This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3428324

Towards Efficient Fine-tuning of Language Models with Organizational Data for Automated Software Review

Mona Nashaat[1], James Miller[2]
[1] Electrical Engineering Department, Faculty of Engineering, Port Said University, Egypt
[2] Electrical and Computer Engineering Department, University of Alberta, Canada

***Abstract*— Large language models like BERT and GPT possess significant capabilities and potential impacts across various applications. Software engineers often use these models for code-related tasks, including generating, debugging, and summarizing code. Nevertheless, large language models still have several flaws, including model hallucination. (e.g., generating erroneous code and producing outdated and inaccurate programs) and the substantial computational resources and energy required for training and fine-tuning. To tackle these challenges, we propose CodeMentor, a framework for few-shot learning to train large language models with the data available within the organization. We employ the framework to train a language model for code review activities, such as code refinement and review generation. The framework utilizes heuristic rules and weak supervision techniques to leverage available data, such as previous review comments, issue reports, and related code updates. Then, the framework employs the constructed dataset to fine-tune LLMs for code review tasks. Additionally, the framework integrates domain expertise by employing reinforcement learning with human feedback. This allows domain experts to assess the generated code and enhance the model performance. Also, to assess the performance of the proposed model, we evaluate it with four state-of-the-art techniques in various code review tasks. The experimental results attest that CodeMentor enhances the performance in all tasks compared to the state-of-the-art approaches, with an improvement of up to 22.3%, 43.4%, and 24.3% in code quality estimation, review generation, and bug report summarization tasks, respectively.**

***Index Terms*— Artificial intelligence, Software engineering, Large Language Models, Reinforcement learning, Software reviews**

I. INTRODUCTION

CODE review plays an essential role in the field of software engineering, where developers examine and evaluate code contributions to ensure software quality, adhere to coding standards, and identify potential issues. Review typically starts after a developer completes a coding task and submits code changes to a version control system like Git. Then, reviewers, who are also team members with relevant expertise in the codebase and the specific changes being made, examine the code changes, looking for various aspects, including correctness, maintainability, and performance. Then, they provide feedback, comments, and suggestions within the code review platform. Traditionally, this procedure heavily depends on manual labor, which consumes significant time and is susceptible to human error.

As a result, there has been a paradigm shift towards automating code review activities [1–3]. Recent research [4–9] leverages various techniques, including traditional machine learning [4,5], deep learning [8,9], and natural language processing [6,7], to automate different tasks for code review. For instance, Chatley and Jones [10] propose a static analysis tool to generate review comments. The tool experiments with different mining algorithms to detect missing files or files with low code quality and provides suitable feedback. Also, deep learning models (i.e., CNN and LSTM) have been employed [9] to learn the required features from the original and revised code. The proposed models [9] learn the feature representation of modified lines in the code by analyzing their execution correlation with the surrounding context. Then, an autoencoder is utilized to acquire the revision features from both the initial and the modified segments. Similarly, another study [8] trains a CNN model to grasp the feature representation of the code and make predictions regarding the approval or rejection of the change request.

Furthermore, Large Language Models (LLMs) [11] have significantly advanced code review tasks by improving code quality while reducing the burden on human reviewers. Research [11–13] has investigated training large language models to automate different activities related to code review, such as recommending potential reviewers [14], refactoring code [15], analyzing bug reports [16], and generating review comments [11,12]. For example, one study [12] employs transformer architecture to generate review feedback by learning the correlation between the code changes and their corresponding review comments. The model collects review comments from several software projects. Then, the model utilizes this data with continuous training to pre-train a large language model. Then, the model is fine-tuned for comment

Mona Nashaat was a postdoctoral fellow at the University of Alberta and is now with the Department of Electrical Engineering at Port Said University, Egypt. Her e-mail addresses are MonaNashaat@eng.psu.edu.eg and nashaata@ualberta.ca.

James Miller is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton T6G 2R3, Canada. E-mail: jimm@ualberta.ca.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3428324

generation. Similarly, Zhang et al. [17] trained a large language model with new training objectives for code review tasks such as bug fixing and comment generation.

Although existing research has investigated pre-training LLMs or utilizing existing models, both approaches pose several challenges. On the one hand, pretraining such domain-specific models from scratch frequently necessitates significant computational resources, posing a challenge for organizations willing to create private LLMs. On the other hand, using existing pre-trained LLMs raises significant privacy concerns for organizations, particularly regarding handling sensitive information. This data exchange over the model API may include proprietary or confidential information, potentially exposing organizations to data breaches or leaks. Finally, fine-tuning, while often effective, does not guarantee good results for several reasons. For example, the quality of the training data significantly affects the model performance. If the fine-tuning dataset is noisy, biased, or not representative of the target task, the fine-tuned model may not generalize well to new data.

Hence, this paper introduces CodeMentor, an innovative framework designed to automate code review tasks by harnessing the capabilities of large language models. This framework utilizes organizational data to establish a fine-tuning process for large language models, enabling in-house training of these models even when computational resources are limited. To tackle the computational hurdles of fine-tuning large language models, the framework applies instruction-tuning approaches [18] and parameter-efficient fine-tuning (PEFT) [19]. To evaluate CodeMentor, we compare it with four state-of-the-art techniques. The experimental results empirically verify that the proposed model can enhance performance in all tasks, proving the effectiveness of our framework.

The primary contributions of this research can be listed as follows:

- **Using domain-specific customization:** CodeMentor integrates domain-specific heuristics to augment the collected data. This process guarantees that the training data is highly relevant and tailored to the specific requirements of code-related tasks, which is less commonly addressed in existing personalized machine learning models.
- **The use of data augmentation:** The framework generates new data samples from the initial organizational data, representing a novel approach to data augmentation. This technique enhances the diversity and richness of the training dataset.
- **The integration with advanced fine-tuning techniques:** Our approach combines the prepared data with advanced fine-tuning techniques [19] and reinforcement learning with human feedback (RLHF). This integration is specifically implemented to optimize the efficiency of LLMs in code-related tasks, providing a unique contribution to the field.
- **Focus on computational efficiency:** Using LoRA for fine-tuning significantly reduces the computational resources required for training, making our approach more accessible and practical for organizations with limited computational capabilities.
- **A thorough empirical assessment:** We empirically validate the effectiveness of the framework across four distinct downstream tasks related to code review activities.
- **An ablation study that assesses the effectiveness of each phase of the framework**: The study excludes each component to evaluate its impact on the overall performance of the framework.

While we recognize that similar strategies have been explored in other domains, the novelty and significance of our work lie in the thoughtful integration and adaptation of these techniques to address the specific requirements of the software engineering domain.

The paper is structured as follows: Section 2 presents an overview of the research background. In Section 3, we delve into the design details of the components of CodeMentor. Section 4 represents the implementation details of the system. Section 5 outlines the evaluation configuration and presents the experimental results. Section 6 discusses related research, and Section 7 discusses the limitations and threats to validity. Lastly, Section 8 concludes the paper.

II. BACKGROUND

This section briefly overviews the fundamental concepts related to this research, including automating the code review process and recent advances in large language models.

A. Automation of Code Review Tasks

The review process in software projects is typically supported by software tools integrated with version control systems. Key participants in this process include the code author(s) and the reviewer(s). Initially, the code author(s) modify the code to introduce new features or resolve issues in the previous version. Then, the code author(s) create a pull request. Both the original code and the revised version are included in the request. Typically, the author(s) describe the intended alterations and may include a reference to a specific issue documented in a bug-tracking system. Subsequently, one or more reviewers are chosen to assess the code changes and provide feedback. During this phase, the reviewer(s) examine the code changes for potential flaws or recommended improvements. Then, in response to the feedback, the author makes necessary revisions and presents an updated code version. This sequence of actions constitutes a review round. After several rounds of review, the pull request is either approved (leading to the integration of changes into the main branch) or rejected.

Numerous studies have suggested various solutions to automate different tasks related to the review process [20–25]. For example, some studies encompass reviewers' assignments [20,21], review generation [22,23], and evaluating review quality [24,25]. Previous studies focus on review recommendations [20,21]. For instance, one study [20] examines the preference of reviewers and authors and merges preference matrices to assign reviewers for a given request.

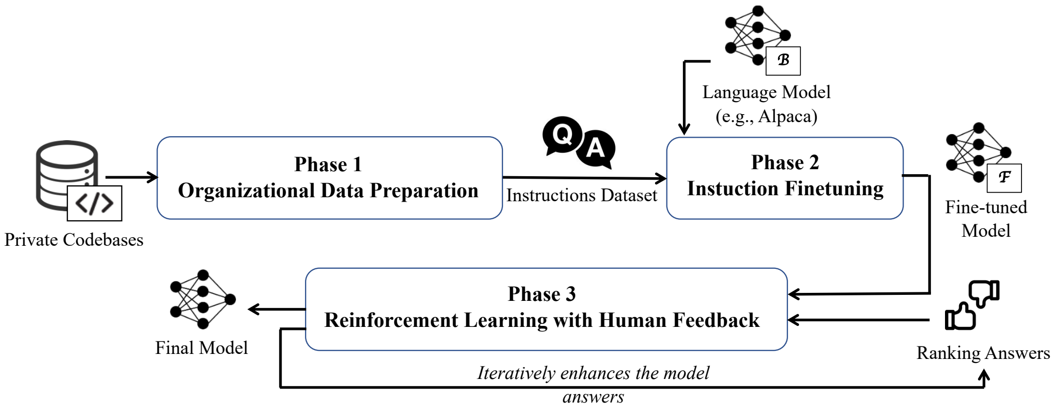

Fig. 1. An overview representation of the proposed framework.

Another study [21] incorporates the reviewer profiles and the properties of the submitted changes to find the similarities before recommending reviewers. The model [21] analyzes the similarity between the change request to be reviewed and the historical code reviewed by the reviewers. It uses the obtained similarity score to select the reviewer who matches the most.

Alternatively, other research emphasizes automating the review generation [22,23]. For instance, Guo et al. [22] proposed an interactive framework to break down the required changes into smaller tasks and analyze the change effect using program-dependence relationships. Similarly, other research [23] utilizes a convolutional network to categorize the review tasks that require manual review. Another deep learning approach [26] is proposed to generate review comments. The model examines a large corpus of <code, comments> pairs and employs historical comments to recommend comments for a given code. However, most of these methods [23,26] ignore looking at the inherent structures of the code. Likewise, although these studies have covered different code review tasks, they have yet to explore the use of organizational data to train their models, which we focus on in this study.

B. Large Language Models

In recent years, there has been a quantum leap in the scale and capabilities of language models. Models like GPT and its successors have demonstrated an extraordinary capacity to understand and generate human-like text. These models have been used across various applications, from natural language understanding [27] to creative content generation [28], and have sparked discussion about their ethical implications and responsible use [29]. The evolution of large language models continues, with ongoing research focusing on improving their efficiency [30,31], fine-tuning for specific tasks [32,33], and addressing concerns related to bias and safety in these AI systems [34].

The breakthrough came with the advent of neural networks and deep learning techniques. In the early 2010s, researchers started experimenting with neural language models. Still, it was only in the introduction of models like Word2Vec [35] that the power of distributed representations of words became evident. These models [35] paved the way for more sophisticated neural architectures, leading to the emergence of the Transformer architecture [36]. Unlike traditional neural networks like recurrent neural networks, Transformers [36] exhibit notable advantages in handling challenging tasks [37–39], such as document classification [38] and question answering [37].

Furthermore, Transformers leverage an attention mechanism [40] in the context of sequence transduction tasks. In sequence transduction, the model is trained to convert an input sequence, such as a sentence, into an output sequence, such as the same sentence translated into a different language. Consequently, Transformers adopt a structure reminiscent of sequence-to-sequence models [41], featuring an encoder-decoder configuration. Within this framework, the model comprises two primary networks. The initial network serves as an encoder to acquire an internal representation of the input vector. In contrast, the second network, acting as a decoder, employs this representation to produce the sequence in the desired target transformation.

Lately, transformers have found applications across a spectrum of language-processing tasks. The emergence of pre-trained models like BERT [42], RoBERTa [43], and GPT [44] have substantially elevated performance in downstream tasks like code generation [45,46], code summarization [47,48], and code review [15,49]. These models have been trained on extensive code bases; thus, they can understand the semantics and contextual representation of the code. Some studies [45,46] have worked on code LLMs to enhance language-to-code generation. For example, Ni et al. [45] proposed a model to evaluate the correctness of the code generated from LLMs. Similarly, Liu et al. [46] proposed a system to assess the code generated by ChatGPT based on different evaluation criteria. Another study [50] proposes a model to generate code based on a defined list of steps that describe the implemented solution. Moreover, large language models have been applied to code summarization using few-shot and zero-shot learning [47,48]. For instance, in one study [48], authors fine-tuned a code LLM with the CodeXGLUE dataset for code summarization. Another research [47] has experimented with zero-shot learning to evaluate the performance of different LLMs for code summarization. The study [47] concludes that the model summary can benefit from the evaluation of domain experts.

Also, many studies [15,49] have proposed code LLMs to automate activities related to code review. For example, Li et al. [15] presented a code LLM trained using a sizeable pre-training dataset from open-source projects. Then, the model is fine-tuned for review tasks. Similarly, another model is presented [49] by

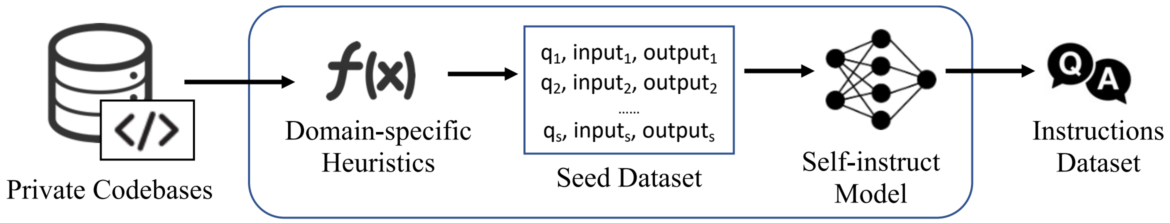

Fig. 2. An overview of the first phase (organizational data preparation).

training a transformer to learn the code changes and implement the required changes. Alternatively, in our implementation, instead of pre-training code LLMs, the framework utilizes the review data available in the organization to fine-tune LLMs. This enables a competitive performance on downstream tasks while reducing the required computational cost.

III. *CodeMentor*: The Proposed Framework

We propose *CodeMentor*, a fine-tuning framework tailored for LLMs. The framework aims to empower in-house LLM training on hardware with limited computational power. This approach uses organizational data, lowers the computational cost, and preserves competitive performance standards. The workflow makes fine-tuning a pre-trained model for a domain-specific task easier, making LLMs more affordable and available to organizations looking to include state-of-the-art natural language understanding in their applications.

Specifically, the framework uses instruction-tuning techniques [18] and parameter-efficient fine-tuning [19] to solve the computational challenges of fine-tuning large language models. Furthermore, to enhance performance and user experience, it adopts reinforcement learning with human feedback to give explicit feedback on the ranked responses to the model. This kind of feedback helps the model to understand the desired behavior and estimate the model's performance on specific tasks. An overview illustration of the architecture of the framework is shown in Fig. 1. As the figure shows, it contains three phases. The first phase is organizational data preparation, which includes analyzing the organization's code repositories and constructing the dataset needed to create a domain-specific model. In the second phase, instruction fine-tuning, the model is fine-tuned based on collected data to solve specific tasks. In this phase, it adjusts the model parameters based on the instructional question-answer pairings that were generated during the first phase. Its third phase is reinforcement learning with human feedback, which uses the feedback from domain experts to adapt the model to their preferences further. Each of these phases is explained in the following subsections.

A. Phase 1: Organizational Data Preparation

During this stage, the framework analyzes the code bases available in the organization and generates the instruction data that will be used in the following stages. The dataset includes a set of instructions $D = \{i_k\}_{k=1}^{N}$ where each instruction i defines a task (e.g., provide a review for a code block), input (e.g., the code to be reviewed), and output (e.g., the generated review comment). So, each instruction $i_k = \left\{t_k, \{in_j, out_j\}_{k\,j=1}^{\;M}\right\}$ consists of a task t_k defined in natural language, and a set of *{in, out}* pairs. To create a high-quality instruction dataset, the framework utilizes the self-instruct model [51] to generate the instruction data automatically. The self-instruct tool helps improve the instruction-following capabilities of pre-trained language models by generating instructional data autonomously. This method reduces reliance on extensive manual annotation. It uses an iterative process where a language model generates, filters, and refines its instructional data, enhancing its ability to follow natural language instructions. The process begins with a small set of seed instructions (the seed dataset), which the model uses to create new instructions and corresponding input-output pairs. These are filtered for quality and added back to the task pool iteratively. The framework allows models to enhance their performance without extensive manual annotation. This method significantly increases the diversity and quantity of instructional data available for fine-tuning.

A detailed description of the first phase is shown in Fig. 2. As the figure shows, the proposed framework automatically analyzes the existing codebases to create a small seed dataset. Typical review data employed in software projects usually comprises multiple documents. They can be formally defined as $D_r = \{d_i\}_{i=1}^{L}$ where d represents a code document, and $D_{submitted} \in D_r$ contains all the code blocks $\{b_i\}_{i=1}^{S}$ submitted for review and $D_{reviewed} \in D_r$ include all the code blocks $\{c_i, b_i\}_{i=1}^{S}$ after revision along with the associated reviewer comment. Therefore, the framework applies data programming strategies from our previous work [52,53] to extract the required knowledge from the codebases found in the organization. We formalize a set of custom domain heuristics

```
def heuristic_fn_review_extraction (d):
   return (d.body, d.pull_request) if d.body is not empty; else ABSTAIN;
def heuristic_fn_quality_estimation (d):
   return false if d.body.get_review_comments is not empty; else return true
```

Fig. 3. An example of the domain heuristics functions.

```
INSTRUCTION_DICT{
Prompt_1: (
            "The following instruction describes a task, coupled with an input that offers additional context."
            "Provide a reply that adequately accomplishes the task. "
            "### Instruction:
            {"Review this code block and give a suitable review"}
            ### Input: {"the code block is {code_block}}
            ### Response: {review_comment}
             )
Prompt_2: (
            "The following instruction describes a task, coupled with an input that offers additional context."
            " Provide a reply that adequately accomplishes the task. "
            "### Instruction:
            {"Rewrite this code block according to the attached review comment"}
            ### Input: {"the code block is {code_block} \n the review is {review_comment} }
            ### Response: {refined_code}
             )
}
```

Fig. 4. An example of the instructions format generated in the second phase.

to generate the instruction data required for each downstream task. An example of domain heuristics used by the model is illustrated in Fig. 3. The figure describes a sample of code review extraction and code quality estimation examples. The proposed framework applies these heuristics to extract the required data for the seed dataset, including tasks, inputs, and output samples related to code review activities. In our framework, we consider four review tasks: review comment generation, code quality estimation, bug report summarization, and code refinement.

After generating the seed dataset, the framework applies a bootstrapping process by sampling the task instructions from the seed dataset to generate a task pool that contains a larger set of instructions and instances. It employs these initial instructions to query a model G to produce similar outputs to each sampled task as:

$$(i_k, in_{j,k}) = out_{j,k} \text{ for } i \in (1, M_k) \text{ and k} \in (1, N) \qquad (1)$$

where M is the number of inputs defined for each task, and N is the number of generated instructions. These generated instructions undergo a filtration process to eliminate instructions with bad quality or similar ones, and the refined data is reintroduced into the task pool. This cycle is repeated several times, yielding a substantial repository of instruction data for fine-tuning the model in the following phase.

B. Phase 2: Instruction Fine-tuning

In this phase, we fine-tune a language model to obey instructions in natural language to improve its performance. Hence, the framework utilizes a collection of instructions in a prompt-answer format to train the model. The main goal of instruction tuning is to increase the model's ability to handle different tasks and generalize to new or unknown tasks more successfully. Therefore, the framework utilizes a subset of the instruction dataset generated from the first phase to fine-tune a

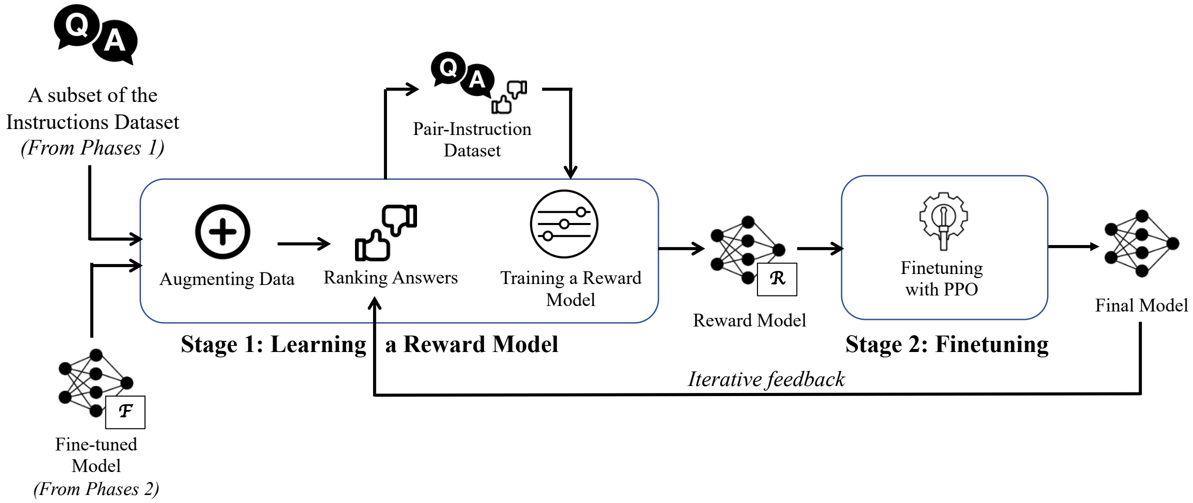

Fig. 5. A detailed overview of fine-tuning the model with reinforcement learning.

large language model. The rest of the dataset is kept for the following phase to align the model with human preferences. Thus, we familiarize the model with code review tasks in this subset. The instruction data has a standard format as {instruction, input, output}. The exact format is used across all the code review tasks. Fig. 4 depicts a sample of the instructions format and instances of instructions for different code review tasks used in this phase.

Furthermore, the framework employs Low-Rank Adaptation of Large Language Models (LoRA) to speed up the training process and decrease memory utilization. LoRA is an effective tuning approach [54] that implements Parameter-Efficient fine-tuning (PEFT). Therefore, we only integrate trainable low-rank matrices [54] into the transformer layer rather than fine-tuning all the model parameters. Suppose the pre-trained weight matrix $w_0 \in R^{d \times k}$, LoRA approximates the weight modification as:

$$w_0 + \Delta w = w_0 + w_d w_k \quad (2)$$

where $w_d \in R^{d \times r}$ and $w_k \in R^{r \times k}$, which splits the weight update matrix into two smaller ones. Thus, during the fine-tuning process, the original pre-trained weights w_0 are kept the same, reducing the likelihood of catastrophic forgetting [54]. The approach only updates w_d and w_k since they become the trainable weights. Furthermore, since the rank-decomposition matrices significantly reduce the number of trainable parameters, the updated pass can be formally presented in (3). The output of this phase is a fine-tuned version of the base language model that is trained for code review tasks.

$$\bar{h} = w_0\, x + \Delta w\, x = w_0\, x + w_d w_k x \quad (3)$$

where x is the input, $h = w_0\, x$ is the output, and $\bar{h}$ represents the updated forward pass.

C. Phase 3: Reinforcement Learning with Human Feedback

Even though the fine-tuned model produced in the last phase has accumulated significant knowledge about code review tasks, it needs an understanding of the preferences of domain experts. To address this, the framework incorporates human feedback in the form of rankings to provide the model with additional guidance during training. Therefore, the framework employs RLHF to let domain experts rank the answers generated from the model. The phase consists of two main stages. First, the framework trains a reward model as the alignment tool that synchronizes human preferences with the learning process. Second, the reward and fine-tuned models are combined as the final model. An overview illustration of the phase is depicted in Fig. 5.

A reward model is trained in the first stage using a subset of the instruction dataset created in the first phase. The dataset is then augmented with an additional answer to each prompt so the user can perform the ranking and choose which answer is accepted. To generate the additional answers, the fine-tuned model created in the last phase produces an additional response for each prompt in the subset. After ranking the answers, the reward model is trained using the instructions and their ranked answers. The training is similar to the supervised fine-tuning process in the second phase. The framework utilizes pairs of outputs for each prompt with their associated ranking score (the reward value). Hence, the model undergoes training to establish connections between outputs and their corresponding reward scores. The objective in this context is to create a reward model that can assess the desirability of potential outputs to domain experts.

Thus, given a prompt, denoted as x, along with two ranked responses provided by domain experts, namely y_1 and y_2, the framework learns a model M_r by utilizing response ranking as rewards. Therefore, the framework optimizes the model M_r to better align with human preferences using the following binary ranking loss:

$$L_r = -\log\left(\sigma\left(s_\theta\left(x, y_p\right) - s_\theta\left(x, y_r\right)\right)\right) \quad (4)$$

where, $s_\theta\left(x, y_p\right)$ represents the score predicted by the model with parameters θ for response y_p of prompt x, y_p is the favored response, and y_r is the rejected one. In other words, the framework treats the collected pairs of human preferences as a binary ranking label problem [55]. Therefore, the response with the higher ranking is designated as the preferred response, with the other response being considered rejected or discarded. Subsequently, the model is trained to generate a score value (a reward) for each response, representing the answer's quality. The response that receives the higher rewards will guide the reinforcement learning policy, encouraging the generation of responses that are more aligned with human preferences.

Then, Proximal Policy Optimization (PPO) is implemented in the second stage to help the fine-tuned model acquire the ability to produce responses that receive higher scores by the reward model. Hence, PPO balances exploration and exploitation throughout training. This balance holds significant importance for RLHF agents, given their necessity to glean insights from human guidance and iterative trial-and-error exploration. This entire procedure forms a reinforcement learning loop, where the LLM takes on the role of the agent, and its responses serve as the actions it undertakes. The state space encompasses user queries and LLM-generated outputs, while the reward is evaluated by how well the LLM output aligns with the application context and the user preferences.

IV. System Implementation

The proposed framework is designed to work with data extracted from version control systems, as almost all software engineering projects involve repositories, which are the backbone for version control and collaborative development. These repositories, often managed using systems like Git, provide a structured and organized environment where code, documentation, and project assets are stored. Thus, to collect a sample of organizational data required for the first phase, we leveraged the GitHub REST API to collect the pull request reviews, which are sets of comments made during the review process of a pull request (PR) bundled together with an associated status and, optionally, a body comment. We designed a series of API requests to access and retrieve the data from GitHub repositories. For pull requests (PRs), we collected their titles, descriptions, creation dates, and authors. Then, for each PR, we retrieved the data for the review comments,

TABLE I
BASIC STATISTICS OF THE GENERATED INSTRUCTION DATA

Number of instances	112,355
Number of instances used in fine-tuning (Phase 2)	89,884
Number of instances used in RLHF (Phase 3)	22,471
Average of instruction length (in words)	27.1
Average of output length (in words)	19.4

TABLE II
BASIS STATISTICS FOR BENCHMARK DATASETS

Dataset	Quality Estimation	Code Review Generation	Code Refinement	Bug Summarization
Training set	197k	83k	106k	56k
Validation set	66k	28k	35k	19k
Testing set	66k	28k	35k	19k

including the comment content and timestamps of those comments.

Then, a set of custom domain heuristics (Fig. 3) is applied in the first phase to extract the required data for the seed dataset for each downstream task. For estimating the code quality, the problem is expressed as a classification problem. Therefore, the user heuristics extracted code blocks from each request with the corresponding review comments. Then, the number of comments is used to give the label, while code changes accompanied by review comments are labeled as low quality, and the code changes with no comments are labeled as high-quality code. As for the review generation task, the data is organized in pairs of the code block and the corresponding review comment. The comment corresponding to each code change is considered as the desired output from the model.

Moreover, to prepare the data required for the bug report summarization task, we extract the issues represented in the repository and apply a heuristic function that classifies the issue as a bug if specific keywords appear in the issue content (i.e., bug, problem, ..). We treat the issue description as the desired output. Finally, in the code refinement task, a custom function extracts the initial committed code, the reviewer's comments, and the final committed code. We use these triplets to build the seed dataset for the third task. During this phase, a seed dataset is formulated with 400 instances (100 instances for each downstream task). An example of the seed datasets generated from the user heuristics is illustrated in Fig. 6.

Then, the self-instruct model is applied to sample the task instruction from the seed dataset and generate a larger task pool. The final generated data has more than 112k instances. Table I provides a statistical summary of the generated data. This dataset will be used to construct the instruction fine-tuning dataset (Phase 2) and the RLHF dataset (Phase 3). Therefore, we split the data into 80% for the fine-tuning dataset and 20% for the RLHF dataset. The final output of this phase is the dataset ready to fine-tune the model in the following phase.

In the second phase, the fine-tuning dataset is used to fine-tune Code Llama [56]. Although the proposed framework can work with any language model, we chose Code Llama [56], which is an advanced language model designed for code-related tasks based on Llama [57]. Different research has proven that Llama is a robust and effective model that can be applied in several domains [58,59]. Hence, we first utilize Code Llama to generate embeddings for the input code snippets. These embeddings encapsulate the semantic and syntactic characteristics of the code, providing a rich representation for further processing. Then, the model is adapted to handle the downstream tasks. For example, to estimate the code quality, we added a separate classifier on top of the embeddings generated by CodeLlama. This classifier is a simple feed-forward network with a ReLU activation function [36] that predicts the quality label. Then, the model is fine-tuned on the fine-tuning dataset. During training, the embeddings produced by CodeLlama are fed into the classifier, which is then trained to predict the correct quality labels with a binary cross-entropy loss function [38].

As for LoRA Training Hyper-parameters, we used a LoRA rank of 8, which balances the trade-off between model capacity and computational efficiency. Also, the learning rate for LoRA training was $3e^{-5}$ and a batch size of 8. The settings are through a series of experiments to optimize model performance while preventing overfitting. To ensure optimal performance, we trained separate models for each task: quality estimation, code review generation, code refinement, and bug summarization. This approach has allowed us to fine-tune the models to the characteristics and demands of each task, therefore improving performance and efficiency. Fine-tuning took 4 hours on a 40GB A100 GPU. The output of this phase is the model fine-tuned to each code review task included in the training dataset.

As for the final phase, we apply RLHF in two main stages. The first step is to train a reward model to rank different answers for the prompt instruction. During this step, we utilize a subset of the dataset generated in the first phase (RLHF dataset). To generate additional responses, we employ the fine-tuned model from the first phase to generate an additional answer for each prompt in the dataset. Consequently, the subset now encompasses 22,471 prompts, each with two corresponding responses, and is prepared for ranking to determine the best answer. To train the reward model, a procedure similar to supervised fine-tuning was followed using the same base model, LLaMA-7B. It is trained on the ranked prompt-answer pairs, resulting in PEFT adapter layers that are then integrated with the base model. This training was completed in approximately 3 hours using a 40GB A100 GPU. In this step, the model learns to maximize the reward signal from the reward model while generating responses to user inputs. The last stage of the RLHF phase involves combining the model fine-tuned in the second phase with the reward model trained in the previous stage. For this process, Proximal Policy Optimization was utilized to fine-tune the model. The output of this stage is the final model used for automating review tasks.

V. EXPERIMENTAL EVALUATION

This section aims to assess the performance of *CodeMentor* in automating code review tasks. The experiments are four-fold and seek to estimate the performance of the proposed

TABLE III
EXPERIMENTAL RESULTS OF COMPARING TECHNIQUES ON CODE QUALITY ESTIMATION

Model	Acc	P	R	F1
Baseline Transformer	0.69	0.78	0.55	0.65
Pre-trained T5	0.71	0.72	0.58	0.64
CodeT5	0.72	0.70	0.63	0.66
CodeReviewer	0.76	0.81	0.67	0.73
CodeMentor	**0.84**	**0.83**	**0.74**	**0.78**

TABLE IV
EXPERIMENTAL RESULTS OF COMPARING TECHNIQUES ON CODE REVIEW GENERATION

Model	BLEU	Human Feedback		
		Tech. Accuracy	Clarity	Relevance
Baseline Transformer	3.72	2.4	1.7	2.1
Pre-trained T5	4.11	3.1	2.3	3.4
CodeT5	4.57	2.9	4.1	3.1
CodeReviewer	4.98	3.7	3.2	3.6
CodeMentor	**5.33**	**3.9**	**4.2**	**3.8**

TABLE V
EXPERIMENTAL RESULTS OF COMPARING TECHNIQUES ON BUG REPORT SUMMARIZATION

Model	BLEU
Baseline Transformer	4.31
Pre-trained T5	5.13
CodeT5	4.88
CodeReviewer	4.53
CodeMentor	**5.73**

framework in the following review tasks:

- *Code Quality Evaluation*: the proposed model evaluates a code change and determines whether it needs a review.
- *Code Review Generation*: in this task, CodeMentor takes code changes and is asked to produce a review comment.
- *Bug report Summarization*: CodeMentor takes an issue report that represents a bug and produces a summary for the report.
- *Code Refinement*: CodeMentor is inquired to refine an input code according to a given review comment.

In the following subsections, we first discuss the datasets, baseline models, and metrics used in the evaluation.

A. Datasets

The datasets used in the evaluation vary depending on the downstream task. First, we utilized the benchmark dataset used in [15] for code quality estimation, code refinement, and review generation tasks. The data [15] included pull requests of GitHub repositories and has been extensively utilized in the literature [2,60]. To estimate the model's performance for the bug report summarization task, the experiments utilize three popular benchmark datasets [61,62], which are the Summary dataset [61], the Authorship dataset [61], and the Eclipse and Mozilla bug reports [62]. The Summary dataset contains 36 bug reports annotated by multiple annotators to create summaries. Similarly, the Authorship dataset comprises 96 bug reports with annotations and summaries. Eclipse and Mozilla bug Reports are obtained from large open-source projects like Eclipse and Mozilla. These datasets are commonly used for evaluating bug summarization approaches [63,64]. Table II summarizes basic statistics for benchmark datasets.

B. Baseline Models

To assess the effectiveness of the proposed model, we benchmark it against four state-of-the-art techniques: the baseline transformer model [36] and three pre-trained code LLMs, which include T5, CodeT5 [65], and CodeReviewer [15].

- **Baseline Transformer**: The model [36] serves as the benchmark for comparison. Its architecture leverages multi-head attention and flexible linear layers, with 12 encoder and decoder layers. This architecture, proven successful in various downstream tasks [37,39,40], provides a baseline for performance evaluation.
- **Pre-trained T5**: This model, presented by Tufano et al. [11], attempts to automate code review using a pre-trained approach. Trained on Stack Overflow and CodeSearchNet data [71], it has 61 million parameters and six layers for both the encoder and decoder.
- **CodeT5**: A unified code understanding and generation model [65], built upon T5's architecture [11], tackles tasks like summarizing, generating, translating, and refining code. The model is trained and fine-tuned using the CodeSearchNet [66] and the CodeXGLUE [67] datasets, respectively.
- **CodeReviewer**: This model [15] is a pre-trained model to automate code review activities. It is trained on a huge GitHub dataset collected specifically for code review activities. Moreover, it possesses four specialized tasks to improve its understanding of the review domain.

C. Evaluation Metrics

This section presents the evaluation metrics used in the experiments. First, as for the code quality evaluation, we utilize the accuracy (Acc), Precision (P), Recall (R), and F1-score (F1) to evaluate the comparing approaches. Second, to evaluate the proposed model in the task of review comment generation and bug report summarization, we utilize the Bilingual Evaluation Understudy [68] (BLEU), which represents the standard metric for text generation tasks like machine translation and code generation. The metric compares the overlap of different-sized word groups (1-grams to 4-grams) between the generated comment and an original one. Additionally, we conducted a human evaluation of the review comments generated by CodeMentor. We engaged a group of eight professional software engineers to assess the quality of the code reviews produced by our framework. The feedback of the group focused on three main aspects:

- **Technical Accuracy:** Software engineers rated whether the review correctly identified issues and suggested improvements that were technically sound.

TABLE VI
EXPERIMENTAL RESULTS OF COMPARING TECHNIQUES ON CODE REFINEMENT

Model	BLEU	EM
Baseline Transformer	0.34	7.93
Pre-trained T5	81.31	17.44
CodeT5	82.11	33.63
CodeReviewer	79.83	37.05
CodeMentor	**86.17**	**43.71**

- **Clarity and Readability:** Software engineers rated how clearly the reviews are written and how easily one can read them.
- **Contextual Understanding and Relevance:** We evaluated the extent to which the reviews show an understanding of the code.

The feedback was collected using a 1-5 scoring scale for every aspect. We use the BLEU score and the exact match (EM) rate for the code refinement task. BLEU score and EM rate capture two different aspects of code generation. The BLEU score considers both exact and partial matches and reflects the overall similarity between the generated code and the ground truth, while the EM rate focuses on exact matches only and is thus a far stricter evaluation criterion. Using both metrics will ensure that the generated code is error-free and matches the target.

D. Experimental Results

In this section, we show the experimental results to evaluate the performance of our proposed model in each task and compare it with other models.

1) Experimental Results for Code Quality Evaluation

The results of the code quality evaluation task are shown in Table III. The table shows that CodeMentor outperforms all other models in terms of accuracy, precision, recall, and F1 score. For the sake of accuracy, CodeMentor is followed by CodeReviewer, CodeT5, T5, and, lastly, Transformer. More importantly, compared to the baseline Transformer, the proposed model improved the precision and recall scores by about 6% and 35%, respectively. Similarly, it outperforms CodeT5 by more than 22% in F1 score. These results indicate that the proposed fine-tuning approach makes the LLM better understand the code changes than the general-purpose models, such as Transformer or T5.

2) Experimental Results for Review Generation

Table IV illustrates the performance of the proposed model compared to other techniques in generating reviews. The results show that the proposed model offers the highest BLEU score among all models tested, which points out its efficiency in generating the code reviews that most fit the reference texts. Remarkably, CodeMentor outperforms the baseline transformer model and the pre-trained T5 model by an increase of more than 43% and 30%, respectively. It performs better than other models, including CodeT5 and CodeReviewer, to catch the lexically-based similarity required to generate accurate code reviews. Although the BLEU scores in all models are relatively low, this is expected due to some inherent difficulties of the generation tasks related to code reviews, which consist of very complicated language and terminology.

Moreover, since relevance evaluates the generated review related to specific code changes, CodeMentor attained a score of 3.8 in relevance, meaning that its reviews are highly relevant and appropriately targeted to the changes made in the code. This comprehensive evaluation underscores CodeMentor's strength and effectiveness in software engineering tasks.

```
{"id": "seed_task_1", "name", "review_comments_generation",
    "instruction": "Evaluate the following code block and write a suitable review comment.",
    "instances": [
    {"input":" the code block is: \n def divide (a, b): \n \t return a / b,
      "output": "The function `add` adds two numbers and returns the result. However, it lacks error handling for
non-numeric inputs. Consider adding type checks or try-except blocks to handle potential errors"],
    "is_classification": false}
{"id": "seed_task_2", "name", "refine_code",
    "instruction": " Please rewrite the provided code block based on the given review ",
    "instances": [
    {"input":" the code block is: \n def divide (a, b): \n \t return a / b",
, and the given review is: “Refactor the code to handle the case where the denominator b is zero, which will raise
a ZeroDivisionError."
      "output":" def divide (a, b):
                    try: return a / b
                   except ZeroDivisionError: return "Error: Division by zero is not allowed.],
    "is_classification": false}
```

Fig. 6. A sample of the instructions generated by the domain-heuristic functions in the seed dataset.

3) Experimental Results for Bug Report Summarization

In the bug report summarization task, Table V compares CodeMentor to other baselines, whereby the proposed model could return the highest BLEU score compared to all the other models tested in this research. While all models could obtain BLEU scores ranging from 4.31 to 5.13, CodeMentor outperformed the comparing techniques by up to 42%. This provides a substantial enhancement above the baseline and other approaches, hence its potential in improving the quality and accuracy of bug report summarization. Also, further techniques could be developed on summarization capabilities for CodeMentor through techniques such as context-aware modeling, fine-tuning using bug report-specific data, and leveraging domain-specific knowledge to improve the relevance and informativeness of generated summaries.

4) Experimental Results for Code Refinement

Moreover, Table VI shows the experimental results of the code refinement task. The results attest that CodeMentor performs remarkably better than other approaches, as evidenced by its highest BLEU score and exact match rate compared to all the models tested. The proposed model outputs review comments identical to the "ground truth" in more than 40% of cases. Compared to the baseline Transformer and pre-trained T5 models, which exhibit lower BLEU scores and EM rates, CodeMentor demonstrates significant advancements in code refinement capabilities. Additionally, while CodeT5 and CodeReviewer achieve competitive BLEU scores and EM rates, CodeMentor's superior performance suggests its effectiveness in capturing the intricacies of code refinement tasks. It could surpass both T5, CodeT5, and CodeReviewer by 6%, 5%, and 8%, respectively, in BLEU score. This demonstrates the model's ability to understand review comments and translate them into precise code revisions.

5) Experimental Results for Computational Cost

To assess the effectiveness of CodeMentor in optimizing the computational cost, we measured the total time required for evaluating CodeMentor and the baseline techniques. Since the experiments utilized pre-trained models, we could not report the training time required for these models. However, training such large models usually requires a massive corpus of data, which can take several weeks. Table VII shows the evaluation time required for CodeMentor and the baseline techniques to complete each task. Table VII shows that CodeMentor reduces the training time by 53.7% for quality estimation, 34% for code review generation, 30.8% for code refinement, and 32.3% for bug summarization relative to the baseline transformer model. This demonstrates the effectiveness of CodeMentor in handling these tasks.

Compared to other models, CodeMentor consistently exhibits lower training times, emphasizing its optimized design and enhanced computational efficiency. This reduction in training time translates to lower operational costs and faster deployment in practical applications. Additionally, the training and finetuning time required for CodeMentor is reported in Section IV. All experiments are conducted on a 40GB A100 GPU.

TABLE VII
COMPUTATIONAL COST REQUIRED FOR COMPARING TECHNIQUES

Model	Total Training Time (hours)			
	Quality Estimation	Code Review Generation	Code Refinement	Bug Summarization
Baseline Transformer	13.4	4.7	5.2	3.1
Pre-trained T5	10.2	3.2	4.9	3.4
CodeT5	13.1	4.6	7.1	4.1
CodeReviewer	12.1	3.8	4.5	3.3
CodeMentor	**6.2**	**3.1**	**3.6**	**2.1**

E. Ablation Study

To evaluate the individual contributions of each phase to the model's overall performance, we conducted an ablation study. This study systematically removes one component at a time and evaluates its impact on the model's performance. We evaluated the following setups of the model:

- **CodeMentor$_{\text{No\_Data}}$:** The model excludes the first phase. Instead, it fine-tunes the model using the benchmark datasets used in the literature [15,61,62]. Specifically, we allocate 70% of the datasets for training, 15% for validation, and 15% for testing. Thus, the model does not incorporate the custom domain heuristics or self-instruct model used in the first phase to create the instruction set. Then, the fine-tuned model is combined with the reward model obtained from the third phase to create the final model.
- **CodeMentor$_{\text{No\_LORA}}$:** The model does not apply LoRA to train. Instead, it fine-tunes Code Llama using the instruction set generated in the first phase. By omitting LoRA, the model does not benefit from the parameter-efficient fine-tuning that LoRA offers. This approach helps isolate and understand the impact of LoRA on the model performance.
- **CodeMentor$_{\text{No\_ RLHF}}$:** The model excludes the RLHF phase and utilizes the fine-tuned model created in the second phase as the final model. Without the RLHF phase, the model does not benefit from the additional layer of refinement that human feedback provides, which helps to align the model's predictions more closely with human expectations and improve its performance on specific tasks.

Table VIII presents the results of the ablation study. The results show that excluding organizational data utilization leads to the worst performance across all tasks. This highlights the critical importance of the domain heuristics and self-instruct model used in the first phase. For example, CodeMentor$_{\text{No\_Data}}$ accuracy in quality estimation is 26% lower than that of the original model. Without data preparation, the model struggles to achieve the same level of understanding and precision in all tasks. Alternatively, the performance of the model without LoRA fine-tuning does not significantly fall behind the original model. For example, the No LoRA model achieves an EM score of 41.75 in code refinement, only around 4% below the full model, indicating that core model capability is maintained by phases one and three, and LoRA is most valuable in its provision of efficiency gains and performance improvement through parameter-efficient fine-tuning. The main advantage of LoRA would be cutting down the need for computational power in training, thus making the process of fine-tuning more efficient.

The model without RLHF performs better than CodeMentor$_{\text{No\_Data}}$ but worse than the full model. Without RLHF, the model's performance is significantly worse, especially for tasks such as code review generation and bug summarization. This serves as a testament to the efficacy of RLHF in aligning the model's outputs with human expectations. The RLHF stage provides an important layer of optimization for the model through feedback, which enhances the model's understanding and performance for specific tasks. Even without RLHF, the model still benefits from instruction fine-tuning and organizational data, proving the significance of these components.

VI. Related Work

This section discusses work related to our research, which is at the intersection of automating software engineering tasks and applying LLMs to code review activities.

A. Automating Code-related Activities

Automating code-related tasks has always interested academia and industry for several decades. Several approaches have been proposed to generate, summarize, or test software code. With recent developments in natural language processing, large language models have come out, pre-trained on enormous text corpora. These models, such as GPT [44] and BERT [42], have acted amazingly in understanding and generating natural language texts. Recent work focuses on adapting the successful pre-trained NLP techniques to the code domain. Feng et al. introduce CodeBERT [69] to address the special challenges of understanding and generating code. Based on the architecture of BERT, the model learns bidirectional representations of context for code snippets. It is pre-trained on enormous-scale code repositories like GitHub and performs state-of-the-art tasks such as code completion, code summarization, or code documentation generation.

Similarly, Gao et al. [70] proposed training transformers with structural properties of code, like local symbolic information and data flow graphs. This approach embeds such information in the self-attention layer to capture the multi-layer structure of code, enhancing the model's capability in code summarization. Further research [71] has investigated the pre-training of a GPT model with more than 2.5 billion parameters. The pretraining included aligned code and the test files to capture the context of code. Then, the model is used to generate more valid tests. On the other hand, Jin et al. [72] proposed a transformer-based framework called InferFix to fix program bugs. The research [72] incorporates a dataset of similar bugs and their corresponding corrections and employs static analysis to identify bugs.

However, relying on static analysis techniques may overlook certain contextual and dynamic aspects of the code, leading to inaccuracies in identifying refactor-prone locations or predicting code defects. Also, most of these efforts [70,71] have explored pre-training large-scale language models, necessitating significant computational resources. For instance, the pre-training of GPT-3, which comprises 175 billion parameters, consumed approximately 355 petaflop/s-days of computational power [73]. Similarly, a typical pre-training session for CodeBERT may take around 1,000 GPU hours to complete on a cluster of high-performance GPUs. This massive computational undertaking spans several weeks and involves the utilization of specialized hardware accelerators. This demonstrates the substantial computational investment necessary to train state-of-the-art code generation models.

B. LLMs for Automating Code Review

Several techniques have been proposed for the automation of code review. For instance, Guo et al. [74] introduced a novel approach to learning meaningful code transformations by analyzing the code before and after going through pull requests. The system provides automated recommendations for code changes from reviewers to help developers get instant feedback upon submission through its code graph embedding and code transformation learning components. Another research [75] trains a model known as RefactorBERT for code quality assessment. The model is trained on over a million commits to predict refactor-prone locations in the code.

Beyond code generation, research has extended the application of machine learning and natural language processing to automate refinement tasks, such as automatic bug fixing and code completion. Wang et al. proposed CodeT5 [65], a unified, pre-trained encoder-decoder Transformer model that optimizes code semantics by using the force of developer-assigned identifiers. The new identifier-aware pre-training task proposed in the model uses user-written code comments, which align the alignment of natural and programming languages more closely. Similarly, Gu et al. [76] trained a large language model with a high-quality dataset of code bases. To clean the dataset, the source code goes through filtering stages to remove syntax errors. Then, the model is trained for code generation. Another research [77] utilizes ChatGPT to refine code based on a given review. First, ChatGPT is pre-trained using the dataset released by Li et al. [15]. Then, the model is fine-tuned to generate refined code and compared against CodeReviewer [15]. The results revealed that the proposed model [77] could outperform CodeReviewer in the code refinement task. However, the effectiveness of these approaches heavily depends on the quality and representativeness of the training data, which may introduce biases or limitations in the models' performance, especially when applied to diverse or evolving codebases. For example, the datasets used to train CodeT5 contain generated comments and source code from GitHub repositories and publicly accessible sources [65]. Hence, these datasets can incorporate biases stemming from textual comments and elements within the code itself. Also, training such models (e.g., CodeT5) can be computationally intensive and resource-demanding, making it challenging to scale up for real-world applications with limited computational resources.

TABLE VIII
EXPERIMENTAL RESULTS OF ESTIMATING THE CONTRIBUTIONS OF EACH PHASE IN CODEMENTOR

Model	Quality Estimation			Code Review Generation	Code Refinement		Bug Summarization
	Acc	**P**	**R**	**BLEU**	**BLEU**	**EM**	**BLEU**
CodeMentor$_{\text{No\_Data}}$	0.62	0.52	0.43	3.83	62.47	27.03	3. 31
CodeMentor$_{\text{No\_LORA}}$	**0.80**	**0.89**	**0.73**	4.81	**85.19**	**41.75**	**5.09**
CodeMentor$_{\text{No\_ RLHF}}$	0.78	0.79	0.71	**5.03**	71.26	37.28	3.23

Furthermore, none of these techniques have incorporated RLHF or parameter-efficient fine-tuning methodologies to augment their training processes. These approaches overlook the potential benefits that RLHF and parameter-efficient fine-tuning could offer to improve model performance and training efficiency in the context of code-related activities, which we precisely accomplish in this study.

VII. LIMITATIONS AND THREATS TO VALIDITY

In this section, we discuss the limitations of the proposed framework and the potential threats to the validity of our findings.

A. Limitations

Proposed framework limitations include data biases, availability of training data, and dependence on domain experts. First, the framework is particularly dependent on data from the organization. This dependence means the model is prone to the same biases as an organization's specific practices, preferences, and historical patterns, which may not be valid in other coding environments. In this case, the framework must be fine-tuned again to different coding environments. Also, considering the framework relies mainly on data from an organization, there is less chance of being exposed to a great variety of coding practices and styles present in more comprehensive public datasets. This limitation may lead to a lack of comprehension of code semantics and overfitting to specific data characteristics. However, in Section V, the experimental evaluation used a set of publicly available datasets to test and evaluate the proposed framework. Besides, we evaluated CodeMentor on multiple downstream tasks, showing model ability in generalization across different contexts. Experimentally, the insights obtained from the experiments empirically demonstrate that the framework can generalize well to new and unknown code bases.

Another limitation of CodeMentor is the computational resource constraints. Although the framework is designed to be efficient, fine-tuning LLMs remains resource-intensive. Organizations with limited computational resources may find it challenging to implement and maintain the framework, especially when dealing with large codebases or frequent model updates. Additionally, integrating reinforcement learning with human feedback requires domain experts to provide meaningful feedback for model refinement. This dependence can be a bottleneck, particularly where such expertise is scarce.

B. Threats to Validity

To ensure the robustness of our findings, we identified and addressed several threats to validity, categorized as construct validity, internal validity, and external validity:

1) Construct Validity

First, it is crucial to ensure that tasks for fine-tuning and evaluation reflect practical code review activities. While we included a diverse set of tasks, some aspects of code review may still be underrepresented. Second, the accuracy of the instructions generated in the training data is vital. We used a bootstrapping process and self-instruction techniques to enhance instruction diversity and relevance.

2) Internal Validity

The effectiveness of the third phase of the framework depends on the quality of feedback from domain experts. Clear guidelines and the involvement of multiple experts helped ensure diverse and consistent feedback.

3) External Validity

The first threat to external validity is the dataset representativeness since organizational data might not generalize to other contexts with different coding practices. We selected data from various projects to capture a broad range of scenarios, but further validation with external datasets is needed. Also, models may perform differently on codebases with distinct characteristics. Additional testing and fine-tuning are recommended when deploying the framework in new environments.

Hence, future work will involve further validation and refinement to enhance the framework's generalizability and applicability in different settings, such as different programming languages, different datasets, and integration with existing code review tools.

VIII. CONCLUSIONS

Code review plays a pivotal role in software engineering, ensuring code quality and adherence to standards. However, traditional manual review processes are time-consuming and error-prone, leading to a shift towards automating code review activities. Large Language Models have further advanced code review tasks. Nevertheless, pre-training and utilizing existing models present challenges regarding computational resources and privacy concerns. To address these challenges, this paper introduces CodeMentor, a novel framework for few-shot learning tailored to the organizational data. By leveraging heuristic rules, weak supervision techniques, and reinforcement learning with human feedback, CodeMentor enables the training of language models specifically for code review activities. To assess the proposed framework, the empirical evaluation considers four tasks related to code review. The experimental results illustrate that CodeMentor could perform better than four state-of-the-art techniques.

REFERENCES

[1] J. von der Mosel, A. Trautsch, and S. Herbold, "On the Validity of Pre-Trained Transformers for Natural Language Processing in the Software Engineering Domain," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1487–1507, 2023, doi: 10.1109/TSE.2022.3178469.

[2] R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli, and G. Bavota, "Code Review Automation: Strengths and Weaknesses of the State of the Art," *IEEE Transactions on Software Engineering*, vol. 50, no. 2, pp. 338–353, 2024, doi: 10.1109/TSE.2023.3348172.

[3] D. Badampudi, M. Unterkalmsteiner, and R. Britto, "Modern Code Reviews—Survey of Literature and Practice," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, 2023, doi: 10.1145/3585004.

[4] B. Wu, B. Liang, and X. Zhang, "Turn tree into graph: Automatic code review via simplified AST driven graph convolutional network," *Knowledge-Based Systems*, vol. 252, p. 109450, 2022, doi: https://doi.org/10.1016/j.knosys.2022.109450.

[5] K. A. Tecimer, E. Tüzün, H. Dibeklioglu, and H. Erdogmus, "Detection and Elimination of Systematic Labeling Bias in Code Reviewer Recommendation Systems," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 181–190. doi: 10.1145/3463274.3463336.

[6] 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.

[7] X. Ye, Y. Zheng, W. Aljedaani, and M. W. Mkaouer, "Recommending pull request reviewers based on code changes," *Soft Computing*, vol. 25, no. 7, pp. 5619–5632, 2021, doi: 10.1007/s00500-020-05559-3.

[8] H.-Y. Li *et al.*, "DeepReview: Automatic Code Review Using Deep Multi-Instance Learning," in *Advances in Knowledge Discovery and Data Mining: 23rd Pacific-Asia Conference, Macau, China*, 2019, pp. 318–330. doi: 10.1007/978-3-030-16145-3_25.

[9] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic Code Review by Learning the Revision of Source Code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 4910–4917, 2019, doi: 10.1609/aaai.v33i01.33014910.

[10] R. Chatley and L. Jones, "Diggit: Automated code review via software repository mining," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 567–571. doi: 10.1109/SANER.2018.8330261.

[11] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.

[12] L. Li *et al.*, "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*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 1009–1021. doi: 10.1145/3540250.3549099.

[13] Z. Li *et al.*, "On the feasibility of specialized ability stealing for large language code models," *arXiv preprint arXiv:2303.03012*, 2023.

[14] H. A. Çetin, E. Doğan, and E. Tüzün, "A review of code reviewer recommendation studies: Challenges and future directions," *Science of Computer Programming*, vol. 208, 2021.

[15] Z. Li *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.

[16] Y. Li, T. Zhang, X. Luo, H. Cai, S. Fang, and D. Yuan, "Do pre-trained language models indeed understand software engineering tasks?," *IEEE Transactions on Software Engineering*, 2023.

[17] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for Source Code and Natural Language Editing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, in ASE '22. New York, NY, USA: Association for Computing Machinery, 2023.

[18] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Advances in Neural Information Processing Systems*, 2022, pp. 24824–24837.

[19] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, and S. Paul, "PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods." 2022.

[20] Y. Wang, X. Wang, Y. Jiang, Y. Liang, and Y. Liu, "A code reviewer assignment model incorporating the competence differences and participant preferences," *Foundations of Computing and Decision Sciences*, vol. 41, no. 1, pp. 77–91, 2016.

[21] M. Fejzer, P. Przymus, and K. Stencel, "Profile based recommendation of code reviewers," *Journal of Intelligent Information Systems*, vol. 50, no. 3, pp. 597–619, Jun. 2018, doi: 10.1007/s10844-017-0484-1.

[22] B. Guo, Y.-W. Kwon, and M. Song, "Decomposing Composite Changes for Code Review and Regression Test Selection in Evolving Software," *Journal of Computer Science and Technology*, vol. 34, no. 2, pp. 416–436, Mar. 2019, doi: 10.1007/s11390-019-1917-9.

[23] M. Staron, M. Ochodek, W. Meding, and O. Söder, "Using Machine Learning to Identify Code Fragments for Manual Review," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications*, 2020, pp. 513–516. doi: 10.1109/SEAA51224.2020.00085.

[24] S. Mahbub, M. E. Arafat, C. R. Rahman, Z. Ferdows, and M. Hasan, "ReviewRanker: A Semi-Supervised Learning Based Approach for Code Review Quality Estimation," *arXiv preprint arXiv:2307.03996*, 2023.

[25] E. Fregnan, F. Petrulio, and A. Bacchelli, "The evolution of the code during review: an investigation on review changes," *Empirical Software Engineering*, vol. 27, no. 7, Sep. 2022, doi: 10.1007/s10664-022-10205-7.

[26] 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.

[27] B. Min *et al.*, "Recent Advances in Natural Language Processing via Large Pre-Trained Language Models: A Survey," *ACM Comput. Surv.*, vol. 56, no. 2, Sep. 2023, doi: 10.1145/3605943.

[28] K. Zhou, J. Yang, C. C. Loy, and Z. Liu, "Learning to Prompt for Vision-Language Models," *International Journal of Computer Vision*, vol. 130, no. 9, pp. 2337–2348, Sep. 2022, doi: 10.1007/s11263-022-01653-1.

[29] Z. Li, "The dark side of chatgpt: Legal and ethical challenges from stochastic parrots and hallucination," *arXiv preprint arXiv:2304.14347*, 2023.

[30] J. Cao, Z. Gan, Y. Cheng, L. Yu, Y.-C. Chen, and J. Liu, "Behind the Scene: Revealing the Secrets of Pre-trained Vision-and-Language Models," in *Computer Vision – ECCV 2020*, 2020, pp. 565–580.

[31] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "MPNet: Masked and Permuted Pre-training for Language Understanding," in *Advances in Neural Information Processing Systems*, 2020, pp. 16857–16867.

[32] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A Systematic Evaluation of Large Language Models of Code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, in MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1–10. doi: 10.1145/3520312.3534862.

[33] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356. doi: 10.1109/SP46215.2023.10179324.

[34] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," *arXiv preprint arXiv:2307.10169*, 2023.

[35] X. Rong, "word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[36] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2017, pp. 5998–6008.

[37] T. Shao, Y. Guo, H. Chen, and Z. Hao, "Transformer-Based Neural Network for Answer Selection in Question Answering," *IEEE Access*, vol. 7, pp. 26146–26156, 2019, doi: 10.1109/ACCESS.2019.2900753.

[38] Z. Shaheen, G. Wohlgenannt, and E. Filtz, "Large scale legal text classification using transformer models," *arXiv preprint arXiv:2010.12871*, 2020.

[39] M. Nashaat, A. Ghosh, J. Miller, and S. Quader, "TabReformer: Unsupervised Representation Learning for Erroneous Data Detection," *ACM/IMS Trans. Data Sci.*, vol. 2, no. 3, May 2021, doi: 10.1145/3447541.

[40] T. Luong, H. Pham, and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation," in *Proceedings of the 2015*

Conference on Empirical Methods in Natural Language Processing, Lisbon, Portugal, 2015, pp. 1412–1421. doi: 10.18653/v1/D15-1166.

[41] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional Sequence to Sequence Learning," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, in ICML'17. Sydney, NSW, Australia, 2017, pp. 1243–1252.

[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, 2019.

[43] Y. Liu *et al.*, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *arXiv:1907.11692 [cs]*, 2019.

[44] L. Floridi and M. Chiriatti, "GPT-3: Its Nature, Scope, Limits, and Consequences," *Minds and Machines*, vol. 30, no. 4, pp. 681–694, Dec. 2020, doi: 10.1007/s11023-020-09548-1.

[45] A. Ni *et al.*, "LEVER: Learning to Verify Language-to-Code Generation with Execution," in *Proceedings of the 40th International Conference on Machine Learning*, 2023, pp. 26106–26128.

[46] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT," *arXiv preprint arXiv:2308.04838*, 2023.

[47] X. Pu, M. Gao, and X. Wan, "Summarization is (Almost) Dead," *arXiv preprint arXiv:2309.09558*, 2023.

[48] T. Ahmed and P. Devanbu, "Few-Shot Training LLMs for Project-Specific Code-Summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, in ASE '22. New York, NY, USA, 2023. doi: 10.1145/3551349.3559555.

[49] 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*, IEEE, 2021.

[50] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," *arXiv preprint arXiv:2303.06689*, 2023.

[51] Y. Wang *et al.*, "Self-instruct: Aligning language model with self generated instructions," *arXiv preprint arXiv:2212.10560*, 2022.

[52] M. Nashaat, A. Ghosh, J. Miller, and S. Quader, "Asterisk: Generating Large Training Datasets with Automatic Active Supervision," *ACM/IMS Transactions on Data Science*, vol. 1, no. 2, 2020, doi: 10.1145/3385188.

[53] M. Nashaat, A. Ghosh, J. Miller, and S. Quader, "Semi-Supervised Ensemble Learning for Dealing with Inaccurate and Incomplete Supervision," *ACM Trans. Knowl. Discov. Data*, vol. 16, no. 3, 2021.

[54] E. J. Hu *et al.*, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[55] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27730–27744, 2022.

[56] B. Roziere *et al.*, "Code LLAMA: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[57] H. Touvron *et al.*, "LLAMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[58] Z. Zheng *et al.*, "A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends." 2024.

[59] W. Wang *et al.*, "Visionllm: Large language model is also an open-ended decoder for vision-centric tasks," *arXiv preprint arXiv:2305.11175*, 2023.

[60] J. Lu, L. Yu, X. Li, L. Yang, and C. 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)*, 2023, pp. 647–658. doi: 10.1109/ISSRE59848.2023.00026.

[61] Y. Koh, S. Kang, and S. Lee, "Deep Learning-Based Bug Report Summarization Using Sentence Significance Factors," *Applied Sciences*, vol. 12, no. 12, 2022, doi: 10.3390/app12125854.

[62] S. Banerjee, J. Helmick, Z. Syed, and B. Cukic, "Eclipse vs. Mozilla: A Comparison of Two Large-Scale Open Source Problem Report Repositories," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, 2015, pp. 263–270.

[63] H. A. Ahmed, N. Z. Bawany, and J. A. Shamsi, "CaPBug-A Framework for Automatic Bug Categorization and Prioritization Using NLP and Machine Learning Algorithms," *IEEE Access*, vol. 9, 2021.

[64] C. Qian, M. Zhang, Y. Nie, S. Lu, and H. Cao, "A Survey on Bug Deduplication and Triage Methods from Multiple Points of View," *Applied Sciences*, vol. 13, no. 15, 2023, doi: 10.3390/app13158788.

[65] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." 2021.

[66] C. Wu and M. Yan, "Learning Deep Semantic Model for Code Search using CodeSearchNet Corpus." 2022.

[67] S. Lu *et al.*, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation." 2021.

[68] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002.

[69] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[70] S. Gao *et al.*, "Code Structure–Guided Transformer for Source Code Summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, 2023.

[71] N. Rao, K. Jain, U. Alon, C. L. Goues, and V. J. Hellendoorn, "CAT-LM Training Language Models on Aligned Code And Tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 409–420. doi: 10.1109/ASE56229.2023.00193.

[72] M. Jin *et al.*, "InferFix: End-to-End Program Repair with LLMs." 2023.

[73] T. Brown *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, 2020, pp. 1877–1901.

[74] S. Guo, M. Li, X. Ge, H. Li, R. Chen, and T. Li, "Constructing meaningful code changes via graph transformer," *IET Software*, vol. 17, no. 2, pp. 154–167, 2023, doi: https://doi.org/10.1049/sfw2.12097.

[75] K. Jesse, C. Kuhmuench, and A. Sawant, "RefactorScore: Evaluating Refactor Prone Code," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 5008–5026, 2023, doi: 10.1109/TSE.2023.3324613.

[76] Q. Gu, "LLM-Based Code Generation Method for Golang Compiler Testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2023. New York, NY, USA, 2023, pp. 2201–2203. doi: 10.1145/3611643.3617850.

[77] Q. Guo *et al.*, "Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, in ICSE '24. New York, NY, USA, 2024. doi: 10.1145/3597503.3623306.

Mona Nashaat, received her BS and master's degrees in computer engineering from the Faculty of Engineering, Port Said University. She received her PhD in Software Engineering and Intelligent Systems from the Department of Electrical and Computer Engineering, University of Alberta. Her research focuses on software engineering, artificial intelligence, and big data.

James Miller received the BSc and PhD degrees in computer science from the University of Strathclyde, Scotland. After working as a principal scientist at the United Kingdom's National Electronic Research Initiative and then as a senior lecturer at the University of Strathclyde, he joined the University of Alberta as a professor. He has published over one hundred journal and conference articles and sits on the Journal of Empirical Software Engineering editorial board.