Exploring the Capabilities of LLMs for Code Change Related Tasks

LISHUI FAN*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JIAKUN LIU*, Singapore Management University, Singapore
ZHONGXIN LIU†, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
DAVID LO, Singapore Management University, Singapore
XIN XIA, Zhejiang University, China
SHANPING LI, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

Developers deal with code-change-related tasks daily, e.g., reviewing code. Pre-trained code and code-change-oriented models have been adapted to help developers with such tasks. Recently, large language models (LLMs) have shown their effectiveness in code-related tasks. However, existing LLMs for code focus on general code syntax and semantics rather than the differences between two code versions. Thus, it is an open question how LLMs perform on code-change-related tasks.

To answer this question, we conduct an empirical study using >1B parameters LLMs on three code-change-related tasks, i.e., code review generation, commit message generation, and just-in-time comment update, with in-context learning (ICL) and parameter-efficient fine-tuning (PEFT, including LoRA and prefix-tuning). We observe that the performance of LLMs is poor without examples and generally improves with examples, but more examples do not always lead to better performance. LLMs tuned with LoRA have comparable performance to the state-of-the-art small pre-trained models. Larger models are not always better, but Llama 2 and Code Llama families are always the best. The best LLMs outperform small pre-trained models on the code changes that only modify comments and perform comparably on other code changes. We suggest future work should focus more on guiding LLMs to learn the knowledge specific to the changes related to code rather than comments for code-change-related tasks.

CCS Concepts: • **Software and its engineering → Software development techniques**; **Software maintenance tools**.

Additional Key Words and Phrases: Code-change-related task, large language model, empirical study

1 INTRODUCTION

After the launch of a project, developers constantly change code to introduce new features and maintain existing code (e.g., performing refactoring and fixing bugs) [12, 16, 18]. A code change contains the added, deleted, or modified (deleted then added) code span across one or more files and is often expressed in a combination of the

*Both authors contributed equally to the paper
†Corresponding author

Authors' addresses: Lishui Fan, flscode@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China; Jiakun Liu, jkliu@smu.edu.sg, Singapore Management University, Singapore; Zhongxin Liu, liu_zx@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore; Xin Xia, xin.xia@acm.org, Zhejiang University, China; Shanping Li, shan@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China.

ACM 1557-7392/2024/12-ART
https://doi.org/10.1145/3709358

code versions before and after the change, or in plain text such as "diff". Developers need to handle many code-change-related tasks due to the ubiquitous nature of code changes. For example, in their daily work, developers need to understand existing code changes in code repositories [20], update the comments accompanied with the changed code [26], write commit messages [60], and review the code changed by other developers [39]. These code-change-related tasks are important for project maintenance but can cost significant efforts and slow down the development process [4, 52]. Therefore, it is necessary to automate or provide tool support for them [52].

To deal with code-change-related tasks, prior studies have proposed a series of machine-learning-based [35, 37, 80] and deep-learning-based approaches [28, 51, 62]. Recently, researchers proposed to leverage pre-trained code models or pre-trained code-change-oriented models to tackle code-change-related tasks and achieved state-of-the-art performance [47, 50, 107]. For example, CCT5 [50], pre-trained based on CodeT5 [93] using 1.5M code change samples with five code-change-oriented pre-training objectives, has achieved the state-of-the-art performance on diverse code-change-related tasks. Recent studies have shown that by significantly increasing the model size and expanding the pre-training data, pre-trained models can be more powerful and can demonstrate various emergent abilities [94]. For example, the Bloom model [71] contains 176B parameters, 789 times more than CCT5 (223M), is pre-trained on 363B natural language tokens, and significantly outperforms all the models with less than 1B parameters on natural language processing tasks. The difference in model size and training data may make Bloom (or other similar models) perform better than small pre-trained models.

Recent work has pre-trained several >1B parameters large language models (LLMs) with massive code corpora for code-related tasks [45, 66, 82][1]. For example, Meta released the Code Llama models [82], which are initialized from the Llama 2 models [89] and trained on 500B tokens from a code-heavy dataset. However, these LLMs may not perform well for code-change-related tasks. This can be the case as they are pre-trained with massive code snippets to learn the general syntactic and semantic knowledge of code, while code changes are more about the differences between two code snippets. Although we can represent a code change as a diff or other forms to help LLMs distinguish the changed parts from the context, LLMs are not pre-trained with the data of such formats. Therefore, It is still an open question whether using >1B parameters LLMs can effectively boost code-change-related tasks. (as compared to smaller <1B parameters LLMs). To the best of our knowledge, there is a lack of an in-depth investigation of LLMs for code-change-related tasks. In this work, we would like to fill this gap and investigate: **How do LLMs perform on code-change-related tasks?**

To answer this question, we select representative and popular >1B parameters LLMs including InCoder [19], CodeGen [73], Llama 2 [89], and Code Llama [82]. We consider three emerging code-change-related tasks: code review generation [17], commit message generation [6], and just-in-time comment update [62]. We start the exploration of LLMs from prompt engineering. Prompt engineering is the most convenient way to apply LLMs because it does not change model parameters. In the realm of prompt engineering, In-Context Learning (ICL) is recognized as one of the most typical and effective approaches [22, 43, 68]. It is a technique that formulates input by integrating task descriptions, exemplars, and query problems, and subsequently instructs LLMs to generate predictions. To this end, we first would like to answer:

RQ1: How do LLMs perform when applying In-Context Learning on code-change-related tasks?

Hereon, we refer to LLMs with ICL as LLM-ICLs. To answer this RQ, we apply LLM-ICLs with different numbers of examples on code-change-related tasks. We find the performance of LLMs is poor without examples. With one example provided, the performance of LLMs generally drastically improves. However, more examples do not always lead to better performance. The effectiveness of LLMs depends on the data lengths in the task and the context length allocated to the model. Besides, larger models do not always have better performance, but models in Code Llama family always perform the best in the selected tasks related to code changes.

[1]For ease of explanation, we refer to >1B parameters LLMs as LLMs, and those <1B parameters LLMs as small pre-trained models.

To further explore the capabilities of LLMs on code-change-related tasks, we allow updating LLM parameters for code-change-related tasks. Parameter-Efficient Fine-tuning Techniques (PEFT) are currently the most common technique for updating LLM parameters. It focuses on updating a few parameters while freezing the rest, allowing the model to efficiently adapt to different tasks. To this end, we summarize our second research question as:

RQ2: How do LLMs perform when applying Parameter-Efficient Fine-tuning Techniques on code-change-related tasks?

Hereon, we refer to LLMs tuned using PEFT as LLM-PEFTs. We explore the capabilities of two most common and popular PEFT methods, i.e., LoRA [29] and prefix-tuning [46]. LoRA can directly change the parameters of LLMs by optimizing the low-rank decomposition of LLMs' self-attention modules. Prefix-tuning prepends a sequence of continuous trainable vectors to the input and the hidden states of each transformer layer. We observe that LLMs tuned with LoRA results in significantly better performance compared to LLMs tuned using prefix-tuning. Similarly, we also find that larger models do not necessarily have better performance, even within the same LLM family. We also observe that the Llama 2 and Code Llama families are the best-performing LLMs across the three code-change-related tasks.

Additionally, to help developers with the tasks related to code changes, previous researchers have proposed a series of tools based on small pre-trained models. We would like to further understand:

RQ3: How do LLMs perform on code-change-related tasks compared to small pre-trained models?

To answer this RQ, we compare the performance of LLMs to that of small pre-trained models, i.e., CodeT5 [93] (the small models pre-trained with code) and CCT5 [50] (the small models pre-trained with code changes), on the selected code-change-related tasks. We observe that LLM-ICLs are similar to or better than CodeT5 on the tasks related to code changes, but inferior to CCT5 . With parameter updates, LLM-PEFTs outperform LLM-ICLs across tasks. Even though LLM-PEFTs have fewer parameters updated compared to the total number of parameters of small pre-trained models, LLM-PEFTs can still achieve comparable performance to CCT5.

We notice that the code change can be presented in two different formats: in the diff format, or in two consecutive code snippets corresponding to the code before and after the change. Intuitively, the input to an LLM may affect the performance of the LLM. Therefore, we aim to explore:

RQ4: How do LLMs perform with different input formats on code-change-related tasks?

To answer this RQ, we compare the performance of LLMs with different input formats on the selected code-change-related tasks. We observe that LLM-ICLs perform better when the input is in the "diff" format on the just-in-time comment update task. For LLM-PEFTs, we find no significant difference among the three tasks with different input formats. This may be because the updated parameters in the LLM have learned to compare the differences between two pieces of code and capture the patterns of the changed parts.

Finally, to further understand the better performance of LLM-PEFTs, we would like to explore:

RQ5: When do LLMs perform better?

We select the best-performing LLMs from the previous RQs and characterize their performance on different types of code changes. We observe that LLM-PEFTs generally outperform LLM-ICLs on all types of code changes, indicating that compared to ICL, PEFT can comprehensively improve the performance of LLMs on all code change categories. LLM-PEFTs outperform the fully fine-tuned small models on the code changes that only revised documents in code (i.e., code comment), and on other code changes that perform code featuring code refactoring or modify both code and documentation, LLM-PEFTs perform comparably to fully fine-tuned small models.

We also conduct a human evaluation on commit message generation to further understand the effectiveness of LLMs. The results show that the commit messages generated by LLM-PEFT are the most expressive and concise without losing much adequacy.

Our study illuminates the opportunities presented by LLMs, necessitating further investigations into their application in code-change-related tasks. For example, we find LLMs tend to learn more knowledge related to

documentation changes. We should focus more on guiding LLMs to learn the knowledge specific to the changes in code, such as the knowledge of refactoring, when tuning LLMs for code-change-related tasks.

In summary, this paper makes the following contributions:

- To the best of our knowledge, this paper presents the first comprehensive empirical study on the capabilities of LLMs in code-change-related tasks. We conduct experiments using a broad range of LLMs on three code-change-related tasks with different techniques and different input formats.
- Our comprehensive exploration and analysis highlight findings about how to apply LLMs to code-change-related tasks for different code-change types in different scenarios.
- We discuss the implications of our findings and demonstrate the future work for code-change-related tasks in the era of LLMs.

2 STUDY DESIGN

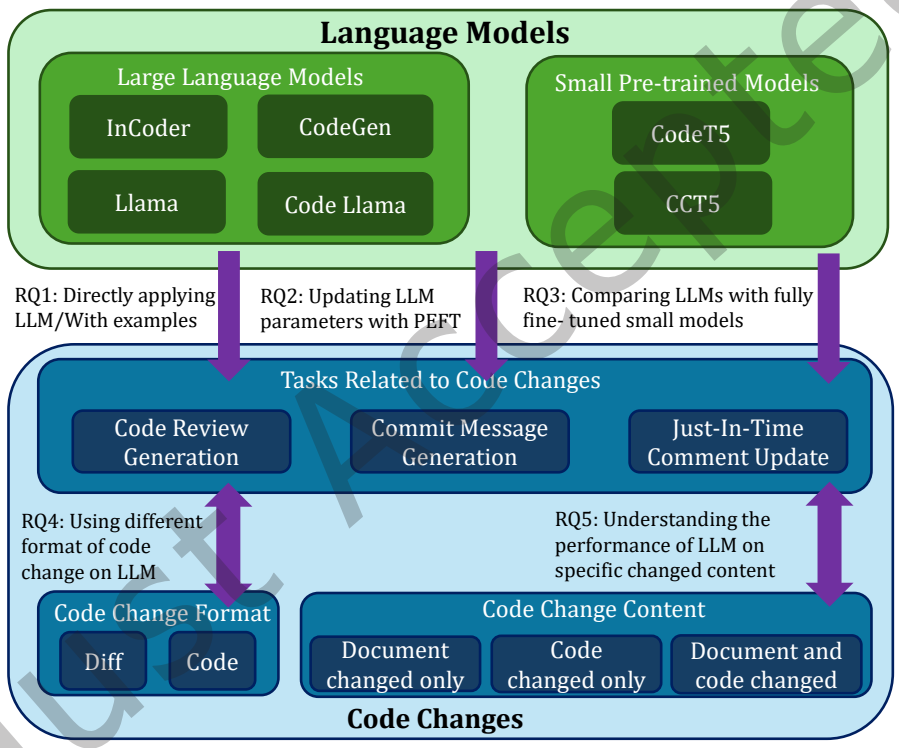


Fig. 1. The overall framework of this study.

Figure 1 presents the overview of our study. We first present the selected code-changes-related tasks in Section 2.1. Then, in Section 2.2, we present the state-of-the-art LLMs and small pre-trained models. We present the approaches we would like to explore to utilize these models (to answer RQ1, RQ2, and RQ3) in Section 2.3. Following that, we present the design of input in Section 2.4 (to answer RQ4) and the design of analyzing the impact of code changes types in Section 2.5 (to answer RQ5). Finally, we present the implementation of the work in Section 2.6.

2.1 Task, Dataset and Evaluation Metrics

To perform our empirical study, we consider three emerging code-change-related tasks that have been popularly researched in recent years, namely, code review generation [47, 84], commit message generation [15, 57], and just-in-time comment update [74, 110].

2.1.1 Code Review Generation (CRG). The primary goal of CRG is to automatically generate reviewers' suggestions from code changes to provide immediate high-quality feedback to developers when they commit to the version control system. This task takes as input a code change included in the commit and generates the corresponding code review comment as output.

Datasets. Following prior studies [50, 58], we use the dataset for the CRG task constructed by Li et al. [47]. They collected a total of 138,127 diff-review pairs from popular open-source projects on GitHub written in nine different programming languages. Each pair consists of a real-world code change along with the corresponding code review comment.

Metrics. To evaluate the generated text, following Li et al.[47]'s work, we utilize the BLEU-4 metric. BLEU-4 computes the overlap of n-grams between the generated and the reference texts, where n ranges from 1 to 4.

2.1.2 Commit Message Generation (CMG). The primary goal of CMG is to automatically produce a concise natural language description to summarize the content and intention of a commit submitted to the version control system. The task takes as input the code change in the commit and generates the corresponding code commit message as output.

Dataset. Following prior studies [50, 83], we use the Multi-programming language Commit Message Dataset (MCMD) constructed by Tao et al. [86]. They considered 5 programming languages, i.e., Python, Java, JavaScript, C#, and C++, and collected the top 100 starred projects in each programming language from GitHub respectively (500 projects in total).[2] Then, for each programming language, they randomly sampled 450,000 commits as well as their corresponding commit messages from the collected projects (2,250,000 commits in total). Note that the original MCMD dataset only provides the tokenized source code data. For example, the code snippet *this.indices.limit(indices.length);* has been tokenized into *this . indices . limit (indices . length) ;*. Considering that the input format can impact the performance of LLMs and different models may use different tokenizers, we used regular expressions to restore the data in MCMD by removing additional spaces. For example, *this . indices . limit (indices . length) ;* will be converted back to *this.indices.limit(indices.length);*. Each model will be provided with the de-tokenized data and will use its own tokenizer. It is worth noting that we only changed the format of the input, while the content of the input remained unchanged.

Metrics. Following prior studies [64, 86], we use B-Norm as the evaluation metric. B-Norm is a variant of BLEU [75], which assesses the lexical overlap between the generated text and the label. Prior work has shown that B-Norm demonstrates the highest correlation with human evaluations on CMG [86]. For convenience, on the CMG task, we also refer to B-Norm as BLEU.

2.1.3 Just-In-Time Comment Update (JITCU). The primary goal of just-in-time (JIT) comment update is to automatically update comments after code changes, which can avert outdated comments and boost the maintainability of software. This task takes as input a code change and the comment associated with the modified code in the before-change version and generates the updated comments after the change.

Dataset. Following prior studies[49, 50], we use the dataset that was first constructed by Liu et al. [63] and then further cleaned by Lin et al. [49] for JITCU. This dataset contains a total of 98,622 comment update instances collected from 1,496 high-quality Java projects on GitHub. Each data entry comprises co-changes between methods and header comments.

[2]In this paper, we refer to them as CMG-Python, CMG-Java, CMG-JavaScript, CMG-C#, and CMG-C++ respectively.

Metrics. Following prior studies [50, 61, 107], we use GLUE and ACC as the evaluation metrics. GLEU is similar to BLEU, and is widely used to evaluate text-editing systems. Additionally, we also use Accuracy (shortened as ACC), which is computed as the percentage of the test instances where the generated comments are the same as the ground truth.

Following prior studies [38, 62], we also use paired bootstrap resampling with 1000 resamples [38] to perform statistical significance tests for each evaluation metric.

2.2 Small Pre-Trained Models and Large Language Models

To understand the effectiveness of LLMs on code-change-related tasks, we plan to compare them with the state-of-the-art (SOTA) approaches that can be applied to diverse code-change-related tasks. These approaches are all pre-trained models with fewer than 1B parameters [50, 60, 94]. Thus we refer to them as small pre-trained models.

For small pre-trained models, we select the state-of-the-art models on code-related tasks and code-change-related tasks, namely CodeT5 [93] and CCT5 [50], in our experiments.

CodeT5 is an encoder-decoder model. It is pre-trained with the code data from CodeSearchNet [31] and the C and CSharp code data from BigQuery[3] using four pre-training tasks, such as Identifier Tagging and Masked Identifier Prediction. Existing code-change-oriented pre-trained models [47, 50, 107] are all built upon CodeT5. Following prior studies [2, 47, 107], we use the base model with 223M parameters.

CCT5 is the state-of-the-art code-change-oriented pre-trained models. It is pre-trained with 39.6 GB of code change data collected from 35K popular GitHub repositories in six programming languages. Five code-change-oriented tasks are used for pre-training, including masked language modeling for code change, code diff generation, code diff to natural language generation, etc.

For LLMs, following prior work [95], we use the following criteria:

- We select open-source LLMs and exclude closed-source LLMs. Because the parameters of closed-source LLMs are inaccessible, making the investigation of fine-tuning techniques unfeasible or expensive.
- We select state-of-the-art LLMs in the software engineering field, especially those released recently at the time we conducted this study (i.e., September 2023)..
- We select a model family in the general domain, facilitating the comparison between the LLMs in the code and general domains
- We select models across different parameter sizes to study the implications of parameter sizes.

Consequently, we select four LLM families, CodeGen [73], InCoder [19], Code Llama [82] and Llama 2 [89].

Table 1. Small pre-trained models and LLM families included in our study. For the models reported with two parameter sizes, we use both of them in our work.

Models		Dataset	Parameters	Domain
Small PT Models	CodeT5	CodeSearchNet	223M	Code
	CCT5	CodeChangeNet	223M	Code Change
LLMs	InCoder	-	1.3B	Code
	CodeGen	The Pile / BigQuery	2B/6B	Code
	Llama 2	-	7B/13B	General
	Code Llama	-	7B/13B	Code

[3]https://console.cloud.google.com/ marketplace/details/github/github-repos

InCoder models are built upon XGLM [53] and is pre-trained using the infilling task. We utilize InCoder-1.3B in our experiments.

CodeGen models are built upon GPT-Neo [3] and GPT-J [91], and is pre-trained using the autoregressive language modeling task. In our experiments, we choose CodeGen-Multi-2B and CodeGen-Multi-6B, which are pre-trained on the code data in a wide range of programming languages collected from BigQuery.

Llama 2 is pre-trained on 2 trillion tokens of data using the autoregressive language modeling task. In our experiments, considering the computation cost, we employ both Llama 2-7B and Llama 2-13B in our experiments.

Code Llama family are based upon Llama 2. It is pre-trained on a 500B token corpus with both code and natural language texts using the infilling objective. Similar to Llama 2, in our experiments, we employ both Code Llama-7B and Code Llama-13B in our experiments. We exclude the variants that are further trained on Python corpus (too specific) or instruction database (different capability) as they are not suitable for our task.

2.3 Techniques to apply Pre-Trained Models

2.3.1 Techniques to apply LLMs. To understand the capability of LLMs on code-change-related tasks, we first explore prompt engineering. This is because prompt engineering does not require updating model parameters, and it is convenient and cheap. Then, we explore the techniques of changing parts of model parameters, namely parameter-efficient fine-tuning (PEFT) techniques. Due to computational resource limitations, we do not explore the method of changing all model parameters, namely full fine-tuning, for LLMs.

Prompt Engineering design proper prompts to guide the pre-trained model to perform a specific downstream task. We choose the most popular technique, i.e., in-context learning (ICL), in prompt engineering. ICL is a technique that guides LLMs to generate contextually appropriate content without updating parameters by providing examples or demonstrations of the task in the prompt. Prior studies have demonstrated the effectiveness of LLM-ICLs on domain-specific tasks [22, 43, 68].

We explore the capability of ICL on code-change-related tasks by varying the number of similar examples, denoted as n, where $n \in 0, 1, 2, 3, 4$. $n = 0$ corresponds to the zero-shot scenario [5], enabling us to explore the capability of LLMs without any prior knowledge of the task. For each test sample, we employ the BM25 method to retrieve similar samples from the training set as examples. BM25, a well-established information retrieval technique, has been shown to perform well to retrieve demonstrations for ICL by previous studies [21, 68].

Parameter-efficient fine-tuning (PEFT) adapts LLMs to downstream tasks by updating a minimal subset of model parameters, either inherent or newly added to the model, with task-specific datasets. We choose two widely used PEFT techniques: LoRA and prefix-tuning. Both techniques are popular due to their effectiveness and efficiency on code-related tasks, such as code generation [27, 54, 95], code completion [8, 76, 97], and code summarization [40, 92, 98].

- ***LoRA*** is proposed to update partial parameters of LLMs by optimizing the low-rank decomposition of the attention module's matrices. When using LoRA, we need to configure two hyper-parameters γ and α, where γ is the rank of the update matrices and α is a scaling factor that helps stabilize the training. Following prior work [29], we consider $\gamma = 8$ and $\alpha = 16$.
- ***Prefix-tuning*** wraps the input with additional context by prepending trainable continuous vectors (prefixes) to the input and the hidden states of each transformer layer. When using prefix-tuning, we need to configure the number of trainable vectors n. Following previous work [46], we set $n = 8$.

2.3.2 Techniques to apply small pre-trained models. Since small pre-trained models are used to help understand the performance of LLMs, we would like to apply them with the technique that can get their best performance. We do not explore the performance of small pre-trained models with ICL, because it is hard for small pre-trained models to comply with instructional prompts without parameter updates [70, 79]. Prior work has shown that the performance improvements brought by PEFT are often not as large as those brought by full fine-tuning [48, 99],

Table 2. An overview of the input design in code change tasks under two different input formats. The *{input_tokens}* are different in the diff input format and code input format.

Tasks	Input Design(diff/code)
CRG	### Instruction: Please write a code review according to the (diff hunk/code before and after the diff hunk). {input_tokens} ### Answer:
CMG	### Instruction: Please write a commit message according to the (diff hunk/code before and after the diff hunk). {input_tokens} ### Answer:
JITCU	### Instruction: Please write a new comment according to the original comment and the (diff hunk/code before and after the diff hunk). {input_tokens} ### Answer:

and full fine-tuning is affordable and common [41, 50, 93] for small pre-trained models. Therefore, we apply small pre-trained models with full fine-tuning.

Full-Parameter Fine-tuning. This technique updates all the parameters of a model for one or more tasks. We fine-tune the small pre-trained models, i.e., CodeT5 and CCT5, using full-parameter fine-tuning to achieve the best performance.

2.4 Input Design

Table 2 shows the prompt templates for the selected tasks. Each template contains "### Instruction:", the description of the task, examples, and the code change needed to process. Finally, we use "### Answer:" to introduce the response, such as a commit message, code comment, or code review comment. Specifically, each example is composed of a code change and the corresponding expected response (i.e., the ground truth appended after "### Answer:"). Additionally, for the JITCU task, where each input contains an original comment, we add "original comment message:\n" to the end of the code change. For the techniques that do not need examples (LLM-ICL without example and PEFT), there is no example in the prompt. Note that while the format of these prompt templates may not be optimal, its effectiveness has been demonstrated in prior studies [77, 78].

To investigate the impact of different formats of code change on the model performance, we explore the code changes presented in the diff format or in the code format. Specifically, the diff format highlights the lines of code that have been added or removed with "+" or "-" at the beginning of the line, respectively; while the code format puts code snippets before and after the change together. To use them in LLM, for diff format, we add the "diff hunk:\n" before the diff text; for code format, we add "code change before:\n" and "code change after:\n" before the code snippet before and after the change, respectively, and then connect the two parts with "\n".

2.5 Analyzing the Impact of Code Change Types

To understand the performance of different models on different types of code changes, for each code-change-related task, we select the best-performing models from small pre-trained models, LLM-ICLs, and LLM-PEFTs. Then we randomly selected 592 samples from the test datasets with a 90% confidence level and a 5.6% confidence interval. The selected dataset contains 196 samples from CRG, 196 samples from JITCU, and 200 samples from CMG (40 for each language). Two authors manually annotated the change category of each sample independently. We use the taxonomy of code changes proposed in prior studies [25, 90] as a starting point and refine the taxonomy based on our observation. Specifically, Tufano et al. [90] categorize code changes into two types: Refactoring and Behavioral changes. Refactoring changes are less likely to alter code functionality, whereas behavioral changes directly impact code behavior. Guo et al. [25], based on these two categories and their annotation results, classify code changes into four categories: Documentation, Feature, Refactoring, and Documentation-and-Code. Considering both Feature and Refactoring categories only involve code modifications, We refine Guo et al.'s [25] taxonomies by merging these two categories as a new category named Code-Only. Merging them leads to a more concise taxonomy. As a result, our taxonomy consists of three categories of code changes, namely Doc-only, Code-only, and Doc-and-Code.

- **Doc-only code change** represents the code changes that only add, modify or remove documentation.
- **Code-only code change** represents the code changes which only modify code entities. Furthermore, this category can be divided into two subcategories: ① **Feature code change** represents the code-only code changes where the functional logic is modified. ② **Refactor code change** refers to the code-only code changes that perform code refactoring, including renaming code entities, swapping two code snippets, and updating code based on coding standards.
- **Doc-and-Code code change** represents the code changes that include both documentation and code modifications.

After two authors independently annotated the selected samples, a discussion was held to solve the disagreements. We did not invite others because all the disagreements were resolved during the discussion. The final Cohen's Kappa coefficient [67], which is used to assess the inter-rater agreements, was 0.758. Finally, we report the performance of different models on different types of code changes.

2.6 Implementation

Our implementation is based on the Huggingface[4] library. Specifically, we use this library to download the models and their tokenizers and to conduct all experiments in our paper. Note that we do not apply prefix-tuning to the InCoder model because there is an implementation issue with adding the virtual tokens of prefix-tuning to the base model of InCoder in the Huggingface library. Such an issue has also been mentioned by other developers[5] and researchers [95].

Following prior studies [7, 95], we use half-precision for LLMs to fit them into our GPU, and full precision for the small pre-trained models to ensure their performance. To make a fair comparison between models, following the prior studies [21, 81, 100, 103, 105] that considered the non-trivial costs of fine-tuning LLMs, we sample 16,000, 2,000 and 2,000 samples from the original dataset (or each sub-dataset in MCMD [86]) as the training, validation, and test sets for each task. Note that prior studies [50, 83] split training, validation, and test sets from the whole dataset to fully explore the performance of the small model; considering the different sizes of the training data, our experimental results can be different from theirs [50, 83].

For each task, we use a maximum length of 1024 for the input of each sample. Because we find the input lengths of over 98% of the samples in the training sets of the three tasks are less than 1024. Particularly, we allow

[4]https://huggingface.co/

[5]https://github.com/huggingface/peft/issues/811

ICL to use another 1024 tokens to provide examples in the prompt. In detail, when the number of examples is n, the maximum length for each example is set to $\frac{1024}{n}$ (n >0). When the input length exceeds the maximum length, we proportionally truncate the code snippets before and after the change at the same time or truncate the diff, depending on the input format. For JITCU, if the input length exceeds the maximum length, we prioritize truncating the code or diff while preserving the integrity of the original comments. For output, we consider a maximum length of 100 tokens. This is because for over 97% of the examples in the training sets of the three tasks, their reference output has less than 100 tokens.

3 EXPERIMENTAL RESULTS

Here, we present the results of the experiments to answer the five research questions.

3.1 RQ1: How Do LLMs Perform When Applying In-Context Learning on Code-Change-Related Tasks?

Table 3. [RQ1] - BLEU scores of LLM-ICLs on CRG. The darker color of the cells means better performance.

Models	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
InCoder-1b	1.44	3.22	3.5	3.47	3.19	3.01	2.88	2.78	2.8
CodeGen-2b-nl	0.16	0.22	0.95	1.23	1.38	1.45	1.51	1.58	1.6
CodeGen-6b-nl	0.56	0.76	0.93	1.18	1.34	1.34	1.6	1.6	2.0
Llama 2-7b	0.65	4.4	4.42	4.55	4.49	4.72	4.79	4.91	4.88
Llama 2-13b	1.72	4.25	4.42	4.62	4.68	4.72	4.73	4.89	5.0
Code Llama-7b	0.9	4.55	4.74	4.83	5.02	4.97	5.03	5.04	4.93
Code Llama-13b	2.27	4.6	4.65	4.79	4.8	4.86	4.79	4.95	4.94

Table 3, 4 and Table 5 show the performance of different LLMs with different settings on the selected code-change-related tasks. Due to space limitations, for CMG, we only present the experiment results on the Python and Java datasets. This is because Python and Java are the most popular programming languages to date. We present all the experiment results of LLM-ICLs on CMG in Appendix A.

3.1.1 Ability of Directly Applying LLM. We observe that the performance of the LLMs is poor without examples across LLMs and tasks. One possible reason is that code changes are different from the pre-training data (e.g., code) of LLMs, indicating that LLMs do not have the knowledge related to code change. With one example provided, the performance of LLMs generally drastically improves. This indicates that examples in prompt can generally improve the performance of LLMs and the ability to understand code changes can be stimulated via examples. However, InCoder-1b experienced performance fluctuations after increasing the number of examples. The reason may be that InCoder-1b has relatively fewer parameters, and it is still challenging for it to understand and capture the content and associations of multiple examples in the input [36].

Finding 1: LLMs lack the knowledge specific to code changes. Providing examples in the prompt can generally improve their performance on the tasks related to code changes.

Table 4. [RQ1] - BLEU scores of LLM-ICLs on Python and Java sub-datasets of CMG. The darker color of the cells means better performance.

Lang	Model	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
Java	InCoder-1b	3.59	2.97	2.8	2.91	2.58	2.86	2.79	3	2.71
	CodeGen-2b-nl	1.9	2.21	2.41	2.54	2.41	2.34	2.21	2.09	2.26
	CodeGen-6b-nl	2.1	2.87	3.28	3.17	3.13	3.07	3.09	3.16	3.11
	Llama 2-7b	2.23	3.48	3.63	3.95	3.97	4.08	4.16	4.04	4.18
	Llama 2-13b	2.42	3.68	3.8	4.58	4.84	5.39	5.75	5.65	6.00
	Code Llama-7b	1.78	5.55	5.29	5.32	5.38	5.37	6.07	6.19	6.12
	Code Llama-13b	2.68	4.44	3.76	3.97	4.01	4.12	4.28	4.98	4.74
Python	InCoder-1b	5.11	3.97	4.12	4.28	4.32	4.3	4.48	3.98	4.19
	CodeGen-2b-nl	2.58	3.23	3.24	3.28	3.26	3.27	3.21	3.24	3.44
	CodeGen-6b-nl	3.4	4.19	4.71	4.43	4.34	4.19	4.24	4.35	4.35
	Llama 2-7b	3.53	4.41	4.59	5.58	6.19	6.37	6.3	6.05	6.33
	Llama 2-13b	4.41	5.12	6.15	6.75	7.17	7.36	7.52	7.28	7.63
	Code Llama-7b	1.51	6.19	7.28	7.79	8.36	8.39	8.13	8.18	8.07
	Code Llama-13b	3.63	5.26	6.42	7.28	7.7	7.59	6.95	7.3	6.29

Table 5. [RQ1] - Performance of LLM-ICLs on JITCU. We report the GLEU and ACC. The darker color of the cells means better performance.

Model	metric	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
InCoder-1b	GLEU	6.54	11.94	21.92	28.93	27.55	28.13	26.27	26.47	18.41
CodeGen-2b-nl		0.00	0.06	1.18	1.20	1.44	1.86	2.30	2.01	1.95
CodeGen-6b-nl		0.00	2.68	5.68	13.18	13.26	15.85	16.33	17.36	17.05
Llama 2-7b		4.90	51.49	55.55	44.52	44.24	44.92	39.35	37.29	38.51
Llama 2-13b		8.46	53.18	58.68	47.30	47.39	46.06	41.25	40.76	41.20
Code Llama-7b		0.03	56.59	59.87	47.91	48.01	47.72	44.06	42.40	41.05
Code Llama-13b		0.60	52.62	54.90	44.32	43.04	45.48	39.97	37.93	36.02
InCoder-1b	ACC	0.60	1.80	3.10	3.80	3.65	3.95	4.35	4.30	2.90
CodeGen-2b-nl		0.00	0.00	0.25	0.20	0.15	0.30	0.40	0.35	0.25
CodeGen-6b-nl		0.00	0.10	0.40	1.55	1.65	1.70	1.35	1.70	1.25
Llama 2-7b		0.45	14.25	17.85	13.65	13.20	12.05	8.15	7.40	8.00
Llama 2-13b		3.05	16.20	20.50	17.30	16.90	15.85	12.10	11.10	10.60
Code Llama-7b		0.05	21.55	23.65	18.95	20.00	19.30	15.15	14.35	13.30
Code Llama-13b		0.50	18.85	19.65	15.95	15.95	16.95	10.25	9.70	9.20

3.1.2 Impact of Different Numbers of Examples. We observe that as the number of examples in prompt increases, the performance of LLMs increases and then decreases across LLMs and tasks under the condition of limited input length. For instance, on JITCU, the GLEU/ACC scores of Llama 2/Code Llama increase at first, until the number of examples in the prompt is 2 and then decrease after adding more examples. On CRG, the BLEU scores of Code Llama-7b achieve the highest 5.04 when the number of examples in the prompt is 7. This may result from the limitation of input context length. As indicated in Section 2.6, we allocate 1024 tokens for the examples in the prompt. When there are too many examples in tasks, we often need to truncate the length of each example in tasks due to the limited input length of LLMs. For example, after being tokenized by the tokenizer of Code Llama, the average length of the samples in the training sets has more than 211/184 tokens on the CRG and JITCU tasks, respectively. With more examples, we need to truncate the examples by removing more tokens from the examples. This hinders LLMs from understanding the information from each example and can result in a drop in performance. And, due to the varying length distributions of data across different tasks, the number of examples required for a model to achieve the best performance varies for different tasks.

> **Finding 2**: More examples do not always lead to better performance. The effectiveness of LLMs often depends on the distribution of data lengths in the task and the context length allocated to the model.

3.1.3 Impact of Model Size. Prior studies showed that within the same LLM family, larger models are associated with better performance [73, 82, 89]. This also applies to some extent to tasks related to code change. For example, in the Llama 2 and CodeGen families, better performance is achieved by larger models. One possible reason is that the larger models can have a broader understanding and capacity to integrate more context effectively due to the vast number of parameters, thus performing better in these families. **However, in the Code Llama family, better performance on three tasks is achieved by smaller models.** A possible reason can be that these smaller models might have just the right capacity to capture the essential patterns without being bogged down by excessive complexity [87, 88]. Therefore, the observed performance differences highlight that model size and performance can be task-specific and that bigger is not always better when it comes to using ICL within different LLM families.

> **Finding 3**: Within the same LLM family, larger models do not always have better performance.

3.1.4 Impact of Model Family. We also find that the **Code Llama family performs the best across code-change-related tasks compared to other model families**. This may be because Code Llama is based on Llama 2 and has been pre-trained on a large amount of (1) code data and (2) code-related natural language data, enabling it to better understand the information in code and natural language at the same time. For instance, **on CRG, CMG-Java, CMG-Python, and JITCU, the best LLM-ICLs are all Code Llama-7b.** Code Llama-7b demonstrates the strongest learning capabilities. As the number of examples increases, the performance of Code Llama-7b improves the most on average. For example, on the CMG-Java, the best Code Llama-7b with examples outperforms its 0 shot setting by 3.47 times. In comparison, Llama 2-13b outperforms its 0 shot setting by 2.48 times. This suggests that although the model may struggle to comprehend the task without example, Code Llama-7b can rapidly learn and capture task-relevant features when provided with examples. These findings highlight the advantage of Code Llama-7b in adapting to new tasks. **When applying LLMs to new code-change-related tasks, we recommend using Code Llama family, especially Code Llama-7b.**

> **Finding 4**: The Code Llama family, especially Code Llama-7b, performs the best in the selected tasks related to code changes.

3.2 RQ2: How Do LLMs Perform When Applying Parameter-Efficient Fine-Tuning Techniques on Code-Change-Related Tasks?

Table 6. [RQ2] - Performance of LLM-PEFTs on the CRG and JITCU tasks. The darker color of the cells means better performance.

Model	CRG		JITCU			
	BLEU-4		GLEU		ACC	
	LORA	Prefix	LORA	Prefix	LORA	Prefix
InCoder-1b	2.50	-	38.23	-	12.10	-
CodeGen-2b-nl	0.27	0.61	0.26	1.13	0.55	0.05
CodeGen-6b-nl	1.11	0.87	4.00	29.36	1.15	3.35
Llama 2-7b	5.74	1.12	63.30	0.25	32.25	0.00
Llama 2-13b	5.29	2.42	61.81	0.00	32.45	0.00
Code Llama-7b	4.04	0.33	65.23	0.06	34.40	0.00
Code Llama-13b	5.61	1.17	64.22	0.27	34.90	0.00

Table 7. [RQ2] - Performance of LLM-PEFTs on the CMG task. The darker color of the cells means better performance.

Models	Java		C#		CPP		Python		JavaScript	
	LORA	Prefix	LORA	Prefix	LORA	Prefix	LORA	Prefix	LORA	Prefix
InCoder-1b	5.13	-	4.95	-	5.54	-	5.57	-	5.47	-
CodeGen-2b-nl	2.23	1.87	3.27	2.71	3.37	3.60	3.77	3.52	3.55	3.89
CodeGen-6b-nl	3.65	2.78	4.77	2.62	4.94	3.08	5.27	4.81	5.14	3.59
Llama 2-7b	12.30	1.77	11.97	1.05	11.02	0.13	13.35	0.05	13.84	0.03
Llama 2-13b	11.73	0.72	11.57	1.00	10.31	0.69	12.85	0.84	13.39	1.03
Code Llama-7b	13.06	1.12	12.05	1.20	11.01	0.89	13.37	0.61	13.79	0.85
Code Llama-13b	11.93	1.68	12.12	2.98	11.77	2.51	12.92	3.93	13.87	1.60

Table 6 and Table 7 show the results of LLM-PEFTs on the three code-change-related tasks.

3.2.1 Comparison Between LoRA and Prefix-Tuning. We observe that when performing PEFT on code change-related tasks, LLMs tuned with LoRA results in significantly better performance compared to LLMs tuned with prefix-tuning. For example, on CRG and CMG, the average BLEU scores of Code Llama-13b using LoRA are 5.61 and 12.52, respectively; while those of Code Llama-13b using prefix-tuning are 1.17, and 2.54, respectively. This indicates that LoRA can help LLMs learn more knowledge related to code changes compared to prefix-tuning. These results are also in line with the different mechanisms of LoRA and prefix-tuning. Specifically, LoRA updates the self-attention modules in LLMs, making it relatively easy to learn new knowledge that is not well covered by the pre-training data. However prefix-tuning, which only prepends some trainable vectors to the input of each layer, plays a similar role to soft prompts, i.e., stimulating the learned knowledge in LLMs. Considering existing

LLMs are not specially trained for code-change-related tasks and have limited knowledge of code changes, it is reasonable that LoRA outperforms prefix-tuning.

Table 8. [RQ2] – An example of the CMG task.

Diff	public class AnalystWorker implements Runnable {
	/** Open a single point channel to the broker
	* to receive high-priority requests immediately
	*/
	private synchronized void openSideChannel () {
	+ if (sideChannelOpen)
	+ return;
	+
	LOG.info("Opening side channel for single point requests.");
	new Thread(() ->{
	sideChannelOpen = true;

Gold	resolve race condition where two side channels could open.
Llama 2-13b + LoRA	Fixed a bug in AnalystWorker where it would re-open a side channel
Llama 2-13b + Prefix-Tuning	data class in it. java, to.

Table 8 presents an example of CMG and the commit messages generated by two Llama 2-13b models that are fine-tuned with LoRA and prefix-tuning, respectively. We observe that the commit message generated by Llama 2-13b with LoRA successfully captures the changed parts in the diff, while the Llama 2-13b with prefix-tuning only generates some low-quality keywords. Similar phenomena are also reported by prior studies [9, 11, 14].

Finding 5: When applying LLM-PEFTs, tuning LLMs using LoRA achieve a significantly better performance compared to those using prefix-tuning.

3.2.2 Impact of Model Size. When tuning LLMs with LoRA, the performance difference between large and small models in the same LLM family is not significant. For example, on CRG, Code Llama-13b outperforms Code Llama-7b. While, on JITCU, smaller models in the Llama 2 family and the Code Llama family outperform the larger models in terms of GLEU scores. One possible reason is that smaller models may have just sufficient adaptability and efficiency when fine-tuning with LoRA, allowing them to capture relevant features effectively for specific tasks. The complexity and capacity of larger models may not provide additional benefits on these tasks related to code changes and might even introduce unnecessary complexity. This means that when tuning LLM using LoRA on tasks related to code changes, developers should consider both the bigger and smaller models within the same model family at the same time.

Finding 6: When tuning LLMs using LoRA, larger models do not necessarily have better performance, even within the same LLM family.

3.2.3 Impact of Model Family. We observe that when tuning LLMs with LoRA, Llama 2 and Code Llama families are the best-performing LLMs on the three code-change-related tasks. Specifically, on CRG and CMG-Python, the Llama 2 performs the best; however, on other tasks, Code Llama performs better. Nonetheless, the performance gap between the two model families is not significant. For example, on CRG, the best-performing Llama 2 model (Llama 2-7b) has a BLEU score of 5.74, which is only approximately 2.32% higher than the best-performing Code Llama model (Code Llama-13b) with a score of 5.61. One possible reason is that, though Code Llama is based on Llama 2 and has been pre-trained on a large-scale code corpus, tuning LLMs using LoRA on code-change-related tasks requires the model to learn new knowledge that is not well covered by the pre-training

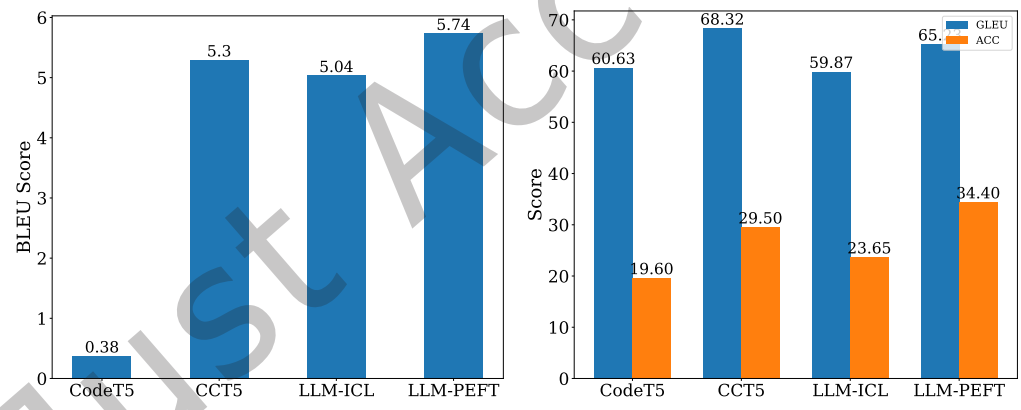
data. This makes these two model families perform similarly on code-change-related tasks. **We recommend exploring both Llama and Code Llama families when tuning LLMs using LoRA**.

> **Finding 7**: When tuning LLM using LoRA on tasks related to code changes, utilizing models specifically pre-trained on code-related tasks (such as Code Llama) offers limited benefits. Llama 2 and Code Llama families are the best-performing LLMs.

3.3 RQ3: How Do LLMs Perform on Code Change-Related Tasks Compared to Small Pre-trained Models?

Table 9. [RQ3] - Performance of the small pre-trained models after fully fine-tuned on code-change-related tasks. The deeper the color, the better the performance.

Model	CRG	CMG					JITCU	
	BLEU	CPP	C#	Java	JavaScript	Python	GLEU	ACC
		BLEU	BLEU	BLEU	BLEU	BLEU		
Codet5	0.38	2.85	3.73	5.33	7.32	5.32	59.14	17.90
CCT5	5.30	12.99	19.53	15.90	17.61	14.57	68.32	29.50



(a) BLEU on CRG (b) GLEU and ACC on JITCU

Fig. 2. [RQ3] - Comparison between the best performing LLM-ICL, LLM-PEFT, and Small Pre-trained Models on CRG and JITCU

Table 9 presents the performance of the small pre-trained models that are fully fine-tuned on the three code-change-related tasks. We observe that CCT5 outperforms CodeT5 on all tasks. This is because CCT5 is pre-trained with code changes and code-change-related natural language descriptions, which makes it good at handling code-change-related tasks. In contrast, CodeT5 is only pre-trained with code and code-related descriptions. To better show the differences between LLMs and small pre-trained models, we compare the best-performing

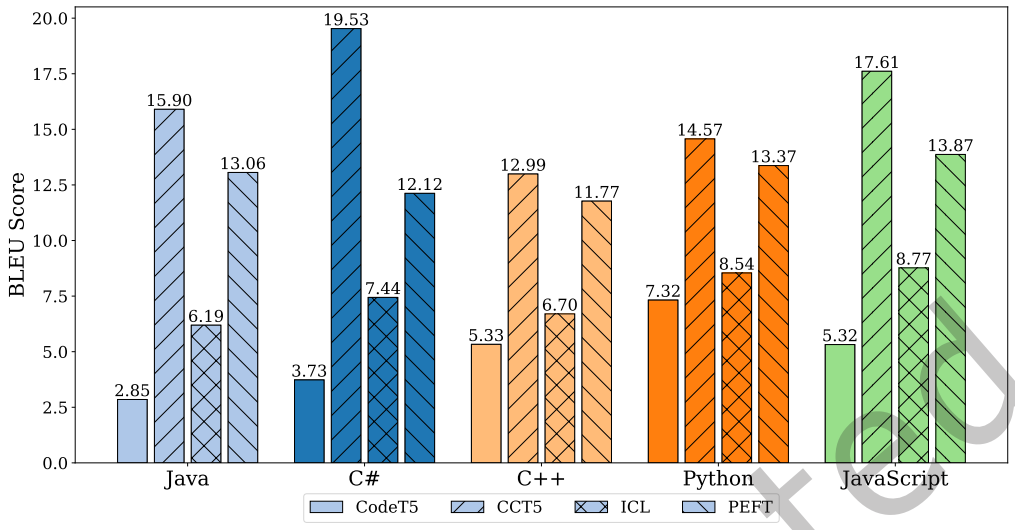


Fig. 3. [RQ3] - Comparison between the best performing LLM-ICL, LLM-PEFT, and Small Pre-trained Models on CMG

LLM-ICL, LLM-PEFT, and small pre-trained models on CRG, CMG, and JITCU. Figure 2a, Figure 3, and Figure 2b visualize the performances of the best performing LLM-ICL, LLM-PEFT, and small pre-trained models on CRG, CMG, and JITCU, respectively.

3.3.1 Comparison between LLM-ICLs and Small Pre-trained Models. The best-performing LLM-ICLs are similar to or better than CodeT5 on the tasks related to code changes. Specifically, the best-performing LLM-ICLs can statistically significantly outperform CodeT5 on CRG, the C++, C#, and Python sub-datasets on CMG. For example, Code Llama-7b with 7 examples outperforms CodeT5 by 4.66 on CRG. Besides, there is no significant difference between the best-performing LLM-ICL and CodeT5 on JITCU in terms of GLEU. For example, CodeT5 only outperforms the best-performing LLM-ICL (i.e., Code Llama-7b with 2 examples) by 0.76 on JITCU. These results indicate that LLMs are promising on code-change-related tasks.

The best-performing LLM-ICLs are statistically significantly inferior to CCT5 on all tasks. Specifically, CCT5 outperforms Code Llama-7b with 7 examples on CRG by 0.26. Recall that CCT5 are pre-trained with code-change-oriented objectives and fine-tuned with task-specific datasets, and the parameters have learned the knowledge related to code changes through updates. This further motivates us to compare LLM-PEFT with small pre-trained model on the tasks related to code changes, both of which can update model parameters.

Finding 10: The best-performing LLM-ICLs are similar or better than small models pre-trained with code, but inferior to small models pre-trained with code changes on the tasks related to code changes.

3.3.2 Comparison between LLM-ICLs and LLM-PEFTs. The best-performing LLM-PEFTs (LLMs tuned with LoRA) outperform the best-performing LLM-ICLs across all tasks, with particularly significant differences on CMG and JITCU. Specifically, the best-performing LLM-PEFT performs slightly better than the best-performing LLM-ICL on CRG; the best-performing LLM-PEFTs outperform the best-performing LLM-ICLs by 123.55%, 89.67%,

77.53%, 60.96% and 103.67% on Java, C#, C++, Python and JavaScript sub-datasets of CMG respectively; the best-performing LLM-PEFTs outperform the best-performing LLM-ICLs by 8.95% and 47.45% in terms of GLEU and ACC on JITCU. This means that if model parameters are allowed to be adjusted, the performance of LLMs on code change-related tasks can be significantly improved.

Finding 11: Tuning LLM with LoRA can significantly improve the performance of LLMs on code-change-related tasks.

Table 10. [RQ2] - The number of parameters updated when fine-tuning each LLM with PEFT

Method	InCoder-1b	CodeGen-2b-nl	CodeGen-6b-nl	Llama 2-7b	Llama 2-13b	Code Llama-7b	Code Llama-13b
LoRA	3,15M(0.23%)	2.62M(0.09%)	4.3M(0.06%)	8,38M(0.12%)	13,1M(0.10%)	8.39M(0.12%)	13.1M(0.10%)
prefix-tuning	-	1.64M(0.06%)	2.7M(0.04%)	2,62M(0.04%)	4.1M(0.03%)	2.62M(0.04%)	4.1M(0.03%)

3.3.3 Comparison between LLM-PEFTs and Small Pre-trained Models. We observe that the best-performing LLM-PEFTs (LLMs tuned with LoRA) statistically significantly outperform CodeT5 on all tasks. Moreover, on CRG, the best-performing LLM-PEFT, i.e., Llama 2-7B tuned using LoRA statistically significantly outperforms the best fully fine-tuned small pre-trained model, i.e., CCT5, by 8.3%. On JITCU, the best-performing LLM-PEFT, i.e., Code Llama-13B tuned using LoRA, outperforms the fully fine-tuned CCT5 by 18.47% in terms of ACC. On CMG, there is no significant difference between the best-performing LLM-PEFT and the best fully fine-tuned small pre-trained model (CCT5) on C++, Java, and Python sub-datasets, and on the Javascript and C# sub-datasets, the best-performing LLM-PEFT are statistically significantly inferior to CCT5. These indicate that the best-performing LLM-PEFTs have comparable performance to CCT5.

Table 10 shows the number of parameters that are updated with different PEFT methods. The number of parameters updated when fine-tuning each LLM with PEFT is fewer than the total number of parameters (i.e., 223M) of the small pre-trained models (as is shown in Table 1). Considering that LLMs are not pre-trained with code-change-oriented objectives, we believe the performance of LLMs can be further improved by pre-training on the objectives and data related to code changes.

Finding 12: The best-performing LLM-PEFTs can have comparable performance to the best-performing fully fine-tuned small pre-trained models on the tasks related to code changes with fewer changed parameters.

3.4 RQ4: How Do LLMs Perform With Different Input Formats on Code-Change-Related Tasks?

Table 11, 12, 13 and Table 14, 15 show the performance LLM-ICLs and LLM-PEFTs using code format as input. To visually demonstrate the performance difference between using code as input and using diff as input, we use blue to indicate a relatively worse performance and red to indicate a relatively better performance. The deeper the color, the greater the difference.

3.4.1 Impact of Different Input Formats on LLM-ICL. We observe that when using LLM-ICL, using diff as the input of LLMs generally outperforms using code as the input of LLMs. Specifically, on JITCU, the performance of LLMs is significantly better when using diff as input compared to using code as input (except for the CodeGen family, where the ACC value is close to 0 across different settings). For example, on CRG, the best-performing LLM-ICL (Code Llama-7b) has a BLEU score of 5.04 when using diff as input, which is only 0.14 lower than the best result obtained using code as input. Additionally, for the best-performing LLM-ICL on JITCU (Code Llama-7B), the ACC score is 23.65 when using diff as input, while it is only 1.6 when using code as input. One possible reason is

Table 11. [RQ3] - BLEU scores of LLM-ICLs with the code input on CRG. Blue indicates worse performance while red indicates better performance, compared to using diff as input. The deeper the color, the greater the difference.

Models	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
InCoder-1b	2.97	3.95	3.82	3.84	3.53	3.42	3.3	3.23	3.04
CodeGen-2b-nl	0.31	0.5	0.58	0.7	0.88	1.03	1.32	1.69	1.92
CodeGen-6b-nl	0.87	0.93	1.11	1.4	1.75	2.07	2.57	2.78	2.95
Llama 2-7b	0.93	4.33	4.43	4.52	4.54	4.67	4.67	4.73	4.8
Llama 2-13b	2.34	4.28	4.45	4.69	4.71	4.61	4.71	4.71	4.71
Code Llama-7b	0.9	4.53	4.5	4.69	4.78	4.79	4.86	4.77	4.9
Code Llama-13b	2.32	4.57	4.57	4.71	4.79	4.83	4.79	5.02	4.9

Table 12. [RQ3] - BLEU scores of LLM-ICLs with the code input format on Python and Java sub-datasets of CMG. Blue indicates worse performance while red indicates better performance, compared to using diff as input. The deeper the color, the greater the difference.

Lang	Model	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
Java	InCoder-1b	3.08	3	2.62	2.74	2.66	2.75	2.79	2.76	2.66
	CodeGen-2b-nl	1.88	1.98	2	2.11	2.28	2.29	2.29	2.36	2.33
	CodeGen-6b-nl	2.46	2.58	2.94	3.24	3.19	3.52	3.43	3.39	3.43
	Llama 2-7b	1.13	3.28	3.51	3.88	3.8	3.63	3.54	3.42	3.54
	Llama 2-13b	1.74	3.35	3.91	4.25	4.23	4.57	4.82	4.91	5.13
	Code Llama-7b	1.81	5.78	5.17	5.86	5.73	5.73	5.74	5.49	5.47
	Code Llama-13b	1.69	4.47	4.08	4.14	4.34	4.49	5.04	4.61	4.28
Python	InCoder-1b	4.54	3.93	4.23	4.01	3.9	3.88	3.94	4.07	3.92
	CodeGen-2b-nl	2.99	3.09	3.01	3.26	3.28	3.48	3.55	3.57	3.38
	CodeGen-6b-nl	4.08	4.35	4.36	4.71	4.62	4.47	4.55	4.81	4.77
	Llama 2-7b	2.12	4.68	4.93	5.19	5.24	5.31	5.05	4.83	4.64
	Llama 2-13b	3.45	4.6	5.56	6.53	6.71	6.75	6.41	5.67	5.57
	Code Llama-7b	1.5	5.6	7.03	7.45	8.07	8.54	8.02	7.16	6.6
	Code Llama-13b	2.31	4.9	6.39	6.75	6.59	7.28	6.62	5.56	5.52

that how to update the comment is highly related to the changed parts in the code. Diffs explicitly annotate the changed lines in code with "+" an "-", making it easier for LLMs to identify and understand what is changed. For two connected code snippets, LLM needs to first understand the two snippets, then determine the updated, deleted, and unchanged lines of code by comparing them, and finally comprehend the change information, which is more complex.

Finding 8: When using LLM-ICL, the input format significantly affects the performance of LLM. Specifically, the model performs better with diff as input, especially on the JITCU task.

Table 13. [RQ3] - GLEU and ACC scores of LLM-ICLs with the code input format on JITCU. Blue indicates worse performance while red indicates better performance, compared to using diff as input. The deeper the color, the greater the difference.

Model	metric	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
InCoder-1b	GLEU	9.99	0.12	0.47	1.15	1.62	1.65	2.00	2.44	3.19
CodeGen-2b-nl		0.00	0.02	1.19	2.10	2.91	4.80	5.80	5.24	4.74
CodeGen-6b-nl		0.00	5.02	13.02	13.45	12.00	8.47	7.17	7.69	7.69
Llama 2-7b		0.32	1.68	8.02	13.50	13.24	14.83	15.00	14.83	14.44
Llama 2-13b		7.50	4.17	15.29	17.49	15.64	16.44	16.23	15.05	14.80
Code Llama-7b		4.89	5.21	9.11	11.82	14.53	15.29	15.59	14.07	13.98
Code Llama-13b		2.57	6.15	21.32	26.07	26.16	27.31	28.48	26.59	25.84
InCoder-1b	ACC	1.50	0.05	0.00	0.05	0.10	0.05	0.15	0.20	0.35
CodeGen-2b-nl		0.00	0.05	0.20	0.60	0.30	0.35	0.25	0.15	0.10
CodeGen-6b-nl		0.00	0.35	0.70	0.95	0.65	0.20	0.30	0.20	0.25
Llama 2-7b		0.25	0.55	1.90	2.55	2.05	2.05	2.00	1.35	1.40
Llama 2-13b		2.60	1.20	4.45	4.35	2.75	2.25	1.80	0.95	1.25
Code Llama-7b		0.00	1.75	1.60	2.55	3.70	3.55	3.10	2.00	1.75
Code Llama-13b		0.55	2.25	4.70	6.15	5.70	5.90	5.85	4.75	4.45

Table 14. [RQ3] - Performance of LLM-PEFTs with the code input format on CRG and JITCU. Blue indicates worse performance while red indicates better performance, compared to using diff as input. The deeper the color, the greater the difference.

Model	CRG		JITCU			
	BLEU-4		GLEU		ACC	
	LoRA	Prefix	LoRA	Prefix	LoRA	Prefix
InCoder-1b	3.54	-	4.98	-	8.85	-
CodeGen-2b-nl	0.46	0.86	0.30	0.03	0.55	0.70
CodeGen-6b-nl	1.65	0.98	3.02	7.55	1.35	1.65
Llama 2-7b	5.71	2.02	62.58	0.13	32.35	0.00
Llama 2-13b	5.16	0.61	62.35	0.00	32.65	0.00
Code Llama-7b	4.40	0.54	63.69	0.09	34.45	0.00
Code Llama-13b	5.40	1.13	63.09	0.16	34.95	0.00

3.4.2 Impact of Different Input Formats on LLM-PEFT. For LLM-PEFTs, there is no significant performance difference between different input formats. For example, when using LLM-PEFT, on CRG, the performance difference between different input formats is only 0.03; on JITCU, the differences in GLEU and ACC are only 1.54 and 0.05, respectively. These facts indicate that LLM-PEFTs are not sensitive to the input format, which is different from the LLM-ICLs. One possible reason is that when representing a code change as two code snippets, it is difficult for LLMs without fine-tuning to compare two code snippets and capture the changed part, while

Table 15. [RQ3] - Performance of LLM-PEFTs with the code input format on CMG. Blue indicates worse performance while red indicates better performance, compared to using diff as input. The deeper the color, the greater the difference.

Model	CPP		C#		Java		JavaScript		Python	
	LoRA	Prefix	LoRA	Prefix	LoRA	Prefix	LoRA	Prefix	LoRA	Prefix
InCoder-1b	5.24	-	5.11	-	4.75	-	4.66	-	5.12	-
CodeGen-2b-nl	3.99	3.47	3.80	2.31	2.95	2.52	3.69	2.70	5.52	4.29
CodeGen-6b-nl	4.43	2.47	4.47	4.45	3.77	2.47	3.87	3.29	4.90	4.09
Llama 2-7b	10.98	2.49	11.86	1.28	12.27	1.54	13.78	0.15	12.94	0.87
Llama 2-13b	10.41	0.59	11.62	0.94	12.01	0.83	13.36	1.27	12.78	0.77
Code Llama-7b	10.82	1.44	11.98	0.41	13.10	1.45	13.72	2.67	13.44	0.65
Code Llama-13b	11.36	1.20	12.10	1.74	12.10	3.26	13.44	2.63	11.42	1.26

LLMs fine-tuned with some code change data can learn to adapt to the input format and thus achieve better performance.

Finding 9: When using LLM-PEFT, the input format does not significantly affect the performance of the LLM.

3.5 RQ5: When Do LLMs Perform Better?

Table 16. [RQ5] - The selected pre-trained models for different tasks under the application of ICL or Finetuning.

	fully fine-tuned small pre-trained model	LLM-ICL	LLM-PEFT
CRG	CCT5	Code Llama-7b	Llama 2-7b
CMG_Java			Code Llama-7b
CMG_C#			Code Llama-13b
CMG_Cpp			Code Llama-13b
CMG_Python			Code Llama-7b
CMG_JavaScript			Code Llama-13b
JITCU			Code Llama-7b

Table 16 shows the best-performing models from small pre-trained models, LLM-ICLs and LLM-PEFTs selected to answer this research question. Table 17 shows the performance of the models on each code change category for each code-change-related task.

We observe that **the best-performing LLM-PEFTs outperform the best-performing LLM-ICLs on all types of code changes** For example, on CRG, the best-performing LLM-PEFTs outperform the best-performing LLM-ICLs on all types of code changes by 86.86%, 16.03%, 3.54%, and 31.76% on each code change type, respectively. This indicates that compared to ICL, PEFT can comprehensively improve the performance of LLMs on all code change categories. Therefore, we believe that LLM-PEFT can be applied to different types of code changes.

Table 17. [RQ5] - Performance of different techniques for each code change category in code-change-related tasks. BLEU for CRG, CMG, GLEU for JITCU. The deeper the color, the better the performance.

Task	Method	Doc	Code		Doc&Code	Total
			Feat	Ref		
CRG	Sample Num	4	149	13	30	196
	Small Pre-trained Models	5.48	4.71	5.04	5.44	4.83
	LLM-ICL	3.12	4.68	3.67	3.81	4.45
	LLM-PEFT	5.83	5.43	3.80	5.02	5.27
CMG	Sample Num	12	79	56	53	200
	Small Pre-trained Models	9.86	11.65	21.71	13.94	15.31
	LLM-ICL	9.68	6.30	9.82	5.37	6.62
	LLM-PEFT	18.19	10.04	19.13	14.30	13.01
JITCU	Sample Num	0	102	81	13	**196**
	LLM-ICL	-	53.58	71.17	34.14	59.97
	LLM-PEFT	-	57.42	75.57	44.45	64.13
	Small Pre-trained Models	-	58.43	72.59	46.38	65.35

Finding 13: PEFT can comprehensively improve the performance of LLMs across all categories of code changes, including changes that only modify documentation, only modify code, and those that modify both code and documentation simultaneously.

We observe that **the best-performing LLM-PEFTs statistically significantly outperform the best-performing fully fine-tuned small models on doc-only code changes.** Specifically, in terms of doc-only code changes on CRG and CMG, the best-performing LLM-PEFTs achieve the BLEU scores of 5.83 and 18.19, respectively, while those of the best-performing fully fine-tuned small models are 5.48 and 9.86. The results indicate that LLM-PEFTs are effective on doc-only code changes. The possible reason is that LLMs are good at understanding natural languages.

In terms of the changes related to source code, the best-performing LLM-PEFTs perform comparably to fully fine-tuned small models, indicating LLM-PEFTs can to some extent understand the changes of functional logic. For example, on JITCU, in terms of code changes related to Ref, the best-performing LLM-PEFTs achieve the BLEU score of 75.57, while that of the best-performing fully fine-tuned small models is 72.59. on CMG, in terms of code changes related to Feat, the best-performing fully fine-tuned small pre-trained models can achieve the BLEU score of 11.65, while that of the best-performing LLM-PEFTs is 10.04. This indicates that compared with small pre-trained models, though LLM-PEFTs can understand the changes that are only related to documentation, LLM-PEFTs still need more knowledge to understand featuring, refactoring, and the mix of code and documentation modifications. One possible reason is that fully fine-tuned small models have learned the knowledge related to source code changes through the pre-training specific to code changes, while LLM-PEFTs are only fine-tuned with some task-specific data. Thus, LLM-PEFTs may have insufficient code-change-specific knowledge, such as the knowledge of refactoring. We recommend that future work consider taking into account code-change-oriented data and objectives during the pre-training of LLM for code-change-related tasks.

Finding 14: LLM-PEFTs can outperform the fully fine-tuned small models on the doc-only code changes and achieve comparable performance to fully fine-tuned small models on other code change types.

4 DISCUSSION

4.1 Human Evaluation

Here, following prior studies [30, 50, 56], we take CMG as an example and conduct a human evaluation to further investigate the effectiveness of LLMs in code-change-related tasks.

Specifically, following prior studies [50, 86], we recruit 4 evaluators who are not the co-authors of this paper. All evaluators have more than 5 years of software development experience and have a deep understanding of Computer Science. We randomly sample 100 commits from the MCMD dataset [86] (50 commits for Java, and 50 commmits for Python). The number of sampled commits is the same as the number of sampled commits used in prior studies related to the human evaluation on the MCMD dataset [50]. We chose these two languages (i.e., Java and Python) because they are currently the most popular programming languages. Additionally, compared to the other three languages, the evaluators are more familiar with these two languages. We then apply the best-performing models from the explored three approaches (small pre-trained models, LLM-ICLs, and LLM-PEFTs) to generate commit messages for these sampled commits. Specifically, in terms of small pre-trained models, we apply CCT5 with the diff input. In terms of LLM-ICL, we apply Code Llama-7b with 7-shot diff input and 5-shot diff input on the Java and Python sub-dataset, respectively. For LLM-PEFT, we apply Code Llama-7b-LoRA with diff input on Java and Python sub-datasets. This process yields 300 generated commit messages.

Following prior studies [50, 86], we ask evaluators to rate each generated message from the following three dimensions, with each dimension scored on a scale of 1 to 5 (1: poor, 2: marginal, 3: acceptable, 4: good, 5: excellent):

(1) **Adequacy**: The extent to which the generated commit message covers the main content of the code changes. A higher score indicates that the generated information more comprehensively summarizes the main content of the code modifications.
(2) **Conciseness**: The extent to which the generated commit message does not contain irrelevant content. A higher score indicates that the information is more concise, without the interference of redundant and irrelevant information.
(3) **Expressiveness**: The readability and understandability of the generated commit message. A higher score indicates that the information is expressed more clearly and easier to understand.

To facilitate the evaluation process, we prepare a questionnaire for each commit. The questionnaire includes the code change, the ground truth commit message, and the commit messages generated by the three compared techniques (small pre-trained models, LLM-ICLs, and LLM-PEFTs). To minimize potential bias, the three techniques are anonymous in the questionnaire, ensuring that the participants are unaware of which technique generated each commit message. Furthermore, each participant completes the questionnaire independently, without any discussion or collaboration with others. Each sample commit will be evaluated by two participants, and we consider the average of the two scores as the final score. If the two scores are significantly different, we will invite a third evaluator to evaluate the commit, and the final score will be the average of the two closest scores.

Same as the conclusion in Section 3.3, the human evaluation further confirms that **LLM-PEFTs can generate commit messages that are more concise and readable compared with other techniques**. Table 18 presents the results of the human evaluation. Overall, the LLM-PEFTs outperform the other techniques in terms of conciseness and expressiveness. The average scores achieved by LLM-PEFTs for adequacy, conciseness, and expressiveness are 3.29, 3.78, and 4.22, respectively. These values are 33.74%, 38.97% and 10.76% higher than those of LLM-ICLs, and -3.52%, 2.92%, and 5.50% higher than those of small pre-trained models, respectively. While the

Table 18. [Discussion] the results of our human evaluations

Approach	Adequacy	Conciseness	Expressiveness
LLM-ICL	2.46	2.72	3.82
Small Pre-trained Model	**3.41**	3.66	4
LLM-PEFT	3.29	**3.78**	**4.23**

adequacy of LLM-PEFTs is slightly lower than that of small pre-trained models, the difference is not significant. This indicates that LLM-PEFTs can generate comparable commit messages compared with small pre-trained models in terms of adequacy.

Interestingly, we also observe that LLM-PEFTs are rated higher than the fully fine-tuned small models in terms of conciseness. This is counter-intuition as it is generally considered that LLMs can generate informative and readable texts, but often with some redundant information [32, 55]. One possible reason is that through fine-tuning, LLMs have learned the preference for this task, i.e., to generate concise summaries rather than lengthy words.

Table 19. [Discussion] - An example of the commit message generation task.

Diff	exports.update_messages=function update_messages(events){
	}
	msgs_to_rerender.push(msg);
	
	- msg.alerted=event.flags.indexOf("has_alert_word")!==-1;
	- msg.mentioned=event.flags.indexOf("mentioned")!==-1||
	- event.flags.indexOf("wildcard_mentioned")!==-1;
	+ message_store.set_message_booleans(msg,event.flags);
	
	condense.un_cache_message_content_height(msg.id);

Gold	Call message.set_message_booleans() in update_messages().
LLM-ICL	Merge pull request from jeannefukumaru/update_message_store
Fully Fine-tuned Small Models	refactor:Use set_message_booleans event for mentions.
LLM-PEFT	message_store.set_message_booleans()

To better understand the quality of the generated commit message, we conduct a case study. Table 19 shows an example of the commit message generation task. We present the changed code, the commit message written by developers, as well as the commit messages generated by LLM-ICL, fully fine-tuned small models, and LLM-PEFT. Based on the diff and the commit messages written by the developers, we can see that the code change is related to *refactoring* the *update_messages()* using the *set_message_booleans()* in *message_store* class. However, the commit messages generated by the three techniques are not perfect. The commit message generated by LLM-ICL mentions a plausible phrase *update_message_store*, but it is not accurate; all the information is hallucinated, but its readability is acceptable. This indicates that the commit message generated by LLM-ICL is not adequate and concise at all, but expressive. The commit message generated by fully fine-tuned small models mentions the *refactor* and *set_message_booleans()*, but misses *set_message_booleans()* and *message_store*; it hallucinates the information *event* and *mentions*, but its readability is acceptable. This indicates that the commit message generated by fully fine-tuned small models is moderate in terms of adequacy and conciseness, and is expressive. The commit message generated by LLM-PEFT mentions the *set_message_booleans()* and *message_store*, but misses *refactoring* and *update_messages()*; there is no hallucinated information, and its readability is acceptable. This indicates that the commit message generated by LLM-PEFT is moderate in terms of adequacy, and is expressive and very concise.

> **Finding 15**: Users rate that compared to other techniques, LLM-PEFT can generate concise and readable commit messages without compromising too much adequacy.

4.2 Implications

When adapting LLMs with ICL to code-change-related tasks, the number of examples in the prompt should be determined by the data length of the task and the context length allocated to the model. In Section 3.1, we observe that the performance of LLM-ICLs is not always positively correlated with the number of examples, and the best-performing LLM-ICLs have different numbers of examples in different tasks. One possible reason is that the lengths of examples in the different tasks are different, and the context length allocated to the model is limited. Therefore, we suggest that practitioners should make a decision based on the distribution of data lengths in the task and the context length allocated to the model when adapting LLMs with ICL to code-change-related tasks.

When using LLMs on code-change-related tasks, the choice of the model family is more important than the model size. In Section 3.1 and Section 3.2, we observe that the performance of LLM is not always positively correlated with the model size even within the same model family, no matter whether using ICL or PEFT. However, we observe that there are differences between the performance of LLMs from different model families. For example, when using LLM with ICL, the best-performing Code Llama outperforms the best-performing Llama 2 and CodeGen across tasks. However, the best-performing smaller Code Llama outperforms the best-performing larger Code Llama across tasks b LLM with ICL. This indicates that the choice of the model family is more important than the model size when using LLMs on code-change-related tasks. Therefore, we suggest that when an LLM is not performing well on code-change-related tasks, practitioners should not only try different model sizes but also try different model families.

When using PEFT, LoRA can help LLMs better handle code-change-related tasks than prefix-tuning. Section 3.2 shows that LLMs tuned with LoRA outperform the LLMs tuned with prefix-tuning in almost all settings. For example, the average performance of LLMs applying LoRA is 3.22 times that of those applying prefix-tuning on CRG. Therefore, we suggest using the LoRA technique to help LLMs handle code-change-related tasks.

LLM-ICLs open up the opportunities to automate the code-change-related tasks suffering from data scarcity. Data scarcity refers to situations where data acquisition is difficult or the amount of data is small. This often occurs in new tasks or scenarios involving user privacy sensitivities. Limited training data cannot guide pre-trained models, such as CodeT5 and CCT5, to learn downstream tasks well through fine-tuning, resulting in their poor performance. We have also tried to adapt CodeT5 and CCT5 with ICL and no fine-tuning to code-change-related tasks, but find they cannot produce meaningful output. In contrast, Section 3.3 has shown that LLM-ICLs can achieve promising performance on code-change-related tasks with only a few examples due to LLMs' emergent abilities. These results imply that it is now possible to automate the code-change-related tasks suffering from data scarcity with LLM-ICLs, which we believe can be a promising research direction.

Pre-training LLMs with code-change-oriented objectives can potentially bring substantial improvements to code-change-related tasks. We can see from Section 3.3 that on CRG and CMG, CCT5 outperforms CodeT5 by substantial margins, and achieves comparable performance to the best LLM-PEFTs. On JITCU, the ACC score of CCT5 is higher than CodeT5 but lower than the best LLM-PEFTs. Comparing CCT5 with CodeT5, we can see that pre-training models using code-change-oriented objectives are beneficial for code-change-related tasks. Comparing LLM-PEFTs with CodeT5, we can know that increasing the sizes of the model and the pre-training data are also beneficial. Inspired by these, a straightforward idea would be pre-training an LLM with

Table 20. Performance of Code Llama under untruncated shots.

Tasks	Metric	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
CRG	BLEU	0.88	4.54	4.59	4.63	4.77	4.86	4.98	**5.01**	4.91
CMG_Java	BLEU	0.62	4.96	5.16	5.82	**5.86**	-	-	-	-
CMG_Python	BLEU	4.45	5.67	6.78	7.19	**7.44**	-	-	-	-
JITCU	GLEU	0.01	56.93	62.53	63.64	64.24	**64.31**	64.11	63.79	63.88
	ACC	0.05	21.7	25.45	26.3	27.65	**27.75**	27	26.75	26.15

code-change-oriented objectives, which can take advantage of the two beneficial designs. We believe this direction would bring substantial benefits for code-change-related tasks.

Practitioners can potentially improve LLM-ICLs for code-change-related tasks by designing novel formats to represent code changes. Diffs and code snippets are the most common formats to represent code changes. In Section 3.4, we investigate the impact of input formats on the effectiveness of LLM-ICLs, and find that LLM-ICLs with diffs as the input format outperforms those representing code changes as code snippets in most of the settings, especially on the JITCUP tasks. This indicates that the representation formats of code changes play an important role in the effectiveness of LLM-ICLs. Thus, it is interesting and promising to improve LLM-ICLs on code-change-related tasks by designing novel representation formats for code changes. For example, we can identify the changed parts in each code change using static analysis tools and explicitly mention these changed parts in the prompt to help LLMs better understand code changes.

To boost code-change-related tasks with LLMs, we should guide LLMs to learn more code-change-specific knowledge. In Section 3.5, we observe that LLM-PEFTs significantly outperform the fully fine-tuned small models in doc-only code changes. This is related to the fact that LLMs are pre-trained with rich sources of natural language texts and thus have the ability to understand and handle tasks related to natural languages. However, the fully fine-tuned small models perform better than LLM-PEFTs on the refactor and doc-and-code code changes. This indicates that PEFT on task-specific datasets is not enough for LLMs to learn code-change-specific knowledge, such as the knowledge of refactoring. We suggest that future researchers should focus more on guiding LLMs to learn code-change-specific knowledge when tuning LLMs for code-change-related tasks. For example, we can leverage the idea of multi-task learning and transfer learning. Before fine-tuning LLMs on the downstream task, we can first tune LLMs on other tasks that are helpful for identifying code-change-specific knowledge, such as refactoring detection, and then fine-tune them on the specific downstream tasks.

4.3 The Impact of Example Truncation to ICL

In Section 3.1.2, we explore the impact of the number of examples when applying LLM using ICL under the condition of the limited input length. We truncate each example when the total length of the input exceeds the limit. However, the truncation can affect the performance of LLMs. We would like to explore the performance of LLM using ICL without truncating examples. Specifically, we select Code Llama to conduct experiments with untruncated shots. Because Code Llama's max input length is large enough (100K) to hold 8 examples, while the max input length of CodeGen and InCoder is 2048 and that of Llama 2 is 4096, which still may lead to truncation. To be consistent with our main experiments, we use Code Llama 7B with diff input format to explore 0-8 shots on CRG and JITCU. We only explore 0-4 shots on Python and Java sub-datasets of CMG, as increasing the number of shots beyond this range would lead to GPU memory overflow, even with a minimal batch size of 1, given our

Table 21. Data leakage rates across LLMs on selected tasks.

Model	CRG	CMG					JITCU	JITCU Clear
		Java	C#	CPP	Python	JavaScript		
InCoder-1b	0	0	0	0	0	0	3.6	0.3
CodeGen-2b-nl	0	0	0	0	0	0	0	0
CodeGen-6b-nl	0	0	0	0	0	0	0	0
Llama-2-7b	0	0	0	0	0	0	2.2	0
Llama-2-13b	0	0	0	0	0	0	5.45	1.9
CodeLlama-7b	0	0	0	0	0	0	0.25	0.05
CodeLlama-13b	0	0	0	0	0	0	0.6	0.4

available computational resources. Table 20 shows the performance of Code Llama with untruncated examples on selected tasks. We can observe that although the results differ between truncated and untruncated versions, both demonstrate consistent performance trends. For example, there is an initial improvement followed by a decline as shots increase on CRG and JITCU. The decline in performance with increasing shots may be attributed to the large number of examples being concatenated and input as a long sequence, which could impair the model's ability to fully comprehend the target sample. Similar findings [10, 69] have been reported in previous studies. Additionally, the number of shots required for LLMs to achieve the best performance varies across different tasks, which is also consistent with our findings on the truncated data.

4.4 Analysis of Data Leakage

In this paper, we conduct experiments on code changes using multiple LLMs. However, these models are pre-trained with large amounts of data, which may raise concerns about potential data leakage. These models may have seen some data in our test sets and merely memorized the results instead of predicting them. To assess the extent of data leakage, we follow the approach of Guo et al. [24] by examining n-gram overlaps between generated content and test data, and considering a sample to suffer from data leakage if an n-gram in the generated content is in the ground truth. Following Guo et al. [24], we set $n = 10$ or $n = 3$ if the length of generated content is less than 10.

As shown in Table 21, we examine the output of each LLM on CRG, CMG, and JITCU with the zero-shot setting. On CRG and CMG, the rate of samples with identical n-grams between the generated content and the test set ground truth (hereon, the data leakage rate) is zero across all LLMs. However, on JITCU, InCoder-1b, Llama 2-7b and Llama 2-13b exhibit relatively higher rates (3.6% 2.2% and 5.45%, respectively). Please note that the goal of JITCU is to update old comments. Given a sample, if its new comment is very similar to its old comment, e.g., only fixing a typo, and the generated comment is nearly identical to the old comment, there can be identical n-grams between the generated and new comments. However, such n-gram overlap is likely to be attributed to the high similarity between the old and new comments, instead of data leakage. To eliminate the influence of such data points, we identify and filter out the samples in which the old and generated comments are nearly identical based on the n-gram overlap mentioned above, and re-calculate the data leakage rates on JITCU with the remaining samples, as shown in the "JITCU Clear" column of Table 21. We can observe that the data leakage rates not only drop substantially but also become negligible. For example, for InCoder-1b, the rate of samples containing identical n-grams decreases from 3.6% to 0.3%. Based on these results, we believe the potential data leakage is negligible and has minimal impact on our experimental findings.

4.5 Threats to Validity

We identify three primary threats to the validity of our study:

1) **The selection of LMs.** To mitigate this threat, we design specific selection criteria for model choice, as demonstrated in Section 2.2. These models include LLMs of different sizes from various families, which are trained with different pre-training data and learning objectives. Additionally, we have also selected robust small pre-trained models, like CCT5, to thoroughly investigate the impact of fine-tuning on LMs.

2) **Representativeness of automated evaluation metrics.** Though automated evaluation metrics (BLEU-4, ACC, GLEU) are widely used in the field of natural language processing, they may not fully reflect the human perception of the text. To address this limitation, we take the commit message generation task as an example and conduct a human evaluation. The evaluation result is consistent with the automated metrics. We believe that the automated metrics are reliable for evaluating the performance of LLMs on code-change-d tasks.

3) **Uncertainty caused by manual data annotation.** In Section 3.3 and Section 4.1, we randomly selected samples based on previous work [50, 86] and manually labelled them. Subjectivity in human decisions is a potential threat during the manual labeling process. To address this threat, following previous studies [50, 86], each sample is labeled by more than one participant. The disagreement between the participants is resolved through discussion. We believe such results are reliable.

5 RELATED WORK

5.1 LLMs in Software Engineering

Pre-trained models have demonstrated impressive capabilities in the field of natural language processing and have shown their excellent performance on a wide range of applications in various domains [47, 50, 65, 107, 109]. Recently, Large Language Models (LLMs) have been introduced equipped with billions of parameters and billions of training samples [104]. LLMs are pre-trained on large-scale text data and learn rich linguistic knowledge and semantic representations which enable them to understand the meaning and structure of natural language. In the field of software engineering, researchers enhanced the general LLMs on code-related tasks and proposed many domain-specific LLMs, e.g., InCoder[19], Code Llama[82], StarCoder[45], and SantaCoder[1]. A series of studies have experimentally demonstrated that these domain-specific LLMs achieve good performance on code-related tasks, e.g., code completion [96, 106, 108], automatic code generation [23, 42, 102], code understanding [13, 72, 101], and code summarization [34, 44, 85]. However, these models were only pre-trained on the code-related tasks and ignored the code-change-related tasks. Pre-training LLM using code-related tasks focuses on the general syntactic and semantic knowledge of code, while code changes are more concerned with the differences between two code snippets. It is still unclear how these LLMs perform on the code-change-related tasks. To fill the gap, in our work, we explore the capabilities of representative LLMs on code-change-related tasks and show the promising directions of using LLMs on code-change-related tasks.

5.2 Techniques for Code-Change-Related Task

Prior studies proposed a series of approaches for code-changes-related tasks. For example, Jiang et al.[33] adapted Neural Machine Translation (NMT) to automatically "translate" diffs into commit messages. Liu et al.[62] focused on the task of comment update. They leveraged a sequence-to-sequence model to learn comment update patterns from code-comment co-changes. Hoang et al.[28] employed the attention mechanism to model the hierarchical structure of a code change, and used multiple comparison functions to identify the differences between the removed and added code. They evaluated the performance of the proposed approach on log message generation, bug fixing patch identification, and just-in-time defect prediction, and the proposed approach outperformed the state-of-the-art techniques. Recently, pre-trained models have been proposed for code-change-related tasks. Researchers pre-trained a large amount of code change data with code-change-oriented objectives [47, 50, 65, 107, 109].

For example, Lin et al. [50] proposed CCT5, which trained with CodeT5 with 5 kinds of code-change-oriented objectives. Zhou et al. [109] proposed CCBERT, which was pre-trained on four proposed self-supervised objectives that are specialized for learning code change representations based on the contents of code changes. Different from these works that only considered the models with millions of parameters, our work explores the capability of code changes on LLMs with more parameters.

The most related work to our paper is the work of Liu et al. [59]. Specifically, Liu et al. investigated using PEFT on small pre-trained models (<1B parameters) on 2 code change-related tasks, i.e., CMG and Just-In-Time Defect Prediction. Different from their work, we focus on LLM (>1B parameters). Such models have been proposed more recently, and their capability on code change-related tasks has not been systematically explored. Besides, we focus on more code change-related tasks, i.e., CRG, CMG, and JITCU. All of these tasks are generation tasks, which are more challenging than the classification task used in Liu et al. [59]'s work (i.e., Just-In-Time Defect Prediction). Furthermore, besides PEFT, we also explore ICL, which shows promising ability with LLMs (>1B parameters). Our results show that on the CRG task, LLMs with ICL can outperform CodeT5 (a small pre-trained code model fine-tuned on code-change-related tasks), and the best-performing LLM-PEFTs have comparable performance to the state-of-the-art fully fine-tuned small models, indicating that LLMs are promising on code-change-related tasks.

6 CONCLUSION AND FUTURE WORK

In this paper, we conduct an empirical study to explore the capabilities of LLMs on code-change-related tasks. Specifically, we adapt LLMs with in-context learning (ICL) and parameter-efficient fine-tuning (PEFT), respectively, to three code-change-related tasks, i.e., code review generation, commit message generation, and just-in-time comment update. We investigate the effects of multiple factors, such as the number of examples for ICL, and the choice of PEFT methods, and we compare the performance of LLMs with that of small pre-trained models. We also explore the impact of the format of code changes and the impact of code change categories on the performance of LLMs. Experimental results show that LLMs are promising for code-change-related tasks, and the best-performing LLMs are often achieved by tuning Llama 2 or Code Llama using LoRA across model sizes. At the same time, LLMs tend to learn the code change knowledge related to documents. We summarize our findings and provide better suggestions to help practitioners better adapt LLMs to code-change-related tasks. In the future, we will try to explore the effects on the performance of LLMs in code-change-related tasks combined with more aspects of code changes, such as the impact of different code change representation forms and the impact of introducing more knowledge related to refactorings to LLMs.

DATA AVAILABILITY

Our replication package, including the source code and used datasets, is available at: https://github.com/ZJU-CTAG/CodeChange.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China (No. 62202420), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. Zhongxin Liu gratefully acknowledges the support of Zhejiang University Education Foundation Qizhen Scholar Foundation.

REFERENCES

[1] Loubna Ben Allal, Raymond Li, et al. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023.
[2] Shamil Ayupov and Nadezhda Chirkova. Parameter-efficient finetuning of transformers for source code. *CoRR*, abs/2212.05901, 2022.

[3] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. If you use this software, please cite it using these metadata.

[4] Amiangshu Bosu and Jeffrey C Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142. IEEE, 2013.

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

[6] Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *Proceedings of the 25th IEEE/ACM international conference on automated software engineering*, pages 33–42, 2010.

[7] Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. Ernie-code: Beyond english-centric cross-lingual pretraining for programming languages. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 10628–10650. Association for Computational Linguistics, 2023.

[8] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, et al. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. *arXiv preprint arXiv:2403.11202*, 2024.

[9] Guanzheng Chen, Fangyu Liu, Zaiqiao Meng, and Shangsong Liang. Revisiting parameter-efficient tuning: Are we really there yet? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 2612–2626. Association for Computational Linguistics, 2022.

[10] Jiuhai Chen, Lichang Chen, Chen Zhu, and Tianyi Zhou. How many demonstrations do you need for in-context learning? In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

[11] Yifan Chen, Devamanyu Hazarika, Mahdi Namazifar, Yang Liu, Di Jin, and Dilek Hakkani-Tur. Inducer-tuning: Connecting prefix-tuning and adapter-tuning. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 793–808. Association for Computational Linguistics, 2022.

[12] Malinda Dilhara, Danny Dig, and Ameya Ketkar. PYEVOLVE: automating frequent code changes in python ML systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 995–1007. IEEE, 2023.

[13] Hao Ding, Ziwei Fan, Ingo Guehring, Gaurav Gupta, Wooseok Ha, Jun Huan, Linbo Liu, Behrooz Omidvar-Tehrani, Shiqi Wang, and Hao Zhou. Reasoning and planning with large language models in code development. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6480–6490, 2024.

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

[15] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. Fira: fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 970–981, 2022.

[16] Anna Maria Eilertsen. Refactoring operations grounded in manual code changes. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume, Seoul, South Korea, 27 June - 19 July, 2020*, pages 182–185. ACM, 2020.

[17] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers: contributions to software engineering*, pages 575–607. Springer, 2011.

[18] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[19] Daniel Fried, Armen Aghajanyan, Jessy Lin, et al. Incoder: A generative model for code infilling and synthesis. 2023.

[20] Thomas Fritz and Gail C Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 175–184, 2010.

[21] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu. Constructing effective in-context demonstration for code intelligence tasks: An empirical study. *arXiv preprint arXiv:2304.07575*, 2023.

[22] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. 2024.

[23] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[24] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
[25] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study. *arXiv preprint arXiv:2309.08221*, 2023.
[26] Shikai Guo, Xihui Xu, Hui Li, and Rong Chen. Deep just-in-time consistent comment update via source code changes. In *13th IEEE International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2022, Beijing, China, November 25-27, 2022*, pages 1–6. IEEE, 2022.
[27] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879, 2023.
[28] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.
[29] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.
[30] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. Correlating automated and human evaluation of code documentation generation quality. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–28, 2022.
[31] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.
[32] Mia Mohammad Imran, Preetha Chatterjee, and Kostadin Damevski. Uncovering the causes of emotions in software developer communication using zero-shot llms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
[33] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE, 2017.
[34] Pengxiang Jin, Shenglin Zhang, Minghua Ma, Haozhe Li, Yu Kang, Liqun Li, Yudong Liu, Bo Qiao, Chaoyun Zhang, Pu Zhao, et al. Assess and summarize: Improve outage understanding with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1657–1668, 2023.
[35] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21:2072–2106, 2016.
[36] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
[37] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 262–265, 2014.
[38] Philipp Koehn. Statistical significance tests for machine translation evaluation. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 388–395, 2004.
[39] Vladimir Kovalenko and Alberto Bacchelli. Code review for newcomers: is it different? In Helen Sharp, Cleidson R. B. de Souza, Daniel Graziotin, Meira Levy, and David Socha, editors, *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 29–32. ACM, 2018.
[40] Jahnavi Kumar and Sridhar Chimalakonda. Code summarization without direct access to code-towards exploring federated llms for software engineering. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 100–109, 2024.
[41] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.
[42] Jia Li, Ge Li, Chongyang Tao, Huangzhao Zhang, Fang Liu, and Zhi Jin. Large language model-aware in-context learning for code generation. *arXiv preprint arXiv:2310.09748*, 2023.
[43] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. Towards enhancing in-context learning for code generation. *arXiv preprint arXiv:2303.17780*, 2023.
[44] Lixuan Li, Bin Liang, Lin Chen, and Xiaofang Zhang. Cross-modal retrieval-enhanced code summarization based on joint learning for retrieval and generation. *Information and Software Technology*, 175:107527, 2024.
[45] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
[46] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4582–4597. Association for Computational Linguistics, 2021.

[47] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

[48] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023.

[49] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. Automated comment update: How far are we? In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 36–46. IEEE, 2021.

[50] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. CCT5: A code-change-oriented pre-trained model. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1509–1521. ACM, 2023.

[51] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach. *IEEE Transactions on Software Engineering*, 49(4):1640–1660, 2022.

[52] I-H Lin and David A Gustafson. Classifying software maintenance. In *1988 Conference on Software Maintenance*, pages 241–247. IEEE Computer Society, 1988.

[53] Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, et al. Few-shot learning with multilingual generative language models. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 9019–9052. Association for Computational Linguistics, 2022.

[54] Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, et al. Mftcoder: Boosting code llms with multitask fine-tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5430–5441, 2024.

[55] Chia-Wei Liu, Ryan Lowe, Iulian V Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. *arXiv preprint arXiv:1603.08023*, 2016.

[56] Fang Liu, Zhiyi Fu, Ge Li, Zhi Jin, Hui Liu, Yiyang Hao, and Li Zhang. Non-autoregressive line-level code completion. *ACM Transactions on Software Engineering and Methodology*, 2024.

[57] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 48(5):1800–1817, 2020.

[58] Shangqing Liu, Yanzhou Li, and Yang Liu. Commitbart: A large pre-trained model for github commits. *CoRR*, abs/2208.08100, 2022.

[59] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, and Yihan Liao. Delving into parameter-efficient fine-tuning in code change learning: An empirical study. *arXiv preprint arXiv:2402.06247*, 2024.

[60] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 373–384. ACM, 2018.

[61] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. Just-in-time obsolete comment detection and update. *IEEE Trans. Software Eng.*, 49(1):1–23, 2023.

[62] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 585–597, 2020.

[63] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating just-in-time comment updating. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 585–597. IEEE, 2020.

[64] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, pages 287–292. Association for Computational Linguistics, 2017.

[65] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658. IEEE, 2023.

[66] Ziyang Luo, Can Xu, Pu Zhao, et al. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568, 2023.

[67] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[68] Nicholas Meade, Spandana Gella, et al. Using in-context learning to improve dialogue safety. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 11882–11910. Association for Computational Linguistics, 2023.

[69] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *EMNLP*, 2022.
[70] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. *arXiv preprint arXiv:2104.08773*, 2021.
[71] Niklas Muennighoff, Thomas Wang, Lintang Sutawika, et al. Crosslingual generalization through multitask finetuning. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 15991–16111. Association for Computational Linguistics, 2023.
[72] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
[73] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.
[74] Sheena Panthaplackel, Li, et al. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435, 2021.
[75] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
[76] Indraneil Paul, Jun Luo, Goran Glavaš, and Iryna Gurevych. Ircoder: Intermediate representations make language models robust multilingual code generators. *arXiv preprint arXiv:2403.03894*, 2024.
[77] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277*, 2023.
[78] Reid Pryzant, Dan Iter, et al. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 7957–7968. Association for Computational Linguistics, 2023.
[79] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.
[80] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the 38th international conference on software engineering companion*, pages 222–231, 2016.
[81] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018.
[82] Baptiste Roziere, Jonas Gehring, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
[83] Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. RACE: retrieval-augmented commit message generation. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 5520–5530. Association for Computational Linguistics, 2022.
[84] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE, 2020.
[85] Chia-Yi Su and Collin McMillan. Distilled gpt for source code summarization. *Automated Software Engineering*, 31(1):22, 2024.
[86] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. A large-scale empirical study of commit message generation: models, datasets and evaluation. *Empirical Software Engineering*, 27(7):198, 2022.
[87] Yi Tay, Mostafa Dehghani, Samira Abnar, Hyung Won Chung, William Fedus, Jinfeng Rao, Sharan Narang, Vinh Q. Tran, Dani Yogatama, and Donald Metzler. Scaling laws vs model architectures: How does inductive bias influence scaling? In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 12342–12364. Association for Computational Linguistics, 2023.
[88] Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. Scale efficiently: Insights from pre-training and fine-tuning transformers. *arXiv preprint arXiv:2109.10686*, 2021.
[89] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
[90] Rosalia Tufano. Automating code review. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 192–196. IEEE, 2023.
[91] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[92] Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. *arXiv preprint arXiv:2401.14727*, 2024.
[93] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021.
[94] Jason Wei, Yi Tay, et al. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.
[95] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462*, 2023.
[96] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In *Forty-first International Conference on Machine Learning*, 2024.
[97] Yue Wu, Yaoxiang Yu, Zhengming Yuan, Siwei Huang, and Bo Cai. Apt: Adaptive prefix-tuning on pretrained models for code intelligence. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. IEEE, 2024.
[98] Rui Xie, Tianxiang Hu, Wei Ye, and Shikun Zhang. Low-resources project-specific code summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
[99] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. *arXiv preprint arXiv:2312.12148*, 2023.
[100] Sanjay Yadav and Sanyam Shukla. Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. In *2016 IEEE 6th International conference on advanced computing (IACC)*, pages 78–83. IEEE, 2016.
[101] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.
[102] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering*, 2024.
[103] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, et al. Evaluating instruction-tuned large language models on code comprehension and generation. *CoRR*, abs/2308.01240, 2023.
[104] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics.
[105] MAE Zeid, KHALED El-Bahnasy, and SE Abu-Youssef. An efficient optimized framework for analyzing the performance of breast cancer using machine learning algorithms. *Journal of Theoretical and Applied Information Technology*, 100(14):5165–78, 2022.
[106] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023.
[107] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
[108] Mingxuan Zhang, Bo Yuan, Hanzhe Li, and Kangming Xu. Llm-cloud complete: Leveraging cloud computing for efficient large language model-based code completion. *Journal of Artificial Intelligence General science (JAIGS) ISSN: 3006-4023*, 5(1):295–326, 2024.
[109] X. Zhou, B. Xu, D. Han, Z. Yang, J. He, and D. Lo. Ccbert: Self-supervised code change representation learning. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 182–193, Los Alamitos, CA, USA, oct 2023. IEEE Computer Society.
[110] Hongquan Zhu, Xincheng He, and Lei Xu. Hatcup: hybrid analysis and attention based just-in-time comment updating. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 619–630, 2022.

A APPENDIX

Table 22. Performance of LLM-ICL on CMG with diff as input.

Lang	Model	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
Java	InCoder-1b	3.59	2.97	2.8	2.91	2.58	2.86	2.79	3	2.71
	CodeGen-2b-nl	1.9	2.21	2.41	2.54	2.41	2.34	2.21	2.09	2.26
	CodeGen-6b-nl	2.1	2.87	3.28	3.17	3.13	3.07	3.09	3.16	3.11
	Llama-2-7b	2.23	3.48	3.63	3.95	3.97	4.08	4.16	4.04	4.18
	Llama-2-13b	2.42	3.68	3.8	4.58	4.84	5.39	5.75	5.65	6
	CodeLlama-7b	1.78	5.55	5.29	5.32	5.38	5.37	6.07	6.19	6.12
	CodeLlama-13b	2.68	4.44	3.76	3.97	4.01	4.12	4.28	4.98	4.74
Python	InCoder-1b	5.11	3.97	4.12	4.28	4.32	4.3	4.48	3.98	4.19
	CodeGen-2b-nl	2.58	3.23	3.24	3.28	3.26	3.27	3.21	3.24	3.44
	CodeGen-6b-nl	3.4	4.19	4.71	4.43	4.34	4.19	4.24	4.35	4.35
	Llama-2-7b	3.53	4.41	4.59	5.58	6.19	6.37	6.3	6.05	6.33
	Llama-2-13b	4.41	5.12	6.15	6.75	7.17	7.36	7.52	7.28	7.63
	CodeLlama-7b	1.51	6.19	7.28	7.79	8.36	8.39	8.13	8.18	8.07
	CodeLlama-13b	3.63	5.26	6.42	7.28	7.7	7.59	6.95	7.3	6.29
C#	InCoder-1b	3.78	3.75	3.9	4.04	3.82	3.83	3.7	3.53	3.51
	CodeGen-2b-nl	2.02	2.7	2.61	2.47	2.55	2.62	2.59	2.46	2.54
	CodeGen-6b-nl	2.21	3.28	3.57	3.76	3.71	3.48	3.52	3.48	3.45
	Llama-2-7b	3.97	4.26	4.56	5.04	5.42	5.8	5.57	5.19	5.22
	Llama-2-13b	3.11	4.8	5.37	6.05	5.97	5.98	6.18	6.01	6.11
	CodeLlama-7b	1.57	6.32	6.68	6.84	7.05	6.74	6.71	6.89	6.91
	CodeLlama-13b	2.46	4.49	5.42	5.54	6.65	6.54	6.28	6.51	6.21
C++	InCoder-1b	4.31	3.8	3.7	3.95	3.73	4.06	3.61	3.86	3.99
	CodeGen-2b-nl	2.5	3.3	3.39	3.38	3.39	3.29	3.4	3.18	3.15
	CodeGen-6b-nl	2.96	4.16	4.23	4.38	4.48	4.36	4.33	4.19	4.16
	Llama-2-7b	2.93	4.49	4.38	4.4	4.81	4.88	5.01	5.41	5.42
	Llama-2-13b	3.43	4.32	5.19	5.32	5.52	5.92	5.88	6.19	6.05
	CodeLlama-7b	1.18	5.79	6.18	6.16	6.26	6.22	6.3	6.7	6.51
	CodeLlama-13b	2.6	5.29	5.68	5.58	5.89	5.82	5.82	5.87	5.91
Javascript	InCoder-1b	4.12	3.37	3.35	3.25	3.2	3.37	3.32	3.53	4.07
	CodeGen-2b-nl	2.76	3.2	3.23	2.81	2.72	2.69	2.68	2.75	2.86
	CodeGen-6b-nl	3.11	3.23	3.52	3.95	3.94	3.89	3.62	3.71	3.38
	Llama-2-7b	2.79	3.72	3.63	4.1	4.01	4.9	5.59	5.55	6.11
	Llama-2-13b	3.3	4.56	5.24	6.81	5.97	7.05	7.67	7.87	7.77
	CodeLlama-7b	1.43	4.82	6.17	7.38	7.12	7.79	7.81	7.67	7.86
	CodeLlama-13b	2.24	5.29	4.85	4.94	5.18	5.57	6.24	6.84	6.6

Table 23. Performance of LLM-ICL on CMG with code as input.

Lang	Model	0shot	1shot	2shot	3shot	4shot	5shot	6shot	7shot	8shot
Java	InCoder-1b	3.08	3	2.62	2.74	2.66	2.75	2.79	2.76	2.66
	CodeGen-2b-nl	1.88	1.98	2	2.11	2.28	2.29	2.29	2.36	2.33
	CodeGen-6b-nl	2.46	2.58	2.94	3.24	3.19	3.52	3.43	3.39	3.43
	Llama-2-7b	1.13	3.28	3.51	3.88	3.8	3.63	3.54	3.42	3.54
	Llama-2-13b	1.74	3.35	3.91	4.25	4.23	4.57	4.82	4.91	5.13
	CodeLlama-7b	1.81	5.78	5.17	5.86	5.73	5.73	5.74	5.49	5.47
	CodeLlama-13b	1.69	4.47	4.08	4.14	4.34	4.49	5.04	4.61	4.28
Python	InCoder-1b	4.54	3.93	4.23	4.01	3.9	3.88	3.94	4.07	3.92
	CodeGen-2b-nl	2.99	3.09	3.01	3.26	3.28	3.48	3.55	3.57	3.38
	CodeGen-6b-nl	4.08	4.35	4.36	4.71	4.62	4.47	4.55	4.81	4.77
	Llama-2-7b	2.12	4.68	4.93	5.19	5.24	5.31	5.05	4.83	4.64
	Llama-2-13b	3.45	4.6	5.56	6.53	6.71	6.75	6.41	5.67	5.57
	CodeLlama-7b	1.5	5.6	7.03	7.45	8.07	8.54	8.02	7.16	6.6
	CodeLlama-13b	2.31	4.9	6.39	6.75	6.59	7.28	6.62	5.56	5.52
C#	InCoder-1b	3.06	3.62	3.61	3.77	3.83	3.41	3.76	3.38	3.72
	CodeGen-2b-nl	2.09	2.59	2.73	2.88	3.04	2.99	3.07	2.97	2.83
	CodeGen-6b-nl	2.51	3.15	3.48	3.62	3.7	3.78	3.84	3.69	3.67
	Llama-2-7b	1.37	4.28	4.99	5.31	5.13	5.23	4.98	4.85	4.22
	Llama-2-13b	2.06	4.73	5.48	5.76	5.59	6.07	5.84	5.63	5.1
	CodeLlama-7b	1.63	6.12	7.04	7.44	7.13	7.01	6.62	6.66	5.95
	CodeLlama-13b	1.4	4.33	5.08	5.86	6.51	6.62	6.41	5.78	4.69
C++	InCoder-1b	3.55	3.53	3.37	3.45	3.76	3.52	3.77	3.83	3.95
	CodeGen-2b-nl	2.69	3	3.18	3.16	3.21	3.36	3.4	3.52	3.43
	CodeGen-6b-nl	2.93	3.75	3.8	3.81	4.06	4.13	4.05	4.19	4.34
	Llama-2-7b	2.01	4.41	4.23	4.89	5.15	5.15	4.84	4.54	4.55
	Llama-2-13b	2.62	4.3	5.01	5.63	5.59	5.76	5.74	5.27	5.53
	CodeLlama-7b	1.24	5.12	5.74	6.11	6.39	6.45	6.29	5.74	5.98
	CodeLlama-13b	2.09	5.03	5.46	6.42	6.47	6.18	6.13	5.69	5.61
Javascript	InCoder-1b	3.21	3.06	3.44	3.34	3.21	3.47	3.82	3.67	3.83
	CodeGen-2b-nl	3.09	3.42	3.27	3.16	3.19	3.55	3.28	3.35	3.49
	CodeGen-6b-nl	3.29	3.73	3.93	3.78	3.93	3.77	3.89	3.84	3.83
	Llama-2-7b	1.3	3.74	3.95	4.21	4.15	4.83	4.28	4.58	4.25
	Llama-2-13b	2.22	4.4	4.99	5.75	6.2	6.94	6.31	6.41	6.08
	CodeLlama-7b	1.52	3.98	7.1	7.88	8.17	8.77	8.02	7.82	7.7
	CodeLlama-13b	1.39	4.09	4.38	5.04	5.38	6.19	5.7	5.45	5.65