Code Comment Inconsistency Detection and Rectification Using a Large Language Model

Guoping Rong
Nanjing University
Nanjing, China
ronggp@nju.edu.cn

Yongda Yu*
Nanjing University
Nanjing, China
yuyongda@smail.nju.edu.cn

Song Liu
Nanjing University
Nanjing, China
522023320112@smail.nju.edu.cn

Xin Tan
Nanjing University
Nanjing, China
522023320141@smail.nju.edu.cn

Tianyi Zhang
Nanjing University
Nanjing, China
522023320208@smail.nju.edu.cn

Haifeng Shen
Southern Cross University
Gold Coast, Australia
haifeng.shen@scu.edu.au

Jidong Hu*
Zhongxing Telecom Equipment
Xi'an, China
hi.jidong@zte.com.cn

***Abstract*—Comments are widely used in source code. If a comment is consistent with the code snippet it intends to annotate, it would aid code comprehension. Otherwise, Code Comment Inconsistency (*CCI*) is not only detrimental to the understanding of code, but more importantly, it would negatively impact the development, testing, and maintenance of software. To tackle this issue, existing research has been primarily focused on detecting inconsistencies with varied performance. It is evident that detection alone does not solve the problem; it merely paves the way for solving it. A complete solution requires detecting inconsistencies and, more importantly, rectifying them by amending comments. However, this type of work is scarce. In this paper, we contribute *C4RLLaMA*, a fine-tuned large language model based on the open-source CodeLLaMA. It not only has the ability to rectify inconsistencies by correcting relevant comment content but also outperforms state-of-the-art approaches in detecting inconsistencies. Experiments with various datasets confirm that *C4RLLaMA* consistently surpasses both *post hoc* and *just-in-time* *CCI* detection approaches. More importantly, *C4RLLaMA* outperforms substantially the only known *CCI* rectification approach in terms of multiple performance metrics. To further examine *C4RLLaMA*'s efficacy in rectifying inconsistencies, we conducted a manual evaluation, and the results showed that the percentage of correct comment updates by *C4RLLaMA* was 65.0% and 55.9% in *just-in-time* and *post hoc*, respectively, implying *C4RLLaMA*'s real potential in practical use.**

***Index Terms*—Code-Comment Inconsistencies, Detection, Rectification, Large Language Model**

I. Introduction

Most code-related activities need program understanding. For example, when updating code, maintainers need to understand the logic of the original code. When developing code in teams, team members need to understand each other's code. The role of comments, which are widely present in code, is to assist readers in better understanding the code [1]–[3]. If a comment is consistent with the code snippet it intends to annotate, it serves its purpose well. Otherwise, Code Comment Inconsistency (*CCI*) would occur, which is detrimental to the understanding of code, but more importantly, it negatively impacts the development, testing, and maintenance of software. Common causes of *CCI* include developers' skill or time constraints [4], developers forgetting to update the comments when making changes to the code [2], [4]–[7], and developers failing to complete/modify the code in accordance with the requirement specifications presented in the comment [4]. TABLE I lists some examples of *CCI*. It is worth noting that these examples only intend to show different forms of *CCI*. They may seem minor; however, inconsistencies in real software projects can contain longer comments and more complex code structures, creating barriers to code understanding and further reducing the understandability and maintainability of code [8], [9].

Developers often rely on comments to understand the main functionality and interface specifications of external code libraries. *CCI* can lead to misunderstandings about the code's functionality and introduce defects during subsequent development [10]. The cost of fixing such defects is often high because they require developers to understand the source code, but they may also require costly communication with the code developers. In a teamwork environment, frequent *CCI* can erode developers' trust in comments [6], and a lack of confidence may make developers ignore comments or even abandon updating them altogether, leading to the failure of the entire code comment paradigm.

To address the *CCI* issues, existing research has been primarily focused on detecting inconsistencies using either the *post hoc* or the *just-in-time* approach. The *post hoc* approach aims to detect *CCI* issues in an existing code version, while the *just-in-time* approach aims to detect *CCI* issues before they are committed. Detection techniques include rule-based (i.e., with predefined rules or patterns) [4], [6] or learning-based (i.e., extracting features by analyzing code comments and related code and then learning the differences between inconsistent code and comments in the extracted features) [8], [9]. Rule-based techniques are easy to implement; however, they often fail to detect implicit or semantic inconsistencies due to the limited coverage of rules. Moreover, it is generally

Corresponding authors: Yongda Yu and Jidong Hu.

TABLE I
CCI EXAMPLES

Code & Comment	Explanation
// Read bandwidth to intermediate memory in GB per second. double intermediate read_gb_per_sec; //Read bandwidth to intermediate memory in GB per second double intermediate write_gb_per_sec;	From the name of the method, it looks like it should be `write` instead of `read`. It looks very much like when the author copied the code and comment, he forgot to change the comment at the same time.
/* * Checks if one of the graphs from unsupported graph type and *throws IllegalArgumentException if it is. The current *unsupported types are graphs with multiple-edges. - * @param graph1 - * @param graph2 +* @param g * @throws IllegalArgumentExceptionprotected */ protected static void assertUnsupportedGraphTypes(Graph g) throws IllegalArgumentException{...}	It looks like the two parameters `graph1` and `graph2` have been merged into 1 parameter `g` in the new version of the code, but the corresponding comment has not been changed, which can obviously be misleading.
/* * Returns the line number in the XML data where the exception occurred. * If there is no line number known, -1 is returned. */ public int getLineNr(){ return this.lineNr; }	It's not easy to tell if the comment correctly describes the functionality and the code doesn't give the full implementation or if the new version of the code ensures that there is no -1 return value. But in either case, the code here is inconsistent with the comment.

challenging to formulate a rule set that can cover a wide spectrum of situations, and as such, some inconsistencies may be misjudged or cannot be detected. While learning-based methods can detect inconsistencies that are difficult to express in rules [1], they require a large amount of labeled data to train the models, and the model performance depends heavily on the quality of the training data [2]. The performance of existing models varies and is generally not high.

It is evident that detecting inconsistencies alone does not solve the *CCI* problem; it merely paves the way for solving it. A complete solution requires detecting inconsistencies and, more importantly, rectifying them by suggesting revisions of incorrect comments.

However, research on the latter is scarce. To the best of our knowledge, there are only two pieces of published work, i.e., the studies conducted by Panthaplackel et al. [11] and Dau et al. [12]. In this paper, we propose *C4RLLaMA*, a fine-tuned large language model based on the open-source CodeLLaMA [13]. *C4RLLaMA* not only has the ability to rectify inconsistencies by correcting relevant comment content but also outperforms contemporary state-of-the-art methods for detecting inconsistencies. We have conducted extensive experiments and a manual evaluation to gauge *C4RLLaMA*'s effectiveness and efficacy in detecting and rectifying *CCI*. The main contributions of this work are as follows.

- We contribute to learning-based *CCI* detection techniques by exploiting a general-purpose large language model.
- We design a targeted loss function for fine-tuning CodeLLaMA so that *C4RLLaMA* can achieve improved *CCI* detection performance and, more importantly, rectify *CCI* by amending comments.
- We introduce a manual evaluation method to gauge *C4RLLaMA*'s efficacy in amending inconsistent comments as a complement to the experimental evaluation method that cannot consider the actual semantic meaning as it relies on text similarity-based metrics.

The rest of the paper is organized as follows. Section II introduces some related work. Section III describes the steps to design and implement *C4RLLaMA*, followed by the evaluation process and results in Section IV. Section V discusses some considerations in *C4RLLaMA*, followed by the validity risks pertinent to the study in Section VI. Section VII concludes the paper with a summary of future work.

II. RELATED WORK

CCI detection is presently a vibrant field of research, with a variety of methods being proposed [6], [11], [14]. These methods predominantly fall into two categories: rule-based and learning-based. However, the rectification of *CCI* issues has been relatively underexplored in the literature.

A. Rule-based CCI detection

Rule-based methods detect inconsistencies between code and comments by employing predefined rules or patterns. These rules may involve checking the consistency of parameter names, return values, exceptions, and so forth, between the code and its comments. This method is relatively straightforward to implement and does not require training data. However, its limitations lie in the restricted coverage of the rules, which often fail to detect certain implicit or semantic inconsistencies.

For instance, @tComment [6] is a rule-based method that identifies inconsistencies by testing the specifications in Javadoc comments. To be specific, it generates random tests through javadoc comments to test whether the code implementing the method is consistent with the content of the comments, with the main focus being on 'null' values or exception handling. However, it falls short of detecting whether the descriptions in the comments align with the code's other important behavior. Similarly, SmartCoCo [15] is a rule-based method that detects inconsistencies between code and comments in smart contracts. It employs constraint propagation and binding techniques to first extract constraints from code and comments, then propagate them to variables and functions, and finally verify their compliance with the binding condition. Gao et al. [16] utilize a rule-based method to discern the relationship between comments and code. They identify a set of rules to determine whether a code modification impacts the validity of a 'TODO' comment, and if it does, the comment is removed.

B. Learning-based CCI detection

Learning-based methods use machine learning or deep learning techniques to detect inconsistencies between code and comments. This is achieved by utilizing neural networks or

language models to ascertain the consistency between code and comments. Such methods may be able to detect inconsistencies that are difficult to articulate as rules. However, they necessitate a substantial volume of annotated data for model training, and the model's performance is heavily reliant on the quality of the training data [17], while the interpretability of the model may be poor [18]. For example, Rabbi and Siddik [19] use a bi-directional recurrent neural network to encode code and comments separately and then employ a fully connected layer to compute their similarity. A binary classification loss function is used to train the model to determine whether the code and comments are consistent. The work conducted by Steiner and Zhang [14] uses a pre-trained language model (BERT) to detect code and comment inconsistencies. It adopts a long text processing technique (Longformer [20]) to process code and comments that exceed the BERT limit and then designs a binary loss function to train the model to determine whether the code and comments are consistent. Panthaplackel et al. [11] uses a deep neural network to detect whether comments need to be updated when the code changes. It aligns the semantics of code and comments through an attention mechanism and then introduces a multicategorical loss function to train the model to determine whether comments need to be kept, updated, or deleted.

C. Detection timing: post hoc *VS.* just-in-time

Researchers have found that the consistency of code and comments may be closely related to code changes [21]. For example, when the functionality or logic of the code changes, the corresponding comments usually need to be updated accordingly, otherwise, inconsistency occurs. The co-evolutionary relationship between code and comments was explored by Fluriluri et al., who conducted experiments on three different open-source software systems and found that code and comments rarely co-evolve [22]. Thus existing research investigates different timing of detecting *CCI* problems. For example, some studies focus on the detection of *CCI* in existing source code repositories (aka, *post hoc* detection) [23]–[27], while others focus on the detection of *CCI* that occur immediately after code changes (aka, *just-in-time* detection) [16], [28]–[31]. These two different modes of detecting *CCI* have little impact on the rule-based approach. However, for the learning-based approach, as the 'Diff' information indicating the difference between the two versions can often be put into the training data, it may have some positive impact on the final performance of the model.

D. Pre-trained models in SE tasks pertinent to natural and programming languages

Pre-trained language models can learn universal language representations through pre-training on large-scale corpora, thereby achieving better performance on downstream tasks through fine-tuning [32]. Many researchers are trying to amalgamate Natural Language (NL) and Programming Language (PL) to achieve mutual conversion between NL and PL. For example, CodeBert [33] demonstrates superior performance in code search and document generation tasks by jointly training PL and NL. CodeReviewer [34] designs targeted pre-training tasks for NL and PL in code review scenarios, achieving good performance in PL review and NL review comment generation.

With the development of large language models (LLMs), researchers have begun to use them to automatically generate PL or NL. For example, CodeLLaMA [13] achieves the most advanced performance in open-source models in multiple code benchmark evaluations by pre-training and instruction fine-tuning on general text and code data. WizardCoder [35] enhances the performance of large model code generation by applying Code Evol-Instruct technology (generating higher quality datasets through self-instruct and instruction fine-tuning). To generate NL, Geng et al. [36] uses a small amount of context learning to make the large model Codex [37] perform better in the field of multi-intent comment generation than the most advanced supervised learning methods. The work carried out by Liang and Huang [38] introduces two different types of Transformer encoders that learn the non-Fourier and Abstract Syntax Tree (AST) structure relative position representation of the source code, thereby improving code semantics and syntax learning to be superior to other deep learning-based models from multiple metrics. In the field of code review, large models also have played a role. For example, Lu et al. [39] use large language models and Parameter-Efficient Fine-Tuning (PEFT) [40] technology to automate the code review process framework, maintaining a fairly high code review performance while reducing time and space costs.

In essence, the majority of the research mentioned above involves a unidirectional transformation between PL and NL, either converting PL into NL or vice versa. The focus of our work is on the matching problem of NL and PL, followed by rectifying inconsistent comments should the *CCI*issue exist, which is significantly different from the aforementioned studies. In this regard, we have found a limited number of studies. The work conducted by Panthaplackel et al. [11] uses GRU (which is not a pre-trained model) to detect and resolve *CCI*. Owing to temporal constraints, this work did not employ large language models such as LLaMA and GPT. Another work is DocChecker [12], which uses a pre-trained language model (Unixcoder [41]) to detect and resolve *CCI*. However, DocChecker only deals with the summary information, which is merely one of many types of code comment information.

III. Methodology

In this section, we will detail the methodology for building *C4RLLaMA*. As shown in Fig.1, the process mainly consists of: (1) constructing a large model training dataset; (2) defining optimization tasks for *CCI* detection and rectification; and (3) implementing *C4RLLaMA* by using low-parameter fine-tuning methods to fine-tune the pre-trained base large language model.

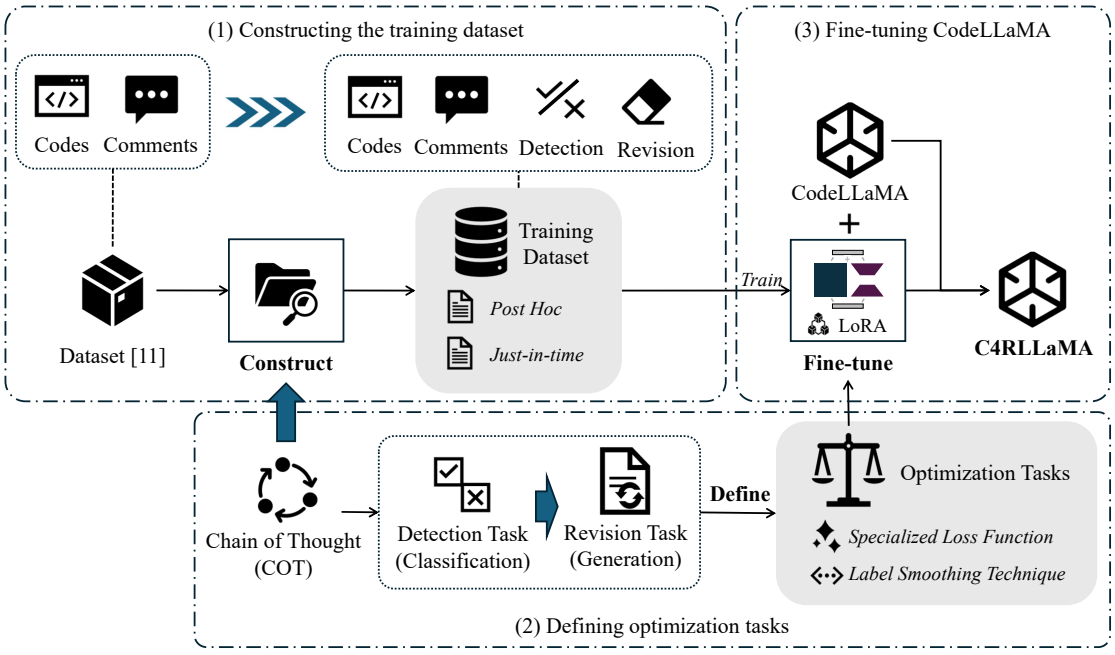

Fig. 1. The process of constructing *C4RLLaMA*

A. Problem statement

For *CCI* we adhere to a concept that has been consistently employed across multiple studies [11], [14]. The primary aim of our research is to harness the power of a large language model to understand textual data, which includes both code and its corresponding comments. This understanding enables us to identify inconsistencies between the two and subsequently amend the comments to rectify these inconsistencies. It is important to note that the comments discussed in this paper differ from code summarization [42], which is a one-way transformation from PL to NL. In contrast, our work involves two-way consistency detection, assuming the existence of both code and natural language. In many cases, the content of comments interspersed within lines of code does not precisely reflect the content encapsulated in most code summaries. In fact, the code summaries in the dataset [11] used in our study represent only one type of comment.

To facilitate a lucid and precise depiction of our approach, we initially define a set of symbols, as illustrated in TABLE II.

TABLE II
NOTATION DEFINITION

C_n	Comments of the nth version
M_n	Code implementation of the nth version
$I \in \{true, false\}$	Judgement result of the code comment consistency
Ins	Instructions for the detection and revision to *CCI* issues
R	Revision of the comments to resolve *CCI* issues

B. Constructing the training dataset

To ensure the validity of comparison with existing studies, we also use the dataset widely applied in several studies on the *CCI* topic [11], [14]. Curated from OSS projects, the dataset consists of 40688 data items, each of which contains a pair of consecutively submitted comments C and code M, denoted as $(C_1, M_1), (C_2, M_2)$, which includes comment elements such as '@return', '@param' and 'Summary'. Based on an underlying assumption that all consistency issues will be rectified promptly, a code change does not cause a consistency problem, $C_1 = C_2$; conversely, if the change raises a consistency problem, $C_1 \neq C_2$.

However, to match the input data format requirements of CodeLLaMA, we need to do some preprocessing of the data. First, we consider adopting Chain-of-Thought (COT) in the training process. COT has been proven to be effective in improving the performances of large language models [43], [44]. Its core principle lies in decomposing complex tasks and generating results step by step through recursion, thus avoiding overly complex reasoning processes. To enable COT, we first transform the data using a LLaMA [45] template shown in TABLE III. An example of using the template to process the original data is shown in TABLE IV, where we present the preprocessing results for both the *post hoc* and the *just-in-time* modes, respectively. We use a zero-shot prompting strategy, which is more comparable to the fine-tuning technology, i.e., to solve the problems without providing examples. Besides, as we want to support both modes of *CCI* detection and rectification at the same time, we need to further process the dataset to construct the training dataset. It is important to note that our processing of the training data only changes the format of the data to cater to CodeLLaMA and adds nothing new to the dataset.

1) Post hoc: *Post hoc* targets source code and comments that already exist in the repository and aims to detect whether

TABLE III
LLAMA TEMPLATE

[INST]
<<SYS>>
{{ system_prompt }} // System prompts for service providers to define model roles or restrictions, which we do not need to use.
<</SYS>>
{{ inputs }} // the input, i.e., the prompt triggers the model to perform consistency check as well as the source code and the corresponding comment
[/INST]
{{ outputs }} // the output, i.e., the result of consistency check and the revised comment if inconsistency occurs.

TABLE IV
DATA PROCESSING EXAMPLE

Post hoc example	[INST] <<SYS>><</SYS>> **Inputs** **Data type** Is the given code consistent with the corresponding summary? ```code public static JSONElement parse(InputStream is, String encoding) throws ConverterException { return parse(GrailsIOUtils.toString(is, encoding)); } ```**Data type** ```summary Parses the given JSON and returns ether a JSONObject or a JSONArry ``` **Label Of Detection Result** [/INST]The given code is inconsistent with the corresponding summary. The corresponding summary to the given code: ``` **Revision Outputs** Parses the given JSON and returns either a JSONObject or a JSONArray ``` **Outputs**
Just-in-time example	[INST] <<SYS>><</SYS>> **Inputs** **Data type** Did the changes cause any issues with consistency in the summary? ```changes <KEEP> private static format get sample format (format container format , format sample format) { if (container format = = null) { return sample format ; } <KEEP_END> <DELETE> int width = container format . width = = - 1 ? format . no _ value : container format . width ; int height = container format . height = = - 1 ? format . no _ value : container format . height ; <DELETE_END> <KEEP> return sample format . copy with container info (container format . id , container format . bitrate , <KEEP_END> <INSERT> container format . <INSERT_END> <KEEP> width , <KEEP_END> <INSERT> container format . <INSERT_END> <KEEP> height , container format . language) ; } <KEEP_END> ```**Data type** ```summary Derives a sample format corresponding to a given container format, by combining it with sample level information obtained from a second sample format. ``` **Label Of Detection Result** [/INST]The given changes is consistent with the corresponding summary. **Outputs**

there is a *CCI* issue. The task is thus defined as follows: first, the code and its comments are analyzed to determine whether there is a *CCI* issue, denoted as $(Ins, C, M) \rightarrow I, I \in \{true, false\}$. Considering the dataset characteristics, we chose to use M_2 as M and C_1 as C as a way to implement the judgment of whether the code modification is consistent with the original comment. i.e., when $C_1 = C_2$, it is determined that $I = false$, meaning there is no *CCI* issue; conversely, when $C_1 \neq C_2$, it is determined that $I = true$, meaning there is one *CCI* issue. Next, when $I = true$, a revision to address *CCI* is required, denoted as $(Ins, C, M, I) \rightarrow R$. In our dataset, the revision result is denoted as $R = C_2, (C_1 \neq C_2)$. Based on the above definition, we have developed Python scripts to process the original dataset [11] to construct the dataset for *CCI* detection and revision in *post hoc* mode.

2) Just-in-time: Just-in-time targets the code commit scenario, aiming at detecting *CCI* issues before the code is committed to a repository. The task is defined as follows: first, determine whether there is a *CCI* issue by analyzing code changes and their comments, denoted as $(Ins, C, Diff(M_n, M_{n+1})) \rightarrow I, I \in \{true, false\}$. In the dataset of this study, we chose to use M_1 as M_n, M_2 as M_{n+1}, and C_1 as C, as a way to implement the determination of whether a change triggers *CCI* issues: when $C_1 = C_2$, it is determined that $I = false$, meaning there is no *CCI* issue; conversely, when $C_1 \neq C_2$, it is determined that $I = true$, meaning there is a *CCI* issue. Next, if there is a *CCI* issue, i.e., $I = true$, a revision to the corresponding comment is required, denoted as $(Ins, C, Diff(M_n, M_{n+1}), I) \rightarrow R$. For this dataset, the revision results in $R = C_2, (C_1 \neq C_2)$. Similarly, we also process the original dataset and construct the dataset for training *C4RLLaMA* in *just-in-time* mode.

C. Defining Optimization Tasks

To enable COT in model training and inference, the tasks that come first in the COT significantly affect the results of the subsequent tasks [43]. For this, the *CCI* detection, which is essentially a judgment task, may significantly affect the performance of subsequent *CCI* rectification tasks. We therefore custom-design a loss function to increase the weight of the judgment task to highlight the importance of the accuracy of the judgment task, as follows:

$$\mathcal{L}_{ID} = \alpha \mathcal{L}_I + (1 - \alpha) \sum_{i=1}^{n} logP(x_i | x_{<i})$$

where α represents the weight of the code-comment consistency judgment task, usually α takes the value of 0.5, and $P(x_i|x_{<i})$ denotes the probability distribution of token x_i predicted by the model based on the input sequence $x_{<i}$, which is a commonly used method to calculate the loss in fine-tuning large models.

We note that the original dataset [11] has a small number of labeling issues, which also have been confirmed by other researchers [46]. To mitigate the noise brought by mislabelling, we employ the Label Smoothing (LSM) technique [47] and define the following sub-loss function for the *CCI* detection task:

$$\mathcal{L}_I = log((1 - \epsilon)P(I|Ins, C, M) + \frac{\epsilon}{K})$$

where ϵ is the degree of smoothing of categorical labels, generally taken as $\epsilon = 0.1$, and $P(I|Ins, C, M)$ denotes the probability distribution of judging the consistency of the comments and code implementation, given the model directives, the comments and the code implementation. K is the length of the word list of the large language model. The method

mitigates the data noise problem by preventing the model from giving overly deterministic answers to noisy data.

D. Fine-tuning CodeLLaMA

Fine-tuning can significantly enhance the ability of large language models to solve problems for specific tasks. We also use the LLaMA template (as shown in TABLE III) for training and inference. Large language models typically have a large number of parameters, and fully fine-tuning them tends to require significant computational resources [48]. Therefore, we adopt a low-parameter fine-tuning strategy to fine-tune CodeLLaMA, a highly-regarded model within the open-source community. CodeLLaMA is built upon the LLaMA2 model [45] and utilizes code data for complementary pre-training, which has demonstrated state-of-the-art performance across numerous code benchmark evaluations and is used as a base model for fine-tuning in a variety of software engineering tasks [49], [50]. Specifically, we chose the Lora [51] method as the low-parameter fine-tuning scheme for the follow-up task. Lora can achieve similar results to full-parameter fine-tuning by using only one-thousandth to one-ten-thousandth of the original model parameter for fine-tuning. LoRA assumes that the parameter changes during the fine-tuning phase have a low intrinsic rank, allowing the parameter changes to be decomposed into the product of low-rank matrices, i.e., $W' = W_0 + \Delta W = W_0 + BA$. Here, W' represents the fine-tuned model parameters, W_0 is the set of pre-trained model parameters, ΔW is the change in model parameters after fine-tuning, $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, with d and k being the dimensions of the model parameters, and satisfying $r \ll \min(d, k)$. During training, the original pre-trained parameter set W_0 is frozen and does not participate in gradient updates; only B and A are updated. Since the number of parameters in the low-rank matrices is much smaller than that of the original model matrix, it allows for fine-tuning the large model with a minimal number of parameters. To further reduce the training cost and improve the convergence speed, we use the Lion (Evo**L**ved S**i**gn M**o**me**n**tum) optimizer [52]. The fine-tuning of the model was performed on 2 A100 40GB graphics cards, training was performed using bf16 precision, and the hyperparameters were set as shown in TABLE V.

TABLE V
TRAINING HYPERPARAMETERS

Param	Lora r	Lora Dropout	Learning Rate	Batch Size	Epoch
Value	8	0.05	1e-4	32	10

IV. EVALUATION

In this section, we evaluate our approach *C4RLLaMA* for *CCI* detection and rectification against several state-of-the-art approaches we have identified. The entire evaluation process is depicted in Fig.2, aiming to answer the following two research questions:

- RQ1: How does *C4RLLaMA* perform on the *CCI* detection tasks for both the *post hoc* and the *just-in-time* modes?
- RQ2: How does *C4RLLaMA* perform on the *CCI* rectification tasks for both the *post hoc* and the *just-in-time* modes?

From a practical point of view, a good *CCI* solution should first perform well in detecting *CCI* issues, followed by its ability to provide accurate revision to reduce developers' efforts in rectifying these *CCI* issues.

A. Experimental settings

As the existing studies are predominantly on *CCI* detection and very few have addressed *CCI* rectification, we use different evaluation settings for RQ1 and RQ2 in our experimental design.

1) CCI detection task evaluation (RQ1): We utilized the widely used dataset from previous work as our training and testing dataset [11], [12], [14] and also adopted the metrics applied in these studies for evaluating *C4RLLaMA*'s accuracy performance including *Precision*, *Recall*, *F1-score* and *Accuracy*. To ensure a fair and consistent evaluation, we established specific selection criteria for the benchmark approaches. If retraining was necessary, we mandated that these approaches utilize the identical dataset as employed in our study. Additionally, we required the algorithm or code to be either provided or publicly accessible. In cases where retraining was not required, we opted for commercially available large language models, specifically ChatGPT and GPT4, recognized for their robust performance. Consequently, we identified the following approaches for benchmarking.

- CodeBERT BOW [11]: Build on CodeBERT [33], this approach can support both *post hoc* and *just-in-time* tasks with relatively good performance.
- SEQ, GRAPH & HYBRID: Three new baseline approaches proposed by Panthaplackel et al. [11] to detect *CCI* issues, which are distinguished by different encoding methods for different contents,i.e., comment, AST tree, respectively. SEQ encodes comments using the GRU network, GRAPH encodes comments using the AST tree, and HYBRID is computed using a multi-head attention mechanism combining the above two encoders.
- BERT & Longformer [14]: Steiner et al. used BERT and Longformer to detect *CCI*. By utilizing the feature of Longformer's ability to handle longer sequence lengths to reduce the information loss, it achieves state-of-the-art performance for the *CCI* detection tasks.
- DocChecker [12]: Proposed by Dau et al., it achieves fairly good performance by supplementary pre-training based on UniXcoder [41]. As the paper only provides performance evaluation on the *post hoc* task, we only compare it on the *post hoc* task.
- ChatGPT&GPT 4: As the most famous large language models that can effectively accomplish a variety of software engineering tasks, we use 0-shot prompt engineering to drive them and realize the *CCI* detection task.

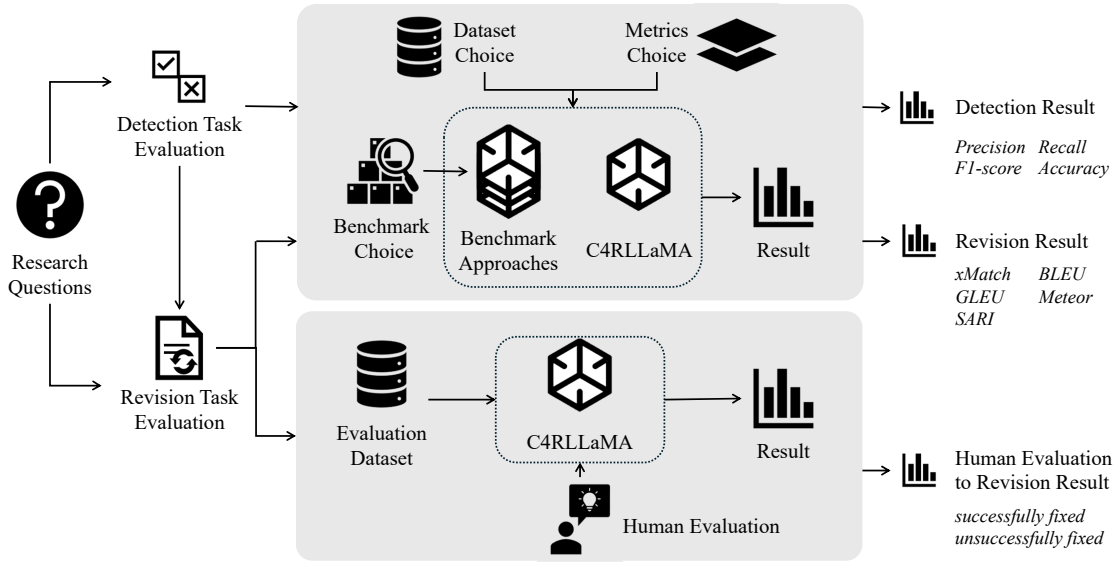

Fig. 2. The process of evaluate *C4RLLaMA*

2) CCI rectification task evaluation (RQ2): As previously discussed, research on the rectification of *CCI* issues is notably sparse. We identified only two studies that provide solutions for *CCI* rectification. However, DocChecker [12] utilizes a different dataset in the *CCI* rectification task, leaving us with only one benchmarking study, namely the work conducted by Panthaplackel et al. [11]. The evaluation metrics used in our study were exactly derived from Panthaplackel et al.'s work [11], i.e., *xMatch*, *BLEU-4* [53], *GLEU* [54], *Meteor* [55], and *SARI* [56]. While the rest are commonly used metrics, *xMatch* tests the extent to which two texts are identical. The test dataset was constructed by following the baseline approaches, i.e., by copying consistent comments when there is no *CCI* issue and using corrected comments generated by the rectification approach when there is a *CCI* issue. We use human evaluation as complementary evidence to further assess the efficacy of the revision content, which is mainly done by evaluating the semantics to see whether the modification has really solved the *CCI* issue. There are only two types of evaluation results, *successfully fixed* and *unsuccessfully fixed*, according to the following criteria: *successfully fixed* requires that the resulting new comment is semantically related to the corresponding code and does not contain explicit errors; otherwise, it is considered *unsuccessfully fixed*. The criteria provide a clear basis to determine whether an incorrect comment had been correctly rectified, which is fairly achievable by senior students majoring in software engineering. We then randomly selected 800 entries (at a 95% confidence level with a confidence interval range of less than 4%) to serve as the dataset for human evaluation.

B. Evaluation results

We present the evaluation results in this subsection.

1) RQ1: Detecting CCI issues: Results are presented in TABLE VI and TABLE VII. It is clear that our proposed *C4RLLaMA* approach significantly outperforms various previous approaches, both in *post hoc* and *just-in-time* modes. Specifically, in *post hoc* mode, our fine-tuned 7B model achieves *F1* and *Accuracy* of 89.0% and 89.6%, respectively, which are 2.6% and 2.7% better than Longformer, the state-of-the-art approach. It is worth mentioning that ChatGPT and GPT4 do not seem to show an advantage in detecting *CCI* issues, which, to a fair degree, confirms the importance of fine-tuning to improve the performance of large language models on specific tasks. Meanwhile, in *just-in-time* mode, our method *C4RLLaMA* also performs well, with *F1* and *Accuracy* reaching 84.1% and 84.5%, respectively, which are 3.2% better than 'GRAPH+feature' and 2.7% better than 'SEQ+feature', two state-of-the-art approaches. ChatGPT and GPT4 have very high *Recall*s in this mode, which, however, is dragged down by the lower *Precision*; hence their *F1* is only about 66.7%, and their *Accuracy* is only around 50%. In particular, it should be noted that to obtain better model performance, in *just-in-time* mode, SEQ, GRAPH, and HYBRID all add extra tags to the code 'Diff' information, and these tags change the original structure and content of the 'Diff', which we suspect degrades

TABLE VI
RESULTS FOR *post hoc CCI* DETECTION .

Approach	Precision	Recall	F1	Accuracy
CodeBert BOW	68.9	73.2	70.7	69.8
SEQ	60.6	73.4	66.3	62.8
GRAPH	62.6	72.6	67.2	64.6
HYBRID	56.3	80.8	66.3	58.9
BERT	72.1	71.9	72.0	72.1
Longformer	92.7	81.0	86.4	87.3
DocChecker	-	-	74.3	72.3
gpt-3.5-turbo-1106	62.6	62.0	62.3	62.5
gpt-4-0125-preview	59.2	81.7	68.7	62.6
CodeLLaMA-7B	**94.0**	**84.6**	**89.0**	**89.6**

TABLE VII
RESULTS FOR *just-in-time CCI* DETECTION.

Approach	Precision	Recall	F1	Accuracy
CodeBERT BOW	67.4	76.8	71.6	69.6
SEQ	80.7	73.8	77.1	78.0
GRAPH	79.8	74.4	76.9	77.6
HYBRID	80.9	74.7	77.7	78.5
SEQ + features†	88.4	73.2	80.0	81.8
GRAPH + features†	83.8	78.3	80.9	81.5
HYBRID + features†	88.6	72.4	79.6	81.5
BERT	76.4	78.3	77.3	77.1
Longformer	80.3	76.5	78.3	78.8
gpt-3.5-turbo-1106	50.0	**100.0**	66.7	50.0
gpt-4-0125-preview	51.5	98.8	66.7	52.8
C4RLLaMA	85.8	82.6	84.1	84.5
C4RLLaMA with standard Diff*	**95.9**	87.3	**91.4**	**91.8**

† As a variation, the 'features' are derived from their prior work [11], which typically includes linguistic (e.g., POS tags) and lexical (e.g., comment/code overlap) features.
* We use the standard formatted 'Diff' to evaluate *C4RLLaMA*.

the performance of the general purpose large language model. Therefore, we processed the 'Diff' information to restore its original form and re-evaluated *C4RLLaMA*, and the results are shown in the bottom row of TABLE VII. In the case of using the standard format 'Diff', the *C4RLLaMA*'s metrics for detecting *CCI* in *just-in-time* mode improved significantly, with its *F1* and *Accuracy* remarkably reaching 91.4% and 91.8%, respectively, which largely corroborates the previous speculation that the special 'Diff' format drags down the *C4RLLaMA*'s performance. The performance of an LLM is largely influenced by the data in the pre-training [57]. The just-in-time dataset [11] uses a customized special format of 'Diff', while the post-hoc dataset uses the regular 'Diff' format. As CodeLLaMA [13] does not reveal the details of its pre-training data, we can only speculate that its training data used the regular instead of the special format of 'Diff', leading to the performance discrepancy between the two datasets.

2) RQ2: Rectifying CCI issues: The *C4RLLaMA* approach also exhibits a distinct advantage in rectifying *CCI* issues, as depicted in TABLE VIII. In Panthaplack et al.'s work [11], the researchers employed diverse strategies for model training. These included a pre-training strategy encompassing solely positive examples and a joint training strategy incorporating both positive and negative examples. From the results in TABLE VIII, it is evident that the *C4RLLaMA* approach emerges as the superior performer in both *post hoc* and *just-in-time* modes on all four metrics including *BLEU-4*, *GLEU*, *SARI* and *Meteor*, reflecting the clear advantage of the large language model in resolving *CCI* issues. In addition, akin to *CCI* detection, the format of the 'Diff' continues to exert a significant influence on the output results as the *C4RLLaMA* approach performs much better with the standard formatted 'Diff' information. The sole metric where the *C4RLLaMA* approach falls short of other approaches is xMatch. This can be primarily attributed to the fact that large language models excel at content generation, whereas the *xMatch* metric requires exact content matches. Consequently, it is unlikely that the *C4RLLaMA* based on the large language model could exhibit an advantage in the *xMatch* metric, even if it yields reasonable revisions.

It should be noted that metrics such as *BLEU-4*, *GLEU*, *SARI*, *Meteor*, and *xMatch* are based on the level of text similarity. They are not the best for evaluating the efficacy of rectifying *CCI* issues, which apparently can only revise comments in the form of natural language. In the first example in TABLE I, a correct change may take many forms such as 'write', 'to write', or 'it is to write'. In turn, even if with the original incorrect form (i.e., keep 'Read' unchanged), the text similarity metrics may still be very high. This inevitably results in the need to understand and analyze the semantic meaning of the revised comment to determine whether the *CCI* issue has been appropriately solved. To this end, we performed a human evaluation of the results created by *C4RLLaMA* to rectify *CCI* based on sampling. The result is depicted in Fig. 3. We can observe that in *post hoc* mode, *C4RLLaMA* can successfully fix 65% of *CCI* issues by revising the comment content, while in *just-in-time* mode, the success rate is 55.9%, indicating *C4RLLaMA*'s high potential for practical applications. We performed a deep analysis of the cases that were determined to be 'unsuccessful fixes', and we identified several situations that are worth noting for future work. First, there are some cases where *C4RLLaMA* only rephrased the inconsistent comment. Although issues such as misspellings were properly rectified, the semantic meaning was still incorrect. This seems to imply that the 7B model we were using in this study has room for improvement in terms of correctly understanding user intent. Second, some *CCI* issues involve external information (e.g., even changes in requirements), which are rather difficult to resolve. Nevertheless, we believe that *CCI* rectification still has a chance to preserve some additional considerations of the original developers at the time of developing the code with respect to other entities associated with the current block of code compared to the use of code summarization techniques for comment-like generation, and thus it is worthwhile to explore better solutions to the *CCI* issue.

TABLE VIII
RESULTS ON *CCI* REVISION.

Approach	xMatch	BLEU-4	GLEU	SARI	Meteor
Jointly trained SEQ	62.3	76.6	75.6	42.0	75.9
Jointly trained GRAPH	59.4	76.6	75.8	42.5	75.1
Jointly trained HYBRID	**62.3**	76.9	75.9	42.3	75.6
Pretrained SEQ	61.4	77.0	76.2	42.4	75.6
Pretrained GRAPH	60.8	76.6	75.8	41.8	74.9
Pretrained HYBRID	61.6	77.2	76.4	42.3	75.8
C4RLLaMA Post hoc	50.0	81.7	82.0	85.2	88.5
C4RLLaMA just-in-time	50.0	77.5	77.5	82.0	86.8
C4RLLaMAjust-in-time with standard 'Diff'*	50.0	**82.6**	**83.0**	**86.7**	**89.2**

* We use the standard formatted 'Diff' to evaluate *C4RLLaMA*.

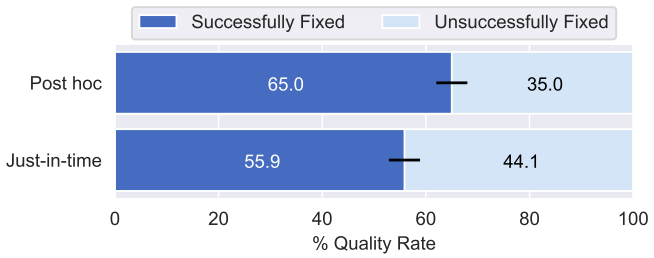

Fig. 3. Human evaluation of the results on *CCI* revision.

V. Discussion

In this work, we fine-tuned a large language model (i.e., CodeLLaMA) to detect and rectify *CCI* issues and conducted empirical studies on publicly available datasets to confirm the validity of our proposed *C4RLLaMA* approach. In this section, we discuss some considerations in our work so far.

a) The reasons for the better performance of C4RLLaMA: The task of code-comment consistency checking necessitates a comprehensive understanding of both the code and comment content, as well as the ability to discern discrepancies between them. This requires the applied approach to possess extensive knowledge to perform these complex logical comparisons. Large language models, pre-trained on vast amounts of data, have the potential to excel in this task by gaining a profound understanding of both code and language [57]–[59]. During the fine-tuning phase, we devise training algorithms that concentrate a portion of the model's capabilities on the task of *CCI* detection and rectification, thereby enhancing the large language model's capabilities. As some studies [11], [46] revealed that noisy data cannot be completely eliminated from datasets curated from Open Source Software (OSS) projects, the loss function we designed in *C4RLLaMA* helps to mitigate the negative impact of this noisy data, thereby improving the final performance in terms of *CCI* detection and rectification.

b) Other implications of resolving CCI issues: Since most large language models, including those oriented towards code generation, do not disclose the details of the dataset used for training, there is currently no information on whether comments are included along with the code fed to the model when it is pre-trained [13], [45]. However, many studies have confirmed that for large language models, data quality is a key factor in model performance [58], [59]. The inclusion of a substantial number of inconsistent comments in the code used for training these models could invariably compromise their performance. In light of this, the research presented in this paper could offer valuable insights for enhancing the quality of code used in training large language models for code-related tasks, particularly from the standpoint of eliminating inconsistent comments.

c) Future improvements: Although *C4RLLaMA* has performed quite well according to the results of this paper, there are still a lot of directions to be explored and problems to be solved. First of all, *C4RLLaMA* currently only uses the model of CodeLLaMA 7B. However, many studies have shown that the amount of model parameters has a significant impact on the resultant model performance [43], [60]. Therefore, It is worth using a large language model with more parameters for the base model. Also, as new and more advanced large language models continue to be released, it is promising that these more capable models can yield better results. Secondly, the data used in this paper is more likely to be examined for just a single comment at a time, which is relatively less difficult. In real-world scenarios, there are often interlinear comments, code documents, and other scenarios with complex semantics, which require the model to fully understand the semantics of the code and natural language, and even include complex logical and mathematical transformations that exist between some of the documents and the code. Although a large language model presents the potential to deal with these types of problems, a lot of exploratory work is still needed.

d) Exploring more proactive ways to solve the comment issue: Comment is undoubtedly helpful to code readers and is hence one of the integral software artifacts. In this paper, as well as in previous related studies, we note that an ex post facto approach to *CCI* issues has generally been taken, i.e., waiting until an *CCI* issue occurs (which is the case even with the *just-in-time* mode, except that it has not been committed to the repository of the code) and then trying to figure out how to correct an error that has already occurred, thus attempting to match the code with improved comments. However, the emergence of the large language models, and in particular the emergence of multiple programming assistants geared towards programming based on large language models, prompt us to wonder if there are more proactive ways to deal with the challenge of commenting code. For example, training a large language model to generate comments directly while programming is a reasonable way for the programmer to pick a more appropriate comment. Then, by integrating this feature in IDEs and using *C4RLLaMA* with the *just-in-time* mode, it is clear that most of the *CCI* issues can be avoided. This paper reveals, to a large extent, that large language models have impressive capabilities of understanding both source code and natural language, and in this sense, a more proactive means of coping with comments is already emerging. What is lacking may be their practical use, which should be explored as a future research direction.

VI. Threats To Validity

This section discusses some of the factors that could potentially impact the findings and conclusions of our study.

The claim regarding large language model: We do not intend to stir up controversy, and in fact, there are no well-recognized criteria or definitions for distinguishing large language models. In this paper, we refer to the general-purpose large language models represented by ChatGPT and LLaMA. These large language models are significantly different from previous models, such as Bert and Longformer, in terms of model structure and the number of parameters, and it is also

evident that the performance of large models is significantly improved by the phenomenon of 'emergence' due to a large number of parameters [57].

The limited dataset: This study employs a dataset, originally compiled by a prior researcher, that exclusively contains data in the Java language. The dataset is limited to three specific data types relevant to comments: 'Summary', '@param', and '@return'. Consequently, the task of *CCI* detection is inherently constrained by these types of comment data to a certain extent. It is widely acknowledged that the performance of machine learning models is largely dictated by the quality and relevance of the data they are trained on. While large language models can achieve a degree of generalization across different programming languages due to pre-training, they still require training on meticulously curated data (e.g., data in the relevant language, specific comment types, etc.) to yield optimal results. This underscores the importance of careful data preparation in the pursuit of more desirable outcomes in machine learning tasks.

The definition of consistency/inconsistency: We take the same treatment of consistency/inconsistency in our work as in previous studies, i.e., since the datasets provided in these studies are labeled and already have a clear distinction between consistent and inconsistent items, we directly follow these labels to determine whether there is a *CCI* issue. However, we also found that there are naturally multiple ambiguous interpretations of so-called consistency, and there are clearly differences in the magnitude of the impact of the *CCI* issue. We find that these phenomena have not been addressed in a targeted manner in existing research so far and undoubtedly have some implications for the conclusions of related studies (e.g., do some of the inconsistencies, similar to those in TABLE I, really need to be addressed?). This paper is no exception. Nevertheless, as we have made great efforts to ensure the validity of comparisons, we believe the conclusions of this paper are still correct. However, from the perspective of guiding practice, a more nuanced distinction between *CCI* issues may be needed.

The baseline labeling issue regarding dataset: The dataset from previous researchers used in this study assumed that the collected items were relatively well maintained, i.e., that all *CCI* issues were fixed in a timely manner, a requirement that the researchers who provided the dataset also acknowledged would be difficult to fully satisfy in reality. This may have led to a small number of untimely modifications being recognized in the dataset as not having *CCI* issues. And we also found that some of the *CCI* rectifications were not fixing code comments, but rather fixing spelling. These facts could have a negative effect on the performance of *C4RLLaMA* regarding accuracy. However, from our sampling results, this type of phenomenon does not occur frequently, so we believe this risk is manageable.

Unable to replicate original algorithm for comparative evaluations: Owing to the absence of certain information in the original dataset (e.g., AST tree), we were unable to replicate the algorithms in study [11] with complete accuracy. However, we believe this does not affect the comparative evaluation presented in VI,VII, and VIII as the data is presumed to represent the optimal performance of the benchmark methods in their respective studies. Despite this, it did prevent us from conducting a human evaluation to compare the effectiveness of the *CCI* issue rectification between our *C4RLLaMA* method and the benchmark methods [11]. Consequently, it remains an open question as to whether the rectification results of the *C4RLLaMA* method at the semantic level outperform those of the benchmark methods

No comparison with code summarization techniques: Code summarization enables developers to quickly understand a given implementation by generating natural language from the given code implementation. Ideally, developers can avoid *CCI* issues by using code summarization to generate various comments after modifying the code implementation. Alternatively, after identifying the *CCI* issue, the developer can simply delete the corresponding comment and use the content generated by the code summarization as the new code comment. However, considering that code summaries and comments do present not exactly the same content, and our dataset is specific to code comments. As such, it is impossible to conduct a fair comparison. Therefore, we did not analyze code summaries comparatively during the evaluation process.

Exclusion of GPT series in CCI rectification comparative study: In our comparative study of *CCI* rectification, we did not include the GPT series of large language models. This omission does not preclude the potential of these models to outperform others in addressing *CCI* issues. However, it is important to note that the GPT series has demonstrated suboptimal performance in detecting *CCI* issues. It is generally more practical for practitioners to first detect *CCI* issues before attempting to rectify them. Therefore, the effectiveness of the GPT series in *CCI* revision remains an open question.

Constraints to the large language model: As the training of large language models consumes a lot of resources, we only carried out the related work described in this paper on CodeLLaMA, one of the most representative code large language models. Also, due to resource constraints, we only used a model of 7B parameters. In fact, there are also several large language models oriented to coding tasks, each of which also has several different parameter size settings. All these factors obviously have an impact on the results of our study. Limited by time and GPU resources, we were not able to conduct experiments on each of them. We recognize this validity risk and will always adopt an open mind to try different large language models at the right time. On the other hand, as one of the most well-known open-source large language models, the LLaMA series has been widely studied, so the *C4RLLaMA* approach based on CodeLLaMA is laid on a solid foundation, and the results deserve to be generalized.

Errors from human evaluation: Despite using clear criteria and consistency checks to ensure the accuracy of human evaluations, errors in manual evaluation are still possible, which may, to a certain degree, affect the results. We mitigated this risk by employing evaluators with a background in

software engineering, ensuring they have the necessary expertise to enact the criteria when determining the consistency of rectified comments and the corresponding source code. Besides, this risk is also largely mitigated by the high degree of consistency between the results of human evaluation and objective evaluations based on multiple metrics, as shown in TABLE VIII.

VII. Conclusion and Future Work

In this study, we present a novel approach, *C4RLLaMA* for the detection and rectification of *CCI*. Leveraging the power of large language models, we fine-tune our model based on CodeLLaMA to achieve state-of-the-art performance on both tasks. Specifically, *C4RLLaMA* successfully detects approximately 90% of *CCI* issues, with over half of these issues being automatically and correctly rectified, irrespective of whether the model operates in *post hoc* or *just-in-time* mode.

This research represents an early attempt to utilize a large language model to address *CCI* issues. Our findings underscore the immense potential of large language models in resolving *CCI* issues, while also highlighting areas for further enhancement. For instance, one promising avenue for future research involves fine-tuning a large language model to directly generate code comments. Additionally, exploring the performance boundaries of *C4RLLaMA* by employing a more powerful large language model as a base could prove beneficial. Such an approach may significantly automate the resolution of *CCI* issues, thereby making it a practical solution for large-scale applications.

Artifacts

In order to facilitate the verification or replication of our work, we provide the dataset, algorithmic code, as well as associated instructions used in the study, as detailed in the following URL. https://github.com/aiopsplus/C4RLLaMA

References

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

[2] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[3] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st international conference on program comprehension (icpc)*. Ieee, 2013, pp. 83–92.

[4] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 590–601.

[5] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*/," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.

[6] F. Salviulo and G. Scanniello, "Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[7] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210.

[8] B. P. Lientz, "Issues in software maintenance," *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, pp. 271–278, 1983.

[9] C. S. Hartzman and C. F. Austin, "Maintenance productivity: Observations based on an experience in a large system environment," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, 1993, pp. 138–170.

[10] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting apis," in *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 25–35.

[11] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 427–435.

[12] A. Dau, J. L. Guo, and N. Bui, "Docchecker: Bootstrapping code large language model for detecting and resolving code-comment inconsistencies," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, 2024, pp. 187–194.

[13] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[14] T. Steiner and R. Zhang, "Code comment inconsistency detection with bert and longformer," *arXiv preprint arXiv:2207.14444*, 2022.

[15] S. Hao, Y. Nan, Z. Zheng, and X. Liu, "Smartcoco: Checking comment-code inconsistency in smart contracts via constraint propagation and binding," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 294–306.

[16] Z. Gao, X. Xia, D. Lo, J. Grundy, and T. Zimmermann, "Automating the removal of obsolete todo comments," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 218–229.

[17] Y. Gong, G. Liu, Y. Xue, R. Li, and L. Meng, "A survey on dataset quality in machine learning," *Information and Software Technology*, p. 107268, 2023.

[18] Y. Dong, H. Su, J. Zhu, and B. Zhang, "Improving interpretability of deep neural networks with semantic information," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4306–4314.

[19] F. Rabbi and M. S. Siddik, "Detecting code comment inconsistency using siamese recurrent network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 371–375.

[20] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[21] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, pp. 367–394, 2009.

[22] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

[23] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 112–122.

[24] A. Cimasa, A. Corazza, C. Coviello, and G. Scanniello, "Word embeddings for comment coherence," in *2019 45th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2019, pp. 244–251.

[25] M. Iammarino, L. Aversano, M. L. Bernardi, and M. Cimitile, "A topic modeling approach to evaluate the comments consistency to source code," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[26] F. Rabbi, M. N. Haque, M. E. Kadir, M. S. Siddik, and A. Kabir, "An ensemble approach to detect code comment inconsistencies using topic modeling." in *SEKE*, 2020, pp. 392–395.

[27] W. Ouyang and B. Hua, "R: Towards detecting and understanding code-document violations in rust," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2021, pp. 189–197.

[28] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 154–163.

[29] A. Sadu, "Automatic detection of outdated comments in open source java projects," Ph.D. dissertation, ETSI_Informatica, 2019.

[30] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards detecting inconsistent comments in java source code automatically," in *2020 IEEE 20th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 2020, pp. 65–69.

[31] Y. Huang, Y. Chen, X. Chen, and X. Zhou, "Are your comments outdated? towards automatically detecting code-comment consistency," *arXiv preprint arXiv:2403.00251*, 2024.

[32] 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.

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[34] "Codereviewer: Pre-training for automating code review activities," *arXiv preprint arXiv:2203.09095*, 2022.

[35] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[36] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[37] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[38] H.-M. Liang and C.-Y. Huang, "Integrating non-fourier and ast-structural relative position representations into transformer-based model for source code summarization," *IEEE Access*, 2024.

[39] 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)*. IEEE, 2023, pp. 647–658.

[40] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "Peft: State-of-the-art parameter-efficient fine-tuning methods," https://github.com/huggingface/peft, 2022.

[41] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[42] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, feb 2021. [Online]. Available: https://doi.org/10.1145/3434280

[43] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[44] P. Lu, S. Mishra, T. Xia, L. Qiu, K.-W. Chang, S.-C. Zhu, O. Tafjord, P. Clark, and A. Kalyan, "Learn to explain: Multimodal reasoning via thought chains for science question answering," *Advances in Neural Information Processing Systems*, vol. 35, pp. 2507–2521, 2022.

[45] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[46] Z. Xu, S. Guo, Y. Wang, R. Chen, H. Li, X. Li, and H. Jiang, "Code comment inconsistency detection based on confidence learning," *IEEE Transactions on Software Engineering*, 2024.

[47] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[48] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu *et al.*, "Instruction tuning for large language models: A survey," *arXiv preprint arXiv:2308.10792*, 2023.

[49] Z. Ma, H. Guo, J. Chen, G. Peng, Z. Cao, Y. Ma, and Y.-J. Gong, "Llamoco: Instruction tuning of large language models for optimization code generation," *arXiv preprint arXiv:2403.01131*, 2024.

[50] P. K. Ojha, A. Gautam, A. Agrahari, and P. Singh, "Sft for improved text-to-sql translation," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 12, no. 17s, pp. 700–705, 2024.

[51] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2021.

[52] X. Chen, C. Liang, D. Huang, E. Real, K. Wang, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, Y. Lu *et al.*, "Symbolic discovery of optimization algorithms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[53] 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, pp. 311–318.

[54] A. Mutton, M. Dras, S. Wan, and R. Dale, "Gleu: Automatic evaluation of sentence-level fluency," in *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, 2007, pp. 344–351.

[55] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[56] W. Xu, C. Napoles, E. Pavlick, Q. Chen, and C. Callison-Burch, "Optimizing statistical machine translation for text simplification," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 401–415, 2016.

[57] 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.

[58] S. Longpre, G. Yauney, E. Reif, K. Lee, A. Roberts, B. Zoph, D. Zhou, J. Wei, K. Robinson, D. Mimno *et al.*, "A pretrainer's guide to training data: Measuring the effects of data age, domain coverage, quality, & toxicity," *arXiv preprint arXiv:2305.13169*, 2023.

[59] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young *et al.*, "Scaling language models: Methods, analysis & insights from training gopher," *arXiv preprint arXiv:2112.11446*, 2021.

[60] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.