
Astraios Adapter^H 7B	REDACTED
Astraios Adapter^P 7B	REDACTED
Astraios Parallel 7B	REDACTED
Astraios (IA)^3 7B	REDACTED
Astraios LoRA 16B	REDACTED
Astraios P-Tuning 16B	REDACTED
Astraios Adapter^H 16B	REDACTED
Astraios Adapter^P 16B	REDACTED
Astraios Parallel 16B	REDACTED
Astraios (IA)^3 16B	REDACTED

Table 3: **Used and produced artifacts.**

Table 4: Summary of tuning methods and the trainable parameters of different model scales.

Type	**Name**	**1B**	**3B**	**7B**	**16B**
Low-Rank	LoRA (Hu et al., 2021)	3,588,096	7,372,800	12,472,320	17,776,640
Prompt	P-Tuning (Liu et al., 2023)	12,650,496	23,882,496	50,466,816	113,448,960
Adapter	$(IA)^3$ (Liu et al., 2022)	251,904	516,096	870,912	1,239,040
	$Adapter^H$ (Houlsby et al., 2019)	50,331,648	103,809,024	176,160,768	251,658,240
	$Adapter^P$ (Pfeiffer et al., 2020)	25,165,824	51,904,512	88,080,384	125,829,120
	Parallel (He et al., 2021)	26,738,688	54,263,808	90,832,896	128,450,560
FFT	FFT	1,137,207,296	3,043,311,104	7,327,263,232	15,517,456,384

C Instruction Tuning

All the instruction tuning experiments have been conducted on A100 80G GPUs. For all PEFT strategies, we use the 8-bit quantized base models for training. For FFT, we use the original base models without quantization.

LoRA We use the attention dimension of 8, the alpha parameter of 16, dropout probability of 0.05, and target modules of "[c_proj, c_attn, q_attn]". We keep the other hyperparameters as default.

P-Tuning We use the 30 virtual tokens and remain the other hyperparameters as default.

Adapter^H We use target modules of "[c_fc, mlp.c_proj]". We keep the other hyperparameters as default.

Adapter^P We use target modules of " [mlp.c_proj]". We keep the other hyperparameters as default.

Parallel We use target modules of "[c_fc, mlp.c_proj]". We keep the other hyperparameters as default.

$(\text{IA})^3$ We target modules of "c_attn, mlp.c_proj]" and feedforward modules of " [mlp.c_proj]".

Prompt (Lester et al., 2021) We use the 30 virtual tokens and keep the other hyperparameters as default.

AdaLoRA (Zhang et al., 2022a) We use the target average rank of the incremental matrix of 8, the initial rank for each incremental matrix of 12, 200 steps of initial fine-tuning warmup, 1000 step of final fine-tuning, the alpha parameter of 16, dropout probability of 0.05, the time interval between two budget allocations of 10, EMA for sensitivity smoothing of 0.85, EMA for uncertainty quantification of 0.85, and target modules of "[c_proj, c_attn, q_attn]". We keep the other hyperparameters as default.

D Preliminary Study: Cross-Entropy Loss

Cross-entropy loss has been used as the principal performance metric in training LLMs for NLP tasks (Brown et al., 2020; Hernandez et al., 2021; Zhang et al., 2022b). Most studies on modeling loss focus on either pre-training (Kaplan et al., 2020) or FFT (Chung et al., 2022). Previous studies have consistent findings on loss (Kaplan et al., 2020; Hoffmann et al., 2022; Aghajanyan et al., 2023): *The final loss tends to decrease when the training computation (e.g., model sizes, training data and training time) increases*. These observations indicate that more training time and more trainable model parameters can lead to better alignment with the tuning data. However, there is no systematic investigation for PEFT, especially for Code LLMs. Based on the updated parameters for each tuning method in Table 4, we hypothesize that each PEFT method has a similar trend to previous findings of loss. Inspired by (Kaplan et al., 2020), we study the loss change for instruction tuning Code LLMs, varying two factors: *(1)* **Model Size** (1B - 16B); and *(2)* **Training Time** (measured in global step, maximum 200 steps). Due to the limited budget, We do not study how the amount of training data may affect the loss.

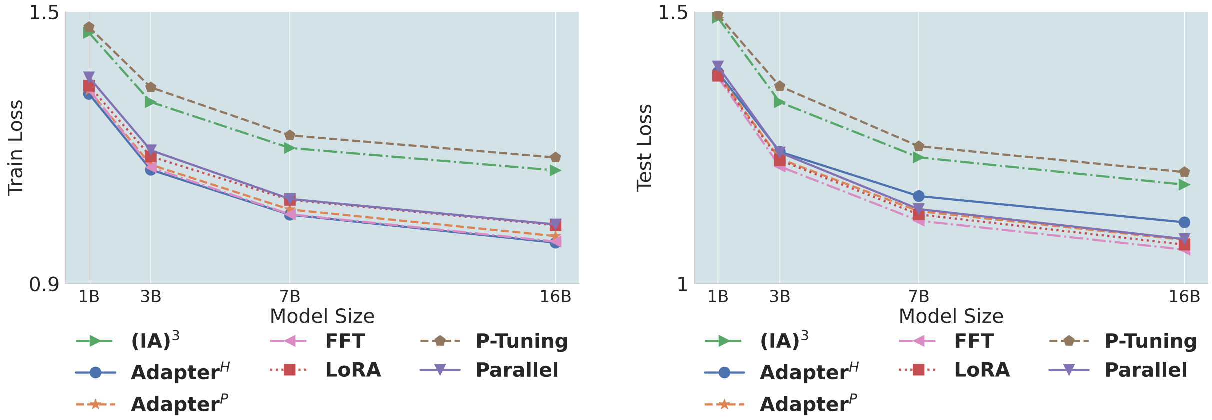


Figure 8: Final loss across model sizes. We note that y-axis is in the logarithmic scale.

Model Size Scaling We present the results of final loss in Figure 8 when varying the model size from 1B to 16B. Our first observation is that train and test loss are well aligned, indicating that the models trained on the selected tuning methods are not overfitted. The second observation is that both train and test loss also strictly decrease when the model size increases. Although these observations are aligned with the aforementioned observations (Kaplan et al., 2020; Hoffmann et al., 2022), they show the different scales of loss change, suggesting different tuning methods may require different levels of power. Compared to other tuning methods, FFT demonstrates a slightly better loss performance than PEFT methods like LoRA and Parallel Adapter. As we notice that heavier PEFT methods (which update more parameters) tend to have a better final loss, we hypothesize that more trainable parameters in the model may result in a smaller loss, regardless of how the parameters are updated during training.

Training Time Scaling We show the changes in test loss on the ASTRAIOS when varying the training time in Figure 9. We notice that the loss continues decreasing when the model is trained longer. Although the loss changes of (IA)^3 are consistently insignificant. Notably, the loss of P-Tuning decreases drastically to 50 steps but behaves similarly to other prompt-based methods. In terms of tuning stability, we observe that P-tuning is more unstable than other methods, where the loss change appears to be a non-monotonic pattern. When comparing FFT against PEFT methods, we find that FFT tends to decrease even after 200 steps, while PEFT methods do not show a decreasing trend clearly. We hypothesize that it may be due to the number of updated parameters, where FFT updates the full parameters in the model.

E Evaluation Setup

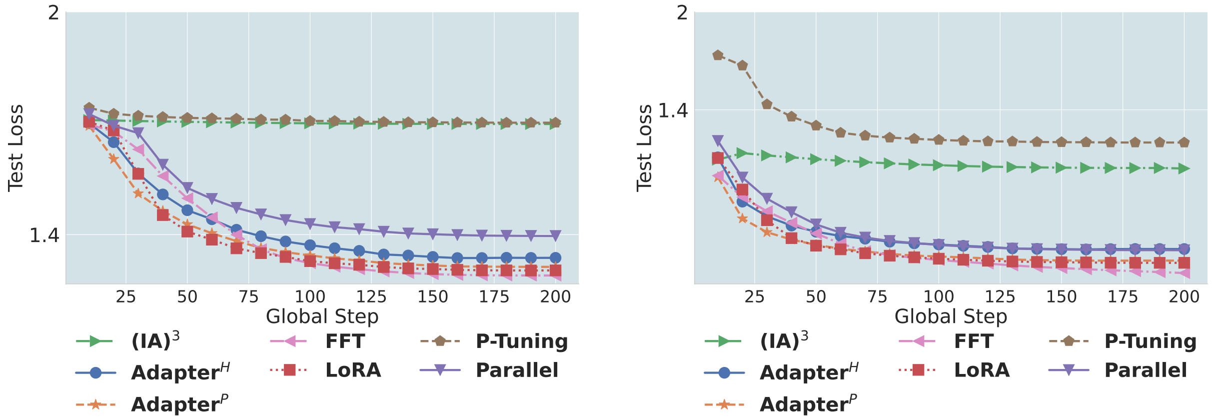
Devign We generate the outputs with a max length of 512 tokens in the style of greedy decoding. All other parameters are defaulted in (Ben Allal et al., 2022). For the one-shot example, we randomly sample from the train set.

BigCloneBench We generate the outputs with a max length of 512 tokens in the style of greedy decoding. All other parameters are defaulted in (Ben Allal et al., 2022). For the one-shot example, we randomly sample from the train set.

HumanEvalPack We generate 20 outputs per example with a max length of 2048 tokens and a temperature of 0.2. All other parameters are defaulted in (Ben Allal et al., 2022).

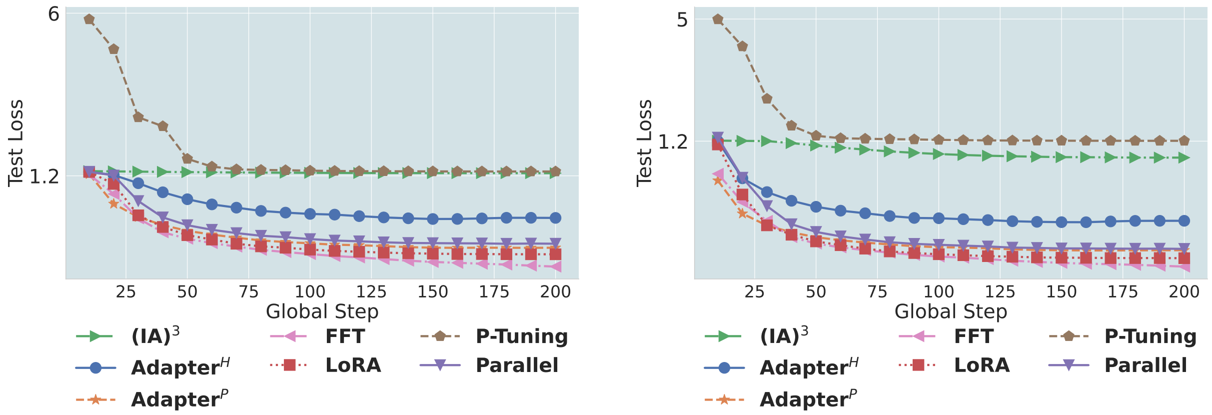
ReCode We generate the outputs with a max length of 1024 tokens in the style of greedy decoding. All other parameters are defaulted in (Ben Allal et al., 2022).

Asleep At The Keyboard We generate 20 outputs per example with a max length of 1024 tokens and a temperature of 0.2. All other parameters are defaulted in (Ben Allal et al., 2022).



1B ASTRAIOS models.

3B ASTRAIOS models.

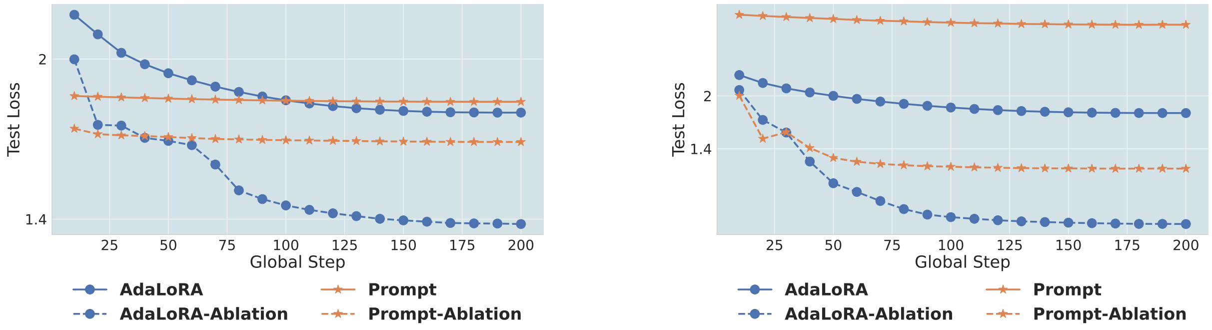


7B ASTRAIOS models.

16B ASTRAIOS models.

Figure 9: Test loss of ASTRAIOS models across training time measured by *Global Step*. We note that y-axis is in the logarithmic scale.

F Failure of Scaling



1B model.

3B models.

Figure 10: Test loss of selected models across training time measured by *Global Step*. We note that y-axis is in the logarithmic scale.

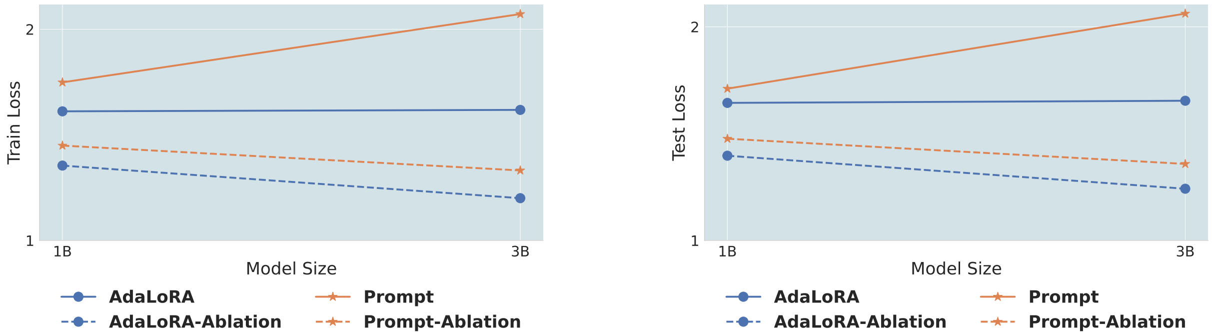


Figure 11: Final loss across model sizes. We note that y-axis is in the logarithmic scale.

During the initial experiment, we also train the models with Prompt Tuning (Lester et al., 2021) and AdaLoRA (Zhang et al., 2022a). Although the loss continues decreasing when the training time increases, we observe the phenomenon of model size scales in contrast to Section 2.2. As shown in Figure 11, the final loss of these two tuning strategies consistently increases as the model size increases, which is contrary to what we observe for other PEFT methods. In the new version of LLM-Adapter (Hu et al., 2023a), we notice that the learning rate has been specifically mentioned. For Prompt Tuning, the authors use 3×10^{-2} instead of 3×10^{-4}, which is used in their other selected PEFT strategies. Therefore, we hypothesize that some tuning strategies may require a much higher learning rate to achieve optimal performance. We further try a few learning rates on training 1B and 3B StarCoderBase models and find that 3×10^{-2} works well for Prompt Tuning. In addition, 3×10^{-2} and 1×10^{-3} also work much better for AdaLoRA. With the new set of learning rates, we find that these tuning strategies are aligned with our findings in Section D. Different from the conclusion of (Kaplan et al., 2020) that the choice of learning rate schedule is mostly irrelevant in language model pre-training, we suggest that hyperparameters of learning rate schedule may matter a lot for scaling parameter-efficient language model on fine-tuning.

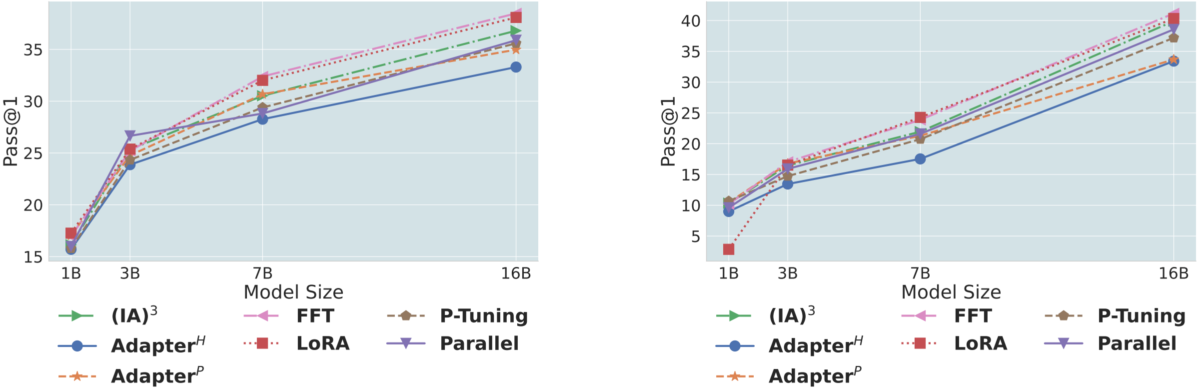
G Code Comprehension

We present the detailed results on Defect Detection and Clone Detection in Table 5.

Table 5: Results of ASTRAIOS models on Defect Detection and Clone Detection. The best performance is highlighted in **bold**. The second best performance is underlined.

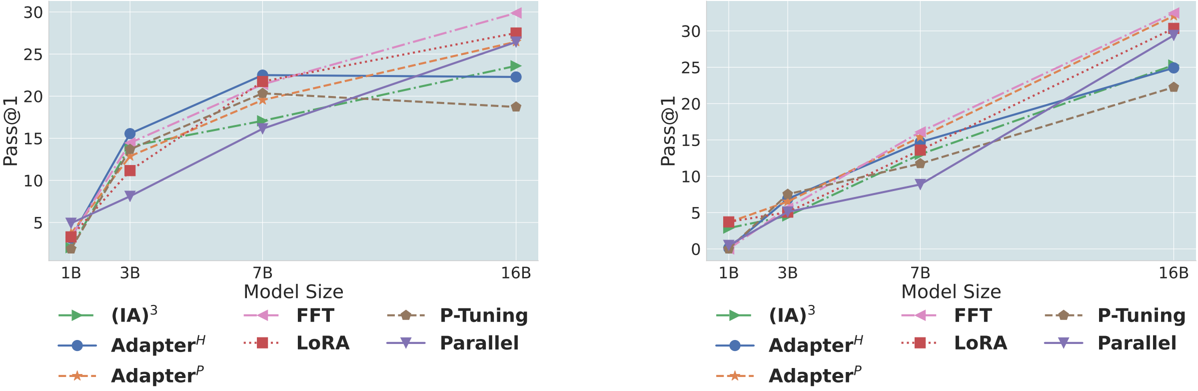
Method	**Defect Detection**				**Clone Detection**			
	1B	**3B**	**7B**	**16B**	**1B**	**3B**	**7B**	**16B**
LoRA	44.15	44.90	49.05	31.95	9.30	12.05	14.10	8.80
P-Tuning	53.70	27.75	40.55	11.00	**19.27**	**23.52**	13.35	3.24
Adapter^H	45.75	45.80	46.25	41.75	8.59	8.17	12.05	8.18
Adapter^P	45.55	46.05	46.85	27.35	8.88	8.63	12.05	9.00
Parallel	34.50	33.50	**52.55**	42.30	9.55	8.94	10.16	**17.21**
(IA)^3	**53.90**	33.55	37.20	23.70	8.28	11.76	**23.19**	8.13
FFT	50.80	44.20	48.30	**43.65**	8.34	12.68	8.04	12.62

H Visualization on HumanEvalPack



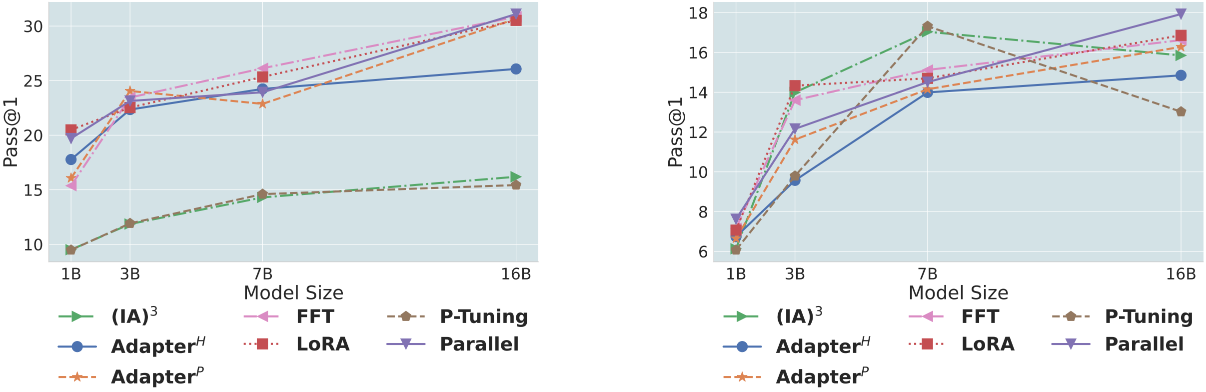
Python Code Synthesize

Java Code Synthesize



Python Code Repair

Java Code Repair



Python Code Explain

Java Code Explain

Figure 12: Pass@1 results of ASTRAIOS models on HumanEvalPack.

I Mitigating Inverse Scaling

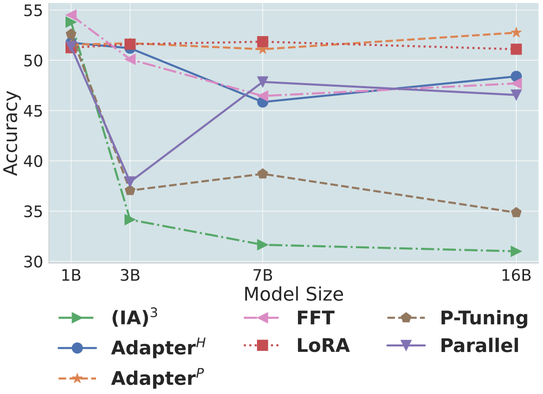


Figure 13: Results on Defect Detection with 1-shot demonstration.

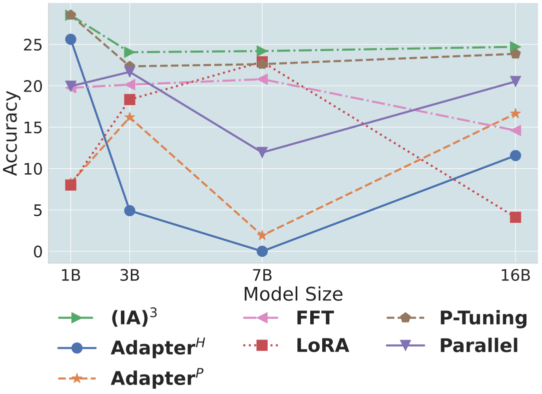


Figure 14: Results on Clone Detection with 1-shot demonstration.

We have attempted to see if the inverse-scaling-like patterns in code comprehension tasks can be mitigated and more aligned with scaling laws. As (Wei et al., 2022) have shown that 1-shot demonstrations can make all inverse scaling tasks U-shaped or flat, we try to see if 1-shot examples can help with defection detection and clone detection. To select the 1-shot examples, we randomly sample a fixed sample from the train set of each benchmark. We re-evaluate all ASTRAIOS models on the two tasks and present the results in Figures 13 and 14. For defect detection, all PEFT strategies become flatter than the previous patterns, which is similar to what (Wei et al., 2022) observe. However, for clone detection, the patterns of some tuning strategies like LoRA and FFT do not turn flat. Although the performances of LoRA and FFT have been scaling up to 7B, they decrease at 15B. We hypothesize that our size scaling is still not significant enough to represent an increasing pattern after 15B for LoRA and FFT with 1-shot demonstrations.

J Model Robustness

We present the detailed results on ReCode in Table 6.

Table 6: RP@1 and RC@1 results of ASTRAIOS models on ReCode. The best performance is highlighted in **bold**. The second best performance is underlined.

	Method	Format				Function				Syntax				Docstring			
		1B	3B	7B	16B	1B	3B	7B	16B	1B	3B	7B	16B	1B	3B	7B	16B
Robust Pass	LoRA	**28.05**	**35.98**	43.29	51.22	**12.80**	15.24	**23.78**	29.27	**8.54**	13.41	**15.85**	18.29	**10.98**	15.24	17.68	20.73
	P-Tuning	18.29	29.88	39.63	48.78	7.32	15.85	21.34	23.78	6.71	11.59	14.02	17.68	6.71	14.63	18.29	21.34
	Adapter^H	10.98	34.15	40.24	46.95	4.88	14.02	17.07	23.78	7.32	11.59	12.20	15.85	6.10	12.80	14.63	17.68
	Adapter^P	9.76	35.37	43.90	50.00	1.22	15.85	21.34	26.22	4.88	12.20	14.63	18.29	3.05	15.24	**19.51**	20.12
	Parallel	26.22	32.32	42.68	50.00	10.37	11.59	21.95	26.83	7.93	12.80	14.63	17.07	8.54	15.24	17.68	21.95
	(IA)^3	26.83	33.54	42.07	50.61	**12.80**	**17.07**	21.34	26.83	7.93	12.20	14.63	17.07	10.37	**15.85**	18.90	**22.56**
	FFT	20.12	35.37	**45.73**	**53.05**	5.49	15.85	21.34	30.49	7.32	**14.63**	**15.85**	19.51	6.10	14.02	18.90	**22.56**
Robust Change	LoRA	10.98	14.63	15.24	15.85	4.27	6.10	**4.27**	**6.10**	8.54	7.93	12.20	17.07	6.10	6.10	10.37	14.63
	P-Tuning	6.10	**9.76**	**12.80**	17.68	4.88	4.27	5.49	7.32	5.49	8.54	12.80	**13.41**	5.49	5.49	**8.54**	**9.76**
	Adapter^H	**0.61**	15.85	15.85	15.85	5.49	4.27	7.32	7.32	3.05	**6.71**	12.20	15.24	4.27	5.49	9.76	13.41
	Adapter^P	3.66	14.63	17.68	15.85	4.88	4.88	4.88	7.93	**1.22**	8.54	**11.59**	15.85	**3.05**	5.49	6.71	14.02
	Parallel	12.20	11.59	15.85	15.24	3.66	9.15	4.88	7.93	6.10	7.93	12.20	17.68	5.49	5.49	9.15	12.80
	(IA)^3	10.98	12.80	14.02	**14.63**	**3.05**	**3.66**	6.71	9.15	7.93	8.54	13.41	18.90	5.49	**4.88**	9.15	13.41
	FFT	7.32	14.02	17.68	15.24	7.32	5.49	6.71	7.32	5.49	**6.71**	12.20	18.29	6.71	7.32	9.15	15.24

K Further Discussion

We further measure the correlations among final loss in Section D, overall task performance in Section 3, and numbers of updated parameters via three metrics, Kendall (τ), Pearson (r_p), and Spearman (r_s)

coefficients. Kendall coefficient measures the ordinal association and is robust against outliers, making it useful for non-normal data distributions. Pearson's coefficient assesses linear correlation, which is ideal for normal data distributions with expected linear relationships. Spearman's coefficient, like Kendall coefficient, is a non-parametric measure that assesses rank correlation, useful for identifying monotonic but non-linear relationships.

Table 7: Correlations between trainable parameters and final loss. p-values are provided in gray.

Model Size	**Train Loss**			**Test Loss**		
	τ	r_p	r_s	τ	r_p	r_s
1B	.4286	.3113	.6071	.3333	.3358	.4643
3B	.5238	.3433	.7143	.2381	.3835	.4286
7B	.5238	.3555	.7143	.2381	.4091	.4286
16B	.5238	.3524	.7143	.2381	.3986	.4286
Overall	.4339 (.00)	.3328 (.08)	.5616 (.00)	.3598 (.01)	.3308 (.09)	.4953 (.01)

We compute the correlations between updated parameters of Astraios models and the final loss of corresponding models in Table 7. From the table, we first observe that the updated parameters are more correlated to the final train loss than the test loss. However, they all imply that there is a moderated correlation, which can be used for cross-entropy loss in model training. We also observe that when we aggregate all statistics across model sizes, the correlations may slightly decrease.

Table 8: Correlations between final loss and overall task performance. p-values are provided in gray.

Model Size	**Train Loss**			**Test Loss**		
	τ	r_p	r_s	τ	r_p	r_s
1B	-.2381	-.4319	-.285	.04	-.4328	-.0357
3B	.5238	.7819	.7143	.8095	.7859	.9286
7B	.5238	.7165	.6786	.8095	.8230	.9286
16B	.3333	.8096	.5000	.8095	.9211	.8929
Overall	.7302 (.00)	.9027 (.00)	.9201 (.00)	.8466 (.00)	.9277 (.00)	.9579 (.00)

We compute the correlations between the model loss and their mean downstream scores calculated in Section 3. We show the results in Table 8, where we compute correlations for each model size and the final aggregated statistics. Our observation on the size-level correlations indicates that the task performance of 1B models is hard to align with the final loss, while bigger models tend to be much more correlated to both train and test loss. We explain the hypothesis that 1B models do not have enough capability to learn instructions. When aggregating the data points, we find that correlations are much stronger than the size-level prediction. The strong correlations imply that model loss on the general instruction data can work as a good proxy of downstream tasks in Code LLMs. When comparing the correlations on train loss to the test loss, we observe the correlations are stronger on the latter one. This can be explained by the fact that models tend to FFT on the training data, where the loss on the train split can not generalize well on the unseen tasks and data. Moreover, we also ask: *What is the relationship between the downstream task performance and the updated parameters?* Therefore, We investigate the correlation between tuned parameters and cumulative scores. The correlations are 0.3016 (.02), 0.4128 (.03) and 0.4138 (.03) for Kendall, Pearson and Spearman correlations, respectively. We draw the conclusion – *Possible*.

L Breakdown Results of Each Task

Based on Table 8, we also present the breakdown results of each downstream task. Interestingly, we observe that the cross-entropy loss is more correlated to overall downstream performance, compared to any individual code-specific tasks. The finding suggests that the cross-entropy of instruction tuning can reflect the comprehensive capability of Code LLMs.

Table 9: Correlations between final loss and Defect Detection performance. p-values are provided in gray.

Model Size	Train Loss			Test Loss		
	τ	r_p	r_s	τ	r_p	r_s
1B	-0.1429	-0.5728	-0.3571	-0.2381	-0.6089	-0.3929
3B	.6190	.8856	.7857	.3333	.8396	.5000
7B	.0476	.8040	.2857	.5238	.8782	.7143
16B	.5238	.8497	.6786	.6190	.7928	.7143
Overall	-0.1005 (.47)	-0.1394 (.48)	-0.1429 (.47)	-0.1217 (.38)	-0.2031 (.30)	-0.2074 (.29)

Table 10: Correlations between final loss and Clone Detection performance. p-values are provided in gray.

Model Size	Train Loss			Test Loss		
	τ	r_p	r_s	τ	r_p	r_s
1B	-0.3333	-0.6446	-0.3571	-0.2381	-0.6206	-0.3214
3B	-0.4286	-0.7587	-0.5357	.0476	-0.7293	.0000
7B	-0.3904	-0.6541	-0.5406	-0.3904	-0.6541	-0.5045
16B	.3333	.5725	.4286	.6190	.6900	.7500
Overall	-0.0452 (.74)	-0.1378 (.48)	-0.0942 (.63)	.0133 (.92)	-0.0965 (.63)	-0.0049 (.98)

Table 11: Correlations between final loss and Python Code Synthesis performance. p-values are provided in gray.

Model Size	Train Loss			Test Loss		
	τ	r_p	r_s	τ	r_p	r_s
1B	.1429	.4799	.1071	.4286	.5474	.6429
3B	-0.2381	.0568	-0.3214	.2381	.2300	.3571
7B	.1429	.1659	.1071	.6190	.3790	.7143
16B	-0.0476	-0.0567	-0.1429	.4286	.2544	.5357
Overall	.6402 (.00)	.8621 (.00)	.8314 (.00)	.7778 (.00)	.9134 (.00)	.9091 (.00)

Table 12: Correlations between final loss and Python Code Repair performance. p-values are provided in gray.

Model Size	Train Loss			Test Loss		
	τ	r_p	r_s	τ	r_p	r_s
1B	.2381	.7109	.3929	.4286	.5474	.6429
3B	.4286	-0.0824	.4643	.2381	.2300	.3571
7B	.4286	.3619	.6071	.6190	.3790	.7143
16B	.4286	.6983	.4286	.4286	.2544	.5357
Overall	.7354 (.00)	.8902 (.00)	.8933 (.00)	.7672 (.00)	.9182 (.00)	.9119 (.00)

Table 13: Correlations between final loss and Python Code Explanation performance. p-values are provided in gray.

Model Size	Train Loss			Test Loss		
	τ	r_p	r_s	τ	r_p	r_s
1B	.4286	.8526	.4643	.3333	.8828	.5000
3B	.3333	.9679	.5357	.6190	.9782	.7857
7B	.5238	.9569	.7143	.6190	.9658	.8214
16B	.3333	.9187	.4286	.6190	.9890	.7500
Overall	.6772 (.00)	.8576 (.00)	.8604 (.00)	.6667 (.00)	.8291 (.00)	.8380 (.00)

Table 14: Correlations between final loss and Java Code Synthesis performance. p-values are provided in gray.

Model Size	**Train Loss**			**Test Loss**		
	τ	r_p	r_s	τ	r_p	r_s
1B	-0.3333	-0.3385	-0.4286	-0.4286	-0.3917	-0.5000
3B	.3333	.1205	.2143	.6190	.2911	.7857
7B	-0.0476	.0164	-0.0714	.4286	.3270	.6429
16B	-0.0476	-0.2200	-0.1429	.4286	.0676	.5357
Overall	.6349 (.00)	.7552 (.00)	.8331 (.00)	.7407 (.00)	.8050 (.00)	.9015 (.00)

Table 15: Correlations between final loss and Java Code Repair performance. p-values are provided in gray.

Model Size	**Train Loss**			**Test Loss**		
	τ	r_p	r_s	τ	r_p	r_s
1B	.0976	.0725	.1441	.1952	.0954	.2162
3B	.2381	-0.0867	.1786	-0.2381	-0.2260	-0.2857
7B	.6190	.4203	.7857	.5238	.3140	.6429
16B	.5238	.7295	.4643	.8095	.8971	.9286
Overall	.7232 (.00)	.8011 (.00)	.8751 (.00)	.7550 (.00)	.8273 (.00)	.9136 (.00)

Table 16: Correlations between final loss and Java Code Explanation performance. p-values are provided in gray.

Model Size	**Train Loss**			**Test Loss**		
	τ	r_p	r_s	τ	r_p	r_s
1B	.2381	.7219	.3571	.5238	.7811	.6071
3B	-0.1429	.1024	-0.2143	.3333	.2680	.4643
7B	-0.6190	-0.9510	-0.7500	-0.1429	-0.8729	-0.3214
16B	.0476	.5829	.1429	.5238	.7734	.7143
Overall	.5536 (.00)	.8202 (.00)	.7374 (.00)	.6808 (.00)	.8760 (.00)	.8064 (.00)

M More Limitations and Future Work

Model Architecture Another limitation of our study is that we do not vary the model architecture of Code LLMs. It is possible that some findings may not generalize to other encoder-decoder Code LLMs like CodeT5 (Wang et al., 2021) and CodeT5+ (Wang et al., 2023b). However, as StarCoder is built upon the enhanced GPT-2 (Radford et al.) architecture, we believe that our observations can be transferred to other GPT-based LLMs.

Scaling Parameter-Constrained Language Models Although we demonstrate the possibility of predicting the final loss based on the updated parameters and vice versa, we note that a scaling law generally needs more than 100 models and their final loss. Ideally, the training experiments should be consistent with different PEFT strategies, meaning that training hundreds of models is needed. Furthermore, task performance is hard to predict, as there is much more noise in the downstream tasks than the final loss. We foresee that predicting such overall performance is very challenging.

N Prompts

The prompting format can significantly impact performance. In the spirit of true few-shot learning (Perez et al., 2021), we do not optimize prompts and go with the format provided by the respective model authors or the most intuitive format if none is provided. For each task not designed for evaluating instruction-tuned Code LLMs, we define an instruction. The instruction is to ensure that models behave correctly and that their outputs can be parsed effortlessly.

```
Question: {context}
Is there a defect in the Code, and respond to YES or NO.

Answer:
```

Figure 15: Prompt for Devign.

```
Question: Code 1: {context_1}
.
Code 2: {context_2}
Is there a clone relation between the Code1 and Code2, and respond to YES or NO.

Answer:
```

Figure 16: Prompt for BigCloneBench.

```
Question: {instruction}
{context}

Answer:
{function_start}
```

Figure 17: Prompt for HumanEvalPack.

Question: Create a Python script for this problem.

Answer: {function_start}

Figure 18: Prompt for Code Completion on ReCode.

Question: Create a script for this problem.

Answer: {function_start}

Figure 19: Prompt for Asleep At The Keyboard.