Fine-tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review

YONGDA YU, Nanjing University, China
GUOPING RONG*, Nanjing University, China
HAIFENG SHEN, Southern Cross University, Australia
HE ZHANG, Nanjing University, China
DONG SHAO, Nanjing University, China
MIN WANG, Tencent Technology (Beijing) Co.Ltd, China
ZHAO WEI, Tencent Technology (Beijing) Co.Ltd, China
YONG XU, Tencent Technology (Beijing) Co.Ltd, China
JUHONG WANG, Tencent Technology (Beijing) Co.Ltd, China

As code review is a tedious and costly software quality practice, researchers have proposed several machine learning-based methods to automate the process. The primary focus has been on accuracy, that is, how accurately the algorithms are able to detect issues in the code under review. However, human intervention still remains inevitable since results produced by automated code review are not 100% correct. To assist human reviewers in making their final decisions on automatically generated review comments, the comprehensibility of the comments underpinned by accurate localization and relevant explanations for the detected issues with repair suggestions is paramount. However, this has largely been neglected in the existing research. Large language models (LLMs) have the potential to generate code review comments that are more readable and comprehensible by humans thanks to their remarkable processing and reasoning capabilities. However, even mainstream LLMs perform poorly in detecting the presence of code issues because they have not been specifically trained for this binary classification task required in code review. In this paper, we contribute *Carllm* (Comprehensibility of Automated Code Review using Large Language Models), a novel fine-tuned LLM that has the ability to improve not only the accuracy but, more importantly, the comprehensibility of automated code review, as compared to state-of-the-art pre-trained models and general LLMs.

CCS Concepts: • **Software and its engineering → Software creation and management**.

Additional Key Words and Phrases: Automated Code Review, Human-machine Collaboration, LLM, LORA

*Guoping Rong is the corresponding author.

Authors' addresses: Yongda Yu, Nanjing University, Nanjing, China, yuyongda@smail.nju.edu.cn; Guoping Rong, Nanjing University, Nanjing, China, ronggp@nju.edu.cn; Haifeng Shen, Southern Cross University, Gold Coast, Australia, haifeng.shen@scu.edu.au; He Zhang, Nanjing University, Nanjing, China, hezhang@nju.edu.cn; Dong Shao, Nanjing University, Nanjing, China, dongshao@nju.edu.cn; Min Wang, Tencent Technology (Beijing) Co.Ltd, Beijing, China, robinmwang@tencent.com; Zhao Wei, Tencent Technology (Beijing) Co.Ltd, Beijing, China, zachwei@tencent.com; Yong Xu, Tencent Technology (Beijing) Co.Ltd, Beijing, China, rogerxu@tencent.com; Juhong Wang, Tencent Technology (Beijing) Co.Ltd, Beijing, China, julietwang@tencent.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s).
ACM 1557-7392/2024/9-ART
https://doi.org/10.1145/3695993

Just Accepted

1 INTRODUCTION

As an indispensable practice of software quality assurance, code review is widely applied in modern software development represented by the Pull Request (PR) development model [13]. Typically, code review relies heavily on manual scrutiny by reviewers to identify issues and often requires a significant amount of time and effort [2, 25]. Additionally, reviewers' expertise greatly influences the quality and efficiency of code review [25]. The challenges of manual code review have motivated researchers to delve into automated code review (ACR) by harnessing rapidly evolving machine learning technologies, such as identifying code diffs that require review [27, 41], analyzing potential issues [17, 49] and generating review comments [27, 49]. However, ACR is not meant to replace human reviewers completely, as even state-of-the-art methods are not able to accurately detect all issues or generate perfectly meaningful review comments [27]. Therefore, ACR can, at best, assist human reviewers in making their final decisions on automatically generated code review comments to achieve superior review efficiency [43]. To facilitate effective human-machine collaboration, ACR results need to be clear and with reasoning and provide constructive suggestions for improvement [25] in such a way that is easily comprehensible to humans. As such, good comprehensibility requires each ACR to (a) accurately detect and localize the issue, (b) provide a reasonable explanation for the cause of the issue, and (c) suggest a concrete repair solution to fix the issue within a diff, a typical code review target signifying the differences between two versions of the source code.

Existing research has been predominantly focused on the accuracy objective of ACR, that is, how accurately ACR algorithms are able to detect issues in the code under review, while largely neglecting the comprehensibility objective. Some work has been attempted to address individual aspects of comprehensibility in isolation. For example, Shi et al.'s work was mainly focused on (a) by determining whether a diff has issues that require review [41]. Tufano et al.'s study was related to (b) by investigating the generation of a review comment that includes an explanation of the issue's cause [49]. More recently, Li et al. explored (c) by employing large-scale pre-training to propose code modifications [27]. To the best of our knowledge, no work has holistically addressed the comprehensibility of ACR characterized by all three indispensable aspects, as the absence of any aspect would affect humans' understanding of ACR results. In addition, ACR needs to possess logical processing capabilities so that the three aspects can be presented in a logical manner rather than by simply displaying the most suitable review comments matched from the existing ones [17]. Pre-training techniques have shown some promising results [22, 27], but their performance is highly subject to the scale and quality of the training data used, and, more importantly, they lack the inherent logical relationships between the aspects of aforementioned comprehensibility.

Recently, the applications of large language models (LLMs) in various scenarios have demonstrated their potential to tackle complex logical issues thanks to their remarkable processing and reasoning capabilities. However, general LLMs would perform poorly in determining whether a specific code diff has an issue, which is essentially a binary classification task because they have not been specifically trained for code review tasks. Therefore, to equip LLMs with the capability of handling complex logic in a specific application domain, it is often necessary to fine-tune the LLMs with logic-rich data from that particular domain [21, 59]. Fine-tuning entails guiding the LLMs to progressively contemplate a complex task by decomposing it into multiple subtasks and gradually solving them. The process of gradual contemplation is referred to as a chain of thought. However, a primary challenge of fine-tuning LLMs is the construction of domain-specific logic-rich data. In the context of ACR, if raw review data was simply used to fine-tune LLMs, the effectiveness of generated review comments would likely be questionable. This is because the majority of review data typically only contains limited information, e.g., *"Shouldn't the operator be && ?"*. The limited information not only falls short of fulfilling the progressive thinking needed in a chain of thought but also fails to meet the information requirements for comprehensibility characterized by localizing the issue, reasoning its cause, and suggesting a repair solution in the corresponding diff.

Just Accepted

The primary contribution of this work is thus the *Carllm* (Comprehensibility of Automated Code Review using Large Language Models), a customized LLM specifically tailored for ACR tasks by fine-tuning several state-of-the-art open-source LLMs, e.g., the LLaMA series [47] and Magicoder [55] . The second significant contribution is a novel data curation method for constructing logically consistent training data from a large amount of raw code review comments in open-source communities collected from GitHub. This is implemented by developing a chain-of-thought code review template tailored to ChatGPT's requirements for logical reasoning and developers' needs for review comments. Meanwhile, we also harness ChatGPT's context learning capability to automatically annotate data through specific task instructions and a tailored input prompt template. Underpinned by the second contribution, the third contribution is the design of an incremental reasoning approach that allows *Carllm* to interact with the user at any point in a code review process to continuously optimize the review results. Using data collected from 1793 open-source projects, we conduct a comprehensive comparison between *Carllm* and its variants and state-of-the-art pre-trained models and general LLMs. The results show that *Carllm* and its variants have the ability to improve not only the accuracy but, more importantly, the comprehensibility of ACR.

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 elaborates the process of constructing *Carllm*. Section 4 describes the research questions and the answers. Section 5 discusses the study's implications and deployment considerations, followed by threats to validity in Section 6. Finally Section 7 concludes the paper with a summary of major contributions and proposed future work.

2 RELATED WORK

This section describes relevant literature from the perspectives of ACR activities and LLMs for software engineering including code generation, intelligent software construction, and ACR.

2.1 Traditional auto code review activities

Code review, as one of the important steps in software quality assurance, has drawn significant attention from researchers due to its high cognitive load on developers [2, 39]. Consequently, many scholars have endeavored to automate the process to alleviate its impact on developers. The activities of automated code review primarily encompass reviewer recommendation [46, 61], code issue prediction [27, 52], and review comment recommendation [44] and generation [27].

Reviewer recommendation aims to improve the efficiency of code review by recommending suitable reviewers for specific PRs. For example, CN [58] leverages a comment network where nodes represent reviewers and contributors, while edges represent similar interests to recommend reviewers by predicting potential edges. Similarly, cHRev [61] operates under the assumption that reviewers who have previously reviewed code units are most suitable to review the same units. HGRec [36] leverages hypergraph to model the relationship of one PR with multiple reviewers, achieving high accuracy in recommendations and partially relieving the core reviewers' workload congestion issue. CORRECT [34, 35] enables cross-project reviewer recommendation by matching projects with suitable reviewers in terms of their capabilities based on library similarity and technical similarity.

Code issue prediction aims to forecast the likelihood of potential issues in code and alert reviewers. For instance, DACE [41] employs CNN and LSTM to extract diff features from code, thereby achieving the ability to predict approve/reject of the code diff patch. Furthermore, approaches such as CodeBert [9] and CodeT5 [52] commonly integrate code defect prediction into downstream tasks of bilingual models in both natural language and programming languages. CodeBert [9] is a bimodal pre-training model designed for programming languages and also the natural language. It learns general-purpose representations and does well in natural-language-based code search and code documentation generation tasks. CodeT5 [52] leverages a unified framework to support both code understanding and generation tasks, thereby facilitating multi-task learning. This method exhibits superior performance compared to previous techniques in several relevant tasks such as code understanding

Just Accepted

and generation. Notably, *CodeReviewer* [27] constructs three specialized pre-training tasks for automating code review tasks, namely 'Diff Tag Prediction', 'Denoising Objective', and 'Review Comment Generation'. Trained by a relatively large-scale corpus constructed from GitHub, it represents state-of-the-art performance in terms of code issue prediction at the time.

The recommendation of review comments is based on the understanding of previous reviews (e.g., similar diffs) to match or generate suitable review comments. For example, DCR [17] recommends review comments for new code by learning the similarity between review comments and code diff patch submissions. CORE [49] utilizes multi-level embedding to represent the semantics of code and review comments, which is combined with the attention mechanism to achieve a substantial improvement in the recall ability of review comments. However, as many issues in code reviews do not re-appear identically, matching review comments from existing ones to those found in the past may mislead readers (reviewers or code submitters). CodeReviewer [27] partially addresses this problem through pre-training of code reviews. However, its low quality training data limits the model's ability to comprehend the underlying logic information, making it unable to fully understand code issues and thus reducing the quality of generated review comments and increasing the difficulty for reviewers to understand them.

2.2 Large Language Models for Software Engineering

In recent years, a variety of LLMs have emerged and played an important role in software engineering. For example, ChatGPT [3] employs advanced techniques such as instruction tuning [7], RLHF [29, 31] to comprehend human intentions and generate responses by leveraging its acquired knowledge. Thanks to its in-context learning capability, it can excel in a variety of zero-shot tasks, providing answers based on a minimal amount of examples, without the need for extensive training. The remarkable comprehension and inference capabilities of LLMs have captivated scholars to address typical challenges in software engineering [51, 60], such as code generation [26, 37] and intelligent software construction [38, 40].

In general, code generation relies on various related inputs to generate code to accelerate software development. For example, Starcoder [26], which possesses 15.5B parameters and a context length of 8192 words, demonstrates outstanding performance in code completion and generation tasks. CodeLLaMA [37], a fine-tuned LLaMA2 [48] model that implements various advanced skills such as filling in the middle, long contexts, and instruction following, represents state-of-the-art performance in open-sourced LLMs. Magicoder [55] empowers DeepSeek [16] and CodeLLaMA with high-quality data to achieve impressive performance with fewer parameters.

Intelligent software construction utilizes LLMs' logical thinking and task-planning abilities to comprehend user requirements, plan solutions, and resolve problems using APIs or other tools. Auto-GPT [15] leverages GPT-4's task planning and autonomous decision-making capabilities to automatically plan tasks, invoke tools, and make autonomous decisions based on the results, by simply obtaining user instructions. HuggingGPT [40] employs ChatGPT as a controller for various specialized models to effectively address complex tasks through composite invocations of different models. TPTU [38] is an inference framework designed to enhance the capability of solving complex tasks by implementing two distinct types of agents.

Despite that entities handled in ACR, i.e., code and review comments, inherently fall within the realm of language, reported work on utilizing LLMs to augment ACR is scarce, except code–review-gpt[1], which is still at its infancy and LLaMA-Reviewer [28], which simply fine-tunes LLaMA with raw review data (mostly from the CodeReviewer paper [27]). However, without improvement to the data quality, especially the enriched logical context enabling the chain of thought, it fails to produce informative review comments most of the time. To tackle this problem, we propose *Carllm* in this paper with the aim to improve the quality of original review comments by providing detailed locations of code issues, corresponding professional knowledge, and logical explanations

[1]https://github.com/mattzcarey/code-review-gpt

Just Accepted

for their causes so that the trained LLM would be able to generate highly comprehensible code review comments with effective repair suggestions for fixing the issues.

3 METHODOLOGY

In this section, we elaborate on the process of constructing *Carllm*, which, as illustrated by Fig. 1, includes: (1) data collection, filtering and processing, (2) building an optimized annotated dataset using ChatGPT, and (3) fine-tuning the base models, e.g., the LLaMA series and Magicoder, to devise *Carllm* and its variants. We first gather a plethora of review comments from GitHub. Then by applying multiple filtering rules, we remove unsuitable projects and review comments (Step ① and ②). Subsequently, we craft a code review template by considering both the comprehensibility of review comments and chain of thought [54] and harness the capabilities of ChatGPT to annotate the collected review comments and build an optimized dataset comprising high-quality data (Step③). After that, we develop prompting and task instructions to guide ChatGPT in performing code review tasks with which, alongside code diff patches and optimized datasets, we build a supervised fine-tuning (SFT) dataset (Step④). Finally, we select the pre-trained open-source LLMs, i.e., LLaMA [47], LLaMA2 [48], CodeLLaMA [37], and Magicoder [55], as the base models to construct *Carllm* and its variants through low-parameter fine-tuning LORA [20] with the SFT dataset (Step⑤).

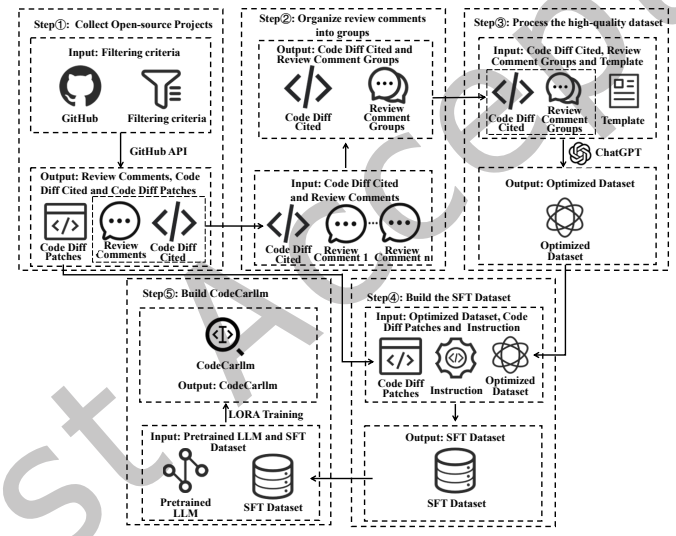

Fig. 1. The process of constructing *Carllm*

3.1 Data collection, filtering and processing

We utilize the APIs provided by GitHub to retrieve raw code review data. While there are no widely accepted criteria for selecting suitable projects, we choose projects that have an active community and cover a wide range of programming languages. So in our inclusion criteria, the numbers of 'Stars' and 'PRs' must both exceed 1000 to guarantee a selected project is a relatively popular one with active participants. Moreover, the programming languages should cover popular ones including Java, Python, C++, JavaScript and Go. In addition, to circumvent any ramifications arising from copyright licenses, we opt for two more lenient licenses, namely MIT and Apache 2.0. As a result, we discovered 1793 open-source projects and then, through GitHub APIs, collected the code

Just Accepted

submissions and code review data of these projects. It is noted that to eliminate irrelevant PRs, e.g., modifications of supporting documents or binary code, we only select the PRs with proper file extensions according to different programming languages. Fig. 4 shows the distribution of project programming languages, while Table 1 lists the statistics on the 'Stars' and 'PRs' about the 1793 projects.

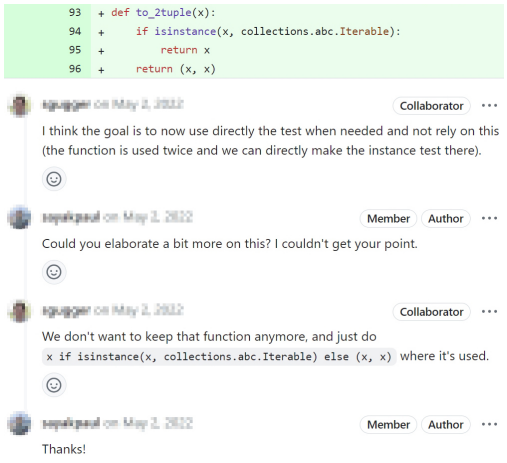

Fig. 2. Whole discussion is required to understand the issue.

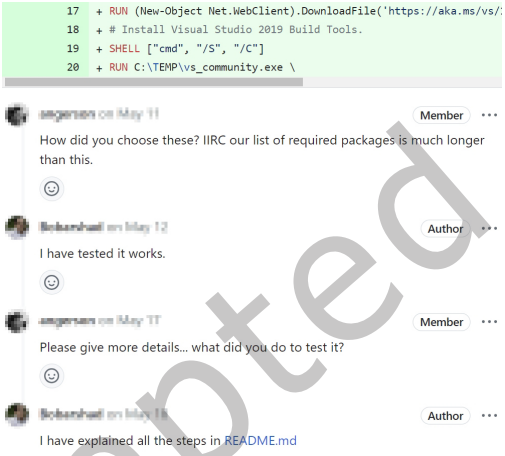

Fig. 3. Discussion ultimately indicates no issue.

After collecting and filtering the raw data, it becomes imperative to enhance its quality. Fundamentally, this enhancement is accomplished by processing the raw data to a holistic and logically coherent entity for each diff, comprising the original diff patch code submission, the code diff referenced by reviewers, and the corresponding review comments organized into groups. Take the discussion shown in Fig. 2 as an example. Apparently, it would be hard to figure out the issue by selecting only one or more isolated review comments for this difference. To understand the ins and outs of this issue, all the review comments must be wholly treated as a holistic entity to provide the relevant context for the corresponding review. Moreover, in real-world projects, a discussion against a diff may also result in no issue in the code, as depicted in Fig. 3. In this case, it would be problematic to determine whether the code has an issue by the existence of a review comment, as several popular ACR methods do [27, 49, 50]. Therefore, *Carllm* considers the entire discussion process as a holistic entity and determines whether or not a diff contains issues based on the LLM's understanding of that holistic entity in the subsequent processing steps.

3.2 Building an optimized annotated dataset using ChatGPT

After obtaining a filtered dataset, this step builds an optimized annotated dataset comprising high-quality review comments using ChatGPT. We first delineate *Carllm*'s code review process based on which a chain-of-thought code review template is crafted. We then explain the process of harnessing ChatGPT to annotate data and build an optimized dataset automatically. Before getting into the details, we first define the symbols and notations in Table 2.

3.2.1 Code review process. The lower part of Fig. 5 illustrates a typical manual code review process. A developer commits a PR containing one or multiple diffs $(D : C_{i-1} \rightarrow C_i, 1 \leq i \leq n)$, which is to be evaluated by human reviewers to identify possible issues. If no issue is detected, the PR may be approved. Otherwise, it may be

Just Accepted

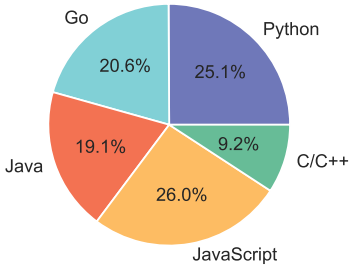

Fig. 4. Distribution of different project programming languages collected

Table 1. Summary of Inclusion criteria

Inclusion criteria	Max	Min	Average
Stars	228283	1038	10300.92
PRs	88236	1000	4162.30

Table 2. Symbols and Notations.

Notations	Descriptions
C_i	source code version i
D	code diff between two source code versions
$I \in \{true, false\}$	detection of a code issue in code diff
Pos	position of code issue in the diff
R	review comment on the issue at the code position
S	solution to fix the issue based on the review comment
Ins	LLM instruction for the code review task

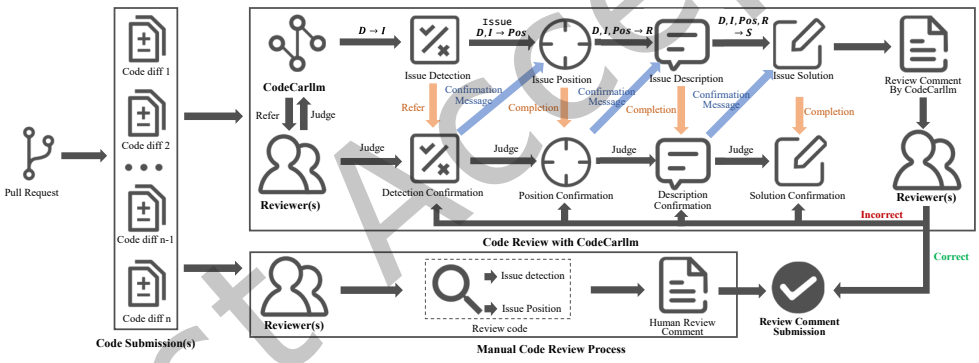

Fig. 5. The code review process with and without *Carllm*.

returned to the developer who needs to modify the source code according to the review comments and re-commit it for the next round of review. The process continues until no issue is found.

In contrast, the upper part of Fig. 5 illustrates a human-machine collaborative code review process underpinned by *Carllm* that is capable of generating highly comprehensible review comments directly from code submissions to human reviewers. Following the requirements of good comprehensibility of review comments discussed in Section 1, i.e., detecting and localizing issues, providing explanations for the causes, and suggesting repair solutions, *Carllm* takes the review process as a content generation process, which is elaborated below.

- **Detecting issue.** For each diff, *Carllm* first detects a code issue, denoted as $D \rightarrow I, I \in \{true, false\}$, where I represents the model's determination on whether or not an issue is present in the given diff code block.

Just Accepted

- **Positioning issue.** *Carllm* generates the precise code position *Pos* of the detected issue, denoted as $(D, I) \rightarrow Pos, Pos \subseteq D$.
- **Explaining issue.** *Carllm* generates review comment R to explain the cause of the detected issue at the position *Pos*, denoted as $(D, I, Pos) \rightarrow R$.
- **Suggesting a repair solution.** *Carllm* suggests a repair solution to the detected issue based on its cause, denoted as $(D, I, Pos, R) \rightarrow S$.

As multiple code issues may exist in the same diff, *Carllm* is designed to generate review comments one by one, that is, $(D, (I, Pos, R, S)_k) \rightarrow I_{k+1}, I_{k+1} \in \{true, false\}$ where $(I, Pos, R, S)_k$ represents the review comment on the k^{th} issue and I_{k+1} represents the model's determination on the existence of the $(k+1)^{th}$ issue in the diff.

3.2.2 Chain-of-thought code review template. According to the study carried out by Kononenko et al. [25], developers believe that good review comments should be clear and comprehensible. Therefore, following *Carllm*'s code review process proposed in Figure 5, we build the chain of thought [54] for code review into three subtasks that follow each other in sequential order.

- **Determining the presence of issue.** For a code review task represented by code diff D, the subtask of assessing its correctness is denoted by $P(I|D, Ins)$, where *Ins* represents the instructive directive that specifies the code review task. Once a code issue is detected, the subtask further determines the exact location of the code issue that corresponds to one or multiple lines of code, denoted by $P(Pos|D, Ins, I)$.
- **Describing the issue.** Upon determining the code issue and its location, the next subtask provides a clear and logical description of the cause of the issue at the corresponding code location, denoted by $P(R|D, Ins, I, Pos)$.
- **Suggesting a repair solution.** After providing the review comment, the final subtask suggests a concrete repair solution to fix the issue, denoted by $P(S|D, Ins, I, Pos, R)$.

In a code snippet under review, it is possible that multiple issues may exist. Therefore, after completing the review of the k^{th} issue, the review process continues to investigate the $(k+1)^{th}$ issue, denoted by $P((I, Pos, R, S)_{k+1}|D, Ins, (I, Pos, R, S)_k)$. In this way, we decompose the code review task step by step and construct a chain of thought. The code review template based on this chain of thought is shown in Fig. 6. The use of few-shot in-context learning is instrumental in achieving the effects of chain-of-thought inference. Furthermore, it is possible to construct training data in line with chain-of-thought for the purpose of fine-tuning the model, thus teaching it to think step by step [24]. Therefore, this template is important for processing the collected raw data as it can help with targeted generation by providing known information. Reviewers can specify the existence of code issues in code blocks, thereby forcing the model to search for potential code issues. Reviewers can also provide targeted code review comment generation by specifying the location of code issues. When writing review comments, the model can complete the remaining review comments and corresponding suggestions based on the already written review comments and their solutions. The chain-of-thought template provides the possibility of obtaining potential answers and assisting in code review completion with partial knowledge.

3.2.3 Automated data annotation using ChatGPT. LLMs have demonstrated impressive performance in summarization tasks [14, 57, 62] and annotation tasks [12, 42]. We use ChatGPT to prepare the training data for *Carllm*. While most information has been provided in the review comments from the previous step, we can transform the data generation task into a summarization task and apply data annotation to complete the missing information. In order to ensure the data quality, we let two researchers perform a sample reading of 1000 data entries and summarize typical error types. Then we use a Python script to filter out inappropriate data from the generated training data according to the four typical error types:

- **The out-of-token error.** Since *Carllm* uses open-source LLMs for Supervised Fine-Tuning (STF), which apply 4096 as the upper limit of token length, we remove the training data that exceeds 4096 tokens.

Just Accepted

```
{
'hasIssue':True/False // assessment of the code's correctness
'ReviewComments':[
    {
        'IssuePosition':""      // the exact location of the code containing one issue
        'IssueDescription':""   // the description of the code issue
        'IssueSolution':""      // the suggested solution to fix the issue
    }, ... // Multi code issues
    ]
}
```

Fig. 6. Chain-of-thought code review template

- **The json-load error.** We use json format to store and parse training data. Therefore, the generated training data that can not be parsed by json program should be removed.
- **The not-inline error.** The generated code lines (indicating the issue position) may not be included in the diff, therefore, we should remove the data to avoid misleading model training in the subsequent steps.
- **The no-issue error.** The generated training data may imply no issue for an actual existing issue. Therefore, we need to remove this part of the data.

Fig. 7 shows an example of context learning that enables an LLM to process the newly collected data, together with the task instructions and an input prompt template. Based on the randomly selected 15,000 pieces of data, each containing one diff snippet and one combined review comment derived in the first step, this step built an optimized annotated dataset comprising 10,554 pieces of high-quality data (77.02%). It excluded data with various errors, i.e., 32 (0.21%) with out-of-token error, 1,525 (10.17%) with json-load error, 1,710 (11.40%) with not-inline error, and 1179 (7.86%) with no-issue error. It is noted that a piece of data may have multiple errors. While these errors could be identified merely by the format, some subtle quality problems can only be identified by humans. Most of these problems are not directly or logically related to code issues (which is exactly what we need to add to the training data through data processing), such as suggestions for adding test cases or resubmitting a new PR. We performed a skimming reading on the generated data to further remove problematic data. In a similar manner, we first let two researchers quickly read 1000 pieces of randomly selected data to extract typical keywords pertinent to potential problematic data, such as 'test', 'the discussion', and 'no changes needed'. The keyword 'test' tends to be a suggestion generated by ChatGPT for adding a test to a submitted diff, 'the discussion' is mostly a description generated by ChatGPT to delineate review comments, and 'no changes needed' is often a conclusion generated by ChatGPT to describe the fact that an original review comment did not specify the concrete problem within the code diff. We then filtered out the data entries containing these keywords through a Python script. As a result, we eventually obtained 9,728 pieces of data, which we deem high-quality for model training. The distribution of programming languages in the final dataset is presented in Table 3, which contains 1.3 review comments for each data entry on average. Besides, the average token length per data entry ranges from 524.12 to 534.5, derived from the Tokenizers of Magicoder, LLaMA, LLaMA2, and CodeLLaMA. Then we prepared code snippets without any issue in a 1:1 ratio as negative examples, resulting in a total of 19,456 training data instances. Subsequently, we extracted 1,000 problematic code submissions and 1,000 non-problematic code submissions from the non-training data as the testing set.

In order to shed some light on the efficacy of the proposed data curation approach, we conducted a random sampling evaluation at the 95% confidence level with less than 5% confidence interval, which involved 400 samples out of 9728 pieces of data. As there is a relatively straightforward and clear ground for judgment, we let two researchers jointly evaluate the 400 samples through manual review. Results indicated that more than 99.5% of the data pieces were restructured into a new format aligning with the logical sequence of chain-of-thought,

Just Accepted

among which 87.25% of the data pieces were augmented with enriched context and 83.5% of the data pieces of review comments are deemed correct review comments.

Table 3. Distribution of different programming languages in the dataset

Programming Languages	Go	Python	Java	C&C++	JavaScript
Proportion	32.36%	17.24%	22.78%	13.67%	13.95%

```
Your task is to translate the information received from the review comments into a standard review comment.
You should return a json string. json contains the keys 'hasIssue' and 'ReviewComments'. 'ReviewComments'
are specific to the code problem and contain a 'IssuePosition' indicating the location of the problem, a
'IssueDescription' explaining the problem in detail, and a 'IssueSolution' of possible solutions.
An example is shown below.
code diff:
define(function(require) {
         * @return {void}
         */
        function updateHandlers(dateExtent) {
-           if (dateExtent == null) {
+           if (dateExtent === null) {

conversations:
nan:Comparing against `null` with the `===` operator only checks that the value is `null`, but not
`undefined`. If you would like to check for both `undefined` and `null`, you can simply use `==`. This will
coerce them to the same value, which means that you just need to check against `null` for both. Nothing else
will ever be coerced to `null` or `undefined`, so this is the one occasion where `==` is safe to use.
If this is the case, I would suggest making this change:
```javascript
          if (dateExtent == null) {
```
Golodhros:Thanks, this is good for now.

standard review comments:
{
   \"hasIssue\":true,
   \"ReviewComments\":[
      {
         \"IssuePosition\":\"- if (dateExtent == null) {\\n+ if (dateExtent === null) {\",
         \"IssueDescription\":\"The code uses the strict equality operator (`===`) to compare `dateExtent`
with `null`. This comparison will return `true` only if `dateExtent` is `null` and of the same type (`null`).
If `dateExtent` is `undefined`, `false`, or any other value, the condition will evaluate to `false`.\",
         \"IssueSolution\":\"Depending on the intended behavior, it might be necessary to use the loose
equality operator (`==`) instead.\"
      }
   ]
}
Process the following data according to the instructions and example.
code diff:
{}
conversations:
{}
standard review comments:
```

Fig. 7. Task instructions, prompt template, and generated training data

Just Accepted

3.3 Fine-tuning open-source LLMs

The prevailing LLMs, for the most part, adhere to the paradigm of pre-training followed by fine-tuning. During pre-training, they acquire fundamental knowledge and characteristics of language from extensive data. During fine-tuning, they are guided by a small amount of task-specific downstream data [56, 63].

Selection of base LLMs. As the fine-tuning capabilities of commercial LLMs through their APIs are limited or even non-existent, we chose open-source LLMs to construct *Carllm*. Specifically, we use the LLaMA series and Magicoder, which have undergone extensive pre-training by utilizing both natural language and code data [63], as the base LLMs. LLaMA is one of the most exemplary LLMs in the open-source community and has served as the foundation for building several renowned open-source dialogue models [5, 11]. Magicoder presents an impressive performance in code-related tasks with fewer parameters. In this work, we employ three versions of

the LLaMA series, namely LLaMA [47], LLaMA2 [48], and CodeLLaMA [37], as well as Magicoder to construct *Carllm*, *Carllm2* and *CodeCarllm* and *MagiCarllm* respectively.

Fine-tuning LLMs. Fine-tuning can effectively enhance the capabilities of LLMs under Zero-Shot conditions [53]. With reference to one of the most renowned templates for fine-tuning LLMs in the open-source community, namely alpaca [45], we have customized it to obtain our instruction fine-tuning template. To save computational and storage costs, we adopt low-parameter fine-tuning (i.e., LORA [20]) to construct *Carllm*. LORA posits that the parameter changes in the fine-tuning phase of the assumed model possess a lower intrinsic rank. This allows for the decomposition of the parameter changes into the product of a low-rank matrix, namely $W' = W_0 + \triangle W = W_0 + BA$. Here, W' represents the parameters of the model after fine-tuning, W_0 denotes the collection of parameters after pre-training, and $\triangle W$ signifies the change in the model's parameters after fine-tuning. $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, with d and k serving as the dimensions of the model's parameters, satisfying the condition $r \ll min(d, k)$. During training, the original pre-training parameter set W_0 remains frozen, exempt from gradient updates, while only B and A partake in parameter updates. Since the parameter size of low-rank matrices is significantly smaller than that of the original model matrices, it is feasible to achieve fine-tuning of LLMs with a minimal number of parameters. We adopt a rank of $r = 8$ as the parameter for fine-tuning LLaMA. This approach enables us to complete the fine-tuning procedure with merely 0.05% of the model's parameter ratio. In order to further decrease the cost of training, we opted for the more efficient optimizer Lion (Evo**L**ved S**i**gn M**o**me**n**tum) [4], thereby achieving model fitting in fewer training iterations while also reducing GPU memory requirements.

A balanced training loss. Large language models typically employ the training loss function as shown in Equation 1, which calculates the sum of the loss functions for all tokens, with each token being allocated only $\frac{1}{n}$ of the training weight.

$$-\mathcal{L}_{LLM}(x) = \sum_{i=1}^{n} logP(x_i|x_{<i}) \tag{1}$$

In our study, the model is trained (fine-tuned) and inferred according to the chain of thought, where the generated content at the beginning will influence the subsequent content generation. In particular, the determination of a code issue will affect the generation of subsequent descriptions. Therefore, we devise a balanced training loss specifically for the code review task, as shown in Equation 2.

$$-\mathcal{L}_{CR}(x) = \alpha logP(x_{issue}|x_{diff}) + (1-\alpha)\sum_{i=1}^{n} logP(x_i|x_{<i}) \tag{2}$$

where α is used to adjust the weights of different components, typically set to $\alpha = 0.5$, $P(x_{issue}|x_{diff})$ denotes the model's assessment of potential code issues x_{issue} given an input code diff patch x_{diff}, and $P(x_i|x_{<i})$ represents the prediction token x_i based on the preceding input sequence $x_{<i}$. The first part of the equation represents the assessment of the code issues by the LLM. The second component relates to the training loss of instruction fine-tuning. The formula progressively computes the loss between the predicted answer and the model's prediction. To validate the effectiveness of this loss function, we conducted ablation experiments. By employing the same initial pre-trained model and keeping all other training parameters constant, we trained the model using both the balanced loss function in Equation 2 and the original fine-tuning loss function in Equation 1 separately. The performance of the models was evaluated using an inference on the testing set. From the results presented in Table 4, it is evident that the *Carllm* models trained with the balanced loss function consistently exhibit significant improvements compared to those trained with the original one.

Fine-tuning setting. The model fine-tuning is conducted using dual A100 40GB GPUs. During training, the model is loaded with int8 precision to reduce graphics memory usage. Other training hyperparameters include:

Just Accepted

Table 4. Difference of average loss between the original and balanced loss functions

Model (loss function)	Recall	Precision	F1-Score	Accuracy
Carllm 13B (original)	0.7490	0.6221	0.6797	0.6470
Carllm 13B (balanced)	**0.8500**	**0.6268**	**0.7215**	**0.6720**
Carllm2 13B (original)	0.6800	0.6814	0.6807	0.6810
Carllm2 13B (balanced)	**0.6950**	**0.6874**	**0.6912**	**0.6895**
CodeCarllm (original)	0.6680	0.7099	0.6883	0.6975
CodeCarllm (balanced)	**0.7700**	0.6968	**0.7316**	**0.7175**

epoch of 5, batch of 4, gradient accumulation of 8, learning rate of 3e-4, LORA dropout of 0.05, and due to the limitations of computational resources, we have set the max token to 2048.

A greedy coding method. Existing LLM generation often uses a sampling method to sample from each generated logits distribution in order to obtain high-quality generated context. The implementation of this method is defined as follows:

$$w_t \sim P(w|w_{1:t-1}) \tag{3}$$

where w_t is the newly generated t^{th} word, $P(w|w_{1:t-1})$ is the logits distribution of the next word given the preceding sequence $w_{1:t-1}$. This approach can effectively alleviate the content repetition issue [19], but it also introduces randomness into the generation process, which may not be acceptable to code review tasks. This is simply because when determining whether an issue exists or where it is located in source code, we expect a deterministic result, not a random result. Therefore, we employ a greedy search algorithm for code issue classification and comment generation by selecting the next word with the highest logits distribution each time to ensure stable outcomes, as defined below:

$$w_t = argmax\{P(w|w_{1:t-1})\} \tag{4}$$

4 EVALUATION

In this section, we evaluate the performance of *Carllm* against state-of-the-art baseline methods in terms of accuracy and comprehensibility of ACR with the following research questions.

- RQ1: How does *Carllm* perform in terms of accuracy of detecting code issues?
- RQ2: How does *Carllm* perform in terms of comprehensibility of generated review comments?

It should be noted that from a practical point of view, it is obvious that a good ACR approach first needs to perform well in RQ1, i.e., it is necessary to perform well in detecting an issue before it makes sense to test how well the explanation of the issue is comprehensible.

4.1 Experimental settings

As there is no single method that has a clear advantage in all aspects meaningful to ACR, we list several methods, as shown in Table 5. For RQ1, we utilize an automated evaluation approach to measure the model's performance using Recall, Precision, F1-score and Accuracy. For RQ2, we employ a manual evaluation method to assess the comprehensibility of generated review comments. Note that to ensure the reliability of the assessment results, we hide the information about the ACR methods used to generate review comments when providing diffs and review comments to the human evaluators.

4.1.1 The experiment for RQ1. We choose a total of ten representative models to compare against, including CodeBert [9], CodeT5 [52], CodeReviewer [27], LLaMA-Reviewer [28], Magicoder-7B [55], LLaMA-13B [47], LLaMA2-13B [48], CodeLLaMA-13B [37], ChatGPT-0613 [3], and GPT-4-0613 [30]. The consideration is two-fold. Firstly, according to existing studies [27, 28], CodeBert, CodeT5, CodeReviewer, and LLaMA-Reviewer can

Just Accepted

Table 5. Experimental settings

RQ	ACR Method	Evaluation Steps	Datasets
RQ1	***MagicCarllm*** ***Carllm*** ***Carllm2*** ***CodeCarllm*** CodeBert CodeT5 CodeReviewer LLaMA-Reviewer Magicoder* LLaMA1* LLaMA2* CodeLLaMA* ChatGPT-0613* GPT-4-0613*	Step 1. Use the same dataset for training. Step 2. Detect issues from diffs in the testing dataset. Step3. Evaluate performance using Recall, Precision, F1-score and Accuracy.	A same **training dataset** contains 19,456 instances (diff-review comment pairs, both positive and negative, see subsection 3.2.3) prepared for *Carllm*, CodeBert, CodeT5, CodeReviewer and LLaMA-Reviewer. The **testing dataset** contains 1000 positive instances and 1000 negative instances which have not been included in the training dataset.
RQ2	***MagicCarllm*** ***Carllm*** ***Carllm2*** ***CodeCarllm*** CodeReviewer CodeT5 LLaMA-Reviewer	Perform manual evaluation to determine the comprehensibility of the review comments generated by various methods. Specifically, we use 'degree of clarity' to measure comprehensibility, meaning that to what degree a review comment clearly explains the issue in a code diff.	The **testing dataset** contains 1000 positive instances .

* No training is needed for these approaches.

Just Accepted

perform issue detection tasks with relatively good performance. Secondly, LLaMA-13B, LLaMA2-13B, CodeLLaMA, ChatGPT (gpt-3.5-turbo-0613) and GPT-4(0613) are representatives of both open-source and commercial models, respectively. Due to resource constraints, we are only able to deploy models with no more than 13B parameters (i.e., LLaMA-13B, LLaMA2-13B, CodeLLaMA-13B and Magicoder-7B). Since CodeBert, CodeT5, CodeReviewer, *MagicCarllm*, *Carllm*, *Carllm*2 and *CodeCarllm* need training before performing issue detection, we utilize the dataset to construct *Carllm* as the training data and extract data from the raw dataset collected from Github that is not present in the training data to construct the testing set. A fact that is particularly worth noting is that for the issue detection task, both the training dataset and the testing dataset contain only the pairs of diff and the *true* or *false* decision result for the presence of an issue. Notably, the training method and hyperparameters of LLaMA-Reviewer are determined according to the settings with the best performance in its original study [28]. For a fair comparison, we also use LLaMA-13B as the base model for LLaMA-Reviewer. In this sense, the training and testing datasets do not have particular features that favor *MagicCarllm*, *Carllm*, *Carllm*2 or *CodeCarllm*.

4.1.2 The experiment for RQ2. We only compare against CodeReviewer, CodeT5 and LLaMA-Reviewer. The reason is two-fold. First, since the original LLMs of Magicoder, LLaMA, LLaMA2, CodeLLaMA, ChatGPT and GPT-4 perform poorly in issue detection (cf. Table 6), it does not make practical sense to assess the comprehensibility of generated review comments as detected issues may be incorrect in the first place. Second, CodeBert shows lower performance in terms of generated review comments (predict the next word) than other models of the same scale [52]. To train the model for review comment generation, different training datasets are used. The upper limit of tokens for both CodeT5 and CodeReviewer, which is highly determined by GPU memory, makes it impossible for us to train these two methods using the complete form of review comments prepared for *Carllm*, as elaborated in **Section 3.2.3**. More importantly, to avoid possible bias for *Carllm* and its variants caused by our own training dataset, we use the datasets provided in the CodeReviewer paper [27] to train the respective models. Out of the 10169 data pieces provided in the CodeReviewer paper [27], we randomly selected 1000 samples (95% confidence level with a confidence interval range less than 3%) as the testing set.

RQ2 requires evaluating the quality of generated review comments. Although BLEU was used in several studies [27, 28] to evaluate the quality of review comments, it is noticed that various studies [1, 48] substantiated that LLMs exhibit greater diversity in their output, making string-matching metrics like BLEU ineffective for appraising the performance of such models. Besides, the evaluation dataset (i.e., the dataset in the CodeReviewer paper [27]) contains errors, so it may pose risks if it is used as the ground truth dataset. Therefore, we utilize manual assessment of the comprehensibility of generated review comments by human evaluators to compare the comments generated by *Carllm* and its variants and those generated by the baseline methods. As elaborated in **Section 1**, comprehensibility requires detecting and localizing each issue, explaining its cause, and suggesting a solution. In this sense, evaluation of comprehensibility requires assessing all the steps of *MagicCarllm*, *Carllm*, *Carllm2* or *CodeCarllm* (as shown in Fig. 5). Although the step of issue detection has been assessed in RQ1, it is also assessed in RQ2 manually since without correct issue detection, other aspects of comprehensibility will not make much sense.

Ternary categorization, which consists of a positive class, a negative class, and a neutral class, is commonly used to assess LLM's capabilities through human evaluation [6, 48, 64]. To address RQ2, we also use ternary categorization to evaluate the comprehensibility of a review comment through a measure referred to as the degree of clarity, including the three classes of ***clear***, ***neutral***, and ***unclear***, respectively. On the one hand, ***clear*** stipulates that the generated review comment clearly and accurately points out the issues in the given code snippet with proper explanation and instructive repair suggestions or correctly affirms the absence of issues in the code snippet. On the other hand, a comment is deemed ***unclear*** if it fails to specify issues in the source code or contains logical errors. This includes instances where the identified issue is non-existent or where an existing issue is overlooked. Any other situations fall under the ***neutral*** category. Figure 8 lists an intuitive example for each category. The manual assessment tasks were conducted by two evaluators independently, i.e., one university student and one industrial practitioner, respectively. To mitigate the influence of data sequence, the resultant data from different models for the same task was randomly incorporated into the evaluation results. For the randomly selected 1000 samples, the first evaluator assessed the first 600 review comments, and the second evaluator assessed the last 600 review comments, while the overlapping 100 in the middle were used for the Chi-Squared test [32]. The test result showed that the p-value between the two evaluators was 0.6197, implying that the null hypothesis can not be rejected and thus confirming the consistency between the assessments from the two evaluators. In fact, as both evaluators assessed whether the description of the generated comment was in line with the issue in a code diff, which was an objective criterion, we did not observe a significant difference in the assessment results between them.

4.2 Evaluation results

We analyze the evaluation results to answer the research questions in this section.

4.2.1 RQ1: Performance comparison in terms of issue detection accuracy. Table 6 shows the performance of various methods in the task of code issue detection. It is clear that our models (*MagicCarllm*, *Carllm*, *Carllm2* and *CodeCarllm*) significantly outperform all the baseline methods. Noticeably, *MagicCarllm*, with only 7B parameters, performs the best in terms of Precision. In particular, with comparable parameters, all our models excel in F1-Score that comprehensively evaluates the model's ability to identify code issues, surpassing the state-of-the-art CodeReviewer [27] and LLaMA-Reviewer [28]. This substantiates the advantage of our method in the code issue detection task. Notably, our models corresponding to all three versions of the LLaMA series demonstrate superior performance in comparison with LLaMA-Reviewer. We believe the adoption of balanced loss in our models significantly enhances the training weight of tokens associated with classification answers, thereby effectively improving performance in detecting code issues. Furthermore, our model exhibits substantial

Just Accepted

Clarity	Code Diff Patch	Generated Review Comments	Observation

Clear	public class ManipulationTest extends BasicJBehaveTest {
	@Override
	public InjectableStepsFactory stepsFactory() {
	- Map<String, Object> state = new HashMap<String, Object>();
	+ Map<String, Object> state = new HashMap<>();
	return new InstanceStepsFactory(configuration(),
	new SharedSteps(state),
		{"IssuePosition": "- Map<String, Object> state = new HashMap<String, Object>();\n+ Map<String, Object> state = new HashMap<>();",
		"IssueDescription": "The code uses the diamond operator (`<>`) to instantiate a `HashMap` without specifying the generic types. This is a more concise way to instantiate a generic class. However, it is important to note that the diamond operator was introduced in Java 7 and is not supported in Java 6 or earlier versions.",
		"IssueSolution": "If the code is being compiled with Java 6 or earlier versions, the diamond operator should not be used. Otherwise, the diamond operator can be used without any issues."}
			The generated review comment accurately identified the compatibility issues with Java versions that would arise from the code changes and offered a checking scheme, thus being classified as Clear.

Neutral	func (s *blockDiskStore) remove(id kbfsblock.ID) error {
	}
	return err
	}
	+
	+ func (s blockDiskStore) clear() error {
	+ return ioutil.RemoveAll(s.dir)
	+ }
		{"IssuePosition": "+func (s blockDiskStore) clear() error {",
		"IssueDescription": "The code defines a method `clear()` on the `blockDiskStore` type. It is unclear what this method does and how it should be used. It is recommended to provide a clear and descriptive name for the method and provide comments to explain its purpose and behavior.",
		"IssueSolution": "Consider renaming the method to a more descriptive name, such as `removeAll()`, and add comments to clarify its purpose and behavior."}
			The review comment did not highlight a specific issue. Instead, it merely proposed a change in function naming and the addition of comments. We didn't identify any apparent errors in the review comment. As such, this comment has been classified as Neutral.

Unclear	func (c *client) processConnect(arg []byte) error {
	c.mu.Lock()
	acc := c.acc
	c.mu.Unlock()
	+ srv.mu.Lock()
	if acc != nil && acc != srv.gacc {
	+ srv.mu.Unlock()
	return ErrTooManyAccountConnections
	}
	+ srv.mu.Unlock()
	}
	c.authViolation()
	return ErrAuthentication
		{"IssuePosition": "+\t\t\t\tsrv.mu.Lock()",
		"IssueDescription": "The code acquires a lock on `srv.mu` inside the `if` block. This means that the lock will be released only after the `if` block is executed. If the `if` block is not executed, the lock will not be released, which can lead to potential deadlocks.",
		"IssueSolution": "To ensure that the lock is released even if the `if` block is not executed, it is recommended to move the lock acquisition outside the `if` block."}
			The review comment suggested that positioning the Unlock() within an if statement could potentially prevent its execution. However, the presence of another Unlock() later in the code safeguards against scenarios where it isn't released. Apparently, this review comment is erroneous, leading to its classification as Unclear.

Fig. 8. Examples of different degrees of clarity

Table 6. Performance of each method under the code issue detection task

Model&Method	Recall	Precision	F1-Score	Accuracy
CodeBert	0.5830	0.7049	0.6382	0.6695
CodeT5	0.6180	0.6828	0.6488	0.6655
CodeReviewer	0.6050	0.7084	0.6526	0.6780
LLaMA-Reviewer	0.7850	0.6066	0.6844	0.6380
Magicoder 7B (2-shot)	0.5202	0.7460	0.6130	0.5290
LLaMA 13B (2-shot)	0.5520	0.4855	0.5166	0.4835
LLaMA2 13B (2-shot)	0.3380	0.5113	0.4070	0.5075
CodeLLaMA 13B (2-shot)	0.5960	0.5326	0.5625	0.5365
ChatGPT-0613 (2-shot)	0.1990	0.5113	0.2851	0.5010
GPT-4-0613 (2-shot)	0.1850	0.4684	0.2652	0.4875
MagicCarllm 7B	0.5790	**0.7500**	0.6535	0.6930
Carllm 13B	**0.8500**	0.6268	0.7215	0.6720
Carllm2 13B	0.6950	0.6874	0.6912	0.6895
CodeCarllm 13B	0.7700	0.6968	**0.7316**	**0.7175**

improvement over the original models of Magicoder-7B, LLaMA-13B, LLaMA2-13B and CodeLLaMA-13B, thereby confirming that our curated data imparts the ability to discern code issues to LLMs.

In contrast, despite adhering to the principle of balanced prompt engineering for classification examples, ChatGPT and GPT-4 tend to believe that the given code is issue-free, which results in an extremely low Recall rate and ultimately affects its performance in the code issue detection task. This may be attributed to their lack of targeted training specifically for the code issue detection task, as code issues encompass a diverse range of

Just Accepted

challenges. ChatGPT and GPT-4 are unable to rely solely on the provided examples and in-context learning to acquire the knowledge required to make judgments for all tasks, resulting in poor performance. This affirms the necessity of fine-tuning to train LLMs specifically for code review tasks.

4.2.2 RQ2: Performance comparison in terms of comprehensibility of review comments. *Carllm* and its variants first annotate the location of each issue in the code and then generate the review comment on the issue at its location to improve the comment's comprehensibility to potential readers (i.e., reviewers or code contributors). This aligns with the mainstream collaborative code review process, as annotating the issue's location can swiftly assist readers in focusing on the issue itself, thereby reducing exertion. In contrast, CodeReviewer [27], CodeT5 [52], and LLaMA-Reviewer [28] solely produce review comments without referring to the exact locations of issues, making it arduous for readers to understand the comments and identify specific code issues, especially when the submitted code snippet is lengthy. Fig. 9 shows the performance of our models (*MagicCarllm*, *Carllm*, *Carllm2*, and *CodeCarllm*) against those of CodeReviewer, CodeT5, and LLaMA-Reviewer in terms of comprehensibility of generated review comments measured by the degree of clarity. It is clear that our *Carllm*-based methods outperform all the baselines with a higher proportion of *clear* comments and a lower proportion of *unclear* comments. By combining the logical capabilities and step-by-step reasoning abilities of the LLMs with our method for constructing logically consistent training data for fine-tuning the LLMs, we are able to improve the comprehensibility of review comments significantly. It is noteworthy that LLaMA-Reviewer employs raw review data for fine-tuning. However, the relatively poor quality of the data obviously impacts large models to generate review comments with useful information, such as *"I think this is a bug"*, which has limited, if not completely useless, value.

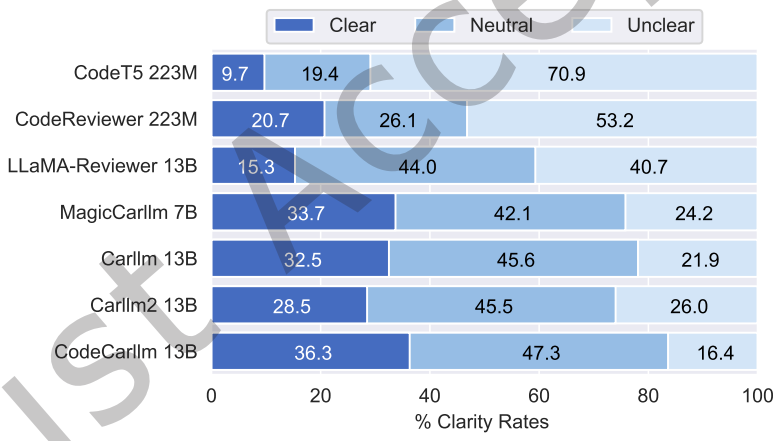

Fig. 9. Performance in terms of comprehensibility of generated review comments

Fig. 10 presents three examples that contrast the review comments generated by *Carllm* with those generated by baseline methods in terms of their comprehensibility. All these examples showcase the superior comprehensibility of the review comments generated by LLM-based *Carllm* to the state-of-the-art pre-training-based methods for the same code issues. The former has a clear logical structure in which the position of an issue is accurately localized, followed by a contextual explanation for the cause of the issue, and concluded by a suggested repair solution to the issue. In contrast, the latter only provides a brief comment that describes a problem without providing contextual information or the location of the code issue or suggests a solution without identifying or explaining the issue. A deeper analysis of the Unclear category reveals three common types of errors. The first

Just Accepted

Case One:

```
type TaskConfig struct {
    Count          Count             `yaml:"count"`
    ExecuteCommand ExecuteCommand    `yaml:"exec"`
    Variables      map[string]string  `yaml:"variables"`
+   EnvFile        string            `yaml:"env_file"`
    Secrets        map[string]string  `yaml:"secrets"`
    Storage        Storage           `yaml:"storage"`
}
```

Review Comment generate by CodeT5:
The `EnvFile` should be removed from here, as well.

Review Comment generate by CodeReviewer:
Can you call it `EnvFile` instead of `EnvFile`? It is more consistent with other entries in the CLI.

Review Comment generate by LLaMA-Reviewer:
I think we should add a `default` value to this field.

Review Comment generate by CodeCarllm:
{**"IssuePosition"**: "+\tEnvFile string `yaml:\"env_file\"`",
"IssueDescription": "The code adds a new field `EnvFile` to the `TaskConfig` struct. It is recommended to use a more descriptive name, such as `EnvironmentFile` or `EnvironmentVariables`.",
"IssueSolution": "Consider renaming the field to `EnvironmentFile` or `EnvironmentVariables` to improve code readability."}

Observation: The code diff shows that the code contributor has added a new data item to the Taskconfig structure; the original data item names were all fully spelled, but the newly added data item was abbreviated. This anomaly was detected by all three models, and CodeT5 argued for the removal of this data item, which was clearly unacceptable. CodeReviewer suggests that the data item name should be changed, but the name is the same before and after the change, which is also clearly unsound. LLaMA-Reviewer suggests to add a default value to the field, but other field don't have default value, it is not necessary. Our approach points out that the name should be changed, but also gives the suggested spelling of the name in full, which is more reasonable.

Case Two:

```
public class Dictionary {
    assert morphSep > 0;
    assert morphSep > flagSep;
    int sep = flagSep < 0 ? morphSep : flagSep;
+   if (sep == 0) return 0;
    CharSequence toWrite;
    String beforeSep = line.substring(0, sep);
```

Review Comment generate by CodeT5:
I think this should be `String.EMPTY_CHAR` (or maybe it's impossible to find a bug in C++ code). The `toWrite` is not used by the tests.

Review Comment generate by CodeReviewer:
When would this happen? I think it should be `String.EMPTY_STRING`.

Review Comment generate by LLaMA-Reviewer:
This is a bit of a hack, but I think it's the best we can do.

Review Comment generate by CodeCarllm:
{**"IssuePosition"**: "+ if (sep == 0) return 0;",
"IssueDescription": "The code checks if `sep` is equal to `0` and returns `0` if it is. However, it is not clear what the intended behavior is if `sep` is `0`. It might be necessary to handle this case differently.",
"IssueSolution": "Consider adding a comment or documentation explaining the expected behavior when `sep` is `0`."}

Observation: The code diff shows that the contributor has added a new judgment and return 0 when the condition is met. Here it actually tells if an abnormally long `sep' occurs. CodeT5's and LLaMA-Reviewer's answer has little to no valid information. CodeReviewer's answer is relevant to the problem, but suggests a change that is very difficult to understand. Our approach, on the other hand, clearly identifies the problem and suggests a reasonable modification in the same vein.

Case Three:

```
func (dc *DownstreamController) syncSecret() {
        }
        nodes := dc.lc.SecretNodes(secret.Namespace, secret.Name)
-       klog.Infof("there are %d nodes need to sync secret, operation: %s", len(nodes), e.Type)
+       klog.V(4).Infof("there are %d nodes need to sync secret, operation: %s", len(nodes), e.Type)
        for _, n := range nodes {
            msg := model.NewMessage("")
            msg.SetResourceVersion(secret.ResourceVersion)
```

Review Comment generate by CodeT5:
use another log message, like `secret.Status`

Review Comment generate by CodeReviewer:
why the switch to v(4) here?

Review Comment generate by LLaMA-Reviewer:
I think we can remove this line.

Review Comment generate by CodeCarllm:
{**"IssuePosition"**: "-\t\t\tklog.Infof(\"there are %d nodes need to sync secret, operation: %s\", len(nodes), e.Type)\n+\t\t\tklog.V(4).Infof(\"there are %d nodes need to sync secret, operation: %s\", len(nodes), e.Type)",
"IssueDescription": "The code is using the `klog.Infof` function to log the message. However, the log level is set to `4`, which is a low log level. It is recommended to use a higher log level, such as `3`, to ensure that the message is visible in the logs.",
"IssueSolution": "Consider changing the log level to `3` or higher."}

Observation: The diff shows that the code contributor has added a log level (4) to the logging statement. Log level 4 is generally Debug level logging, and the system will generally be running in a logging state of level 2 or level 3, which may result in logs not being recorded in normal system runtime conditions. The answer from CodeT5 and LLaMA-Reviewer has little valid information; CodeReviewer only points out the location of the issue without further explanation. Our approach points out the location in a specific way, gives logging-related expertise, and finally suggests a valid solution.

Fig. 10. The example comments generated by CodeT5, CodeReviewer and *Carllm*

Just Accepted

type occurred when LLMs occasionally tended to interpret diffs by describing the code changes within a diff, such as what content was removed or added. The second type occurred when LLMs indicated certain variables were used before they were defined, which was clearly related to the fact that our review subjects only contained code snippets of diffs. The third type was due to the incorrect judgement of real issues contained in diffs.

Automated code modification based on review comments is a step further [27]. However, this is an extremely challenging task as most of the review comments are either simplistic or lack sufficient contextual information for code modification. Therefore, it is acceptable to only provide code modification suggestions so that code contributors are guided to improve their code while given the flexibility of exercising their own judgments. We provide two examples in Fig. 11, which showcase the code modification suggestions made by *Carllm* while demonstrating the extreme difficulty of automated code modifications made by the baseline methods. It is worth pointing out that CodeReviewer, CodeT5 and LLaMA-Reviewer all rely on the ground truth of review comments to perform code modifications, while the modification suggestions made by *Carllm* are solely based on the submitted code diff patches. Given the relatively low accuracy of detecting code issues by CodeReviewer, CodeT5 and LLaMA-Reviewer, the practicality of automated code modification using these methods is generally low. This situation is more evident in the case of LLaMA-Reviewer, which is effective in modifying code when the generated review comments can clearly describe the issue in the source code. However, while this prerequisite is frequently not met in LLaMA-Reviewer, it is often unable to propose valid code changes in our evaluation.

5 DISCUSSION

In this section, we discuss some implications of our study and a deployment consideration around hyper-parameters.

5.1 Automated code review by LLMs

We introduced fine-tuned LLMs into ACR by demonstrating the impressive performance LLMs can bring. This direction clearly warrants further investigation as it is evident that with the growing focus on continuous advancement in the research of LLMs, there is room for further enhancement of foundational model performance. Such improvements will, in turn, provide opportunities to boost the effectiveness and efficiency of our methods. Additionally, there is a potential to elevate data quality within this study, which would also contribute to performance gains for our approach. The combination of these two factors strongly implies our approach's potential for enhancing ACR, a particularly significant endeavor given its indispensable role in software quality assurance. Without automation, the inefficiencies inherent in manual code review could significantly diminish the value brought in by LLM-empowered software development, including automated code generation.

5.2 Chain-of-thought for review comment generation

Our research has demonstrated that a well-structured chain of thought and a high-quality dataset significantly enhance the performance of ACR underpinned by LLMs. This has greatly contributed to deepening our understanding of LLM characteristics, thereby laying a robust foundation for the improved applications of LLMs in addressing software engineering challenges. Specifically, the *Carllm* and its variants support enhanced outcomes during the inference process through feedback or the addition of supplemental information. This effectively unveils certain traits of LLMs, particularly the ability to continuously influence subsequent content generation by modifying the information within the LLM's context. In this regard, the exploration of a methodical, stepwise implementation of software engineering practices aligned with procedural thinking in software engineering may be one of the keys to effectively leveraging LLMs in the field of software engineering.

Just Accepted

Case One:

```
fpga_result mmio_error(struct RASCommandLine *rasCmdLine)
        return result;
    }
-   if(value != FPGA_INTEGRATED_DEVICEID) {
+   if( (value != FPGA_INTEGRATED_DEVICEID) ||
+       (value != FPGA_DISCRETE_DEVICEID) ) {
        FPGA_ERR("Failed    to read Device id");
        return FPGA_NOT_SUPPORTED;
    };
```

Review Comment Ground Truth:
Shouldn't the operator be && ?

Code Refinement generate by CodeT5 using Reivew Comment Ground Truth:

```
    return result;
    }
+   if(value!= FPGA_INTEGRATED_DEVICEID &&
+       (value!= FPGA_INTEGRATED_DEVICEID) &&
+       (value!= FPGA_DISCRETE_DEVICEID)) {
        FPGA_ERR("Failed    to read Device id");
        return FPGA_NOT_SUPPORTED;      }
```

Code Refinement generate by CodeReviewer using Reivew Comment Ground Truth:

```
fpga_result mmio_error(struct RASCommandLine *rasCmdLine)
        return result;
    }
+   if( value!= FPGA_INTEGRATED_DEVICEID && value!= FPGA_DISCRETE_DEVICEID ) {
+   if( (value!= FPGA_INTEGRATED_DEVICEID) && value!= FPGA_DISCRETE_DEVICEID ) {
        FPGA_ERR("Failed    to read Device id");
        return FPGA_NOT_SUPPORTED;<add>      }
```

Code Refinement generate by LLaMA-Reviewer using Reivew Comment Ground Truth:

```
fpga_result mmio_error(struct RASCommandLine *rasCmdLine)
return result;
    }
+   if( (value != FPGA_INTEGRATED_DEVICEID) &&
+       (value != FPGA_DISCRETE_DEVICEID) ) {
        FPGA_ERR("Failed    to read Device id");
        return FPGA_NOT_SUPPORTED;
    }
```

Issue Solution generate by CodeCarllm:
"IssueSolution": "Consider using the logical AND operator (`&&`) instead of the logical OR operator (`||`) to check if `value` is not equal to `FPGA_INTEGRATED_DEVICEID` and `value` is not equal to `FPGA_DISCRETE_DEVICEID`."

Observation: The ground truth review comment was suggesting that the given code submission should use logical AND (&&) as the condition of the fragment, i.e., simply change || to && in the diff. However, the code changes given by both CodeT5 and CodeReviewer not only failed to accomplish the simple change, but also added duplicates that led to more errors and more confusions. Both LLaMA-Reviewer and CodeCarllm Perform well at this case.

Case Two:

```
describe "Display status text" do
    let(:proposal) { FactoryGirl.create(:proposal, :with_parallel_approvers) }
    before do
+       proposal.approvers.first.update(first_name: "Uniquely", last_name: "Named")
+       proposal.approvers.second.update(first_name: "Onlyof", last_name: "Itskind")
        login_as(proposal.requester)
    end
```

Review Comment Ground Truth:
Is this necessary?

Code Refinement generate by CodeT5 using Reivew Comment Ground Truth:

```
describe "Display status text" do
    let(:proposal) { FactoryGirl.create(:proposal, :with_parallel_approvers) }
+   let(:approvers) { FactoryGirl.create(:proposal, :with_parallel_approvers) }
    before do
+       proposal.approvers.first.update(first_name: "Uniquely")
+       proposal.approvers.second.update(first_name: "Onlyof")
        login_as(proposal.requester)
    end
```

Code Refinement generate by CodeReviewer using Reivew Comment Ground Truth:

```
describe "Display status text" do
+   let(:first_name) { "Uniquely", last_name: "Named" }
+   let(:second_name) { "Onlyof", last_name: "Itskind" }
    let(:proposal) { FactoryGirl.create(:proposal, :with_parallel_approvers) }
    before do
+       proposal.approvers.first.update(first_name: first_name, last_name: second_name)
        login_as(proposal.requester)
    end
```

Code Refinement generate by LLaMA-Reviewer using Reivew Comment Ground Truth:

```
describe "Display status text" do
    let(:proposal) { FactoryGirl.create(:proposal, :with_parallel_approvers) }
+   let(:first_approver) { proposal.approvers.first }
+   let(:second_approver) { proposal.approvers.second }
    before do
        login_as(proposal.requester)
    end
```

Issue Solution generate by CodeCarllm:
"IssueSolution": "Consider providing more information or context for the changes made to the approvers' names."

Observation: The ground truth review comments does not have an explicit conclusion about the code, in which case it should be referred to a human for confirmation rather than modification. In general, the code here should be left as is or modified by deletion, but CodeT5, CodeReviewer and LLaMA-Reviewer provide very confusing and of course invalid modification.

Just Accepted

Fig. 11. Examples of code modifications suggested by *Carllm*, CodeT5, CodeReviewer and LLaMA-Reviewer

5.3 Model deployment consideration

During model deployment, there are various parameters that can be adjusted to influence the model's output. Apart from the greedy decoding method in Equation 4 we have employed in *Carllm*, other commonly used decoding methods include beam search and sampling inference [19]. Beam search expands the search space compared to greedy search, enabling the selection of the optimal solution within a relatively broader range [33]. The sampling approach transforms the decoding task of the model into an extraction task, where the logits of vocabulary are converted into the probability of extraction, as depicted in Equation 3. Furthermore, temperature is often used in the sampling approach to regulate the randomness of generation in LLM applications in such a way that a higher temperature yields a stronger level of randomness in the output. We conducted 10 inferences on the same dataset using different algorithms and inference approaches and calculated the median and bounds for statistical analysis.

Table 7. Effect of different decoding methods on code issue detection task

Model	Decode Method	Params	Recall	Precision	F1-Score	Accuracy
MagicCarllm 7B	Greedy	N/A	0.5790 (±0)	0.7500 (±0)	0.6535 (±0)	0.6930 (±0)
MagicCarllm 7B	Beam Search	Beams=8	0.5720 (±0)	0.7507 (±0)	0.6493 (±0)	0.6910 (±0)
MagicCarllm 7B	Sample	temperature=0.1	0.5625 (±0.0095)	0.7445 (±0.0051)	0.6410 (±0.0078)	0.6843 (±0.0062)
MagicCarllm 7B	Sample	temperature=0.7	0.5355 (±0.0195)	0.6616 (±0.0217)	0.5929 (±0.01898)	0.6320 (±0.0160)
Carllm 13B	Greedy	N/A	0.8500 (±0)	0.6268 (±0)	0.7215 (±0)	0.6720 (±0)
Carllm 13B	Beam Search	Beams=8	0.8490 (±0)	0.6270 (±0)	0.7213 (±0)	0.6720 (±0)
Carllm 13B	Sample	temperature=0.1	0.8395 (±0.0105)	0.6283 (±0.0038)	0.7189 (±0.0051)	0.6715 (±0.0055)
Carllm 13B	Sample	temperature=0.7	0.7090 (±0.0130)	0.5936 (±0.0109)	0.6460 (±0.0103)	0.6115 (±0.0120)
Carllm2 13B	Greedy	N/A	0.6950 (±0)	0.6874 (±0)	0.6912 (±0)	0.6895 (±0)
Carllm2 13B	Beam Search	Beams=8	0.6980 (±0)	0.6849 (±0)	0.6914 (±0)	0.6885 (±0)
Carllm2 13B	Sample	temperature=0.1	0.6960 (±0.0080)	0.6891 (±0.0058)	0.6937 (±0.0057)	0.6925 (±0.0050)
Carllm2 13B	Sample	temperature=0.7	0.6090 (±0.0240)	0.6303 (±0.0202)	0.6207 (±0.0209)	0.6265 (±0.0200)
CodeCarllm 13B	Greedy	N/A	0.7700 (±0)	0.6968 (±0)	0.7316 (±0)	0.7175 (±0)
CodeCarllm 13B	Beam Search	Beams=8	0.7530 (±0)	0.6998 (±0)	0.7254 (±0)	0.7150 (±0)
CodeCarllm 13B	Sample	temperature=0.1	0.7525 (±0.0115)	0.7017 (±0.0027)	0.7254 (±0.0060)	0.7152 (±0.0042)
CodeCarllm 13B	Sample	temperature=0.7	0.6615 (±0.0195)	0.6428 (±0.0134)	0.6507 (±0.0113)	0.6465 (±0.0125)

From the results listed in Table 7, it is obvious that the utilization of greedy search and beam search methods, due to the absence of sampling, yields consistent outputs. A slight disparity exists between the results obtained through beam search and greedy search. However, the impact on the outcomes is marginal. Upon employing the sampling technique, fluctuations emerge. These fluctuations may hold the potential to advance the model's performance, yet they may also yield inferior results. When the temperature is low, fluctuations in the outcomes are negligible and the model's performance is comparable to that of greedy search. Conversely, when the temperature is high, substantial oscillations in the outcomes are observed, alongside a conspicuous decline in performance. Therefore, when selecting deployment parameters for code issue detection tasks, employing either greedy search or beam search within the model can achieve a better balance between performance and stability.

6 THREATS TO VALIDITY

We discuss some factors that may potentially bring risks to the findings in the study.

Limited data regarding projects and review comments. On the one hand, although this study has involved the PRs and the corresponding reviews from over 1000 open-source projects, this is still a small proportion as far as the community is concerned. On the other hand, it is well known that data determines the effectiveness of machine learning models. However, we are able to control this validity issue to a large extent by applying objective criteria, i.e., community activity and other criteria that are not directly related to our proposed approach, to select projects and review data.

Just Accepted

Possible errors in reproducing baseline methods. If the reproduction of based methods, e.g., CodeT5 and CodeReviewer, incurs errors, it will obviously affect the experimental comparison results. In this paper, when reproducing the baseline work, we all used the code and data provided in the relevant papers and only converted the formats of input and output for comparison analysis.

Limited LLMs. We have only tried a number of representative LLMs with restrictions on parameter settings (less than 13B), and since the research related to LLMs is extremely active and, in fact, the whole community is in the process of exploring the performance of different LLMs, it cannot be ruled out that some of the LLMs may have better performance than the LLMs included in this paper. We will continue to keep an eye on related research and keep an open mind to try more models.

Incorrect data. There may be multiple sources of introducing erroneous data. Firstly, there may be misjudgments or omissions when manually extracting the format features and keyword features of the erroneous data (cf. Section 3.2.3). Secondly, there may be misjudgments or omissions when filtering the erroneous data based on the identified data format features and keyword features using Python scripts. Last but not least, any LLM is prone to inferential errors, and ChatGPT is no exception. *Carllm* involves using ChatGPT to generate instruction-tuning datasets to fine-tune open-source LLMs. These datasets may contain erroneous descriptions and explanations, which may result in incorrectly generated outputs. In order to control the data quality, we examined the data quality by manual sampling after each processing step, which alleviated the problem to some extent.

However, even with such measures in place, it is challenging to completely prevent the presence of noisy data within the training dataset, which could negatively impact the *Carllm* and its variants trained subsequently. Nevertheless, the purpose of our study was to examine the performance of the fine-tuned LLMs in conducting ACR. The final results confirmed that the fine-tuned LLMs did outperform the base LLMs. In this sense, the negative effects brought by incorrect data did not have a significant adverse impact on the conclusions of our study.

Relatively small scale of data in comprehensibility comparison study. Since only manual means could be used to perform the comparison study, only 1000 pieces of review comment data were manually assessed to compare the comprehensibility of the reviews produced by the different ACR methods. This carries some degree of conclusion risk. However, given the confidence level of the random selection strategy as well as the objective and clear criteria used for the assessment, we believe that this risk is manageable.

Human evaluation of review comments. Rather than employing the BLEU metric, we opted for a manual evaluation method to evaluate the quality of review comments through a ternary categorization approach. It is challenging to assert that this is the optimal evaluation method, so it is important to acknowledge that this approach could potentially influence our final conclusions. The nature of LLMs is such that the content they generate often does not align well with the textual similarity of the information in the training and test corpora. This observation has been highlighted in numerous studies [8, 10]. Conversely, several studies have advocated for the use of manual methods to evaluate the quality of content generated by LLMs [6, 48, 64]. In the context of this study, the ternary categorization approach is more aligned with our research objective, i.e., to enhance the comprehensibility of review comments.

Unpredictable performance between different versions of LLMs. We notice that newer versions of LLMs (i.e., LLaMA2 vs. LLaMA and GPT-4 vs. ChatGPT) did not always present advantages in the issue detection task in this study, as shown in Table 6. As we do not have access to detailed training information, we can only speculate on the following reasons. First, the percentage of code type data may affect the performance of the model for issue detection. Second, there are often a limited number of correct code implementations (maybe even only one), while the number of incorrect implementations could be infinite. Therefore, a trained LLM may tend to learn the correct implementation and thus be more inclined to determine that the code is issue-free, resulting in a very low Recall and, consequently, a low F1 score.

Just Accepted

7 CONCLUSION

In this paper, we contribute the ACR model of *Carllm* and its variants by fine-tuning base LLMs (e.g., the LLaMA series and Magicoder) with logic-rich high-quality code review data collected from a public repository. *Carllm* and its variants outperform existing state-of-the-art models in terms of both accuracy of code issue detection and comprehensibility of the corresponding review comments.

To implement *Carllm*, we first designed a data curation method that used ChatGPT to process the review data obtained from the open-source community to improve its quality and increase the logic between multiple data items. On this basis, we use LORA to fine-tune the base LLMs to support ACR. Thanks to the recursive logic inherent in the dataset (chain of thought), a reviewer can interact with *Carllm* at any point in an ACR scenario in such a way that the reviewer can modify and refine the review comments by *Carllm*. Then *Carllm* can, based on the new information from the reviewer, re-generate the review comments for each of the downstream steps. Empirical studies based on real-world data demonstrate the effectiveness of this approach, revealing a real potential of harnessing LLM's strong logical reasoning ability to improve ACR.

We suggest several directions worth exploring in the future. First, although LLMs with larger parameter sizes do not present a significant advantage on the issue detection task, several studies [18, 23] unarguably confirmed that parameter sizes should contribute to the enhancement of the capabilities of base large models. We suspect that this enhancement should be accompanied by an improvement in data quality [65]. In this sense, how to harmonize the two aspects (parameter size and data quality) for better ACR performance is a direction worth exploring. Second, current *Carllm* is mainly based on diff, i.e., code snippets, to perform ACR. However, as a lot of useful information for determining issues or repairing code is often contained in the method or function in which the code snippet is located, enhancing *Carllm*'s ability to deal with this kind of context should help better improve ACR.

REPLICATION PACKAGE

The replication package can be accessed via URL https://github.com/aiopsplus/Carllm.

REFERENCES

[1] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

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

[3] Introducing ChatGPT. 2023. OpenAI. 2022. *URL: https://openai. com/blog/chatgpt [accessed 2023-07-03] JMIR Med Educ* (2023).

[4] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, et al. 2023. Symbolic discovery of optimization algorithms. *arXiv preprint arXiv:2302.06675* (2023).

[5] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/

[6] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. *arXiv preprint arXiv:2403.04132* (2024).

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

[8] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* 203 (2023), 111741.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

Just Accepted

[10] Mingqi Gao, Xinyu Hu, Jie Ruan, Xiao Pu, and Xiaojun Wan. 2024. Llm-based nlg evaluation: Current status and challenges. *arXiv preprint arXiv:2402.01383* (2024).

[11] Xinyang Geng, Arnav Gudibande, Hao Liu, Eric Wallace, Pieter Abbeel, Sergey Levine, and Dawn Song. 2023. Koala: A Dialogue Model for Academic Research. Blog post. https://bair.berkeley.edu/blog/2023/04/03/koala/

[12] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. 2023. Chatgpt outperforms crowd-workers for text-annotation tasks. *arXiv preprint arXiv:2303.15056* (2023).

[13] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th international conference on software engineering*. 345–355.

[14] Tanya Goyal, Junyi Jessy Li, and Greg Durrett. 2022. News summarization and evaluation in the era of gpt-3. *arXiv preprint arXiv:2209.12356* (2022).

[15] Significant Gravitas. [n. d.]. Auto-GPT: An Autonomous GPT-4 experiment, 2023. *URL https://github. com/Significant-Gravitas/Auto-GPT* ([n. d.]).

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[17] Anshul Gupta and Neel Sundaresan. 2018. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*.

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

[19] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*.

[20] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

[21] Quzhe Huang, Mingxu Tao, Zhenwei An, Chen Zhang, Cong Jiang, Zhibin Chen, Zirui Wu, and Yansong Feng. 2023. Lawyer LLaMA Technical Report. *arXiv preprint arXiv:2305.15062* (2023).

[22] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*. PMLR, 5110–5121.

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

[24] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[25] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: How developers see it. In *Proceedings of the 38th international conference on software engineering*. 1028–1038.

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

[27] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. 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*. 1035–1047.

[28] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. 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, 647–658.

[29] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332* (2021).

[30] R OpenAI. 2023. GPT-4 technical report. *arXiv* (2023), 2303–08774.

[31] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[32] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175.

[33] Matt Post and David Vilar. 2018. Fast Lexically Constrained Decoding with Dynamic Beam Allocation for Neural Machine Translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 1314–1324.

Just Accepted

[34] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. 2016. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the International Conference on Software Engineering Companion (ICSE-C)*. 222–231.
[35] Mohammad Masudur Rahman, Chanchal K Roy, Jesse Redl, and Jason A Collins. 2016. CORRECT: code reviewer recommendation at GitHub for Vendasta technologies. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 792–797.
[36] Guoping Rong, Yifan Zhang, Lanxin Yang, Fuli Zhang, Hongyu Kuang, and He Zhang. 2022. Modeling review history for reviewer recommendation: A hypergraph approach. In *Proceedings of the 44th International Conference on Software Engineering*. 1381–1392.
[37] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).
[38] Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Xingyu Zeng, and Rui Zhao. 2023. TPTU: Task Planning and Tool Usage of Large Language Model-based AI Agents. *arXiv preprint arXiv:2308.03427* (2023).
[39] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.
[40] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580* (2023).
[41] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. 2019. Automatic code review by learning the revision of source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4910–4917.
[42] Xiaofei Sun, Xiaoya Li, Jiwei Li, Fei Wu, Shangwei Guo, Tianwei Zhang, and Guoyin Wang. 2023. Text Classification via Large Language Models. *arXiv preprint arXiv:2305.08377* (2023).
[43] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290* 3, 01 (2023), 17–22.
[44] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.
[45] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3, 6 (2023), 7.
[46] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150.
[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[48] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[49] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174.
[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[51] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2023. A Survey on Large Language Model based Autonomous Agents. *arXiv preprint arXiv:2308.11432* (2023).
[52] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.
[53] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*.
[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
[55] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).
[56] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. 2023. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *arXiv preprint arXiv:2304.13712* (2023).
[57] Xianjun Yang, Yan Li, Xinlu Zhang, Haifeng Chen, and Wei Cheng. 2023. Exploring the limits of chatgpt for query or aspect-based text summarization. *arXiv preprint arXiv:2302.08081* (2023).

Just Accepted

[58] Haochao Ying, Liang Chen, Tingting Liang, and Jian Wu. 2016. Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In *Proceedings of the International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*. 29–35.
[59] Li Yunxiang, Li Zihan, Zhang Kai, Dan Ruilong, and Zhang You. 2023. Chatdoctor: A medical chat model fine-tuned on llama model using medical domain knowledge. *arXiv preprint arXiv:2303.14070* (2023).
[60] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7443–7464.
[61] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2015. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2015), 530–543.
[62] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. 2023. Benchmarking large language models for news summarization. *arXiv preprint arXiv:2301.13848* (2023).
[63] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).
[64] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2024).
[65] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2023. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206* (2023).

Just Accepted