LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning

Junyi Lu[†‡], Lei Yu[†‡], Xiaojia Li[§], Li Yang[*†], Chun Zuo[¶]
[†] Institute of Software, Chinese Academy of Sciences, Beijing, China
[‡]University of Chinese Academy of Sciences, Beijing, China
[§]School of Software, Tsinghua University, Beijing, China [¶]Sinosoft Company Limited, Beijing, China
{lujunyi21, yulei21}@mails.ucas.ac.cn, lixj21@mails.tsinghua.edu.cn,
yangli2017@iscas.ac.cn, zuochun@sinosoft.com.cn

***Abstract*—The automation of code review activities, a long-standing pursuit in software engineering, has been primarily addressed by numerous domain-specific pre-trained models. Despite their success, these models frequently demand extensive resources for pre-training from scratch. In contrast, Large Language Models (LLMs) provide an intriguing alternative, given their remarkable capabilities when supplemented with domain-specific knowledge. However, their potential for automating code review tasks remains largely unexplored.**

In response to this research gap, we present LLaMA-Reviewer, an innovative framework that leverages the capabilities of LLaMA, a popular LLM, in the realm of code review. Mindful of resource constraints, this framework employs parameter-efficient fine-tuning (PEFT) methods, delivering high performance while using less than 1% of trainable parameters.

An extensive evaluation of LLaMA-Reviewer is conducted on two diverse, publicly available datasets. Notably, even with the smallest LLaMA base model consisting of 6.7B parameters and a limited number of tuning epochs, LLaMA-Reviewer equals the performance of existing code-review-focused models.

The ablation experiments provide insights into the influence of various fine-tuning process components, including input representation, instruction tuning, and different PEFT methods. To foster continuous progress in this field, the code and all PEFT-weight plugins have been made open-source.

***Index Terms*—Code Review Automation, Large Language Models (LLMs), Parameter-Efficient Fine-Tuning (PEFT), Deep Learning, LLaMA, Software Quality Assurance**

I. Introduction

Since its formalization by Fagan in 1976 [1], code review has been a cornerstone of software engineering, instrumental in defect identification, quality improvement, and knowledge sharing [2]. However, this primarily manual process imposes a significant workload on developers. Even with modern code review (MCR) practices, which are more streamlined than traditional ones, the effort required is still substantial [3]–[5].

To alleviate this burden, a surge of research has focused on automating the code review process. This includes tasks such as recommending reviewers [6]–[15], evaluating code quality [12], [16]–[21], refining problematic code [20], [22]–[25], and generating potential review comments [20], [23], [26]–[31]. Recent advancements in natural language processing (NLP) have further enabled the use of pre-trained language models (PLMs) for these tasks [20], [23]. However, such domain-specific models often require substantial resources for pre-training from scratch.

[*]Corresponding author.

In contrast, unified large language models (LLMs) demonstrate remarkable performance when scaled to a certain parameter size [12], [13]. They can effectively handle specific tasks without the need for domain-specific pre-training, presenting a promising avenue for code review automation.

In this study, we present LLaMA-Reviewer, a novel framework that leverages LLaMA, a mainstream LLM, for automating code review. We incorporate Parameter-Efficient Fine-Tuning (PEFT) methods to address the computational challenge of LLM fine-tuning. Our approach builds upon the pipeline proposed by Li et al. [20], which comprises 1) review necessity prediction, 2) review comment generation, and 3) code refinement tasks.

We extensively evaluate LLaMA-Reviewer on two public datasets for each sub-task and investigate the impacts of the input representation, instruction tuning, and different PEFT methods. The primary contributions of this work include:

- Introducing the application of LLMs to code review automation tasksm, offering an offline and privacy-conscious alternative to closed-source solutions like OpenAI APIs.
- Proposing a “unified model + PEFT” paradigm to reduce computational demands during code review tasks, with the plug-in model being part of it to optimize storage space requirements, first in software engineering domain.
- Conducting a comprehensive evaluation of two PEFT methods and ablation studies on fine-tuning components.
- Open-sourcing our code, models, and results [32].

Here is the paper’s structure: Section II gives the necessary background; Section III details our proposed approach; Section IV describes the experiment design; Section V discusses evaluation results; Section VI reviews related work; Section VII identifies potential validity threats; Section VIII concludes our findings and proposes future research directions.

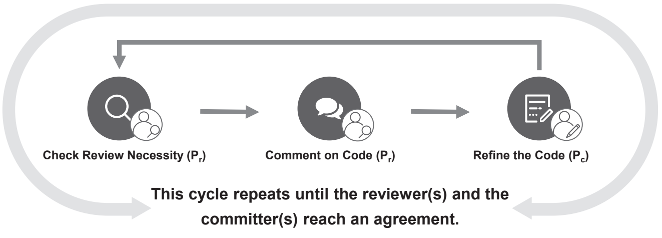

Check Review Necessity (P_r)
Comment on Code (P_r)
Refine the Code (P_c)
This cycle repeats until the reviewer(s) and the committer(s) reach an agreement.

Fig. 1. The cycle of the code review process.

II. Background

This section provides a succinct introduction to three key concepts underlying our research: the automated code review pipeline, large language models (LLMs), and parameter-efficient fine-tuning (PEFT) methods.

A. Automation in Code Review

Modern Code Review (MCR), a technique extensively adopted by both large enterprises and open-source projects, has a relatively consistent core cycle despite varied implementations. This cycle, from the creation of a pull request to its final merge into the main branch or rejection, involves two primary participants: the committers (P_c) and reviewers (P_r). The cycle encompasses three key steps (as shown in Figure 1): review necessity prediction (P_r), commenting on code (P_r), and code refinement (P_c). The objective of automating the code review process is to alleviate the workload for both parties.

The three steps translate into three automation tasks: **1) Review Necessity Prediction**, predicting whether a code diff requires review comments; **2) Review Comment Generation**, automatically generating review comments for a given code diff; and **3) Code Refinement**, refining the code based on prior snippets and review comments. Our study focuses on these tasks, aiming to fully automate the code review process.

B. Large Language Models

The evolution of language modeling (LM) has seen four significant stages: statistical language models (SLMs), neural language models (NLMs), pre-trained language models (PLMs), and the latest development, large language models (LLMs) [33]. PLMs, explicitly pre-trained for certain tasks, have been notably successful in various downstream software engineering tasks. This is demonstrated by models like CodeT5 [34] and PLBART [35]. However, the potential of LLMs in these contexts has yet to be fully explored.

The primary distinction between PLMs and LLMs is their scale of parameters and data size. LLMs are models with $\sim$10B parameters or more, pre-trained with extensive data [33]. Current research suggests that scaling these dimensions improves model performance and gives rise to emergent capabilities [36]. Notably, LLMs can achieve performance on par with PLMs without the necessity for task-specific pre-training, thus alleviating the resource-heavy demands of pre-training.

To be more specific, the currently trending LLMs can be categorized into unified LLMs and code LLMs. The formers are predominantly pre-trained on natural language corpora, enriched with a smaller portion of code, and have been validated as effective in various tasks [37], [38]. The latters are primarily pre-trained on a code-based corpus and they have achieved impressive results in code generation [39]–[42].

In this research, we utilize LLaMA, an open-source unified LLM developed by Meta. This choice stems from four perspectives: 1)Top-performing models (GPT-3.5/GPT-4) for code tasks are unified models, not code LLMs; 2)The increasing trend towards unified models, exemplified by OpenAI's transition from Codex to GPT-3.5 for API usage; 3) The increasing trend towards unified models, exemplified by OpenAI's transition from Codex to GPT-3.5 for API usage; 4) Code LLMs primarily excel in code generation tasks, whereas code review tasks pose different challenges.

C. Parameter-Efficient Fine-Tuning

Despite their effectiveness, the high computational resources required for fine-tuning large language models (LLMs) present a significant challenge. Numerous strategies have been developed to increase the efficiency of the fine-tuning process and lower training costs. These include adapter tuning [43], [44], prefix tuning [45], prompt tuning [46], [47], and low-rank adaptation (LoRA) [48]. These methods freeze the base model's parameters while training a few additional parameters, achieving performance comparable to full-parameter tuning.

In this study, we use two PEFT methods—zero-init attention prefix-tuning [49] and LoRA tuning [48]—to fine-tune LLaMA. These methods do not introduce extra latency to the model and have demonstrated effectiveness in natural language tasks. The PEFT methods' specifics are further elaborated in the subsequent section on our proposed approaches.

III. LLaMA-Reviewer: Proposed Approach

A. Overview

Our framework, illustrated in Figure 2, employs a dual-stage fine-tuning process. We initiate with instruction-following tuning on LLaMA using code-centric domain data, enhancing the model's proficiency in comprehending code review tasks and adhering to task instructions. We then conduct supervised fine-tuning for each sub-task within the code review process using the enhanced LLaMA. To balance parameter efficiency and model performance, we incorporate two key techniques—zero-init attention prefix-tuning and low-rank adaptation (LoRA) tuning—since they have achieved wide acceptance and promising results, especially with LLaMA [49], [50]. These methods, founded on the plugin encapsulation strategy of PEFT, furnish us with lightweight task-specific plugins. These plugins, independent of the base model's weights, can be seamlessly integrated during inference.

We evaluate the effectiveness of our approach using two distinct datasets. For clarity, we will refer to the dataset in CodeReviewer [20] as the "CRer dataset", and the dataset from Tufano et al. [23] as the "Tuf. dataset". Further details on the evaluation process can be found in the subsequent sections.

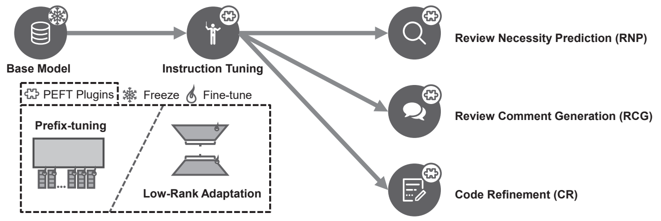


Fig. 2. The overview of LLaMA-Reviewer.

B. *Instruction Tuning on LLaMA*

Research indicates that when LLMs are fine-tuned on a diverse range of multi-task datasets using natural language descriptions, they demonstrate enhanced performance on unseen tasks [51], [52]. Instruction-following tuning aids the model in better interpreting user intentions and following instructions. To initially adapt the pre-trained model for code review tasks, we employ instruction tuning on a code-centric domain.

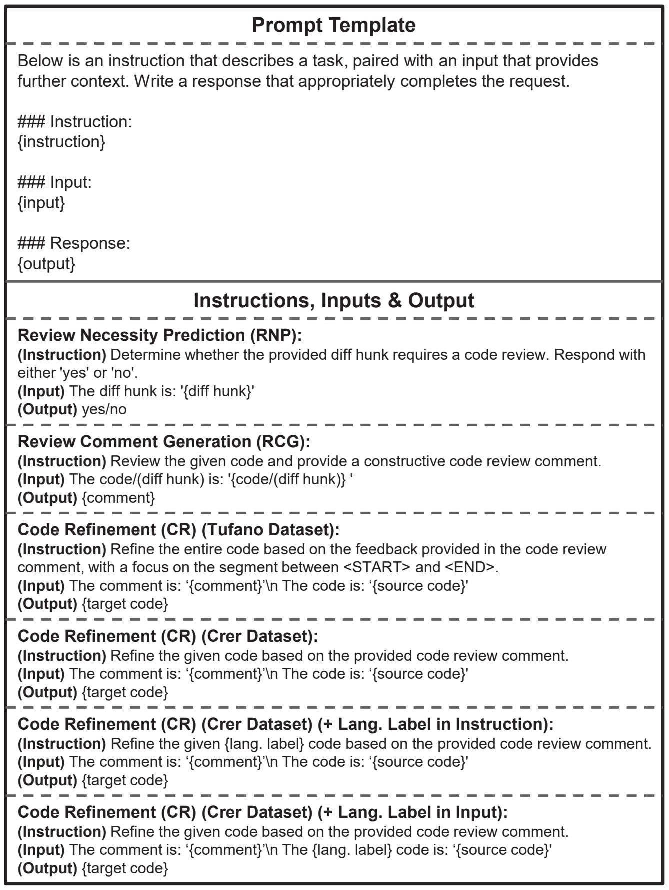
We leverage the primary process and template from Stanford Alpaca [53], modifying the full-parameter fine-tuning to use PEFT methods as explained in Section III-C and Section III-D. Given our code review task's deep relevance to coding, we substitute the original data with its code-domain equivalent, Code Alpaca [54]. Combining data from Alpaca and Code Alpaca was considered but did not lead to performance improvements, as further discussed in the ablation experiments section. The data structure adheres to the {instruction, input (optional), output} format, following the framework from [55]. The same prompt template is used for subsequent sub-tasks to maximize the use of the first stage fine-tuned model. Figure 3 illustrates the prompt template and sub-task instruction, input, and output formats.

C. *Zero-init Attention Prefix-tuning*

Zero-init attention prefix-tuning, an offshoot of prefix-tuning, keeps the base model's weights intact while integrating an extra K prefix tokens into the topmost L layers of LLaMA's transformer. These flexible prompts are concatenated with the original tokens, allowing the computation of multi-head attention using the newly introduced keys and values. During the fine-tuning, only these adaptable prompts are trained.

This method deviates from conventional prefix-tuning by introducing a learnable gating factor during attention calculation. This factor regulates the relevance of the inserted prompt tokens. To facilitate the tuning process, this gating factor is initially zeroed, leading to the 'zero-init' in the name of the method. This factor specifically controls the attention shared between the prompt tokens and the original tokens.

Figure 4 illustrates the details. We take the l layer, one of the topmost L layers as an example. Here, $P_l \in \mathbb{R}^{K\times C}$ represents the K prompt tokens and $T_l \in \mathbb{R}^{M\times C}$ signifies the M original tokens in the l layer. The feature dimension is symbolized by C. When the $(M+1)$-th token t_{M+1} is computed via the attention mechanism, the queries, keys, and values are obtained through several linear layers as follows:



Prompt Template

Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

Instruction:
{instruction}

Input:
{input}

Response:
{output}

Instructions, Inputs & Output

Review Necessity Prediction (RNP):
(Instruction) Determine whether the provided diff hunk requires a code review. Respond with either 'yes' or 'no'.
(Input) The diff hunk is: '{diff hunk}'
(Output) yes/no

Review Comment Generation (RCG):
(Instruction) Review the given code and provide a constructive code review comment.
(Input) The code/(diff hunk) is: '{code/(diff hunk)} '
(Output) {comment}

Code Refinement (CR) (Tufano Dataset):
(Instruction) Refine the entire code based on the feedback provided in the code review comment, with a focus on the segment between <START> and <END>.
(Input) The comment is: '{comment}'\n The code is: '{source code}'
(Output) {target code}

Code Refinement (CR) (Crer Dataset):
(Instruction) Refine the given code based on the provided code review comment.
(Input) The comment is: '{comment}'\n The code is: '{source code}'
(Output) {target code}

Code Refinement (CR) (Crer Dataset) (+ Lang. Label in Instruction):
(Instruction) Refine the given {lang. label} code based on the provided code review comment.
(Input) The comment is: '{comment}'\n The code is: '{source code}'
(Output) {target code}

Code Refinement (CR) (Crer Dataset) (+ Lang. Label in Input):
(Instruction) Refine the given code based on the provided code review comment.
(Input) The comment is: '{comment}'\n The {lang. label} code is: '{source code}'
(Output) {target code}

Fig. 3. The prompt template and instruction, input and output formats.

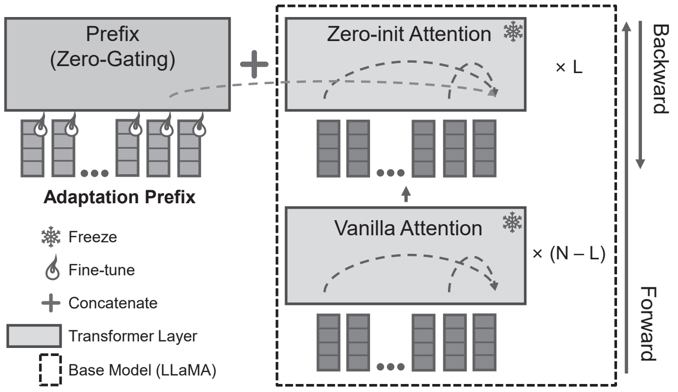


Fig. 4. The details of prefix-tuning on LLaMA.

$$
\begin{aligned}
Q_l &= \mathrm{Linear}_{\mathrm{q}}\left(t_{M+1}\right)\\
K_l &= \mathrm{Linear}_{\mathrm{k}}\left([P_l; T_l; t_{M+1}]\right)\\
V_l &= \mathrm{Linear}_{\mathrm{v}}\left([P_l; T_l; t_{M+1}]\right)
\end{aligned}
\tag{1}
$$

Next, the attention scores are derived:

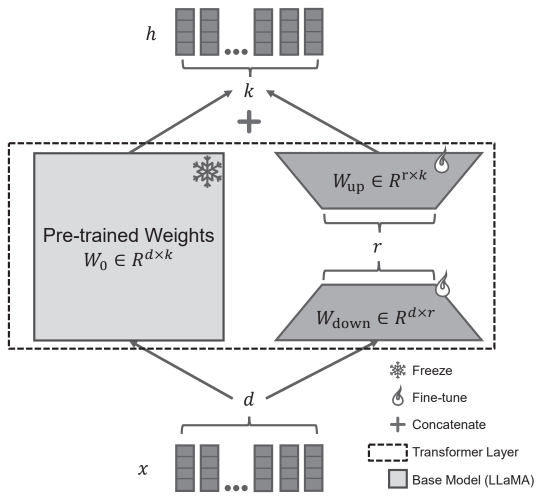


Fig. 5. The core component of Low-Rank Adaptation (LoRA).

$$\begin{aligned} S_l &= \left[S_l^K; S_l^{M+1}\right]^T \\ S_l^K &= Q_l \left(K_l^K\right)^T / \sqrt{C} \in \mathbb{R}^{1 \times K} \\ S_l^{M+1} &= Q_l \left(K_l^{M+1}\right)^T / \sqrt{C} \in \mathbb{R}^{1 \times (M+1)} \end{aligned} \tag{2}$$

The gating factor is integrated post-softmax application:

$$S_l^g = \left[\text{Softmax}\left(S_l^K\right) \cdot g_l; \text{Softmax}\left(S_l^{M+1}\right)\right]^T \tag{3}$$

Ultimately, the output of the t_{M+1} token is produced via a linear projection layer, and it is controlled by prefixes:

$$t_{M+1}^o = \text{Linear}_o\left(S_l^g V_l\right) \in \mathbb{R}^{1 \times C}. \tag{4}$$

D. Low-Rank Adaptation

Low-Rank Adaptation (LoRA) provides a different perspective on Parameter-Efficient Fine-Tuning (PEFT). Unlike full-parameter tuning methods that require all weights to be updated during the fine-tuning phase, LoRA retains the weights of the original model and integrates trainable low-rank matrices to the transformer layers to simulate the weight adjustments. This approximation draws on the principle that the adaptation process inherently has a low "intrinsic rank."

Figure 5 illustrates the core component of LoRA. Suppose $W_0 \in \mathbb{R}^{d \times k}$ represents the pre-trained matrix. The approximation of the weight adjustment from W_0 to $W_0 + \Delta W$ using LoRA can be expressed as:

$$W_0 + \Delta W = W_0 + W_{down} W_{up} \tag{5}$$

Here, $W_{down} \in \mathbb{R}^{d \times r}$ and $W_{up} \in \mathbb{R}^{r \times k}$, with $r \ll \min(d, k)$ being the rank. During the fine-tuning process, W_0 remains unchanged, whereas W_{down} and W_{up} become the trainable parameters. Given an input x and its associated original output h, the adjusted output $\bar{h}$ is computed as:

$$\bar{h} = W_0 x + \Delta W x = h + W_{down} W_{up} x \tag{6}$$

TABLE I
SUMMARY OF THE CODE REVIEW AUTOMATION TASKS

Task	Input	Output	Datasets
Review Necessity Prediction	PL	L (yes/no)	Crer.
Code Review Comment Generation	PL	NL	CRer., Tuf.
Code Refinement	PL, NL	PL	CRer., Tuf.

E. Code Review Automation Tasks

LLaMA-Reviewer is specifically designed to automate three core tasks integral to the code review process, namely review necessity prediction, code review comment generation, and code refinement. These tasks sequentially correspond to stages in a typical code review process. Inputs and outputs for these tasks are categorized into three formats:

- Programming Language (PL) for code snippets,
- Natural Language (NL) for code comments, and
- Binary Labels (L) for decisions on the requirement for further review. This simplifies the decision process to a "yes" (review needed) or "no" (review not needed).

Table I illustrates each task along with its input and output formats, and corresponding dataset references (Crer. for the CRer dataset, and Tuf. for the Tufano dataset). The prompt construction approach is visualized in Figure 3.

1) Review Necessity Prediction: This task involves checking if diff hunks need reviews. Proposed by Li et al. [20], a diff hunk represents a concise piece of code that shows differences between old and new code snippets. Although including additional method context could be beneficial, the lines in the original diff hunk are typically long enough to present a challenge when managed as input.

2) Code Review Comment Generation: This task generates pertinent comments for a given code snippet. Two perspectives are considered: a line-level perspective that focuses on the content of individual lines of code (using the CRer dataset), and a method-level perspective that provides a more holistic view of the code's context (using the Tufano dataset).

3) Code Refinement: Code refinement entails making minor adjustments or reordering the existing code to enhance its quality. Given the nature of these minor modifications, the input and output codes often bear strong similarities. Inputs are formatted according to the Tufano and CRer datasets. Typically, We omitted language type information as the baseline models did so and it is only available in one task of one published dataset, to ensure a fair comparison. The implications of such information are further explored in our evaluation section.

IV. EXPERIMENTAL DESIGN

This section details our experimental design, outlining the fundamental research questions driving our investigation, the datasets employed, the evaluation metrics used, and a thorough summary of the baseline models and our implementation specifics.

A. Research Questions

To evaluate the effectiveness of our proposed framework, we posit the following research queries:

(RQ1) How effective is a large language model in automating code review tasks, compared to state-of-the-art methods?

Motivation. The rapid advancements in AI-Generated Content (AIGC) and the known correlation between model capabilities and their size have led to the widespread use of fine-tuned, pre-trained large language models (LLMs). Although programming language data plays a significant role in augmenting a model's capabilities, the application of such data to code-related tasks—particularly those demanding proficiency in both natural language (NL) and programming language (PL), such as automated code review—remains largely unexplored. In this study, we employ the smallest variant of LLaMA as our base large language model to assess its effectiveness in automating code review tasks compared to state-of-the-art methods, notably task-specific pre-trained models like CodeReviewer [20]. The model's performance is scrutinized across each task to identify its strengths and areas that warrant enhancement. Specifically, we pose the following questions:

- (RQ1.1) How effective is a large language model at **checking review necessity** (classification)?
- (RQ1.2) How proficient is a large language model at **generating code review comments** (NL generation)?
- (RQ1.3) How capable is a large language model at **refining code** based on comments (PL generation)?

(RQ2) How does the representation of input data impact the performance of large language models?

Motivation. Given the fixed pre-training data format, there may be discrepancies between this format and that required for specific tasks. To evaluate the capabilities of the large language model, we delve into two critical factors:

1) Code Formatting. We evaluate the model's performance on raw code input and input with modified formatting (including eradicating consecutive spaces) to determine which can be handled more effectively.
2) Programming Language Labels. With various programming languages, we examine the impact of specifying the language type in the input and the importance of the label's location within the prompt template.

In light of these considerations, we pose two sub-questions:

- (RQ2.1) How does code formatting influence the model's performance?
- (RQ2.2) How do the inclusion and placement of programming language labels affect the model's performance?

(RQ3) How does instruction tuning influence the performance of subsequent sub-tasks?

Motivation. Instruction tuning, the inclusion of code-related instructions in the initial stage, aims to infuse domain knowledge and help the model understand sub-tasks better. The effectiveness of this approach and its compatibility with various parameter-efficient fine-tuning (PEFT) methods remains a rich area of exploration. Additionally, given the dual nature of code review tasks involving natural language (NL) and programming language (PL), a comparison between using exclusively PL-related instructions and a mix of NL and PL-related instructions is warranted. Consequently, we ask:

- (RQ3.1) What is the impact of the initial instruction tuning stage on zero-init attention prefix-tuning?
- (RQ3.2) How does the initial instruction tuning stage influence low-rank adaptation (LoRA)?
- (RQ3.3) Which approach yields superior results in code review tasks: using a mix of NL and PL-related instructions or relying solely on PL-related instructions?

(RQ4) What implications arise from different parameter-efficient fine-tuning (PEFT) methods?

Motivation. Two PEFT methods are employed to alleviate computational resource demands during fine-tuning. However, achieving an optimal balance between efficiency and effectiveness is challenging. Accordingly, we intend to analyze the results obtained from these two methods. Furthermore, in the context of low-rank adaptation (LoRA), the rank r determines the number of trainable parameters; thus, we perform an ablation study to investigate the impact of rank r. Lastly, we compare the two methods concerning parameter count and storage space requirements with prior methods. To this end, we explore the following sub-questions:

- (RQ4.1) Which PEFT method performs better, and why?
- (RQ4.2) Within LoRA, how does the rank r influence trainable parameters and the overall performance?
- (RQ4.3) How do the two PEFT methods compare to previous approaches in terms of parameter efficiency and storage space requirements?

B. Datasets

We utilize two prominent code review datasets: the dataset from CodeReviewer of Li et al. [20] (hereafter the CRer dataset), and the dataset from Tufano et al. [23] (hereafter the Tufano dataset). The rationale for our choice includes:

- Unlike other datasets in the literature [30], [56] that cover specific sub-tasks, the chosen datasets encompass the entire code review process.
- Both the CRer and Tufano datasets are derived from a diverse range of repositories, providing broad coverage. This contrasts with other datasets [30], [56] that draw from a limited repository pool, potentially leading to bias due to their restricted scope.
- The Tufano dataset is an enhanced version compared to those used in earlier studies [22], [25], [57], making it preferable for its recency and comprehensiveness.
- Both datasets have been seminal in the field, each offering unique features that contribute to our study.

The CRer dataset, a multi-language corpus, is drawn from GitHub repositories and adheres to a diff-aware, line-grained format. It preserves inline comments and docstrings within code snippets and retains consecutive spaces. This dataset is divided into three sub-datasets, each dedicated to a specific aspect of code review: review necessity prediction, code review comment generation, and code refinement.

The Tufano dataset, in contrast, is language-specific (Java) and aggregates data from both GitHub and Gerrit. It uses a function-grained format, removes comments and consecutive spaces, and does not reflect differences between the associated commit and the base branch. For code refinement tasks, it denotes areas of focus within the comments using "⟨START⟩" and "⟨END⟩" markers. We utilize two subsets of this dataset for code review comment generation and code refinement.

Table II offers a detailed statistical summary of datasets.

C. Evaluation Criteria

We employ task-specific metrics to gauge the performance of our model across the code review tasks.

For review necessity prediction, we approach it as a binary classification problem where 'requiring a review' is the positive class. Thus, we use precision, recall, and F1-score as evaluation metrics to quantify the model's categorization accuracy.

For the code review comment generation and code refinement tasks, which involve response generation, we use the BLEU-4 score, which measures the overlap of n-grams for n ranging from 1 to 4. This follows the evaluation method employed in CodeReviewer [20].

We do not use the codeBLEU metric proposed by Tufano et al. [23], due to its incompatibility with the CRer dataset. The CRer dataset's structure and language variety make it unsuitable for this metric.

For all tasks, we consider the top-1 result, aligning with our objective of automating the code review process to lighten developers' workload by focusing on the most relevant feedback.

D. Baselines

We chose our baselines based on the tasks and datasets' unique needs. Table III displays the chosen baselines.

We omitted the studies of [25], [56], [57] from our baseline selection, as they are designed for smaller datasets or require additional input information.

E. Implementation Details

We implemented our approach using the xturing[1] and Lit LLaMA[2] frameworks, respectively. All experiments were conducted on NVIDIA A100-SXM4-80GB GPU platforms, with the token length limit set to 2048 and a batch size of 64. We used the AdamW optimizer and trained the models for 5 epochs for review necessity prediction and 10 epochs for both code review comment generation and code refinement tasks.

For zero-init attention prefix-tuning, we used a learning rate of 0.009, weight decay of 0.02, prefix prompt length of 10, and a prefix layer of 30. In Low-rank Adaptation (LoRA), we set the learning rate to 0.0003, weight decay to 0.01, LoRA rank to 16, and the LoRA scaling factor to 16. Ablation experiment settings may vary based on each experiment's requirements. The LoRA rank and prefix-tuning setting are based on empirical experience [49], [50]. More details on the specific hyper-parameters are available in our materials.

[1]https://github.com/stochasticai/xturing

[2]https://github.com/Lightning-AI/lit-llama-main

Baseline implementation was tailored for each situation. For the CRer dataset results, we used the findings reported in the CodeReviewer paper [20], as we made no modifications to the dataset. We reproduced the CommentFinder [31] results on both datasets using their publicly available code. All other baseline results were derived from their provided models.

V. Evaluation

In this section, we sequentially address each research question, presenting the results obtained from our experiments and drawing conclusions for each RQ. Our discussion begins with an assessment of LLaMA-Reviewer's performance, followed by an exploration of input representation's influence and the initial stage of instruction tuning. We conclude with an analysis of the implications arising from different parameter-efficient fine-tuning (PEFT) methods.

A. RQ1: Evaluating the Performance of LLaMA-Reviewer

This subsection assesses LLaMA-Reviewer's performance across each task, explains the observed results, and summarises the conclusions as key findings.

1) (RQ1.1) Review Necessity Prediction Performance: In our review necessity prediction experiments, we focused exclusively on the CRer dataset, as it is the only dataset that provides the necessary data for this task. The results are presented in Table IV, with the last row displaying the outcomes for LLaMA-Reviewer. The preceding rows show results derived from [20]. We consider the class requiring a review as positive. Importantly, the results for LLaMA-Reviewer with prefix-tuning are not included in this task due to its rigid training structure, which is not conducive to performing classification tasks.

LLaMA-Reviewer achieves superior recall with a comparable F1 score, as the results show, indicating that it can identify a larger number of problematic code snippets potentially prompting discussion in the subsequent code review process. This capability is crucial for reviewers as the primary objective of code review is to uncover as many potential issues as possible. In this scenario, LLaMA-Reviewer can reduce the number of code snippets that need review after filtering, without overlooking a significant number of problematic snippets.

We obtained these results through threshold adjustment. Furthermore, with a threshold of 0.5, identical to the original generation setting, LLaMA-Reviewer achieves a precision of 88.61%, exceeding all baselines. This performance is also meaningful in real-world scenarios, where problematic code snippets are less common than normal ones, indicating that false positives can place an additional burden on reviewers.

2) (RQ1.2) Review Comment Generation Performance: We evaluated the code review comment generation task using both the Tufano and CRer datasets. Table V illustrates the results, with the symbol "–" signifying missing values. It is noteworthy that CommentFinder [31] does not include parameter numbers as it does not utilize a deep learning method. We have

TABLE II
STATISTICAL OVERVIEW OF THE TUFANO DATASET AND THE CRER DATASET.

Dataset	Review Necessity Prediction			Review Comment Generation			Code Refinement			Lang #	Gran.	Indent. & Consec. Spaces	Diff. & Comm.
	Train #	Valid #	Test #	Train #	Valid #	Test #	Train #	Valid #	Test #				
Tufano	–	–	–	∼134k	∼17k	∼17k	∼134k	∼17k	∼17k	1	Func.	✘	✘
Crer	∼226k	∼31k	∼31k	∼118k	∼10k	∼10k	∼150k	∼13k	∼13k	9	Line.	✔	✔

TABLE III
SUMMARY OF THE BASELINES.

Baseline (Description)	**Tasks**	**Datasets**	**References**
Transformer-s (6-layer encoder/decoder, trained from scratch)	RCG, CR	Tuf.	[23], [58]
Transformer-b (12-layer encoder/decoder, trained from scratch)	RNP, RCG, CR	CRer	[20], [58]
Tufano et al. (Pre-trained Transformer-s on their datasets)	RCG, CR	CRer, Tuf.	[23]
CodeT5 (Pre-trained Transformer-b model for code understanding and generation)	RNP, RCG, CR	CRer	[20], [34]
CodeReviewer (Pre-trained with Transformer-b and specific code-review-related pre-training tasks)	RNP, RCG, CR	CRer	[20]
CommentFinder (Retrieval-based review comment recommendation)	RCG	CRer, Tuf.	[31]
AUGER (Re-pre-trained T5 with extra review tags input)	RCG	Tuf.	[30]

Task abbreviations: RNP (Review Necessity Prediction), RCG (Review Comment Generation), CR (Code Refinement).
Dataset abbreviations: CRer (CRer dataset), Tuf. (Tufano dataset).

TABLE IV
RESULTS OF REVIEW NECESSITY PREDICTION ON CRER DATASET.

Model	Layers	Model Params	Trainable Params	Storage Space	Prec.	Recall	F1
Transformer-b	24	∼220M	∼220M	850M	74.50	46.07	56.93
Tufano et al.	12	∼60M	∼60M	231M	70.82	57.20	63.29
CodeT5	24	∼220M	∼220M	850M	70.36	58.96	64.16
CodeReviewer	24	∼220M	∼220M	850M	78.60	65.63	71.53
LLaMA-Reviewer (LoRA)	**32**	**∼6.7B**	**∼8.4M**	**16M**	**60.99**	**83.50**	**70.49**

TABLE V
RESULTS OF REVIEW COMMENT GENERATION.

Model	L.	Model Params	Trainable Params	Storage Space	BLEU-4	
					Crer.	Tuf.
Transformer-s	12	∼60M	∼60M	231M	–	6.94*
Transformer-b	24	∼220M	∼220M	850M	4.76	–
Tufano et al.	12	∼60M	∼60M	231M	4.39	7.39*
CodeT5	24	∼220M	∼220M	850M	4.83	–
CodeReviewer	24	∼220M	∼220M	850M	5.32	–
CommentFinder	–	–	–	∼100M	3.82*	4.19*
AUGER	24	∼220M	∼220M	850M	–	3.03*
Ours (Prefix)	**32**	**∼6.7B**	**∼1.2M**	**2.4M**	**5.16**	**4.66**
Ours (LoRA)	**32**	**∼6.7B**	**∼8.4M**	**16M**	**5.70**	**5.04**

* Denotes results achieved by their given code or model.

opted not to report certain baseline results due to incongruity in granularity between the proposed methods and the dataset, which makes the results meaningless.

The results indicate that LLaMA-Reviewer surpasses all baselines on the CRer dataset, especially when employing Low-Rank Adaptation (LoRA) for fine-tuning. This superior performance highlights the potential of large language models (LLMs). Even though LLaMA wasn't specifically pre-trained for code review tasks like CodeReviewer [20], its superior performance with a limited amount of tuning outstrips that of smaller models. The results on the Tufano dataset are relatively less ideal, which we will further discuss in RQ2.1.

A plausible explanation for this enhanced performance is the congruence between the task of natural language generation and LLaMA's pre-training corpus. Furthermore, the impressive performance on the CRer dataset could be attributed to the use of code differences and the raw code format, mirroring the conditions of the pre-training stage. Given the complexity of the code review comment generation task in comparison to the other tasks [20], the larger model size of LLaMA provides a distinct advantage.

3) (RQ1.3) Code Refinement Performance: We evaluated the code refinement task on both the Tufano and CRer datasets. The results are shown in Table VI, where the symbol "–" signifies missing values.

On both datasets, despite not outperforming all models, LLaMA-Reviewer competes closely with CodeReviewer [20] or Tufano et al. [23], the models pre-trained specifically for code review and the corresponding data format and exceeds the performance of the other baselines. Considering that we used the smallest version of LLaMA and limited tuning epochs, this outcome suggests potential improvements.

The LLaMA-Reviewer's advantage over most baselines primarily arises from its large model size and the nature of the pre-training data. The gap between LLaMA-Reviewer and

TABLE VI
RESULTS OF CODE REFINEMENT.

Model	L.	Model Params	Trainable Params	Storage Space	BLEU-4	
					Crer.	Tuf.
Transformer-s	12	∼60M	∼60M	231M	–	77.54*
Tufano et al.	12	∼60M	∼60M	231M	77.03	78.33*
CodeT5	24	∼220M	∼220M	850M	80.82	–
CodeReviewer	24	∼220M	∼220M	850M	82.61	–
Ours (Prefix)	**32**	**∼6.7B**	**∼1.2M**	**2.4M**	**76.71**	**77.04**
Ours (LoRA)	**32**	**∼6.7B**	**∼8.4M**	**16M**	**82.27**	**78.23**

* Denotes results achieved by their given code or model.

TABLE VII
ROLE OF LANGUAGE LABEL (LORA $r = 8$).

Model	Lang. Label	Placement	BLEU-4
LLaMA-Reviewer (LoRA) w/o Instruction Tuning	✘	–	**81.87**
	✔	Instruction	81.07
	✔	Input	81.33
LLaMA-Reviewer (LoRA) w/ Instruction Tuning	✘	–	81.59
	✔	Instruction	**82.00**

CodeReviewer [20] or Tufano et al. [23] is due to the differences between their target tasks and LLaMA's pre-training tasks, as well as the input formats. However, task-specific pre-training from scratch, as with CodeReviewer [20], is resource-intensive, creating a barrier for enhancement through model size expansion. Instead, integrating domain knowledge into a single pre-trained model and applying parameter-efficient fine-tuning methods could be more cost-effective.

Interestingly, the relative simplicity of the code refinement task compared to the review comment generation task may have paradoxically lowered the LLaMA-Reviewer's BLEU score. This is because the model, trained to generate diverse predictions mimicking human behavior, might produce more diverse, yet valid, refinements which diverge from the singular ground truth, thereby decreasing textual similarity.

Answer to RQ1: LLaMA-Reviewer relatively more excels in generating review comments (NL) and identifies more problems in necessity prediction while maintaining competitive performance in code refinement.

B. RQ2: The Influence of Input Representation

In this subsection, we investigate the impact of input representation using the results from RQ1 and additional ablation experiments. We address each sub-question in turn before drawing an overall conclusion.

1) (RQ2.1) Consequences of Code Formatting: To evaluate the effect of code formatting, we examine the results of code comment generation and code refinement tasks using both the CRer and Tufano datasets.

The results presented in Section V-A show superior relative performance on the CRer dataset compared to the Tufano dataset. Despite the differences in data distribution, the primary distinction between these two datasets lies in their code formatting. The code in the CRer dataset is more rudimentary and akin to the format used during LLaMA's pre-training, while the code in the Tufano dataset has undergone sophisticated processing. These results suggest that a code representation similar to the one used during pre-training allows the model to better leverage its understanding of code structures and semantics.

2) (RQ2.2) Role of Language Label: We conducted experiments to evaluate the impact of language labels solely on the CRer dataset, as it includes multiple programming languages. Language labels, determined based on the programming language of the code, were integrated either into the instruction or the input, as shown in Figure 3. The remaining settings, borrowed from the code refinement task with low-rank adaptation as a fine-tuning method, were kept constant.

Contrary to expectations, the results presented in Table VII reveal that adding a language label does not enhance performance without an initial phase of instruction tuning. This could be due to the model's difficulty in associating label information with the task without pre-existing domain knowledge. However, once instruction tuning is implemented, the labels indeed contribute positively to the model's performance. Through a paired bootstrap resampling test, we determined that the use of a language label improves performance over its absence, evidenced by a p-value of 0.0032.

Although language labels have demonstrated their value, we chose not to include them in other experiments in order to maintain consistency with previous research [20] and to ensure a fair comparison.

Answer to RQ2: LLaMA-Reviewer performs better when the input representation resembles that used during pre-training. Additional natural language information, such as language labels, can be better leveraged by the model through instruction tuning.

C. RQ3: The Impact of Instruction Tuning

To answer RQ3, we perform experiments with models trained with and without the preliminary stage of instruction tuning, employing both zero-init attention prefix-tuning and Low-Rank Adaptation (LoRA). We also introduce supplementary experiments with additional natural language instructions [53] during LoRA instruction tuning to identify the optimal instruction tuning data (as posed by RQ3.3). The experiments are conducted on the CRer dataset using LoRA, and the results from the code review tasks are documented in Table VIII.

1) (RQ3.1) Consequences for Zero-init Attention Prefix-tuning: Our results suggest that instruction tuning is not conducive to prefix tuning. This could be attributed to the structure of prefix tuning, which employs the prefix solely to control attention and keeps attention across the postfix fixed. As such, its ability to capture general domain knowledge is limited. Moreover, the zero-init prefix attention, which contributes significantly to the efficacy of prefix tuning, is undermined when instruction tuning is added.

TABLE VIII
IMPACT OF INSTRUCTION TUNING (LORA r = 8).

Method	I. Tuning	Dataset	RNP (F1)	RCG	CR
LoRA	✗	–	**70.20**	5.58	**81.87**
	✔	PL	69.34	**5.64**	81.59
	✔	PL + NL	69.82	5.23	81.17
Prefix-tuning	✗	–	–	**5.16**	**76.71**
	✔	PL	–	5.02	76.04

2) *(RQ3.2) Consequences for Low-Rank Adaptation:* Unlike prefix tuning, instruction tuning in conjunction with LoRA improves performance across most tasks. For review necessity prediction, it elevates the precision of predictions from 81.56% to 83.99% when using the PL dataset as the sole tuning set, even though it doesn't enhance the f1-score. For review comment generation, it increases the BLEU score.[3] For code refinement, although no substantial improvement is detected, we infer from Section V-B that it augments the model's ability to incorporate label information. Instruction tuning with LoRA helps the base model comprehend instructional intent, which, in turn, benefits subsequent task tuning, particularly review comment generation, given its complexity and multi-intent nature.

3) *(RQ3.3) Influence of Instruction Types:* We focus on the "PL+NL" and "PL" rows using LoRA, which denote the use of both Alpaca and Code Alpaca data and only Code Alpaca data, respectively. Interestingly, even though code review tasks are closely related to natural language, incorporating Alpaca data for instruction tuning diminishes performance across all tasks. This trend may be linked to the extensive diversity of natural language instructions in Alpaca. Vast instructions in the Alpaca dataset are with verbs like rewrite and classify, and they seem to be overwhelming for code review tasks due to their wide range of tasks.

Answer to RQ3: Instruction tuning can potentially enhance task performance or the capacity to handle additional natural language information. However, the effectiveness is minor due to the word habit inconsistency among instruction and downstream datasets.

D. RQ4: Influence of Parameter-Efficient Fine-Tuning

To explore the impact of Parameter-Efficient Fine-Tuning (PEFT) methods, we conduct additional experiments adjusting the rank r of LoRA, specifically setting the rank to 8 and 16. The investigation of hyper-parameters for prefix-tuning is ommited as it was fully analyzed in previous work [49]. The outcomes are detailed in Table IX. For comparative analysis, we also include the results of prefix-tuning.

1) *(RQ4.1) Comparison between PEFT Methods:* The results decisively indicate that LoRA surpasses prefix-tuning across all tasks. We attribute this enhanced performance to two primary aspects. First, the prefix-tuning method implemented in our study has fewer trainable parameters than LoRA, impeding its adaptability from the base model. Second, in contrast to prefix-tuning which relies on prefixes to control the base model, LoRA approximates full-parameter tuning, an attribute crucial when the target output significantly differs from the pre-training format.

[3]This effect is more pronounced with learning rate 5e-5. Post 10 epochs, the results with and without instruction tuning are 5.43 and 5.27, respectively.

TABLE IX
INFLUENCE OF PARAMETER-EFFICIENT FINE-TUNING METHODS.

Tuning Method	r	Trainable Params	Storage Space	RNP (F1)	RCG (BLEU)	CR (BLEU)
Prefix	–	∼1.2M	2.4M	–	5.16	76.71
LoRA	8	∼4.2M	8M	69.34	5.64	81.59
LoRA	16	∼8.4M	16M	**70.49**	**5.7**	**82.27**

2) *(RQ4.2) Impact of LoRA Rank r:* Increasing the LoRA rank r from 8 to 16 improves the performance of LLaMA-Reviewer. This improvement is intuitive, as a higher rank augments the number of trainable parameters, bringing the model closer to full-parameter tuning. Nevertheless, the chief aim of using PEFT methods is to limit trainable parameters and conserve computational resources. Thus, striking a balance between performance and efficiency is a vital consideration.

3) *(RQ4.3) Efficiency of PEFT Methods:* As shown in the table, PEFT methods reduce the number of trainable parameters to under 1% while still ensuring acceptable performance. Given that PEFT methods keep the weights of the base model constant, the storage space decreases drastically from 13GB to under 20MB. These independent plug-in weights make PEFT methods an ideal fit for multi-task processes, such as the automation of code review activities.

Answer to RQ4: Among PEFT methods, LoRA is more suitable for automating code review tasks. By choosing an appropriate LoRA rank, LLaMA-Reviewer can achieve competitive performance with less than 1% of the trainable parameters and significantly reduced storage space requiements.

VI. RELATED WORK

A. Tuning on LLaMA

While OpenAI's proprietary ChatGPT and GPT series have driven substantial progress in the AI field, their closed-source nature has generated reservations among researchers. Addressing this concern, Meta introduced their open-source Large Language Model (LLaMA) [37], which has swiftly emerged as a pivotal asset in the AI landscape due to its remarkable performance capabilities.

A variety of LLaMA-based tuned models have shown exceptional performance, rivaling the ChatGPT and GPT series. Stanford's Alpaca [53] represents an early significant development in this area, tuning LLaMA using a dataset generated from ChatGPT. Subsequent notable works have pursued a range of objectives, including language-specific adaptations [59], [60], text quality enhancement [61], [62], multi-modal input accommodation [49], [63], [64], and code-related capabilities improvement [54].

Due to the substantial computational demands of full parameter tuning, researchers have gravitated towards Parameter-Efficient Fine-Tuning (PEFT) methods for tuning LLaMA. One such example is Alpaca LoRA, which achieves performance comparable to Alpaca with only 0.13% of the training parameters, resulting in approximately a 60 times speedup [50]. LLaMA-adapter, introduced by Zhang et al. [49], represents a further contribution, employing a zero-init attention prefix-tuning method.

Our study focuses on evaluating LLaMA's performance on code review-related tasks and employs state-of-the-art PEFT methods to enhance training efficiency.

B. Automation of Code Review Activities

Code review is an essential, albeit time-consuming, aspect of software development, sparking considerable interest in automation strategies for activities such as reviewer recommendation [6]–[15], code quality assessment [12], [16]–[21], problematic code refinement [20], [22]–[25], and review comment suggestion [20], [23], [26]–[31]. This paper focuses on the pipeline proposed by Li et al. [20], which comprises review necessity prediction, code review comment generation, and code refinement.

Early studies on review necessity prediction primarily examined diff hunk acceptance, with Shi et al. [16] pioneering a CNN and LSTM-based framework, DACE, and Hellendoorn et al. [17] utilizing a Transformer to account for inter-diff hunk relations within a single pull request (PR). However, the field has since evolved, with Li et al. [20] shifting focus towards identifying diff hunks requiring review and integrating this into a pipeline with a code review-specific pre-trained model.

Initial efforts in code review comments leveraged retrieval-based methods for historical comment extraction. Gupta et al. [27] introduced an LSTM-based model, DeepMem, to recommend comments for new code snippets based on their relationship with code changes, while Siow et al. [28] enhanced retrieval through attention-based LSTM semantic information capture. The field has since shifted towards generating review comments with the rise of deep learning. Tufano et al. [23] pioneered this approach, pre-training a model on both code and technical language, with subsequent efforts by CodeReviewer [20] and AUGER [30] using a code review-specific pre-trained model and review tags, respectively, for improved results. At the same time, CommentFinder [31] presents an efficient retrieval-based alternative.

For code refinement, early efforts are often aligned with automatic bug-fixing techniques [65]–[67]. Pioneering the adaptation of this task to code review, Tufano et al. [68] focused on learning from code changes implemented in PRs. Later researchers incorporated code review comments into task input to better simulate code refinement [22], [23]. Addressing the challenge of new tokens, AutoTransforms [25] employed a byte-pair encoding (BPE) approach, additionally using a diff-aware method, D-ACT [56], to boost performance in cases closely related to single differences between the code base and initial commit. However, they did not account for the influence of code review comments and excluded data with similar input. CoditT5 [57], a pre-trained model explicitly designed for editing, used part of Tufano's dataset for validation as a downstream task. Similarly, CodeReviewer [20] developed a model based on their pre-trained model, specifically tailored for the code review process.

Despite significant progress in automating code review tasks, previous research often neglects the potential of unified large language models (LLMs). As the model size and training data continue to grow, unified LLMs are improving their performance at a rapid pace and show comparable performance to those task-specific pre-trained models. For task-specific pre-trained models, building from scratch is resource-intensive and time-consuming. Thus, in this study, we leverage LLaMA, a mainstream unified large language model, to investigate the evolving potential of LLMs and assess their suitability for tasks that encompass both programming and natural language, such as code review tasks.

VII. Conclusion and Future Work

In this paper, we introduced LLaMA-Reviewer, a framework for automating the code review process using large language models (LLMs) and parameter-efficient fine-tuning (PEFT) techniques. We demonstrated that, despite using the smallest version of LLaMA with only 6.7B parameters and less than 1% of trainable parameters over a limited number of tuning epochs, LLaMA-Reviewer can match the performance of state-of-the-art code-review-focused models. Additionally, by applying plug-in type models, we significantly reduce storage space requirements.

Our findings also suggest that aligning the input representation with the format used during pre-training could better leverage the capabilities of LLMs. Additionally, an initial stage of instruction tuning can improve task performance and increase the model's ability to process additional natural language information. Our results also showed that low-rank adaptation with an appropriate rank is preferable for tasks with specific input and output formats.

Looking ahead, we aim to broaden our exploration of large language models, considering models of various sizes and types, and further investigate PEFT methodologies. We are also interested in examining more closely the relationship between the token length of prompt templates, code snippets, comments, and the sequence block of the pre-trained models.

Acknowledgment

This work was supported by Chinese Academy of Sciences-Dongguan Science and Technology Service Network Plan (No.202016002000032), and Alliance of International Science Organizations (No. ANSO-CR-KP-2022-03).

References

[1] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.

[2] D. Spadini, G. Çalikli, and A. Bacchelli, "Primers or reminders? the effects of existing review comments on code review," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1171–1182.

[3] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open-source software projects: Parameters, statistical models, and theory," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–33, 2014.

[4] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[5] Q. Shan, D. Sukhdeo, Q. Huang, S. Rogers, L. Chen, E. Paradis, P. C. Rigby, and N. Nagappan, "Using nudges to accelerate code reviews at scale," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 472–482.

[6] E. Sülün, "Suggesting reviewers of software artifacts using traceability graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1250–1252.

[7] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal, C. Maddila, S. Mehta, and B. Ashok, "Whodo: Automating reviewer suggestions at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 937–945.

[8] A. Chueshev, J. Lawall, R. Bendraou, and T. Ziadi, "Expanding the number of reviewers in open-source projects by recommending appropriate developers," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 499–510.

[9] S. Rebai, A. Amich, S. Molaei, M. Kessentini, and R. Kazman, "Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations," *Automated Software Engineering*, vol. 27, no. 3, pp. 301–328, 2020.

[10] E. Mirsaeedi and P. C. Rigby, "Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1183–1195.

[11] E. Sülün, E. Tüzün, and U. Doğrusöz, "Rstrace+: Reviewer suggestion using software artifact traceability graphs," *Information and Software Technology*, vol. 130, p. 106455, 2021.

[12] I. X. Gauthier, M. Lamothe, G. Mussbacher, and S. McIntosh, "Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 30–41.

[13] D. Kong, Q. Chen, L. Bao, C. Sun, X. Xia, and S. Li, "Recommending code reviewers for proprietary software projects: A large scale study," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 630–640.

[14] G. Rong, Y. Zhang, L. Yang, F. Zhang, H. Kuang, and H. Zhang, "Modeling review history for reviewer recommendation: A hypergraph approach," *arXiv preprint arXiv:2204.09526*, 2022.

[15] P. Pandya and S. Tiwari, "Corms: a github and gerrit based hybrid code reviewer recommendation approach for modern code review," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 546–557.

[16] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 4910–4917.

[17] V. J. Hellendoorn, J. Tsay, M. Mukherjee, and M. Hirzel, "Towards automating code review at scale," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1479–1482.

[18] H. Hijazi, J. Cruz, J. Castelhano, R. Couceiro, M. Castelo-Branco, P. de Carvalho, and H. Madeira, "ireview: an intelligent code review evaluation tool using biofeedback," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 476–485.

[19] D. Wang, Y. Ueda, R. G. Kula, T. Ishio, and K. Matsumoto, "Can we benchmark code review studies? a systematic mapping study of methodology, dataset, and metric," *Journal of Systems and Software*, vol. 180, p. 111009, 2021.

[20] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. 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*, 2022, pp. 1035–1047.

[21] H. Hijazi, J. Duraes, R. Couceiro, J. Castelhano, R. Barbosa, J. Medeiros, M. Castelo-Branco, P. De Carvalho, and H. Madeira, "Quality evaluation of modern code reviews through intelligent biometric program comprehension," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2022.

[22] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[23] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," *arXiv preprint arXiv:2201.06850*, 2022.

[24] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.

[25] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: automated code transformation to support modern code review process," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 237–248.

[26] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 931–940.

[27] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[28] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[29] Y. Hong, C. K. Tantithamthavorn, and P. P. Thongtanunam, "Where should i look at? recommending lines that reviewers should pay attention to," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1034–1045.

[30] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "Auger: automatically generating review comments with pre-training models," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1009–1021.

[31] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder: a simpler, faster, more accurate code review comments recommendation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 507–519.

[32] *LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning*. Zenodo, May 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7991113

[33] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[34] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[35] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[36] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[37] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[38] OpenAI, "Gpt-4 technical report," 2023.

[39] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, “Incoder: A generative model for code infilling and synthesis,” *arXiv preprint arXiv:2204.05999*, 2022.

[40] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[41] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” *arXiv preprint arXiv:2203.13474*, 2022.

[42] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, “Codet5+: Open code large language models for code understanding and generation,” *arXiv preprint arXiv:2305.07922*, 2023.

[43] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, “Parameter-efficient transfer learning for nlp,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 2790–2799.

[44] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, “Towards a unified view of parameter-efficient transfer learning,” *arXiv preprint arXiv:2110.04366*, 2021.

[45] X. L. Li and P. Liang, “Prefix-tuning: Optimizing continuous prompts for generation,” *arXiv preprint arXiv:2101.00190*, 2021.

[46] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” *arXiv preprint arXiv:2104.08691*, 2021.

[47] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang, “Gpt understands, too,” *arXiv preprint arXiv:2103.10385*, 2021.

[48] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2021.

[49] R. Zhang, J. Han, A. Zhou, X. Hu, S. Yan, P. Lu, H. Li, P. Gao, and Y. Qiao, “Llama-adapter: Efficient fine-tuning of language models with zero-init attention,” *arXiv preprint arXiv:2303.16199*, 2023.

[50] E. J. Wang, “Tloen/alpaca-lora: Instruct-tune llama on consumer hardware.” [Online]. Available: https://github.com/tloen/alpaca-lora/

[51] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, “Finetuned language models are zero-shot learners,” *arXiv preprint arXiv:2109.01652*, 2021.

[52] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[53] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, “Stanford alpaca: An instruction-following llama model,” https://github.com/tatsu-lab/stanford_alpaca, 2023.

[54] S. Chaudhary, “Code alpaca: An instruction-following llama model for code generation,” https://github.com/sahil280114/codealpaca, 2023.

[55] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, “Self-instruct: Aligning language model with self generated instructions,” *arXiv preprint arXiv:2212.10560*, 2022.

[56] C. Pornprasit, C. Tantithamthavorn, P. Thongtanunam, and C. Chen, “D-act: Towards diff-aware code transformation for code review under a time-wise evaluation.”

[57] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, “Coditt5: Pretraining for source code and natural language editing,” in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[58] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[59] Z. L. Chenghao Fan and J. Tian, “Chinese-vicuna: A chinese instruction-following llama-based model,” 2023. [Online]. Available: https://github.com/Facico/Chinese-Vicuna

[60] Y. Cui, Z. Yang, and X. Yao, “Efficient and effective text encoding for chinese llama and alpaca,” *arXiv preprint arXiv:2304.08177*, 2023. [Online]. Available: https://arxiv.org/abs/2304.08177

[61] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, “Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality,” March 2023. [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/

[62] B. Peng, C. Li, P. He, M. Galley, and J. Gao, “Instruction tuning with gpt-4,” *arXiv preprint arXiv:2304.03277*, 2023.

[63] D. Zhu, J. Chen, X. Shen, X. Li, and M. Elhoseiny, “Minigpt-4: Enhancing vision-language understanding with advanced large language models,” *arXiv preprint arXiv:2304.10592*, 2023.

[64] H. Liu, C. Li, Q. Wu, and Y. J. Lee, “Visual instruction tuning,” 2023.

[65] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[66] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning,” in *Proceedings of the aaai conference on artificial intelligence*, vol. 31, no. 1, 2017.

[67] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[68] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, “On learning meaningful code changes via neural machine translation,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.