`CodeAgent`: Autonomous Communicative Agents for Code Review

Xunzhu Tang[1], Kisub Kim[2], Yewei Song[1], Cedric Lothritz[3], Bei Li[4], Saad Ezzini[5],
Haoye Tian[6,*], Jacques Klein[1], and Tegawendé F. Bissyandé[1]

[1]University of Luxembourg
[2]Singapore Management University
[3]Luxembourg Institute of Science and Technology
[4]Northeastern University
[5]Lancaster University
[6]The University of Melbourne

Abstract

Code review, which aims at ensuring the overall quality and reliability of software, is a cornerstone of software development. Unfortunately, while crucial, Code review is a labor-intensive process that the research community is looking to automate. Existing automated methods rely on single input-output generative models and thus generally struggle to emulate the collaborative nature of code review. This work introduces `CodeAgent`, a novel multi-agent Large Language Model (LLM) system for code review automation. `CodeAgent` incorporates a supervisory agent, QA-Checker, to ensure that all the agents' contributions address the initial review question. We evaluated `CodeAgent` on critical code review tasks: (1) detect inconsistencies between code changes and commit messages, (2) identify vulnerability introductions, (3) validate code style adherence, and (4) suggest code revision. The results demonstrate `CodeAgent`'s effectiveness, contributing to a new state-of-the-art in code review automation. Our data and code are publicly available (`https://github.com/Daniel4SE/codeagent`).

1 Introduction

Code review (Bacchelli and Bird, 2013; Bosu and Carver, 2013; Davila and Nunes, 2021) implements a process wherein software maintainers examine and assess code contributions to ensure quality and adherence to coding standards, and identify potential bugs or improvements. In recent literature, various approaches (Tufano et al., 2021, 2022) have been proposed to enhance the performance of code review automation. Unfortunately, major approaches in the field ignore a fundamental aspect: the code review process is inherently interactive and collaborative (Bacchelli and Bird, 2013). Instead, they primarily focus on rewriting and adapting the submitted code (Watson et al., 2022; Thongtanunam et al., 2022; Staron et al., 2020). In this respect, an effective approach should not only address how to review the submitted code for some specific needs (e.g., vulnerability detection (Chakraborty et al., 2021; Yang et al., 2024a)). Still, other non-negligible aspects of code review should also be considered, like detecting issues in code formatting or inconsistencies in code revision (Oliveira et al., 2023; Tian et al., 2022; Panthaplackel et al., 2021). However, processing multiple sub-tasks requires interactions among employees in different roles in a real code review scenario, which makes it challenging to design a model that performs code review automatically.

Agent-based systems are an emerging paradigm and a computational framework in which autonomous entities (aka agents) interact with each other (Li et al., 2023a; Qian et al., 2023; Hong et al., 2023) to perform a task. Agent-based approaches have been proposed to address a spectrum of software engineering tasks (Qian et al., 2023; Zhang et al., 2024; Tang et al., 2023; Tian et al., 2023), moving beyond the conventional single input-output paradigm due to their exceptional ability to simulate and model complex interactions and behaviors in dynamic environments (Xi et al., 2023; Yang et al., 2024b; Wang et al., 2023). Recently, multi-agent systems have leveraged the strengths of diverse agents to simulate human-like decision-making processes (Du et al., 2023; Liang et al., 2023; Park et al., 2023), leading to enhanced performance across various tasks (Chen et al., 2023; Li et al., 2023b; Hong et al., 2023). This paradigm is well-suited to the challenge of code review, where multiple reviewers, each with diverse skills and roles, collaborate to achieve a comprehensive review of the code..

This paper. Drawing from the success of agent-based collaboration, we propose a **multi-agent-based framework** `CodeAgent` to simulate the

---

[*] Corresponding author.

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 11279–11313
November 12-16, 2024

dynamics of a collaborative team engaged in the code review process, incorporating diverse roles such as code change authors, reviewers, and decision makers. In particular, A key contribution of `CodeAgent` is that we address the challenge of prompt drifting (Zheng et al., 2024; Yang et al., 2024c), a common issue in multi-agent systems and Chain-of-Thought (CoT) reasoning. This issue, characterized by conversations that stray from the main topic, highlights the need for strategies to maintain focus and coherence (Greyling, 2023; Chae et al., 2023). This drift, often triggered by the model-inspired tangents or the randomness of Large Language Models (LLMs), necessitates the integration of a supervisory agent. We employ an agent named QA-Checker (for "Question-Answer Checker") that monitors the conversation flow, ensuring that questions and responses stay relevant and aligned with the dialogue's intended objective. Such an agent not only refines queries but also realigns answers to match the original intent, employing a systematic approach grounded in a mathematical framework.

To evaluate the performance of `CodeAgent`, we first assess its effectiveness for typical review objectives such as detecting vulnerabilities 4.1 and validating the consistency and alignment of the code format 4.2. We then compare `CodeAgent` with state-of-the-art generic and code-specific language models like ChatGPT (OPENAI, 2022) and CodeBERT (Feng et al., 2020). Finally, we assess the performance of `CodeAgent` compared to the state-of-the-art tools for code revision suggestions (Tufano et al., 2021; Thongtanunam et al., 2022; Tufano et al., 2022). Since each of these related works presents a specific dataset, we also employ them toward a fair comparison. Additionally, we also collect pull requests from GitHub, featuring an extensive array of commits, messages, and comments to evaluate advanced capabilities.The experimental results reveal that `CodeAgent` significantly outperforms the state-of-the-art, achieving a 41% increase in hit rate for detecting vulnerabilities. `CodeAgent` also excels in consistency checking and format alignment, outperforming the target models. Finally, `CodeAgent` showcases its robustness for code revision by presenting superior average edit progress.

We summarize our contributions as follows:

- To the best of our knowledge, we are the first to propose an autonomous agent-based system for practical code review in the field of software maintenance.
- We build a new dataset comprising 3 545 real-world code changes and commit messages. This dataset, which includes all relevant files and details in a self-contained format, is valuable for evaluating advanced code review tasks such as vulnerability detection, code style detection, and code revision suggestions.
- We demonstrate the effectiveness of the QA-Checker. This agent monitors the conversation flow to ensure alignment with the original intent, effectively addressing the common prompt drifting issues in multi-agent systems.

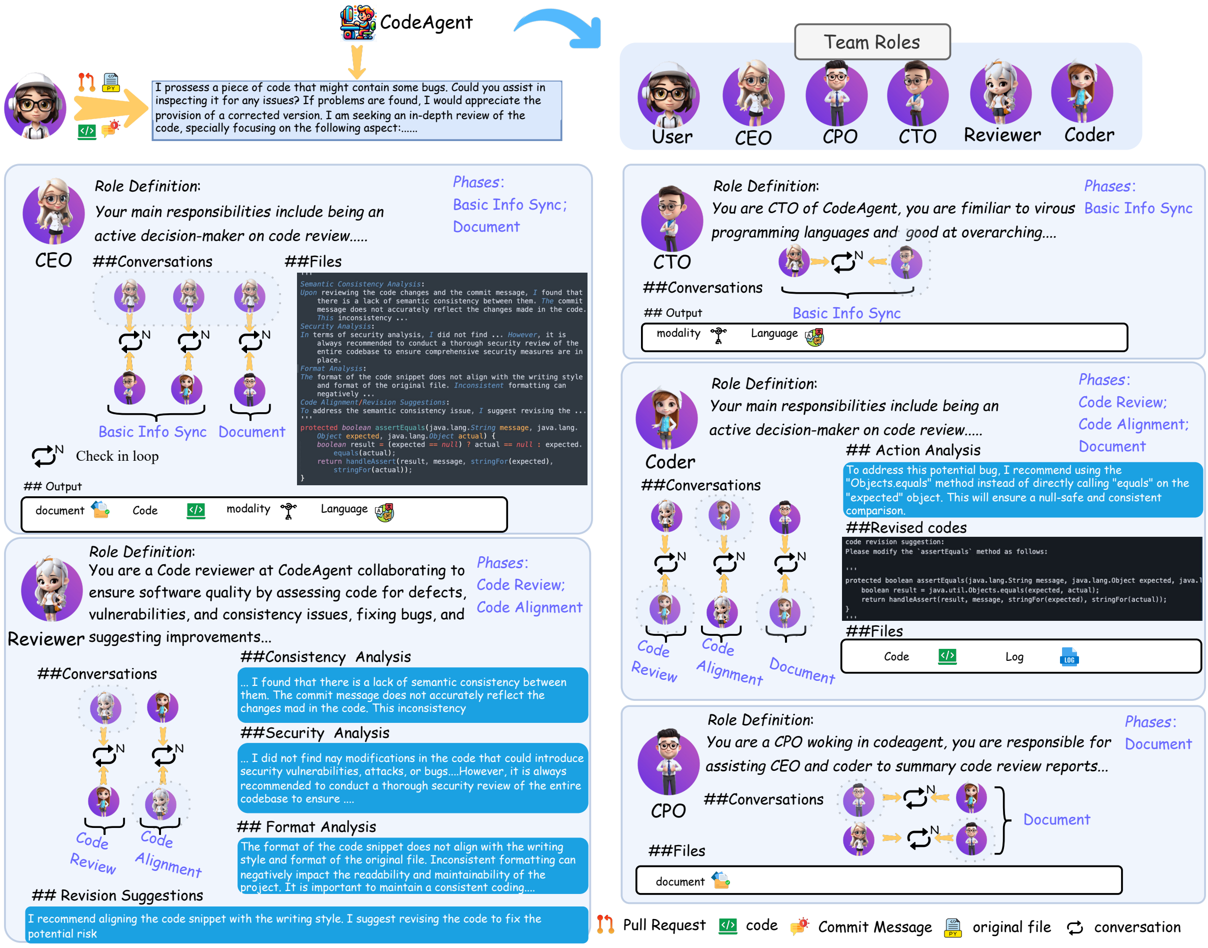
Experimental evaluation highlights the performance of `CodeAgent`: In vulnerability detection, `CodeAgent` outperforms GPT-4 and CodeBERT by 3 to 7 percentage points in terms of the number of vulnerabilities detected. For format alignment, `CodeAgent` outperforms ReAct by approximately 14% in recall for inconsistency detection. On the code revision task, `CodeAgent` surpasses the state of the art in software engineering literature, achieving an average performance improvement of about 30% in the Edit Progress metric (Zhou et al., 2023).

2 `CodeAgent`

This section details the methodology behind our `CodeAgent` framework. We discuss tasks and definition in Sec 2.1, pipeline in Section 2.2, defined role cards in Section 2.3, and the design of the QA-Checker in Sec 2.4.

2.1 Tasks

We define *CA*, *VA*, *FA*, and *CR* in as following:
CA (Zhang et al., 2022): Consistency analysis between code change and commit message; the task is to detect cases where the commit message accurate describes (in natural language) the intent of code changes (in programming language).
VA (Braz et al., 2022): Vulnerability analysis; the task is to identify cases where the code change introduces a vulnerability in the code.
FA (Han et al., 2020): Format consistency analysis between commit and original files; the task is to



I prossess a piece of code that might contain some bugs. Could you assist in inspecting it for any issues? If problems are found, I would appreciate the provision of a corrected version. I am seeking an in-depth review of the code, specially focusing on the following aspect:......

Your main responsibilities include being an active decision-maker on code review.....

```
'''
Semantic Consistency Analysis:
Upon reviewing the code changes and the commit message, I found that
    there is a lack of semantic consistency between them. The commit
    message does not accurately reflect the changes made in the code.
    This inconsistency ...
Security Analysis:
In terms of security analysis, I did not find ... However, it is
    always recommended to conduct a thorough security review of the
    entire codebase to ensure comprehensive security measures are in
    place.
Format Analysis:
The format of the code snippet does not align with the writing style
    and format of the original file. Inconsistent formatting can
    negatively ...
Code Alignment/Revision Suggestions:
To address the semantic consistency issue, I suggest revising the ...
'''
protected boolean assertEquals(java.lang.String message, java.lang.
    Object expected, java.lang.Object actual) {
    boolean result = (expected == null) ? actual == null : expected.
        equals(actual);
    return handleAssert(result, message, stringFor(expected),
        stringFor(actual));
}
```

You are a Code reviewer at CodeAgent collaborating to ensure software quality by assessing code for defects, vulnerabilities, and consistency issues, fixing bugs, and suggesting improvements...

... I found that there is a lack of semantic consistency between them. The commit message does not accurately reflect the changes mad in the code. This inconsistency

... I did not find nay modifications in the code that could introduce security vulnerabilities, attacks, or bugs....However, it is always recommended to conduct a thorough security review of the entire codebase to ensure ....

The format of the code snippet does not align with the writing style and format of the original file. Inconsistent formatting can negatively impact the readability and maintainability of the project. It is important to maintain a consistent coding....

I recommend aligning the code snippet with the writing style. I suggest revising the code to fix the potential risk

You are CTO of CodeAgent, you are fimiliar to virous programming languages and good at overarching....

Your main responsibilities include being an active decision-maker on code review.....

To address this potential bug, I recommend using the "Objects.equals" method instead of directly calling "equals" on the "expected" object. This will ensure a null-safe and consistent comparison.

```
code revision suggestion:
Please modify the `assertEquals` method as follows:

'''
protected boolean assertEquals(java.lang.String message, java.lang.Object expected, java.l
    boolean result = java.util.Objects.equals(expected, actual);
    return handleAssert(result, message, stringFor(expected), stringFor(actual));
}
'''
```

You are a CPO woking in codeagent, you are responsible for assisting CEO and coder to summary code review reports...

Figure 1: **A Schematic diagram of role data cards of simulated code review team and their conversations within `CodeAgent`.** We have six characters in `CodeAgent` across four phases, including "Basic Info Sync", "Code Review", "Code Alignment", and "Document". Code review is a kind of collaboration work, where we design conversations between every two roles for every step to complete the task.

validate that the code change formatting style is not aligned with the target code.

CR (Zhou et al., 2023): Code revisions; this task attempts to automatically suggest rewrites of the code change to address any issue discovered.

2.2 Pipeline

We defined six characters and four phases for the framework. The roles of the characters are illustrated in Figure 1. Each phase contains multiple conversations, and each conversation happens between agents. The four phases consist of 1) Basic Info Sync, containing the roles of chief executive officer (*CEO*), chief technology officer (*CTO*), and Coder to conduct modality and language analysis; 2) Code Review, asking the *Coder* and *Reviewer* for actual code review (i.e., target sub-tasks); 3) Code Alignment, supporting the *Coder* and *Reviewer* to correct the commit through code revision and suggestions to the author; and 4) Document, finalizing by synthesizing the opinions of the *CEO*, *CPO* (Chief Product Officer), *Coder*, and *Reviewer* to provide the final comments. In addition to six defined roles, the proposed architecture of `CodeAgent` consists of phase-level and conversation-level components. The waterfall model breaks the code review process at the phase level into four sequential phases. At the conversation level, each phase is divided into atomic conversations. These atomic conversations involve task-oriented role-playing between two agents, promoting collaborative communication. One agent works as an instructor and the other as an assistant. Communication follows an instruction-following style, where agents interact to accomplish a specific subtask within each conversation, and each conversation is supervised by QA-Checker. QA-Checker is used to align the

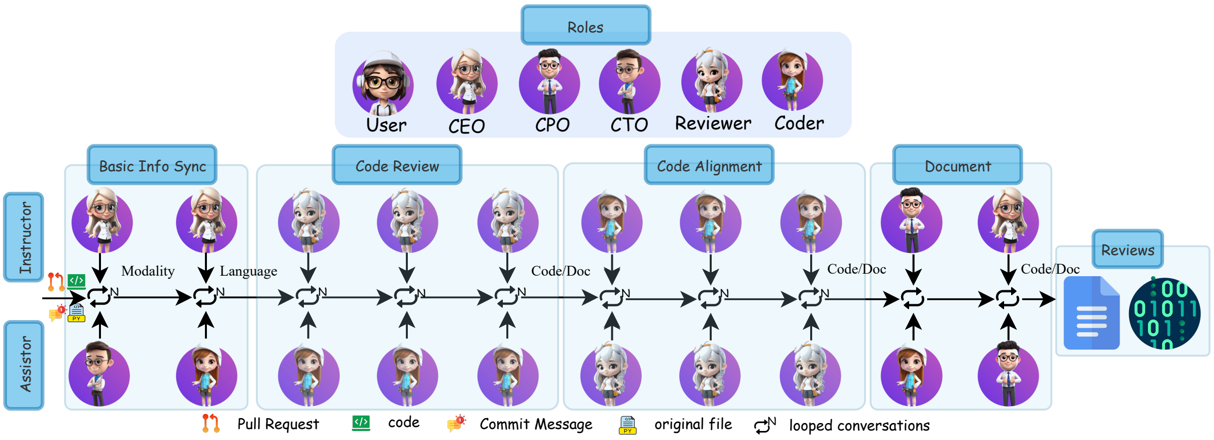


Figure 2: `CodeAgent`'s pipeline/scenario of a full conversation during the code review process among different roles. "Basic Info Sync" demonstrates the basic information confirmation by the CEO, CTO, and Coder; "Code Review" shows the actual code review process; "Code Alignment" illustrates the potential code revision; and "Document" represents the summarizing and writing conclusion for all the stakeholders. All the conversations are being ensured by the Quality Assurance checker until they reach the maximum dialogue turns or meet all the requirements.

consistency of questions and answers between the instructor and the assistant in a conversation to avoid digression. QA-Checker will be introduced in Section 2.4.

Figure 2 shows an illustrative example of the `CodeAgent` pipeline. `CodeAgent` receives the request to do the code review with the submitted commit, commit message, and original files. In the first phase, *CEO*, *CTO*, and *Coder* will cooperate to recognize the modality of input (e.g., document, code) and language (e.g., Python, Java and Go). In the second phase, with the help of *Coder*, *Reviewer* will write an analysis report on consistency analysis, vulnerability analysis, format analysis and suggestions for code revision. In the third phase, based on analysis reports, *Coder* will align or revise the code if any incorrect snippets are identified with assistance from *Reviewer*. *Coder* cooperates with *CPO* and *CEO* to summarize the document and codes about the whole code review in the final phase.

2.3 Role Card Definition

As shown in Figure 1, we define six characters in our simulation system (`CodeAgent`), including *User*, *CEO*, *CPO*, *CTO*, *Reviewer*, *Coder*, and they are defined for different specific tasks.

All tasks are processed by the collaborative work of two agents in their multi-round conversations. For example, as a role *Reviewer*, her responsibility is to do the code review for given codes and files in three aspects (tasks *CA*, *VA*, and *FA* in Sec 2.1) and provide a detailed description of observation. *Reviewer*'s code review activity is under the assistance with *Coder* as shown in Figure 2. Meanwhile, with the Reviewer's assistance, *Coder* can process the code revision as shown in the 'Revised codes' part in the *Coder* card in Figure 1. Apart from *Reviewer*, *Coder* also cooperates with *CTO* and *CEO* in the simulated team.

Each role and conversation, input and output of each conversation is designed in Figure 1. Further information about role definition details is provided in our Appendix-Section C.1.

2.4 Self-Improving CoT with QA Checker

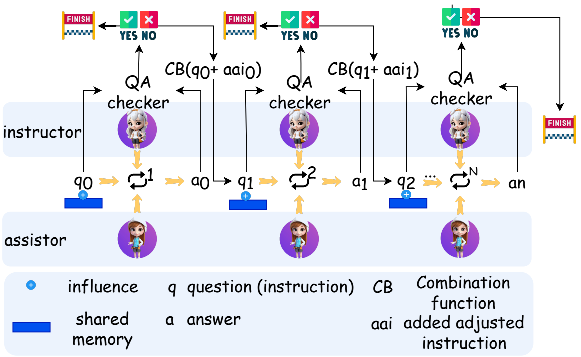


Figure 3: This diagram shows the architecture of our designed Chain-of-Thought (CoT): Question-Answer Checker (QA-Checker).

QA-Checker is an instruct-driven agent, designed to fine-tune the question inside a conversation to drive the generated answer related to the question. As shown in Figure 3, the initial question (task instruction) is represented as q_0, and the first answer of the conversation between $Reviewer$ and $Coder$ is represented as a_0. If QA-Checker identifies that a_0 is inappropriate for q_0, it generates additional instructions attached to the original question (task instruction) and combines them to ask agents to further generate a different answer. The combination in Figure 3 is defined as $q_1 = CB(q_0 + aai_0)$, where aai_0 is the additional instruction attached. The conversation between two agents is held until the generated answer is judged as appropriate by QA-Checker or it reaches the maximum number of dialogue turns.

Theoretical Analysis of QA-Checker in Dialogue Refinement The QA-Checker is an instruction-driven agent, crucial in refining questions and answers within a conversation to ensure relevance and precision. Its operation can be understood through the following lemma and proof in Appendix A.

3 Experimental Setup

We evaluate the performance of `CodeAgent` through various qualitative and quantitative experiments across nine programming languages, using four distinct metrics. In this section, we will discuss experimental settings, including datasets, metrics, and baselines. For more information, please see Appendix C.

3.1 Datasets

To conduct a fair and reliable comparison for the code revision task, we employ the same datasets (i.e., Trans-Review$_{data}$, AutoTransform$_{data}$, and T5-Review$_{data}$) as the state-of-the-art study (Zhou et al., 2023). Furthermore, we collect and curate an additional dataset targeting the advanced tasks. Table 1 shows our new dataset which includes over 3,545 commits and 2,933 pull requests from more than 180 projects, spanning nine programming languages: Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. It focuses on consistency and format detection, featuring both positive and negative samples segmented by the merged and closed status of pull requests across various languages. The detailed information about the dataset can be seen in Appendix-Section F.

Table 1: Comparison of Positive and Negative Samples in CA and FA (CA and FA are defined in Section 2.1).

Samples	CA		FA	
	Merged	Closed	Merged	Closed
Positive (consistency)	2,089	820	2,238	861
Negative (inconsistency)	501	135	352	94

3.2 Metrics

- **F1-Score and Recall.** We utilized the F1-Score and recall to evaluate our method's effectiveness on tasks *CA* and *FA*. The F1-Score, a balance between precision and recall, is crucial for distinguishing between false positives and negatives. Recall measures the proportion of actual positives correctly identified (Hossin and Sulaiman, 2015).

- **Edit Progress (EP).** EP evaluates the improvement in code transitioning from erroneous to correct by measuring the reduction in edit distance between the original code and the prediction on task *CR*. A higher EP indicates better efficiency in code generation (Dibia et al., 2022; Elgohary et al., 2021; Zhou et al., 2023).

- **Hit Rate (Rate)** We also use hit rate to evaluate the rate of confirmed vulnerability issues out of the found issues by approaches on task *VA*.

3.3 State-of-the-Art Tools and Models

Our study evaluates various tools and models for code revision and modeling. **Trans-Review** (Tufano et al., 2021) employs src2abs for code abstraction, effectively reducing vocabulary size. **AutoTransform** (Thongtanunam et al., 2022) uses Byte-Pair Encoding for efficient vocabulary management in pre-review code revision. **T5-Review** (Tufano et al., 2022) leverages the T5 architecture, emphasizing improvement in code review through pre-training on code and text data. In handling both natural and programming languages, **CodeBERT** (Feng et al., 2020) adopts a bimodal approach, while **GraphCodeBERT** (Guo et al., 2021) incorporates code structure into its modeling. **CodeT5** (Wang et al., 2021), based on the T5 framework, is optimized for identifier type awareness, aiding in generation-based

Table 2: The number of vulnerabilities found by `CodeAgent` and other approaches. As described in Appendix-Section F, we have 3,545 items to evaluate. Rate$_{cr}$ represents the confirmed number divided by the number of findings while Rate$_{ca}$ is the confirmed number divided by the total evaluated number. `CodeAgent` $_{w/o}$ indicates the version without QA-Checker.

	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	`CodeAgent`	`CodeAgent` $_{w/o}$
Find	1,063	864	671	752	693	483	564
Confirm	212	317	345	371	359	449	413
Rate$_{cr}$	19.94%	36.69%	51.42%	49.34%	51.80%	92.96%	73.23%
Rate$_{ca}$	5.98%	8.94%	9.73%	10.46%	10.13%	12.67%	11.65%

The values in gray (nn.nn) denote the greatest values for the confirmed number of vulnerabilities and the rates.

tasks. Additionally, we compare these tools with **GPT** (OPENAI, 2022) by OpenAI, notable for its human-like text generation capabilities in natural language processing. Finally, we involve **COT** (Wei et al., 2022) and **ReAct** (Yao et al., 2022), of which **COT** is a method where language models are guided to solve complex problems by generating and following a series of intermediate reasoning steps and **ReAct** synergistically enhances language models by interleaving reasoning and action generation, improving task performance and interpretability across various decision-making and language tasks.

4 Experimental Result Analysis

This section discusses the performance of `CodeAgent` in the four tasks considered for our experiments. In Appendix Section E, we provide further analyses: we discuss the difference in the execution time of `CodeAgent` in different languages and perform a capability analysis between `CodeAgent` and recent approaches.

4.1 Vulnerability Analysis

Compared to *CA* and *FA*, *VA* is a more complex code review subtask, covering more than 25 different aspects (please see the Appendix-Section G), including buffer overflows, sensitive data exposure, configuration errors, data leakage, etc. Vulnerability analysis being a costly, time-consuming, resource-intensive and sensitive activity, only a low proportion of commits are labeled. We therefore propose a proactive method for data annotion: we execute `CodeAgent` on the 3,545 samples (covering nine languages) and manual verify the identified cases to build a ground truth. Then, we applied CodeBERT (Feng et al., 2020) and GPT on the dataset with the task of vulnerability binary prediction.

Comparison As shown in Table 2, `CodeAgent` successfully identified 483 potential vulnerabilities within a data set of 3,545 samples, with an impressive 449 of these finally confirmed as high-risk vulnerabilities[1]. CodeBERT, a key pre-trained model for code-related tasks, with its parameters frozen for this experiment, initially identified 1,063 items as vulnerable, yet only 212 passed the stringent verification criteria. Similar trends were observed with GPT-3.5 and GPT-4.0, which confirmed 317 and 345 vulnerabilities out of 864 and 671 identified items, respectively. These outcomes are further quantified by the confirmation rates (Rate$_{cr}$) of 19.94% for CodeBERT, 36.69% for GPT-3.5, and 51.42% for GPT-4.0, while `CodeAgent` demonstrated a remarkable Rate$_{cr}$ of 92.96%. Additionally, the analysis of confirmed vulnerabilities against all analyzed items (Rate$_{ca}$) yielded 5.98%, 8.94%, 9.73%, and 12.67% for CodeBERT, GPT-3.5, GPT-4.0, and `CodeAgent`, respectively. Evidently, Table 2 not only highlights `CodeAgent`'s high precision in identifying vulnerable commits but also reveals the progressive improvement from GPT-3.5 to GPT-4.0, likely due to the latter's capacity to handle longer input sequences, with token limits of 4,096 and 32,768, respectively. The integration of sophisticated algorithms like CoT and QA-Checker in `CodeAgent` has significantly enhanced its capabilities in vulnerability detection, surpassing the individual input-output efficiencies of GPT and CodeBERT. Appendix-Sections D and M highlight further details regarding the importance of the QA-checker. Moreover, more experimental results in 9 languages are accessible

---

[1] The verification process involved a rigorous manual examination, extending beyond 120 working hours. Each sample being validated by at least 2 people: a researcher and an engineer

in Appendix-Section J.

In addition, the analysis of vulnerabilities identified by various models reveals interesting overlaps among the models. CodeBERT confirmed 212 vulnerabilities, whereas GPT-3.5, GPT-4.0, and `CodeAgent` confirmed 317, 345, and 449 vulnerabilities, respectively. Notably, the intersection of vulnerabilities confirmed by CodeBERT and GPT-3.5 is 169, indicating a substantial overlap in their findings. Similarly, the intersection between CodeBERT and GPT-4.0 is 170, while a larger overlap of 212 vulnerabilities is observed between GPT-3.5 and GPT-4.0. The combined intersection among CodeBERT, GPT-3.5, and GPT-4.0 is 137, underscoring the commonalities in vulnerabilities detected across these models. Further, the intersections of vulnerabilities confirmed by CodeBERT, GPT-3.5, and GPT-4.0 with `CodeAgent` are 212, 317, and 334, respectively, highlighting the comprehensive coverage and detection capabilities of `CodeAgent`.

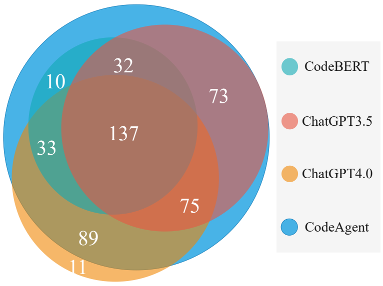


Figure 4: Overlap of vulnerability detection by CodeBERT, GPT-3.5, GPT-4.0, and `CodeAgent`.

Ablation Study. As shown in Table 2, we conducted an ablation study to evaluate the effectiveness of the QA-Checker in `CodeAgent`. Specifically, we created a version of our tool without the QA-Checker, referred to as `CodeAgent` $_{w/o}$. We then compared this version to the full version of `CodeAgent` that includes the QA-Checker. The results demonstrate that `CodeAgent` $_{w/o}$ is substantially less effective in identifying vulnerable issues, yielding lower hit rates (Rate$_{cr}$ and Rate$_{ca}$). This reduction in performance highlights the critical role of the QA-Checker in enhancing `CodeAgent`'s overall effectiveness. More detailed information about the ablation study can be found in Appendix-Section M.

4.2 Consistency and Format Detection

In this section, we will discuss the performance of `CodeAgent` and baselines on metrics like the F1-Score and recall score of task *CA* and *FA*. For *CA* and *FA*, the dataset we have is shown in Table 1 and more detailed data information is shown in Figure 7 in Appendix.

Code Change and Commit Message Consistency Detection. As illustrated in Table 3, we assess the efficacy of `CodeAgent` in detecting the consistency between code changes and commit messages, contrasting its performance with other prevalent methods like CodeBERT, GPT-3.5, and GPT-4.0. This evaluation specifically focuses on merged and closed commits in nine languages. In particular, `CodeAgent` exhibits remarkable performance, outperforming other methods in both merged and closed scenarios. In terms of Recall, `CodeAgent` achieved an impressive 90.11% for merged commits and 87.15% for closed ones, marking a considerable average improvement of 5.62% over the other models. Similarly, the F1-Score of `CodeAgent` stands at 93.89% for merged and 92.40% for closed commits, surpassing its counterparts with an average improvement of 3.79%. More comparable details in different languages are shown in Appendix-Section. K.

Table 3: Comparison of `CodeAgent` with other methods on merged and closed commits across 9 languages on **CA task**. 'Imp' represents the improvement.

Merged	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	63.64	80.08	84.27	80.73	82.04	90.11	5.84
F1	75.00	87.20	90.12	87.62	88.93	93.89	3.77
Closed	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	64.80	79.05	81.75	81.77	83.42	87.15	5.21
F1	77.20	87.35	89.61	89.30	89.81	92.40	3.35
Average	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	64.22	79.57	83.01	81.25	82.73	88.63	5.62
F1	76.01	87.28	89.61	88.46	89.37	93.16	3.79

Format Consistency Detection. In our detailed evaluation of format consistency between commits and original files, `CodeAgent`'s performance was benchmarked against established models like CodeBERT and GPT variants across nine different languages. This comparative analysis, presented in Table 4, was centered around pivotal metrics such as Recall and F1-Score. `CodeAgent` demonstrated a significant edge over the state-of-the-art, particularly in the merged category, with an impressive Recall of 89.34% and an F1-Score of 94.01%. These figures represent an average improvement of 10.81% in Recall and 6.94% in F1-Score over other models. In the closed category, `CodeAgent` continued to outperform, achieving a Recall of 89.57% and an F1-Score of 94.13%, surpassing its counterparts with an improvement of 15.56% in Recall and 9.94% in F1-Score. The overall average performance of CodeAgent further accentuates its superiority, with a Recall of 89.46% and an F1-Score of 94.07%, marking an average improvement of 13.39% in Recall and 10.45% in F1-Score. These results underscore CodeAgent's exceptional capability in accurately detecting format consistency between commits and their original files.

Table 4: Comparison of CodeAgent with other methods on merged and closed commits across the 9 languages on **FA task**. 'Imp' represents the improvement.

Merged	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	60.59	60.72	78.53	70.39	71.21	89.34	10.81
F1	74.14	74.88	87.07	80.69	82.18	94.01	6.94

Closed	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	69.95	73.61	68.46	73.39	74.01	89.57	15.56
F1	80.49	84.19	80.16	83.65	83.90	94.13	9.94

Average	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	65.27	67.17	73.50	71.89	72.61	89.46	15.96
F1	77.32	79.54	83.62	82.17	83.04	94.07	10.45

4.3 Code Revision

We evaluate the effectiveness of CodeAgent in revision suggestion (i.e., bug fixing) based on Edit Progress (EP) metric. We consider Trans-Review, AutoTransform, T5-Review, CodeBERT, GraphCodeBERT, CodeT5 as comparable state of the art. As detailed in Table 5, these approaches exhibit a varied performance across different datasets. In particular, CodeAgent shows remarkable performance in the T5-Review dataset, achieving the highest EP of 37.6%. This is a significant improvement over other methods, which underlines the effectiveness of CodeAgent in handling complex code revision tasks. Furthermore, with an average EP of 31.6%, CodeAgent consistently outperforms its counterparts, positioning itself as a leading solution in automated code revision. Its ability to excel in the T5-Review, a challenging benchmark data, indicates a strong capability to address complex bugs. In addition, its overall average performance surpasses other state-of-the-art models, highlighting its robustness and reliability.

Table 5: Experimental Results for the Code Revision (**CR task**) of CodeAgent and the state-of-the-art works. Bold indicates the best performers.

Approach	**Trans-Review$_{data}$**	**AutoTransform$_{data}$**	**T5-Review$_{data}$**	**Average**
	EP	**EP**	**EP**	**EP**
Trans-Review	-1.1%	-16.6%	-151.2%	-56.3%
AutoTransform	49.7%	**29.9%**	9.7%	29.8%
T5-Review	-14.9%	-71.5%	13.8%	-24.2%
CodeBERT	49.8%	-75.3%	22.3%	-1.1%
GraphCodeBERT	**50.6%**	-80.9%	22.6%	-2.6%
CodeT5	41.8%	-67.8%	25.6%	-0.1%
CodeAgent	42.7%	14.4%	**37.6%**	**31.6%**

5 Related Work

Automating Code Review Activities. Our work contributes to automating code review activities, focusing on detecting source code vulnerabilities and maintaining code consistency. Related studies include Hellendoorn et al. (Hellendoorn et al., 2021), who addressed code change anticipation, and Siow et al. (Siow et al., 2020), who introduced CORE for code modification semantics. Hong et al. (Hong et al., 2022) proposed COMMENTFINDER for comment suggestions, while Tufano et al. (Tufano et al., 2021) and Li et al. (Li et al., 2022) developed tools for code review automation using models like T5CR and CodeReviewer, respectively. Recently, Lu et al. (Lu et al., 2023) incorporated large language models for code review, enhancing fine-tuning techniques.

Collaborative AI. Collaborative AI, involving AI systems working towards shared goals, has seen advancements in multi-agent LLMs (Talebirad and Nadiri, 2023; Qian et al., 2023), focusing on collective thinking, conversation dataset curation (Wei et al., 2023; Li et al., 2023a), and sociological phenomenon exploration (Park et al., 2023). Research by Akata et al. (Akata et al., 2023) and Cai et al. (Cai et al., 2023) further explores LLM cooperation and efficiency. However, there remains a gap in integrating these advancements with structured software engineering practices (Li et al., 2023a; Qian et al., 2023), a challenge our approach addresses by incorporating advanced human processes in multi-agent systems. For a complete overview of related work, please refer to our Appendix-Section B.

6 Conclusion

In this paper, we introduced CodeAgent, a novel multi-agent framework that automates code reviews. CodeAgent leverages its novel QA-Checker system to maintain focus on the review's objectives and ensure alignment. Our experiments demonstrate CodeAgent's effectiveness in detecting vulnerabilities, enforcing code-message consistency, and promoting uniform code style. Furthermore, CodeAgent outperforms existing state-of-the-art solutions in code revision suggestions. By incorporating human-like conversational elements and considering the specific characteristics of code review, `CodeAgent` significantly improves both efficiency and accuracy. We believe this work opens exciting new avenues for research and collaboration practices in software development.

7 Acknowledgments

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014). The author Cedric Lothritz is supported by the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475.

Limitations

Firstly, the generalizability of the system across different software development environments or industries may require further validation and testing. While the system has shown promising results in the provided datasets, its applicability to other contexts remains uncertain without additional empirical evidence. This limitation suggests that the findings may not be fully transferable to all settings within the software development domain. Secondly, the baseline test used in the study might be insufficient. The current testing approach may not fully capture the system's performance, particularly in edge cases or more complex scenarios. This could result in an overestimation of the system's capabilities and an underestimation of its limitations. Further, more comprehensive testing is needed to establish a more robust baseline and to ensure that the system performs reliably across a wider range of conditions.

Ethics Statements

This study was conducted in compliance with ethical guidelines and standards for research. The research did not involve human participants, and therefore, did not require informed consent or ethical review from an institutional review board. All data used in this study were publicly available, and no personal or sensitive information was accessed or processed. The development and evaluation of the `CodeAgent` system were performed with a focus on transparency, reproducibility, and the potential positive impact on the software development community.

References

Elif Akata, Lion Schulz, Julian Coda-Forno, Seong Joon Oh, Matthias Bethge, and Eric Schulz. 2023. Playing repeated games with large language models. *arXiv preprint*.

Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE.

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*, pages 133–142. IEEE.

Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. 2022. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *arXiv preprint*.

Hyungjoo Chae, Yongho Song, Kai Tzu-iunn Ong, Taeyoon Kwon, Minjin Kim, Youngjae Yu, Dongha Lee, Dongyeop Kang, and Jinyoung Yeo. 2023. Dialogue chain-of-thought distillation for commonsense-aware conversational agents. *arXiv preprint arXiv:2310.09343*.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*.

Nicole Davila and Ingrid Nunes. 2021. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177:110951.

Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. 2022. Aligning offline metrics and human judgments of value of ai-pair programmers. *arXiv preprint arXiv:2210.16494*.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Cobus Greyling. 2023. Prompt drift and chaining.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. 2020. Does code review really remove coding convention violations? In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 43–53. IEEE.

Vincent J Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. Towards automating code review at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1479–1482.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 507–519.

Mohammad Hossin and Md Nasir Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. Camel: Communicative agents for" mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*.

Yuan Li, Yixuan Zhang, and Lichao Sun. 2023b. Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents. *arXiv preprint arXiv:2310.06500*.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Codereviewer: Pre-training for automating code review activities. *arXiv e-prints*, pages arXiv–2203.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*.

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)*, pages 647–658. IEEE.

Delano Oliveira, Reydne Santos, Fernanda Madeiral, Hidehiko Masuhara, and Fernando Castor. 2023. A systematic literature review on the impact of formatting elements on code legibility. *Journal of Systems and Software*, 203:111728.

OPENAI. 2022. Chatgpt.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. 2020. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE.

Miroslaw Staron, Mirosław Ochodek, Wilhelm Meding, and Ola Söder. 2020. Using machine learning to identify code fragments for manual review. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 513–516. IEEE.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents.

Xunzhu Tang, Zhenghan Chen, Kisub Kim, Haoye Tian, Saad Ezzini, and Jacques Klein. 2023. Just-in-time security patch detection–llm at the rescue for data augmentation. *arXiv preprint arXiv:2312.01241*.

Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th international conference on software engineering*, pages 237–248.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*.

Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. *arXiv preprint arXiv:2208.04125*.

Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2291–2302.

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)*, pages 163–174. IEEE.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. 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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2023. Unleashing cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *arXiv preprint arXiv:2307.05300*.

Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58.

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*, 35:24824–24837.

Jimmy Wei, Kurt Shuster, Arthur Szlam, Jason Weston, Jack Urbanek, and Mojtaba Komeili. 2023. Multi-party chat: Conversational agents in group settings with humans and models. *arXiv preprint*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. 2024a. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*.

Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F Bissyandé, and Shunfu Jin. 2024b. Cref: an llm-based conversational software repair framework for programming tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 882–894.

Xiaoyu Yang, Jie Lu, and En Yu. 2024c. Adapting multi-modal large language model to concept drift in the long-tailed open world. *arXiv preprint arXiv:2405.13459*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint*.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B Tenenbaum, Tianmin Shu, and Chuang Gan. 2023. Building cooperative embodied agents modularly with large language models. *arXiv preprint*.

Mengxi Zhang, Huaxiao Liu, Chunyang Chen, Yuzhou Liu, and Shuotong Bai. 2022. Consistent or not? an investigation of using pull request template in github. *Information and Software Technology*, 144:106797.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*.

Jonathan Zheng, Alan Ritter, and Wei Xu. 2024. Neo-bench: Evaluating robustness of large language models with neologisms. *arXiv preprint arXiv:2402.12261*.

Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. 2023. Generation-based code review automation: How far are we? *arXiv preprint arXiv:2303.07221*.

Contents (Appendix)

A Details of QA-Checker Algorithm

Lemma A.1. *Let $\mathcal{Q}(Q_i, A_i)$ denote the quality assessment function of the QA-Checker for the question-answer pair (Q_i, A_i) in a conversation at the i-th iteration. Assume $\mathcal{Q}$ is twice differentiable and its Hessian matrix $H(\mathcal{Q})$ is positive definite. If the QA-Checker modifies the question Q_i to Q_{i+1} by attaching an additional instruction aai_i, and this leads to a refined answer A_{i+1}, then the sequence $\{(Q_i, A_i)\}$ converges to an optimal question-answer pair (Q^*, A^*), under specific regularity conditions.*

Proof. The QA-Checker refines the question and answers using the rule:

$$\begin{aligned} Q_{i+1} &= Q_i + aai_i, \\ A_{i+1} &= A_i - \alpha H(\mathcal{Q}(Q_i, A_i))^{-1} \nabla \mathcal{Q}(Q_i, A_i), \end{aligned}$$

where α is the learning rate. To analyze convergence, we consider the Taylor expansion of $\mathcal{Q}$ around (Q_i, A_i):

$$\begin{aligned} \mathcal{Q}(Q_{i+1}, A_{i+1}) \approx \mathcal{Q}(Q_i, A_i) &+ \nabla \mathcal{Q}(Q_i, A_i) \\ &\cdot (Q_{i+1} - Q_i, A_{i+1} - A_i) \\ &+ \frac{1}{2}(Q_{i+1} - Q_i, A_{i+1} - A_i)^T \\ &H(\mathcal{Q}(Q_i, A_i))(Q_{i+1} - Q_i, A_{i+1} - A_i). \end{aligned}$$

Substituting the update rule and rearranging, we get:

$$\begin{aligned} \mathcal{Q}(Q_{i+1}, A_{i+1}) \approx \mathcal{Q}(Q_i, A_i) & \\ &- \alpha \nabla \mathcal{Q}(Q_i, A_i)^T H(\mathcal{Q}(Q_i, A_i))^{-1} \\ &\nabla \mathcal{Q}(Q_i, A_i) \\ &+ \frac{\alpha^2}{2} \nabla \mathcal{Q}(Q_i, A_i)^T H(\mathcal{Q}(Q_i, A_i))^{-1} \\ &\nabla \mathcal{Q}(Q_i, A_i). \end{aligned}$$

For sufficiently small α, this model suggests an increase in $\mathcal{Q}$, implying convergence to an optimal question-answer pair (Q^*, A^*) as $i \to \infty$. The convergence relies on the positive definiteness of $H(\mathcal{Q})$ and the appropriate choice of α, ensuring each iteration moves towards an improved quality of the question-answer pair. □

In practical terms, this lemma and its proof underpin the QA-Checker's ability to refine answers iteratively. The QA-Checker assesses the quality of each answer concerning the posed question, employing advanced optimization techniques that are modeled by the modified Newton-Raphson method to enhance answer quality. This framework ensures that, with each iteration, the system moves closer to the optimal answer, leveraging both first and second-order derivatives for efficient and effective learning.

Further Discussion The QA-Checker computes $\mathcal{Q}(Q_i, A_i)$ at each iteration i and compares it to a predefined quality threshold τ. If $\mathcal{Q}(Q_i, A_i) < \tau$, the QA-Checker generates an additional instruction aai_i to refine the question to $Q_{i+1} = Q_i + aai_i$, prompting the agents to generate an improved answer A_{i+1}.

First, we assume that the quality assessment function $\mathcal{Q}(Q_i, A_i)$ is twice differentiable with respect to the question Q_i. This assumption is reasonable given the smooth nature of the component functions (relevance, specificity, and coherence) and the use of continuous word embeddings. Next, we apply the second-order Taylor approximation to $\mathcal{Q}(Q_{i+1}, A_{i+1})$ around the point (Q_i, A_i):

$$\begin{aligned}\mathcal{Q}(Q_{i+1}, A_{i+1}) \approx \mathcal{Q}(Q_i, A_i) + \nabla\mathcal{Q}(Q_i, A_i)^T \Delta Q_i \\ + \frac{1}{2}\Delta Q_i^T H(\mathcal{Q}(Q_i, A_i))\Delta Q_i + R_2(\Delta Q_i)\end{aligned}$$

where $\Delta Q_i = Q_{i+1} - Q_i$, $H(\mathcal{Q}(Q_i, A_i))$ is the Hessian matrix of $\mathcal{Q}$ evaluated at (Q_i, A_i), and $R_2(\Delta Q_i)$ is the remainder term.

Assuming that the remainder term $R_2(\Delta Q_i)$ is negligible and that the Hessian matrix is positive definite, we can approximate the optimal step ΔQ_i^* as:

$$\Delta Q_i^* \approx -H(\mathcal{Q}(Q_i, A_i))^{-1}\nabla\mathcal{Q}(Q_i, A_i).$$

Substituting this approximation into the Taylor expansion and using the fact that $Q_{i+1} = Q_i + \alpha\Delta Q_i^*$ (where α is the learning rate), we obtain:

$$\begin{aligned}\mathcal{Q}(Q_{i+1}, A_{i+1}) \approx & \mathcal{Q}(Q_i, A_i) - \alpha\nabla\mathcal{Q}(Q_i, A_i)^T \\ & \cdot H(\mathcal{Q}(Q_i, A_i))^{-1}\nabla\mathcal{Q}(Q_i, A_i) \\ & + \frac{\alpha^2}{2}\nabla\mathcal{Q}(Q_i, A_i)^T H(\mathcal{Q}(Q_i, A_i))^{-1} \\ & \cdot \nabla\mathcal{Q}(Q_i, A_i).\end{aligned}$$

The assumptions of twice differentiability, negligible remainder term, and positive definite Hessian matrix provide a more solid foundation for the approximation in Lemma 3.1. For sufficiently small α, this approximation suggests an increase in $\mathcal{Q}$, implying convergence to an optimal question-answer pair (Q^*, A^*) as $i \to \infty$. The convergence relies on the positive definiteness of $H(\mathcal{Q})$ and the appropriate choice of α, ensuring each iteration moves towards an improved quality of the question-answer pair.

The quality assessment function $\mathcal{Q}$ used by the QA-Checker is defined as:

$$\begin{aligned}\mathcal{Q}(Q_i, A_i) = & \alpha \cdot \text{Relevance}(Q_i, A_i) \\ & + \beta \cdot \text{Specificity}(A_i) \\ & + \gamma \cdot \text{Coherence}(A_i)\end{aligned}$$

where:

- Q_i and A_i represent the question and answer at the i-th iteration of the conversation.
- Relevance(Q_i, A_i) measures how well the answer A_i addresses the key points and intent of the question Q_i, computed as:

  $$\text{Relevance}(Q_i, A_i) = \frac{\vec{Q_i} \cdot \vec{A_i}}{|\vec{Q_i}||\vec{A_i}|}$$

  where $\vec{Q_i}$ and $\vec{A_i}$ are vector representations of Q_i and A_i.
- Specificity(A_i) assesses how specific and detailed the answer A_i is, calculated as:

  $$A_i = \frac{\sum_{t\in\text{ContentWords}(A_i)} \text{TechnicalityScore}(t)}{\text{Length}(A_i)}$$

  where ContentWords(A_i) is the set of substantive content words in A_i, TechnicalityScore(t) is a measure of how technical or domain-specific the term t is, and Length(A_i) is the total number of words in A_i.
- Coherence(A_i) evaluates the logical flow and structural coherence of the answer A_i, computed as:

  $$\begin{aligned}\text{Coherence}(A_i) = & \alpha \cdot \text{DiscourseConnectives}(A_i) \\ & + \beta \cdot \text{CoreferenceConsistency}(A_i) \\ & + \gamma \cdot \text{AnswerPatternAdherence}(A_i)\end{aligned}$$

  where DiscourseConnectives(A_i) is the density of discourse connectives in A_i, CoreferenceConsistency(A_i) measures the consistency of coreference chains in A_i, and AnswerPatternAdherence(A_i) assesses how well A_i follows the expected structural patterns for the given question type.

α, β, and γ are non-negative weights that sum to 1, with $\alpha = \beta = \gamma$.

B Complete Related Work

Automating Code Review Activities Our focus included detecting source code vulnerabilities, ensuring style alignment, and maintaining commit message and code consistency. Other studies explore various aspects of code review. Hellendoorn et al. (Hellendoorn et al., 2021) addressed the challenge of anticipating code change positions. Siow et al. (Siow et al., 2020) introduced CORE, employing multi-level embeddings for code modification semantics and retrieval-based review suggestions. Hong et al. (Hong et al., 2022) proposed COMMENTFINDER, a retrieval-based method for suggesting comments during code reviews. Tufano et al. (Tufano et al., 2021) designed T5CR with SentencePiece, enabling work with raw source code without abstraction. Li et al. (Li et al., 2022) developed CodeReviewer, focusing on code diff quality, review comment generation, and code refinement using the T5 model. Recently, large language models have been incorporated; Lu et al. (Lu et al., 2023) fine-tuned LLama with prefix tuning for LLaMA-Reviewer, using parameter-efficient fine-tuning and instruction tuning in a code-centric domain.

Collaborative AI Collaborative AI refers to artificial intelligent systems designed to achieve shared goals with humans or other AI systems. Previous research extensively explores the use of multiple LLMs in collaborative settings, as demonstrated by Talebirad et al. (Talebirad and Nadiri, 2023) and Qian et al. (Qian et al., 2023). These approaches rely on the idea that inter-agent interactions enable LLMs to collectively enhance their capabilities, leading to improved overall performance. The research covers various aspects of multi-agent scenarios, including collective thinking, conversation dataset curation, sociological phenomenon exploration, and collaboration for efficiency. Collective thinking aims to boost problem-solving abilities by orchestrating discussions among multiple agents. Researchers like Wei et al. (Wei et al., 2023) and Li et al. (Li et al., 2023a) have created conversational datasets through role-playing methodologies. Sociological phenomenon investigations, such as Park et al. (Park et al., 2023)'s work, involve creating virtual communities with rudimentary language interactions and limited cooperative endeavors. In contrast, Akata et al. (Akata et al., 2023) scrutinized LLM cooperation through orchestrated repeated games. Collaboration for efficiency, proposed by Cai et al. (Cai et al., 2023), introduces a model for cost reduction through large models as tool-makers and small models as tool-users. Zhang et al. (Zhang et al., 2023) established a framework for verbal communication and collaboration, enhancing overall efficiency. However, Li et al. (Li et al., 2023a) and Qian et al. (Qian et al., 2023), presenting a multi-agent framework for software development, primarily relied on natural language conversations, not standardized software engineering documentation, and lacked advanced human process management expertise. Challenges in multi-agent cooperation include maintaining coherence, avoiding unproductive loops, and fostering beneficial interactions. Our approach emphasizes integrating advanced human processes, like code review in software maintenance, within multi-agent systems.

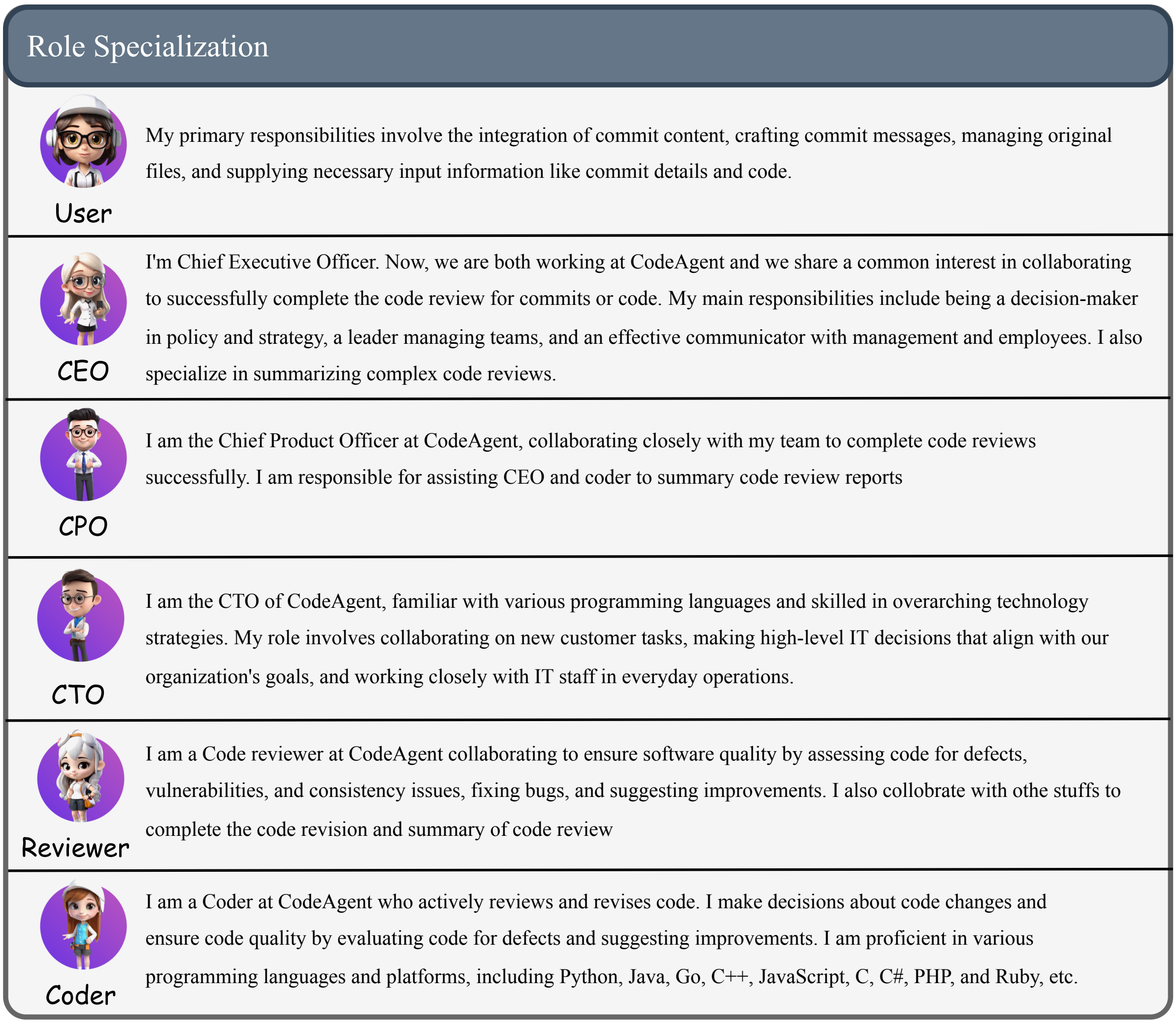
C Experimental Details

In our work, the maximum number of conversation rounds is set at 10.

C.1 Role Definition

Six roles are defined as shown in Figure 5.

Apart from that, for the QA-checker in `CodeAgent`, we define an initial prompt for it, which is shown as follows:



Role Specialization

User	My primary responsibilities involve the integration of commit content, crafting commit messages, managing original files, and supplying necessary input information like commit details and code.
CEO	I'm Chief Executive Officer. Now, we are both working at CodeAgent and we share a common interest in collaborating to successfully complete the code review for commits or code. My main responsibilities include being a decision-maker in policy and strategy, a leader managing teams, and an effective communicator with management and employees. I also specialize in summarizing complex code reviews.
CPO	I am the Chief Product Officer at CodeAgent, collaborating closely with my team to complete code reviews successfully. I am responsible for assisting CEO and coder to summary code review reports
CTO	I am the CTO of CodeAgent, familiar with various programming languages and skilled in overarching technology strategies. My role involves collaborating on new customer tasks, making high-level IT decisions that align with our organization's goals, and working closely with IT staff in everyday operations.
Reviewer	I am a Code reviewer at CodeAgent collaborating to ensure software quality by assessing code for defects, vulnerabilities, and consistency issues, fixing bugs, and suggesting improvements. I also collobrate with othe stuffs to complete the code revision and summary of code review
Coder	I am a Coder at CodeAgent who actively reviews and revises code. I make decisions about code changes and ensure code quality by evaluating code for defects and suggesting improvements. I am proficient in various programming languages and platforms, including Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby, etc.

Figure 5: Specialization of six main characters in `CodeAgent`.

I'm the QA-Checker, an AI-driven agent specializing in ensuring quality and coherence in conversational dynamics, particularly in code review discussions at CodeAgent. My primary role involves analyzing and aligning conversations to maintain topic relevance, ensuring that all discussions about code commits and reviews stay focused and on track. As a sophisticated component of the AI system, advanced algorithms are applied, including chain-of-thought reasoning and optimization techniques, to evaluate and guide conversational flow. I am adept at identifying and correcting topic drifts and ensuring that every conversation adheres to its intended purpose. My capabilities extend to facilitating clear and effective communication between team members, making me an essential asset in streamlining code review processes and enhancing overall team collaboration and decision making.

C.2 Execute Time Across Languages

As depicted in the data, we observe a significant trend in the average execution time for code reviews in `CodeAgent` across various programming languages. The analysis includes nine languages: Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. For each language, the average execution time of code reviews for both merged and closed pull requests (PRs) is measured. The results, presented in Figure 6, indicate that, on average, the execution time for merged PRs is longer than that for closed PRs by approximately 44.92 seconds. This considerable time difference can be attributed to several potential reasons. One primary explanation is that merged PRs likely undergo a more rigorous and detailed review process. They are intended to be integrated into the main codebase, and as such, contributors might be requested to update their commits in the PRs more frequently to adhere to the project's high-quality standards. On the other hand, closed PRs, which are not meant for merging, might not require such extensive review processes, leading to shorter review times on average, which may also be the reason they are not merged into main projects.

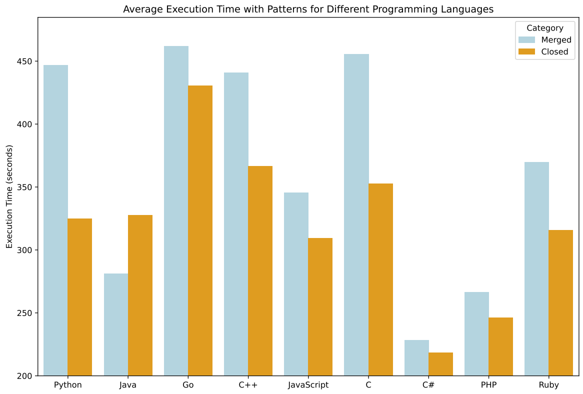


Figure 6: Execution time with `CodeAgent` across different language (count unit: second).

D Comparative Analysis of QA-Checker AI System and Recursive Self-Improvement Systems

In this section, we will delve into the differences between QA-Checker and self-improvement systems (Hong et al., 2023), and underscore the importance of the QA-Checker in role conversations.

D.1 Comparison Table

We begin with a comparative overview presented in Table 6.

D.2 Differences and Implications

The key differences between these systems lie in their application scope, learning mechanisms, and improvement scopes. The QA-Checker is highly specialized, focusing on QA tasks with efficiency and precision. In contrast, recursive self-improvement systems boast a broader application range and adaptability, integrating experiences from diverse projects for systemic improvements.

D.3 Importance of QA-Checker in Role Conversations

In the context of role conversations, the QA-Checker plays a pivotal role. Its specialized nature makes it exceptionally adept at handling specific conversational aspects, such as accuracy, relevance, and clarity in responses. This specialization is crucial in domains where the quality of information is paramount, ensuring that responses are not only correct but also contextually appropriate and informative.

Furthermore, the efficiency of the QA-Checker in refining responses based on advanced optimization techniques makes it an invaluable tool in dynamic conversational environments. It can quickly adapt to the nuances of a conversation, providing high-quality responses that are aligned with the evolving nature of dialogue.

D.4 Conclusion

While recursive self-improvement systems offer broad adaptability and systemic learning, the QA-Checker stands out in its specialized role in QA tasks, particularly in role conversations. Its focused approach to improving answer quality and its efficiency in handling conversational nuances make it an essential component in AI-driven communication systems.

E Capabilities Analysis between `CodeAgent` and Other Methods

Compared to open-source baseline methods such as AutoGPT and autonomous agents such as ChatDev and MetaGPT, `CodeAgent` offers functions for code review tasks: consistency analysis, vulnerability analysis, and format analysis. As shown in Table 7, our `CodeAgent` encompasses a wide range of abilities to handle complex code review tasks efficiently. Incorporating the QA-Checker self-improved module can significantly improve the conversation generation between agents and contribute to the improvement of code review. Compared to COT, the difference and the advantages of `CodeAgent` with QA-Checker are shown in Section D.

F Dataset

Previous Dataset As shown in Zhou et al. (2023), our study incorporates three distinct datasets for evaluating the performance of `CodeAgent`: Trans-Review$_{\text{data}}$, AutoTransform$_{\text{data}}$, and T5-Review$_{\text{data}}$. Trans-Review$_{\text{data}}$, compiled by Tufano et al. (Tufano et al., 2021), derives from Gerrit and GitHub projects, excluding noisy or overly lengthy comments and review data with new tokens in revised code not present in the initial submission. AutoTransform$_{\text{data}}$, collected by

Table 6: Comparative Overview of QA-Checker AI System and Recursive Self-Improvement Systems

Feature/System	QA-Checker AI System	Recursive Self-Improvement System
Application Focus	Specialized for QA tasks with precise task execution	Broad scope, covering various dimensions like software development and learning algorithms
Learning Mechanism	Advanced optimization techniques for iterative improvement in QA	Multi-level learning: learning, meta-learning, and recursive self-improvement
Scope of Improvement	Focused on individual capability in specific QA tasks	Enhances the entire system, including multi-agent interactions and communication protocols
Experience Integration	Based on mathematical models to optimize answer quality	Utilizes experiences from past projects to improve overall performance

Table 7: Comparison of capabilities for `CodeAgent` and other approaches. '✓' indicates the presence of a specific feature in the corresponding framework, '✗ is absence. ChatDev and MetaGPT are two representative multi-agent frameworks, GPT is a kind of single-agent framework, and CodeBert is a representative pre-trained model.

Approaches	Consistency Analysis	Vulnerability Analysis	Format Analysis	Code Revision	COT	QA-Checker
ChatDev (Qian et al., 2023)	✗	✗	✗	✗	✓	✗
MetaGPT (Hong et al., 2023)	✗	✗	✗	✗	✓	✗
GPT (OPENAI, 2022)	✓	✓	✓	✓	✗	✗
CodeBert (Feng et al., 2020)	✓	✓	✓	✓	✗	✗
`CodeAgent`	✓	✓	✓	✓	✓	✓

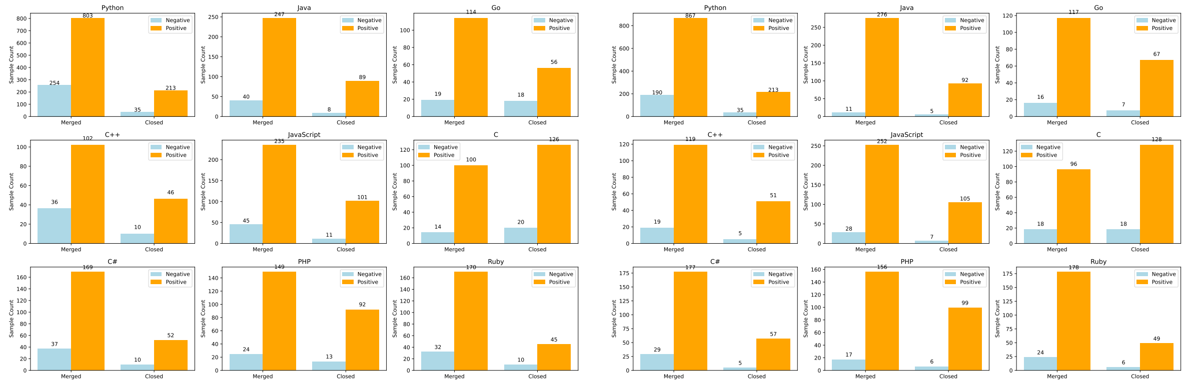
Thongtanunam et al. (Thongtanunam et al., 2022) from three Gerrit repositories, comprises only submitted and revised codes without review comments. Lastly, T5-Review$_{\text{data}}$, gathered by Tufano et al. (Tufano et al., 2022) from Java projects on GitHub, filters out noisy, non-English, and duplicate comments. These datasets are employed for Code Revision Before Review (CRB) and Code Revision After Review (CRA) tasks, with the exception of AutoTransform$_{\text{data}}$ for CRA and Review Comment Generation (RCG) due to its lack of review comments.

New Dataset Design and Collection To enhance our model evaluation and avoid data leakage, we curated a new dataset, exclusively collecting data from repositories created after April 2023. This approach ensures the evaluation of our CodeAgent model on contemporary and relevant data, free from historical biases. The new dataset is extensive, covering a broad spectrum of software projects across nine programming languages.

Dataset Description Our dataset, illustrated in Fig. 8, encapsulates a detailed analysis of consistency and format detection in software development, spanning various programming languages. It includes CA (consistency between commit and commit message (See Sec 2.1)) and FA (format consistency between commit and original (See Sec 2.1)) data, segmented into positive and negative samples based on the merged and closed status of pull requests. For example, in Python, the dataset comprises 254 merged and 35 closed negative CA samples, alongside 803 merged and 213 closed positive CA samples, with corresponding distributions for other languages like Java, Go, C++, and more. Similarly, the FA data follows this pattern of positive and negative samples across languages. Figure 7 graphically represents this data, highlighting the distribution and comparison of merged versus closed samples in both CA and FA categories for each language. This comprehensive dataset, covering over 3,545 commits and nearly 2,933 pull requests from more than 180 projects, was meticulously compiled using a custom crawler designed for GitHub API interactions, targeting post-April 2023 repositories to ensure up-to-date and diverse data for an in-depth analysis of current software development trends.

Table 8: Statistics of Studied Datasets.

Dataset Statistics	#Train	#Valid	#Test
Trans-Review	13,756	1,719	1,719
AutoTransform	118,039	14,750	14,750
T5-Review	134,239	16,780	16,780



(a) Positive and negative data of both merged and closed commits across 9 languages on *CA* task (Sec 2.1).

(b) Positive and negative data of both merged and closed commits across 9 languages on *FA* task (Sec 2.1).

Figure 7: Distribution of positive, negative of both merged and closed data across 9 languages, including ‘python’, ‘java’, ‘go’, ‘c++’, ‘javascript’, ‘c’, ‘c#’, ‘php’, ‘ruby’.

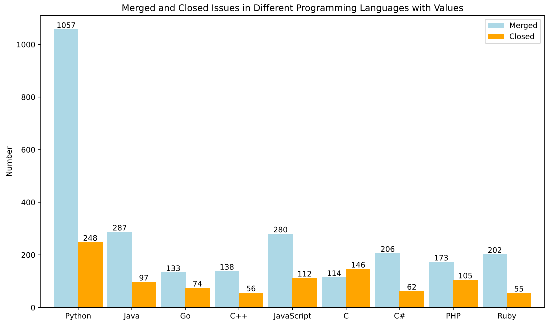


Figure 8: Comparative Visualization of Merged and Closed Commit Counts Across Various Programming Languages

G Key Factors Leading to Vulnerabilities

The following table outlines various key factors that can lead to vulnerabilities in software systems, along with their descriptions. These factors should be carefully considered and addressed to enhance the security of the system.

H Data Leakage Statement

As the new dataset introduced in Section F, the time of the collected dataset is after April 2023, avoiding data leakage while we evaluate `CodeAgent` on **codeData** dataset.

I Algorithmic Description of `CodeAgent` Pipeline with QA-Checker

This algorithm demonstrates the integration of QA-Checker within the `CodeAgent` pipeline, employing mathematical equations to describe the QA-Checker's iterative refinement process.

Algorithm 1 Integrated Workflow of `CodeAgent` with QA-Checker

```
Input: Code submission, commit message, original files
Output: Refined code review document
Initialize phase $p = 1$
while $p \leq 4$ do
    Switch: Phase $p$
    Case 1: Basic Info Sync
    Conduct initial information analysis
    Update: $p = 2$
    Case 2: Code Review
    Perform code review with Coder and Reviewer
    Update: $p = 3$
    Case 3: Code Alignment
    Apply code revisions based on feedback
    Update: $p = 4$
    Case 4: Document
    Finalize review document
    Update: $p = 5$ (End)
    QA-Checker Refinement (Applies in Cases 2 and 3)
    Let $Q_i$ be the current question and $A_i$ the current answer
    Evaluate response quality: $qScore = \mathcal{Q}(Q_i, A_i)$
    if $qScore$ below threshold then
        Generate additional instruction $aai$
        Update question: $Q_{i+1} = Q_i + aai$
        Request new response: $A_{i+1}$
    end if
end while
Return: Refined code review document
```

No.	Vulnerability Factor	Description
1	Insufficient Input Validation	Check for vulnerabilities like SQL injection, Cross-Site Scripting (XSS), and command injection in new or modified code, especially where user input is processed.
2	Buffer Overflows	Particularly in lower-level languages, ensure that memory management is handled securely to prevent overflows.
3	Authentication and Authorization Flaws	Evaluate any changes in authentication and authorization logic for potential weaknesses that could allow unauthorized access or privilege escalation.
4	Sensitive Data Exposure	Assess handling and storage of sensitive information like passwords, private keys, or personal data to prevent exposure.
5	Improper Error and Exception Handling	Ensure that errors and exceptions are handled appropriately without revealing sensitive information or causing service disruption.
6	Vulnerabilities in Dependency Libraries or Components	Review updates or changes in third-party libraries or components for known vulnerabilities.
7	Cross-Site Request Forgery (CSRF)	Verify that adequate protection mechanisms are in place against CSRF attacks.
8	Unsafe Use of APIs	Check for the use of insecure encryption algorithms or other risky API practices.
9	Code Injection	Look for vulnerabilities related to dynamic code execution.
10	Configuration Errors	Ensure that no insecure configurations or settings like open debug ports or default passwords have been introduced.
11	Race Conditions	Analyze for potential data corruption or security issues arising from race conditions.
12	Memory Leaks	Identify any changes that could potentially lead to memory leaks and resource exhaustion.
13	Improper Resource Management	Check resource management, such as proper closure of file handles or database connections.
14	Inadequate Security Configurations	Assess for any insecure default settings or unencrypted communications.
15	Path Traversal and File Inclusion Vulnerabilities	Examine for risks that could allow unauthorized file access or execution.
16	Unsafe Deserialization	Look for issues that could allow the execution of malicious code or tampering with application logic.
17	XML External Entity (XXE) Attacks	Check if XML processing is secure against XXE attacks.
18	Inconsistent Error Handling	Review error messages to ensure they do not leak sensitive system details.
19	Server-Side Request Forgery (SSRF)	Analyze for vulnerabilities that could be exploited to attack internal systems.
20	Unsafe Redirects and Forwards	Check for vulnerabilities leading to phishing or redirection attacks.
21	Use of Deprecated or Unsafe Functions and Commands	Identify usage of any such functions and commands in the code.
22	Code Leakages and Hardcoded Sensitive Information	Look for hardcoded passwords, keys, or other sensitive data in the code.
23	Unencrypted Communications	Verify that data transmissions are securely encrypted to prevent interception and tampering.
24	Mobile Code Security Issues	For mobile applications, ensure proper handling of permission requests and secure data storage.
25	Cloud Service Configuration Errors	Review any cloud-based configurations for potential data leaks or unauthorized access.

In this algorithm, $\mathcal{Q}(Q_i, A_i)$ represents the quality assessment function of the QA-Checker, which evaluates the relevance and accuracy of the answer A_i to the question Q_i. If the quality score $qScore$ is below a predefined threshold, the QA-Checker intervenes by generating an additional instruction aai to refine the question, prompting a more accurate response in the next iteration.

J Detailed Performance of `CodeAgent` in Various Languages on *VA* task

In our comprehensive analysis using `CodeAgent`, as detailed in Table 9, we observe a diverse landscape of confirmed vulnerabilities across different programming languages. The table categorizes these vulnerabilities into 'merged' and 'closed' statuses for languages such as Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. A significant finding is a markedly high number of 'merged' vulnerabilities in Python, potentially reflective of its extensive application or intrinsic complexities leading to security gaps. Conversely, languages like Go, Ruby, and C exhibit notably lower counts in both categories, perhaps indicating lesser engagement in complex applications or more robust security protocols. Table 9 that the 'closed' category consistently presents lower vulnerabilities than 'merged' across most languages, signifying effective resolution mechanisms. However, an exception is noted in C, where 'closed' counts surpass those of 'merged', possibly indicating either delayed vulnerability identification or efficient mitigation strategies. Remarkably, the Rate$_{close}$ is generally observed to be higher than Rate$_{merge}$ across the languages, exemplifying a significant reduction in vulnerabilities post-resolution. For example, Python demonstrates a Rate$_{merge}$ of 14.00% against a higher Rate$_{close}$ of 18.16%. This trend is consistent in most languages, emphasizing the importance of proactive vulnerability management. The Rate$_{avg}$, representing the proportion of confirmed vulnerabilities against the total of both merged and closed items, further elucidates this point, with C++ showing the highest Rate$_{avg}$ at 16.49%. These insights not only underline the diverse vulnerability landscape across programming languages but also highlight the adeptness of `CodeAgent` in pinpointing and verifying vulnerabilities in these varied contexts.

K More detailed experimental results on CA and FA tasks

Detailed experimental results of CA are shown in Figure 9 and Figure 10. Detailed experimental results of FA are shown in Figure 11 and Figure 12.

L Case Study

As shown in Table 10, we can easily localize the figure numbers of case studies for specific programming languages.

L.1 Performance on 9 languages

Table 10: Correlation Table between specific programming language and case study.

Programming Language	Figure No.
Python	13
Java	14
Go	15
C++	16
JavaScript	17
C	18
C#	19
php	20
Ruby	21

L.2 Difference of `CodeAgent`-3.5 and `CodeAgent`-4.0

`CodeAgent`-3.5 and `CodeAgent`-4.0 in this paper has no difference in general code review, however, `CodeAgent`-4.0 is more powerful in processing long input sequences and logic reasoning. As shown in Figure 22, we take one example of consistency detection between commit and commit message and find that `CodeAgent`-4.0 diffs from `CodeAgent`-3.5 in the detailed explanation. `CodeAgent`-3.5 output a report with 15k lines while `CodeAgent`-4.0 outputs a report with more than 17.7k lines. Detailed data is shown in `https://zenodo.org/records/10607925`.

M Ablation study

In this section, we evaluate the performance of different parts in `CodeAgent` in vulnerability analysis. `CodeAgent` is based on chain-of-thought (COT) and large language model (a.k.a. GPT). As shown in Section 4.1, `CodeAgent` outperforms baselines (a.k.a. CodeBERT, GPT-3.5,

Table 9: Vulnerable problems (#) found by `CodeAgent`. Rate$_{merge}$ means the value of confirmed divided by the total number in the merged and Rate$_{close}$ is the value of confirmed divided by the total number in the closed. Rate$_{avg}$ is the value of the confirmed number divided by the total number of the merged and closed.

`CodeAgent`	Python	Java	Go	C++	JavaScript	C	C#	PHP	Ruby
merged (total#)	1,057	287	133	138	280	114	206	173	202
merged (confirmed#)	148	17	11	19	34	9	21	28	20
Rate$_{merge}$	14.00%	5.92%	8.27%	13.77%	12.14%	7.89%	10.19%	16.18%	9.90%
closed (total#)	248	97	74	56	112	146	62	105	55
closed (confirmed#)	45	10	5	13	16	26	7	15	5
Rate$_{close}$	18.16%	10.31%	6.76%	23.2%	14.29%	17.81%	11.29%	14.29%	9.09%
Total number (#)	1,305	384	207	194	392	260	268	278	257
Total confirmed (#)	193	27	16	32	50	35	28	43	25
Rate$_{avg}$	14.79%	7.03%	7.73%	16.49%	12.76%	13.46%	10.45%	14.47%	9.73%

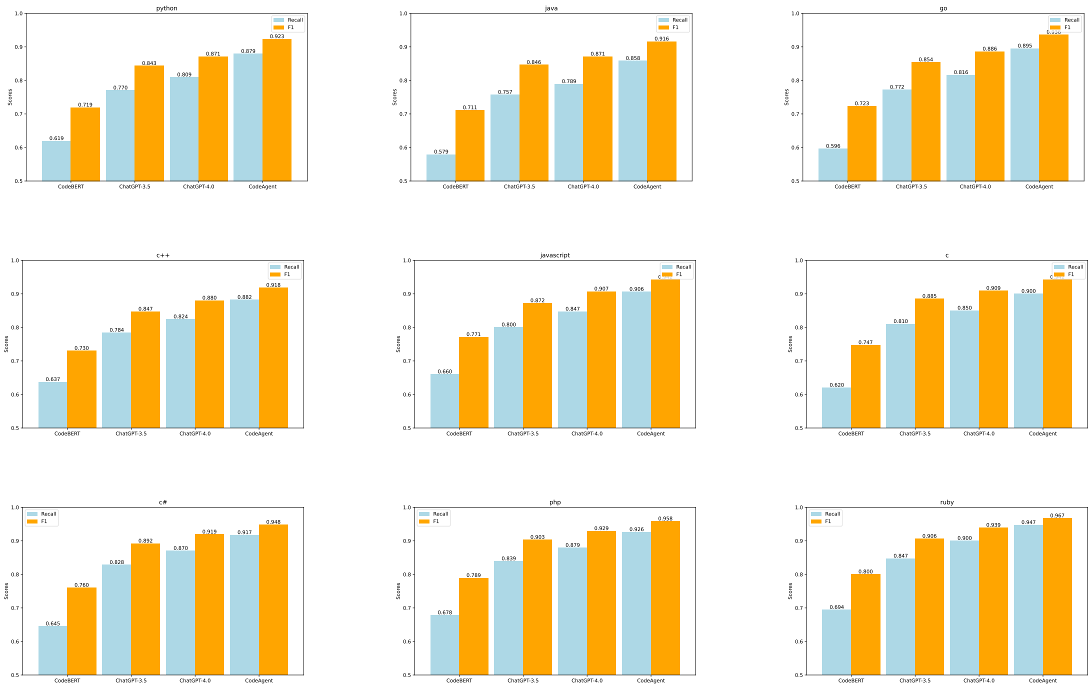


Figure 9: Comparison of models on the **merged** data across 9 languages on **CA task**.

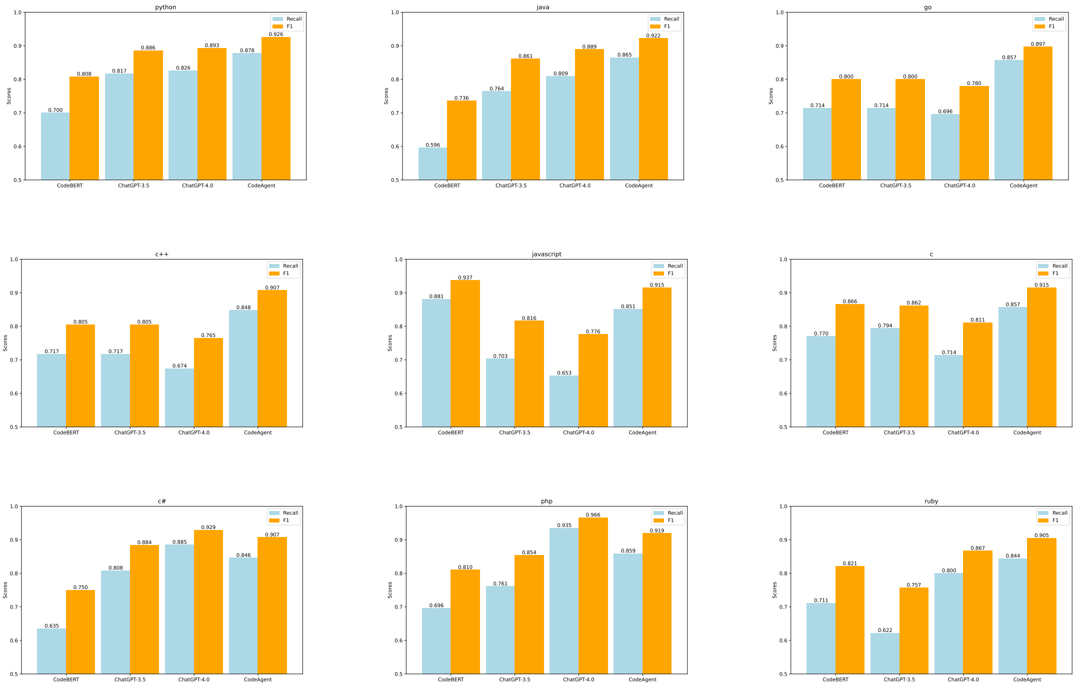


Figure 10: Comparison of models on the **closed** data across 9 languages on **CA task**.

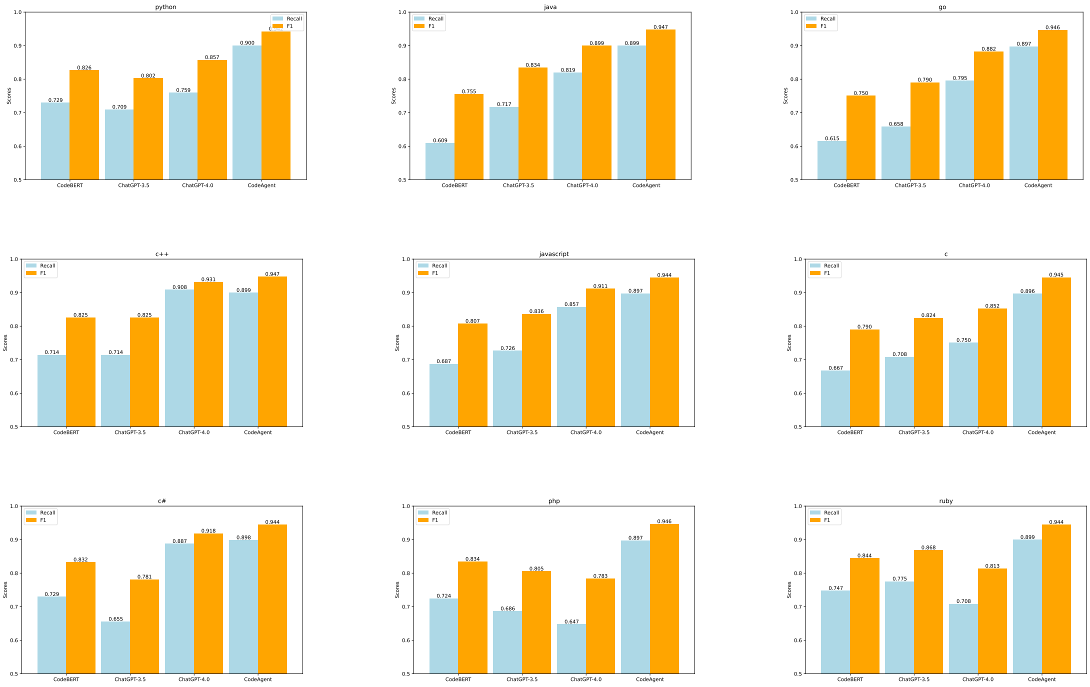


Figure 11: Comparison of models on the **merged** data across 9 languages on **FA task**.

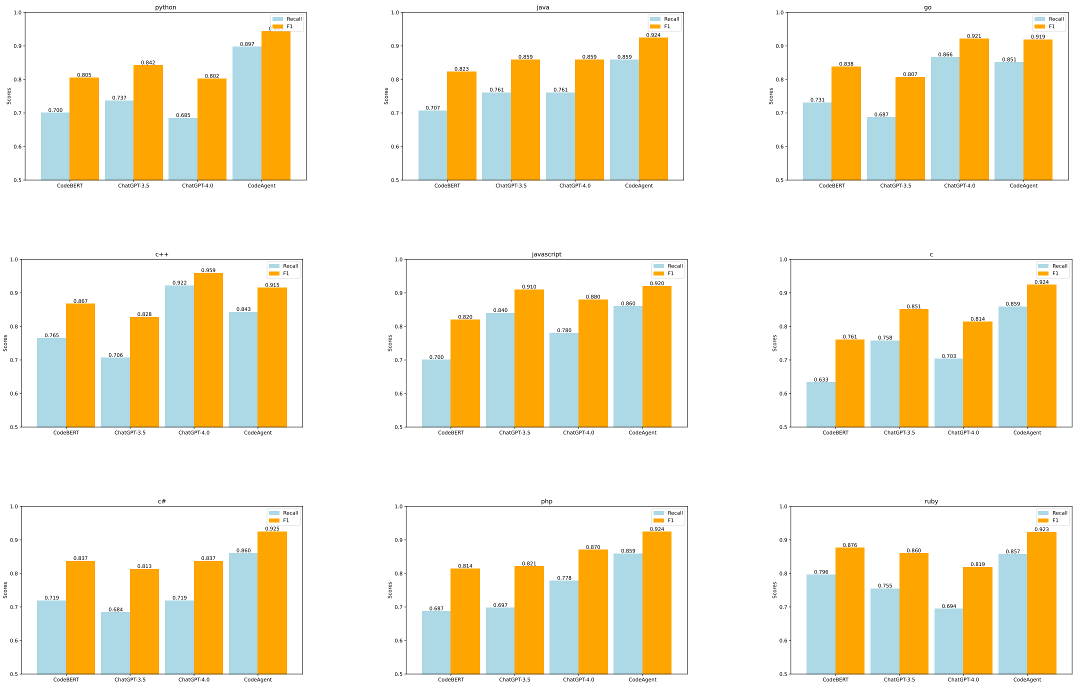


Figure 12: Comparison of models on the **closed** data across 9 languages on **FA task**.

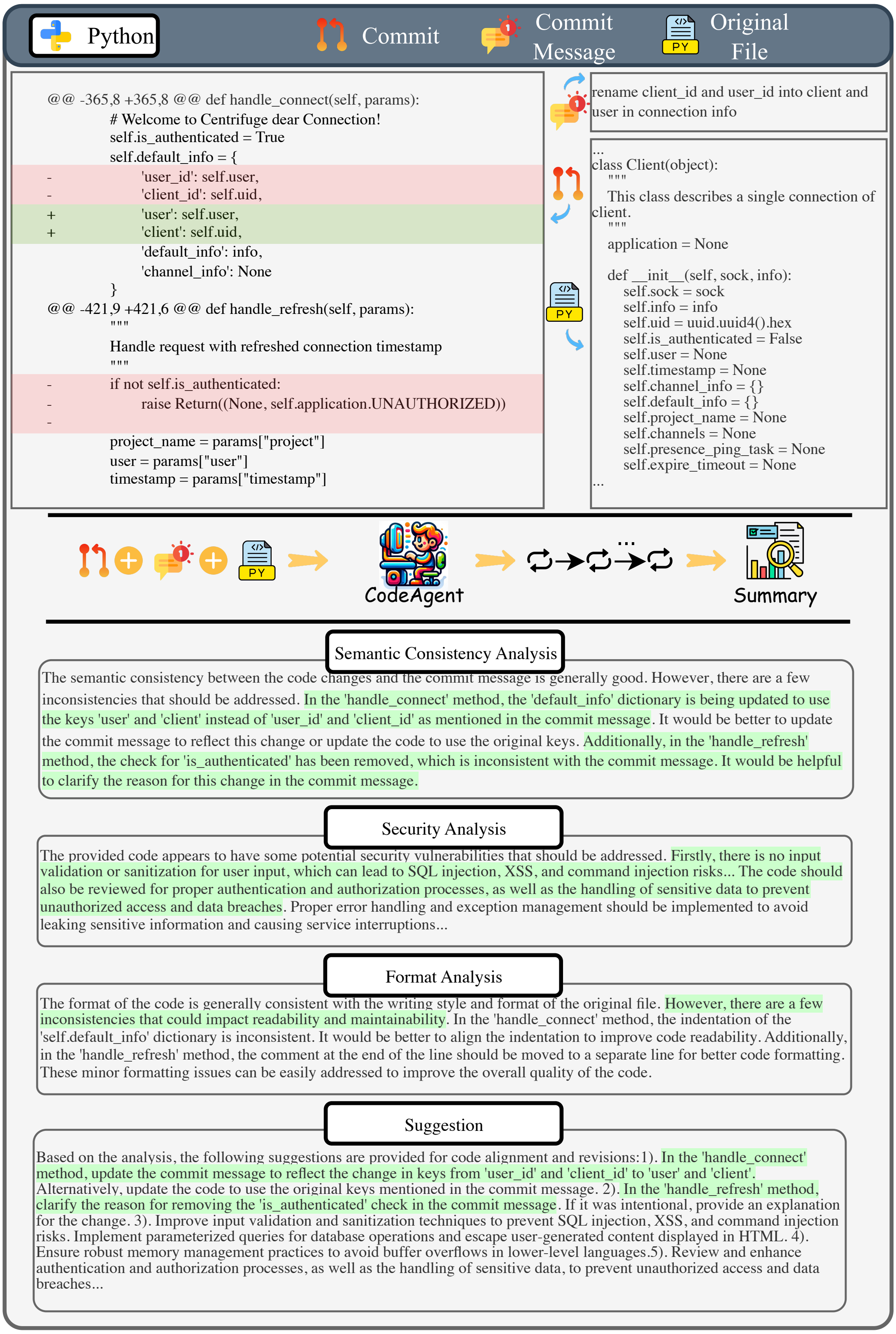
GPT-4.0) across 9 different languages. The performance mainly comes from the combination of COT and QA-Checker. Thus, we design an additional version called `CodeAgent` $_{w/o}$, which means `CodeAgent` without QA-Checker. Then, we use `CodeAgent` $_{w/o}$ to do vulnerability analysis and compare with `CodeAgent`. We first discuss about the result of `CodeAgent` $_{w/o}$ and then discuss about comparison between `CodeAgent` and `CodeAgent` $_{w/o}$.

Overview of Vulnerabilities in `CodeAgent` $_{w/o}$ Table 11 presents the findings of `CodeAgent` $_{w/o}$, a variant of the original `CodeAgent`, in identifying vulnerabilities across different programming languages. The table showcases the number of 'merged' and 'closed' vulnerabilities in languages such as Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. Notably, Python leads in the 'merged' category with a total of 1,057 cases, of which 140 are confirmed, yielding a Rate$_{merge}$ of 13.25%. In contrast, languages like Go and Ruby show lower vulnerability counts in both 'merged' and 'closed' categories. The table also includes Rate$_{close}$ and Rate$_{avg}$, providing insights into the effectiveness of vulnerability management across these languages.

Detailed Comparison between `CodeAgent` and `CodeAgent` $_{w/o}$ Comparing the findings in Table 11 with those in Table 9, we observe some notable differences in vulnerability detection by `CodeAgent` and `CodeAgent` $_{w/o}$. While the overall trend of higher 'merged' vulnerabilities in Python and lower counts in Go and Ruby remains consistent, Table 11 shows a slight reduction in the Rate$_{merge}$ for most languages, suggesting a more conservative confirmation approach in `CodeAgent` $_{w/o}$. Similarly, Rate$_{close}$ and Rate$_{avg}$ values in Table 11 generally indicate a lower proportion of confirmed vulnerabilities compared to Table 9, reflecting potentially different criteria or efficacy in vulnerability assessment. These variations highlight the impact of QA-Checker in `CodeAgent`.

N Cost statement

As shown in Table 12, `CodeAgent`-4 has a higher query time and cost compared to `CodeAgent`-3.5 due to its increased complexity and capabilities. We acknowledge that the integration of AI models and multi-agent systems may introduce complexity and require specialized knowledge for implementation.



```
@@ -365,8 +365,8 @@ def handle_connect(self, params):
        # Welcome to Centrifuge dear Connection!
        self.is_authenticated = True
        self.default_info = {
-            'user_id': self.user,
-            'client_id': self.uid,
+            'user': self.user,
+            'client': self.uid,
             'default_info': info,
             'channel_info': None
        }
@@ -421,9 +421,6 @@ def handle_refresh(self, params):
        """
        Handle request with refreshed connection timestamp
        """
-        if not self.is_authenticated:
-            raise Return((None, self.application.UNAUTHORIZED))
-
        project_name = params["project"]
        user = params["user"]
        timestamp = params["timestamp"]
```

rename client_id and user_id into client and user in connection info

```
...
class Client(object):
    """
    This class describes a single connection of client.
    """
    application = None

    def __init__(self, sock, info):
        self.sock = sock
        self.info = info
        self.uid = uuid.uuid4().hex
        self.is_authenticated = False
        self.user = None
        self.timestamp = None
        self.channel_info = {}
        self.default_info = {}
        self.project_name = None
        self.channels = None
        self.presence_ping_task = None
        self.expire_timeout = None
...
```

Semantic Consistency Analysis

The semantic consistency between the code changes and the commit message is generally good. However, there are a few inconsistencies that should be addressed. In the 'handle_connect' method, the 'default_info' dictionary is being updated to use the keys 'user' and 'client' instead of 'user_id' and 'client_id' as mentioned in the commit message. It would be better to update the commit message to reflect this change or update the code to use the original keys. Additionally, in the 'handle_refresh' method, the check for 'is_authenticated' has been removed, which is inconsistent with the commit message. It would be helpful to clarify the reason for this change in the commit message.

Security Analysis

The provided code appears to have some potential security vulnerabilities that should be addressed. Firstly, there is no input validation or sanitization for user input, which can lead to SQL injection, XSS, and command injection risks... The code should also be reviewed for proper authentication and authorization processes, as well as the handling of sensitive data to prevent unauthorized access and data breaches. Proper error handling and exception management should be implemented to avoid leaking sensitive information and causing service interruptions...

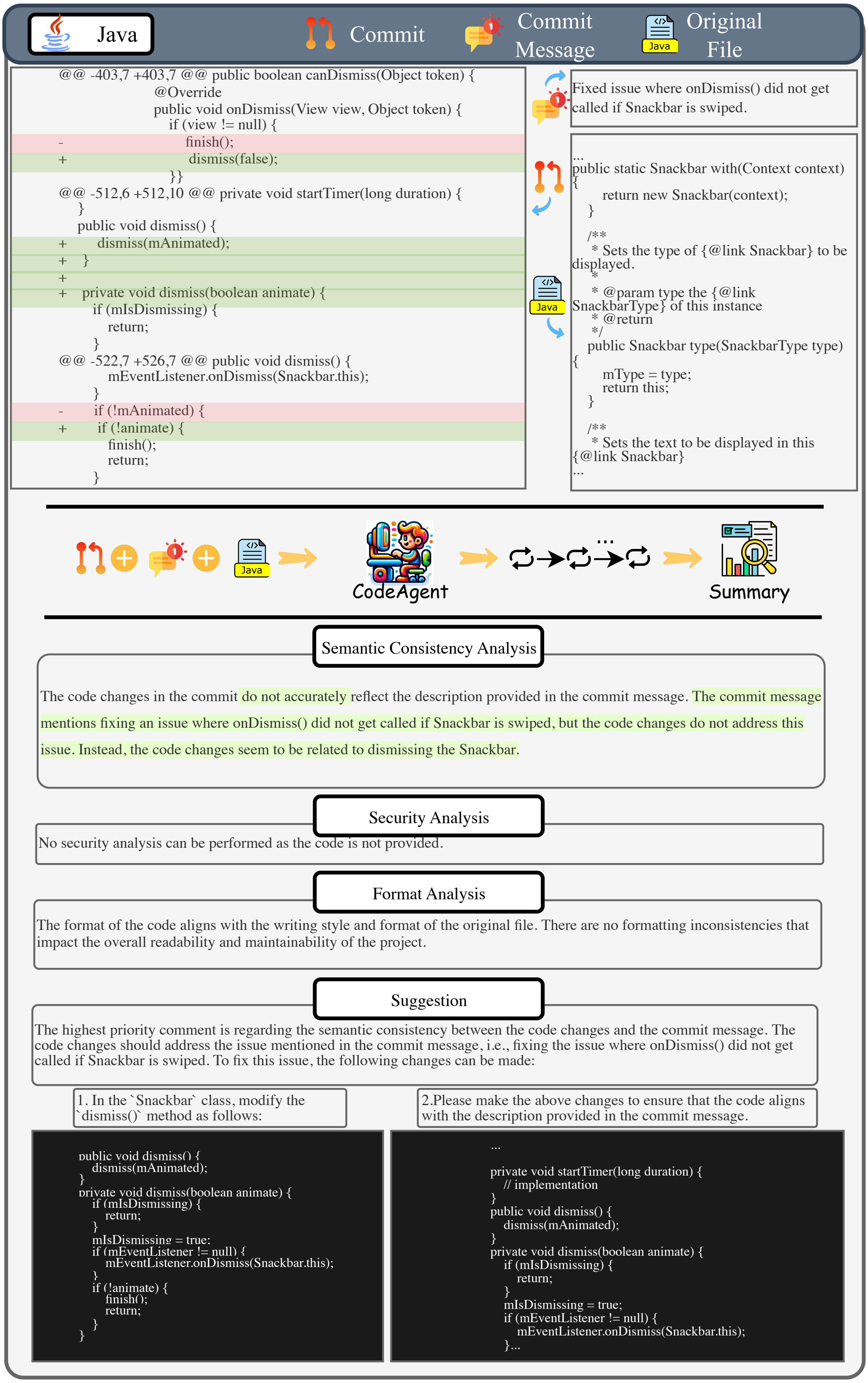
Format Analysis

The format of the code is generally consistent with the writing style and format of the original file. However, there are a few inconsistencies that could impact readability and maintainability. In the 'handle_connect' method, the indentation of the 'self.default_info' dictionary is inconsistent. It would be better to align the indentation to improve code readability. Additionally, in the 'handle_refresh' method, the comment at the end of the line should be moved to a separate line for better code formatting. These minor formatting issues can be easily addressed to improve the overall quality of the code.

Suggestion

Based on the analysis, the following suggestions are provided for code alignment and revisions:1). In the 'handle_connect' method, update the commit message to reflect the change in keys from 'user_id' and 'client_id' to 'user' and 'client'. Alternatively, update the code to use the original keys mentioned in the commit message. 2). In the 'handle_refresh' method, clarify the reason for removing the 'is_authenticated' check in the commit message. If it was intentional, provide an explanation for the change. 3). Improve input validation and sanitization techniques to prevent SQL injection, XSS, and command injection risks. Implement parameterized queries for database operations and escape user-generated content displayed in HTML. 4). Ensure robust memory management practices to avoid buffer overflows in lower-level languages.5). Review and enhance authentication and authorization processes, as well as the handling of sensitive data, to prevent unauthorized access and data breaches...

Figure 13: Example in Python project with sha value: “0d3b94bbd7528ed8649cdab4370a62e508bad82c”



```
@@ -403,7 +403,7 @@ public boolean canDismiss(Object token) {
                    @Override
                    public void onDismiss(View view, Object token) {
                        if (view != null) {
-                           finish();
+                           dismiss(false);
                        }}
@@ -512,6 +512,10 @@ private void startTimer(long duration) {
     }
     public void dismiss() {
+        dismiss(mAnimated);
+    }
+
+    private void dismiss(boolean animate) {
         if (mIsDismissing) {
             return;
         }
@@ -522,7 +526,7 @@ public void dismiss() {
             mEventListener.onDismiss(Snackbar.this);
         }
-        if (!mAnimated) {
+        if (!animate) {
             finish();
             return;
         }
```

Fixed issue where onDismiss() did not get called if Snackbar is swiped.

```
...
public static Snackbar with(Context context)
{
        return new Snackbar(context);
    }

    /**
     * Sets the type of {@link Snackbar} to be
displayed.
     *
     * @param type the {@link
SnackbarType} of this instance
     * @return
     */
    public Snackbar type(SnackbarType type)
{
        mType = type;
        return this;
    }

    /**
     * Sets the text to be displayed in this
{@link Snackbar}
...
```

Semantic Consistency Analysis

The code changes in the commit do not accurately reflect the description provided in the commit message. The commit message mentions fixing an issue where onDismiss() did not get called if Snackbar is swiped, but the code changes do not address this issue. Instead, the code changes seem to be related to dismissing the Snackbar.

Security Analysis

No security analysis can be performed as the code is not provided.

Format Analysis

The format of the code aligns with the writing style and format of the original file. There are no formatting inconsistencies that impact the overall readability and maintainability of the project.

Suggestion

The highest priority comment is regarding the semantic consistency between the code changes and the commit message. The code changes should address the issue mentioned in the commit message, i.e., fixing the issue where onDismiss() did not get called if Snackbar is swiped. To fix this issue, the following changes can be made:

1. In the `Snackbar` class, modify the `dismiss()` method as follows:

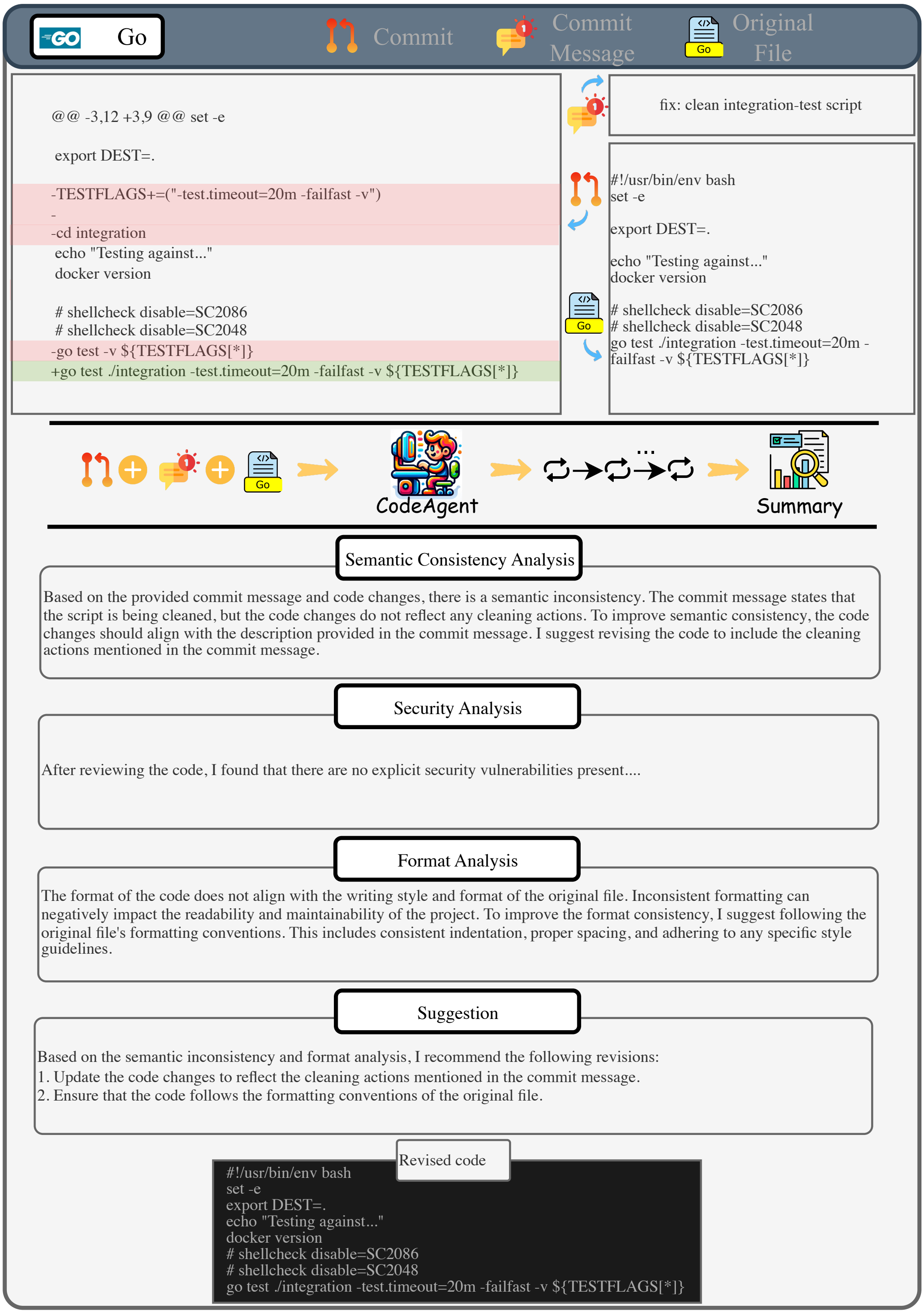
```
public void dismiss() {
    dismiss(mAnimated);
}
private void dismiss(boolean animate) {
    if (mIsDismissing) {
        return;
    }
    mIsDismissing = true;
    if (mEventListener != null) {
        mEventListener.onDismiss(Snackbar.this);
    }
    if (!animate) {
        finish();
        return;
    }
}
```

2.Please make the above changes to ensure that the code aligns with the description provided in the commit message.

```
...

private void startTimer(long duration) {
    // implementation
}
public void dismiss() {
    dismiss(mAnimated);
}
private void dismiss(boolean animate) {
    if (mIsDismissing) {
        return;
    }
    mIsDismissing = true;
    if (mEventListener != null) {
        mEventListener.onDismiss(Snackbar.this);
    }...
```

Figure 14: Example in Java project with sha value: “6e3c6d17d943f5ac70b421653eb167e0c34b119f"



```
@@ -3,12 +3,9 @@ set -e

 export DEST=.

-TESTFLAGS+=("-test.timeout=20m -failfast -v")
-
-cd integration
 echo "Testing against..."
 docker version

 # shellcheck disable=SC2086
 # shellcheck disable=SC2048
-go test -v ${TESTFLAGS[*]}
+go test ./integration -test.timeout=20m -failfast -v ${TESTFLAGS[*]}
```

fix: clean integration-test script

```
#!/usr/bin/env bash
set -e

export DEST=.

echo "Testing against..."
docker version

# shellcheck disable=SC2086
# shellcheck disable=SC2048
go test ./integration -test.timeout=20m -
failfast -v ${TESTFLAGS[*]}
```

Based on the provided commit message and code changes, there is a semantic inconsistency. The commit message states that the script is being cleaned, but the code changes do not reflect any cleaning actions. To improve semantic consistency, the code changes should align with the description provided in the commit message. I suggest revising the code to include the cleaning actions mentioned in the commit message.

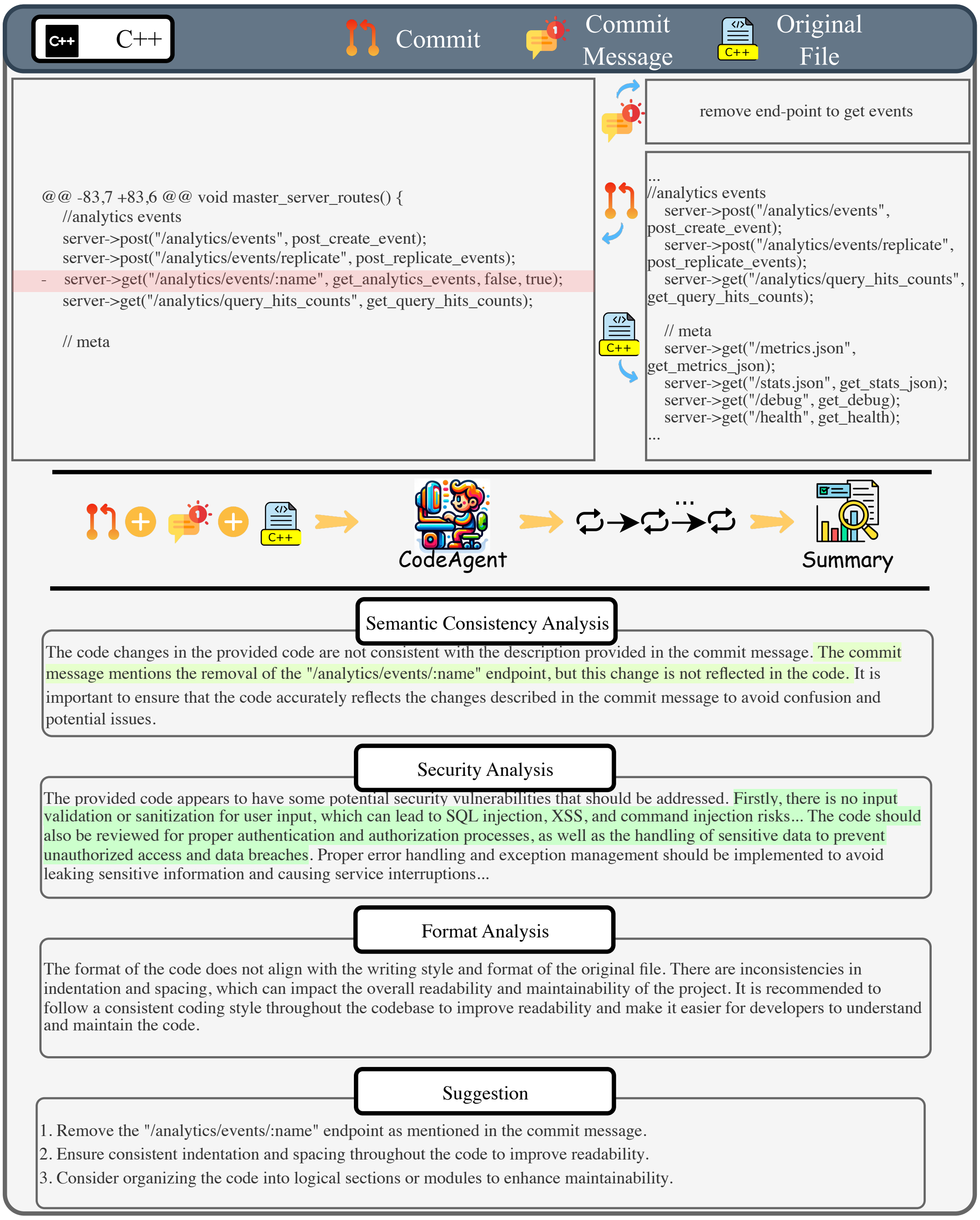
After reviewing the code, I found that there are no explicit security vulnerabilities present....

The format of the code does not align with the writing style and format of the original file. Inconsistent formatting can negatively impact the readability and maintainability of the project. To improve the format consistency, I suggest following the original file's formatting conventions. This includes consistent indentation, proper spacing, and adhering to any specific style guidelines.

Based on the semantic inconsistency and format analysis, I recommend the following revisions:
1. Update the code changes to reflect the cleaning actions mentioned in the commit message.
2. Ensure that the code follows the formatting conventions of the original file.

```
#!/usr/bin/env bash
set -e
export DEST=.
echo "Testing against..."
docker version
# shellcheck disable=SC2086
# shellcheck disable=SC2048
go test ./integration -test.timeout=20m -failfast -v ${TESTFLAGS[*]}
```

Figure 15: Example in Go project with sha value: “a6b00b097b1a8140bd8b60177247dc830374c8ed"



remove end-point to get events

```
@@ -83,7 +83,6 @@ void master_server_routes() {
    //analytics events
    server->post("/analytics/events", post_create_event);
    server->post("/analytics/events/replicate", post_replicate_events);
-   server->get("/analytics/events/:name", get_analytics_events, false, true);
    server->get("/analytics/query_hits_counts", get_query_hits_counts);

    // meta
```

```
...
//analytics events
   server->post("/analytics/events",
post_create_event);
   server->post("/analytics/events/replicate",
post_replicate_events);
   server->get("/analytics/query_hits_counts",
get_query_hits_counts);

   // meta
   server->get("/metrics.json",
get_metrics_json);
   server->get("/stats.json", get_stats_json);
   server->get("/debug", get_debug);
   server->get("/health", get_health);
...
```

Semantic Consistency Analysis

The code changes in the provided code are not consistent with the description provided in the commit message. The commit message mentions the removal of the "/analytics/events/:name" endpoint, but this change is not reflected in the code. It is important to ensure that the code accurately reflects the changes described in the commit message to avoid confusion and potential issues.

Security Analysis

The provided code appears to have some potential security vulnerabilities that should be addressed. Firstly, there is no input validation or sanitization for user input, which can lead to SQL injection, XSS, and command injection risks... The code should also be reviewed for proper authentication and authorization processes, as well as the handling of sensitive data to prevent unauthorized access and data breaches. Proper error handling and exception management should be implemented to avoid leaking sensitive information and causing service interruptions...

Format Analysis

The format of the code does not align with the writing style and format of the original file. There are inconsistencies in indentation and spacing, which can impact the overall readability and maintainability of the project. It is recommended to follow a consistent coding style throughout the codebase to improve readability and make it easier for developers to understand and maintain the code.

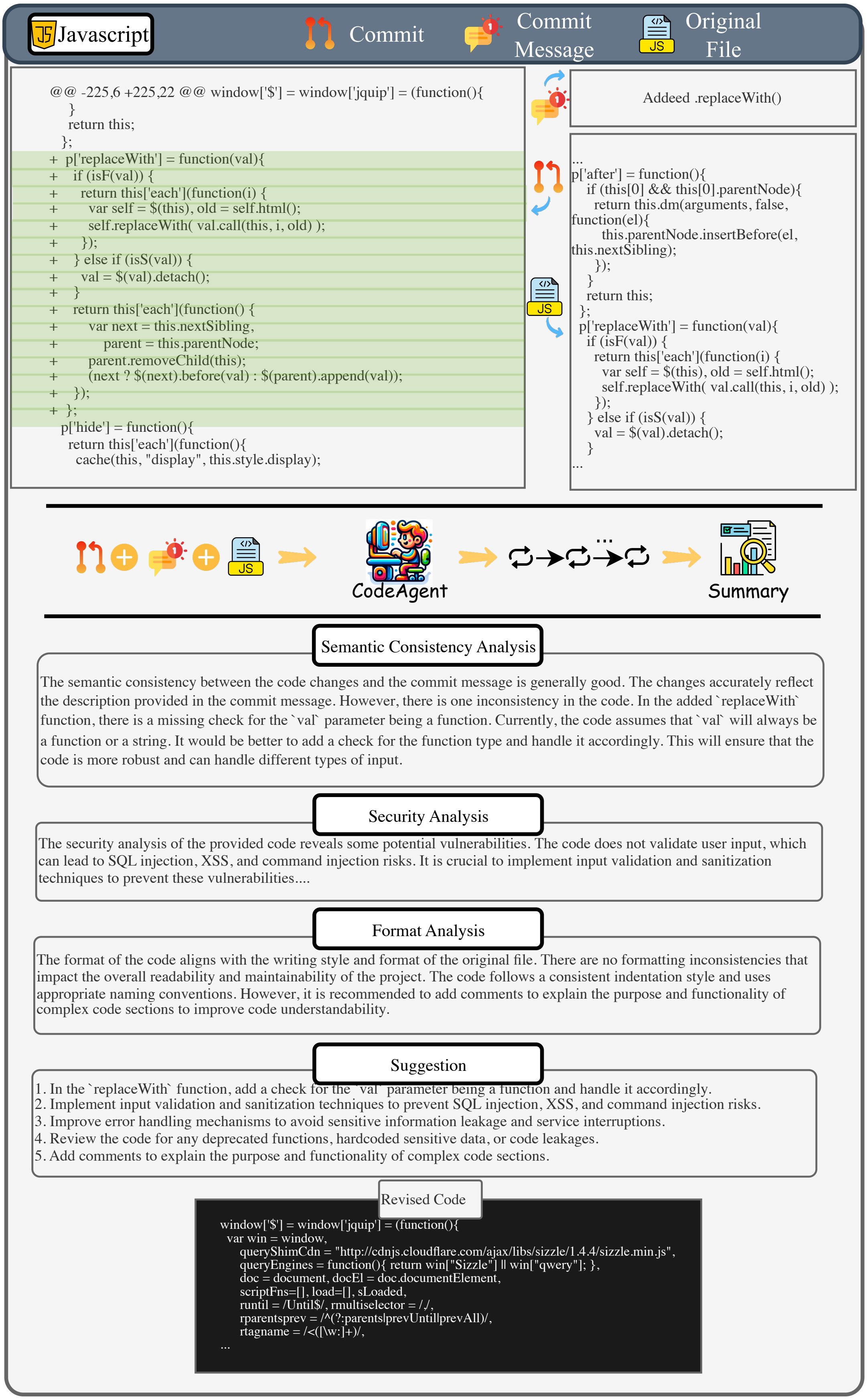
Suggestion

1. Remove the "/analytics/events/:name" endpoint as mentioned in the commit message.
2. Ensure consistent indentation and spacing throughout the code to improve readability.
3. Consider organizing the code into logical sections or modules to enhance maintainability.

Figure 16: Example in C++ project with sha value: “09e88c7c20347cdf513323fa8c616aba0a1b9ea7”

Table 11: Vulnerable problems (#) found by `CodeAgent` $_{w/o}$

`CodeAgent`	Python	Java	Go	C++	JavaScript	C	C#	PHP	Ruby
merged (total#)	1,057	287	133	138	280	114	206	173	202
merged (confirmed#)	140	17	10	12	28	9	21	28	17
Rate$_{merge}$	13.25%	5.92%	7.52%	8.70%	10.00%	7.89%	10.19%	16.18%	8.42%
closed (total#)	248	97	74	56	112	146	62	105	55
closed (confirmed#)	36	9	5	12	16	26	7	15	5
Rate$_{close}$	14.52%	9.28%	6.76%	21.43%	14.29%	17.81%	11.29%	14.29%	9.09%
Total number (#)	1,305	384	207	194	392	260	268	278	257
Total confirmed (#)	176	26	15	24	44	35	28	43	22
Rate$_{avg}$	13.49%	6.77%	7.25%	12.37%	11.22%	13.46%	10.45%	15.47%	8.56%



```
@@ -225,6 +225,22 @@ window['$'] = window['jquip'] = (function(){
    }
    return this;
  };
+  p['replaceWith'] = function(val){
+    if (isF(val)) {
+      return this['each'](function(i) {
+        var self = $(this), old = self.html();
+        self.replaceWith( val.call(this, i, old) );
+      });
+    } else if (isS(val)) {
+      val = $(val).detach();
+    }
+    return this['each'](function() {
+        var next = this.nextSibling,
+            parent = this.parentNode;
+        parent.removeChild(this);
+        (next ? $(next).before(val) : $(parent).append(val));
+    });
+  };
  p['hide'] = function(){
    return this['each'](function(){
      cache(this, "display", this.style.display);
```

Addeed .replaceWith()

```
...
p['after'] = function(){
    if (this[0] && this[0].parentNode){
      return this.dm(arguments, false,
function(el){
        this.parentNode.insertBefore(el,
this.nextSibling);
      });
    }
    return this;
  };
  p['replaceWith'] = function(val){
    if (isF(val)) {
      return this['each'](function(i) {
        var self = $(this), old = self.html();
        self.replaceWith( val.call(this, i, old) );
      });
    } else if (isS(val)) {
      val = $(val).detach();
    }
...
```

Semantic Consistency Analysis

The semantic consistency between the code changes and the commit message is generally good. The changes accurately reflect the description provided in the commit message. However, there is one inconsistency in the code. In the added `replaceWith` function, there is a missing check for the `val` parameter being a function. Currently, the code assumes that `val` will always be a function or a string. It would be better to add a check for the function type and handle it accordingly. This will ensure that the code is more robust and can handle different types of input.

Security Analysis

The security analysis of the provided code reveals some potential vulnerabilities. The code does not validate user input, which can lead to SQL injection, XSS, and command injection risks. It is crucial to implement input validation and sanitization techniques to prevent these vulnerabilities....

Format Analysis

The format of the code aligns with the writing style and format of the original file. There are no formatting inconsistencies that impact the overall readability and maintainability of the project. The code follows a consistent indentation style and uses appropriate naming conventions. However, it is recommended to add comments to explain the purpose and functionality of complex code sections to improve code understandability.

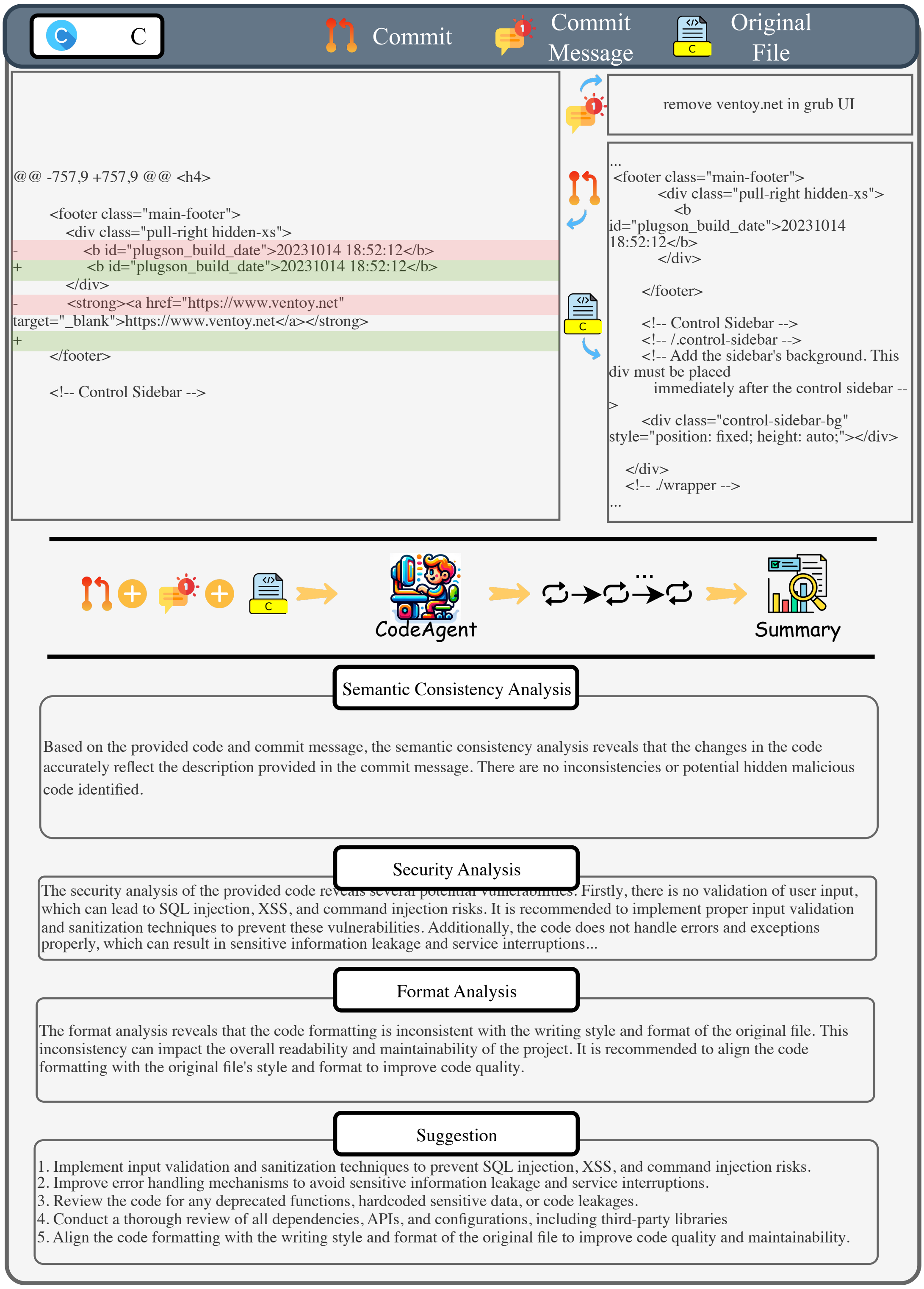
Suggestion

1. In the `replaceWith` function, add a check for the `val` parameter being a function and handle it accordingly.
2. Implement input validation and sanitization techniques to prevent SQL injection, XSS, and command injection risks.
3. Improve error handling mechanisms to avoid sensitive information leakage and service interruptions.
4. Review the code for any deprecated functions, hardcoded sensitive data, or code leakages.
5. Add comments to explain the purpose and functionality of complex code sections.

Revised Code

```
window['$'] = window['jquip'] = (function(){
  var win = window,
      queryShimCdn = "http://cdnjs.cloudflare.com/ajax/libs/sizzle/1.4.4/sizzle.min.js",
      queryEngines = function(){ return win["Sizzle"] || win["qwery"]; },
      doc = document, docEl = doc.documentElement,
      scriptFns=[], load=[], sLoaded,
      runtil = /Until$/, rmultiselector = /,/,
      rparentsprev = /^(?:parents|prevUntil|prevAll)/,
      rtagname = /<([\w:]+)/,
...
```

Figure 17: Example in JavaScript project with sha value: “6e9a80d93b470aef4978677ed0b7c9996fddbe20"



remove ventoy.net in grub UI

```
@@ -757,9 +757,9 @@ <h4>

        <footer class="main-footer">
            <div class="pull-right hidden-xs">
-               <b id="plugson_build_date">20231014 18:52:12</b>
+                <b id="plugson_build_date">20231014 18:52:12</b>
            </div>
-           <strong><a href="https://www.ventoy.net"
target="_blank">https://www.ventoy.net</a></strong>
+
        </footer>

        <!-- Control Sidebar -->
```

```
...
 <footer class="main-footer">
          <div class="pull-right hidden-xs">
             <b
id="plugson_build_date">20231014
18:52:12</b>
          </div>

      </footer>

      <!-- Control Sidebar -->
      <!-- /.control-sidebar -->
      <!-- Add the sidebar's background. This
div must be placed
         immediately after the control sidebar --
>
      <div class="control-sidebar-bg"
style="position: fixed; height: auto;"></div>

   </div>
   <!-- ./wrapper -->
...
```

Semantic Consistency Analysis

Based on the provided code and commit message, the semantic consistency analysis reveals that the changes in the code accurately reflect the description provided in the commit message. There are no inconsistencies or potential hidden malicious code identified.

Security Analysis

The security analysis of the provided code reveals several potential vulnerabilities. Firstly, there is no validation of user input, which can lead to SQL injection, XSS, and command injection risks. It is recommended to implement proper input validation and sanitization techniques to prevent these vulnerabilities. Additionally, the code does not handle errors and exceptions properly, which can result in sensitive information leakage and service interruptions...

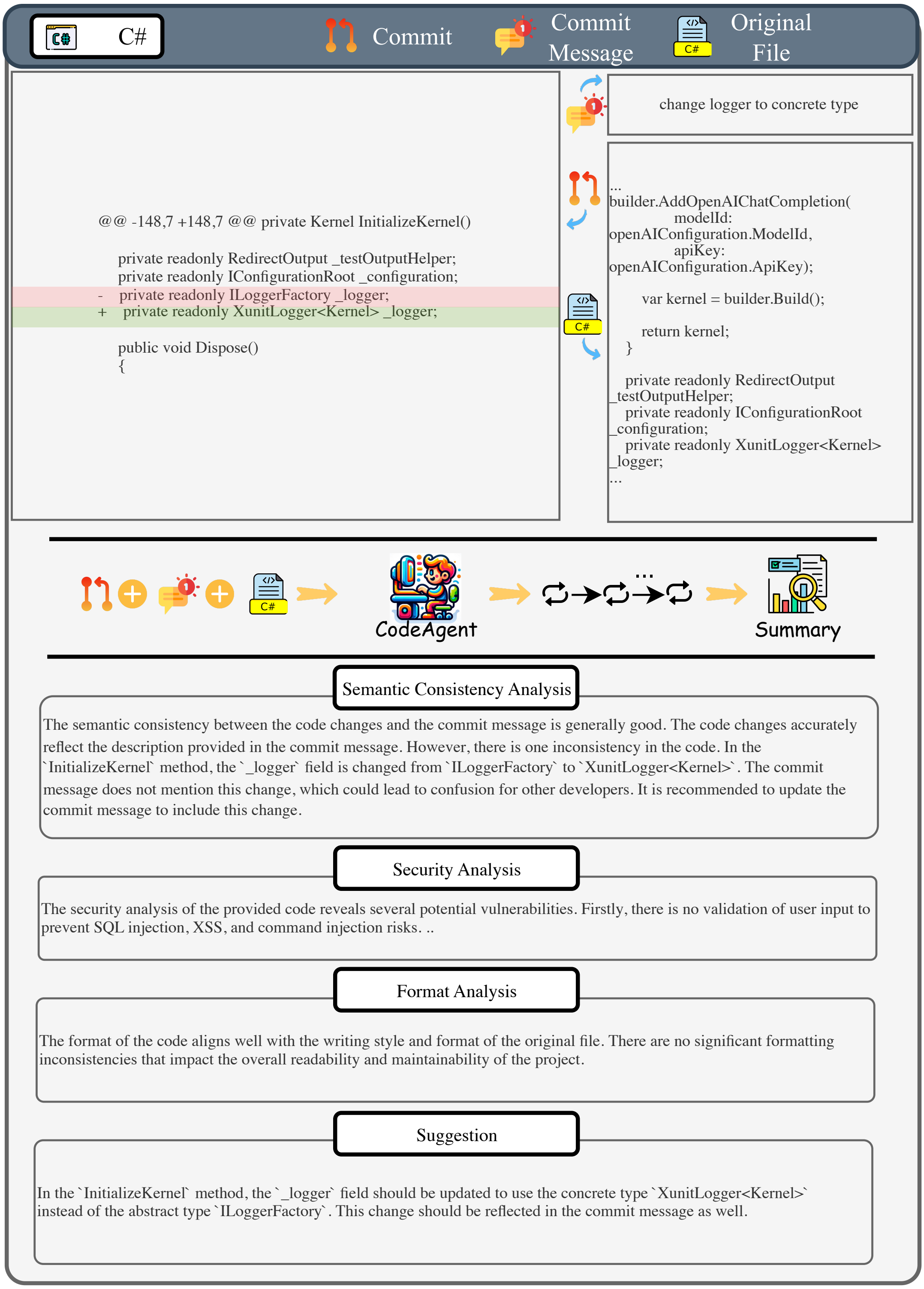
Format Analysis

The format analysis reveals that the code formatting is inconsistent with the writing style and format of the original file. This inconsistency can impact the overall readability and maintainability of the project. It is recommended to align the code formatting with the original file's style and format to improve code quality.

Suggestion

1. Implement input validation and sanitization techniques to prevent SQL injection, XSS, and command injection risks.
2. Improve error handling mechanisms to avoid sensitive information leakage and service interruptions.
3. Review the code for any deprecated functions, hardcoded sensitive data, or code leakages.
4. Conduct a thorough review of all dependencies, APIs, and configurations, including third-party libraries
5. Align the code formatting with the writing style and format of the original file to improve code quality and maintainability.

Figure 18: Example in C project with sha value: “5f93cf43dd6f809d7927abb79884285ad77e8a58"



```
@@ -148,7 +148,7 @@ private Kernel InitializeKernel()

    private readonly RedirectOutput _testOutputHelper;
    private readonly IConfigurationRoot _configuration;
-   private readonly ILoggerFactory _logger;
+   private readonly XunitLogger<Kernel> _logger;

    public void Dispose()
    {
```

change logger to concrete type

```
...
builder.AddOpenAIChatCompletion(
        modelId:
openAIConfiguration.ModelId,
        apiKey:
openAIConfiguration.ApiKey);

    var kernel = builder.Build();

    return kernel;
  }

  private readonly RedirectOutput
_testOutputHelper;
  private readonly IConfigurationRoot
_configuration;
  private readonly XunitLogger<Kernel>
_logger;
...
```

Semantic Consistency Analysis

The semantic consistency between the code changes and the commit message is generally good. The code changes accurately reflect the description provided in the commit message. However, there is one inconsistency in the code. In the `InitializeKernel` method, the `_logger` field is changed from `ILoggerFactory` to `XunitLogger<Kernel>`. The commit message does not mention this change, which could lead to confusion for other developers. It is recommended to update the commit message to include this change.

Security Analysis

The security analysis of the provided code reveals several potential vulnerabilities. Firstly, there is no validation of user input to prevent SQL injection, XSS, and command injection risks. ..

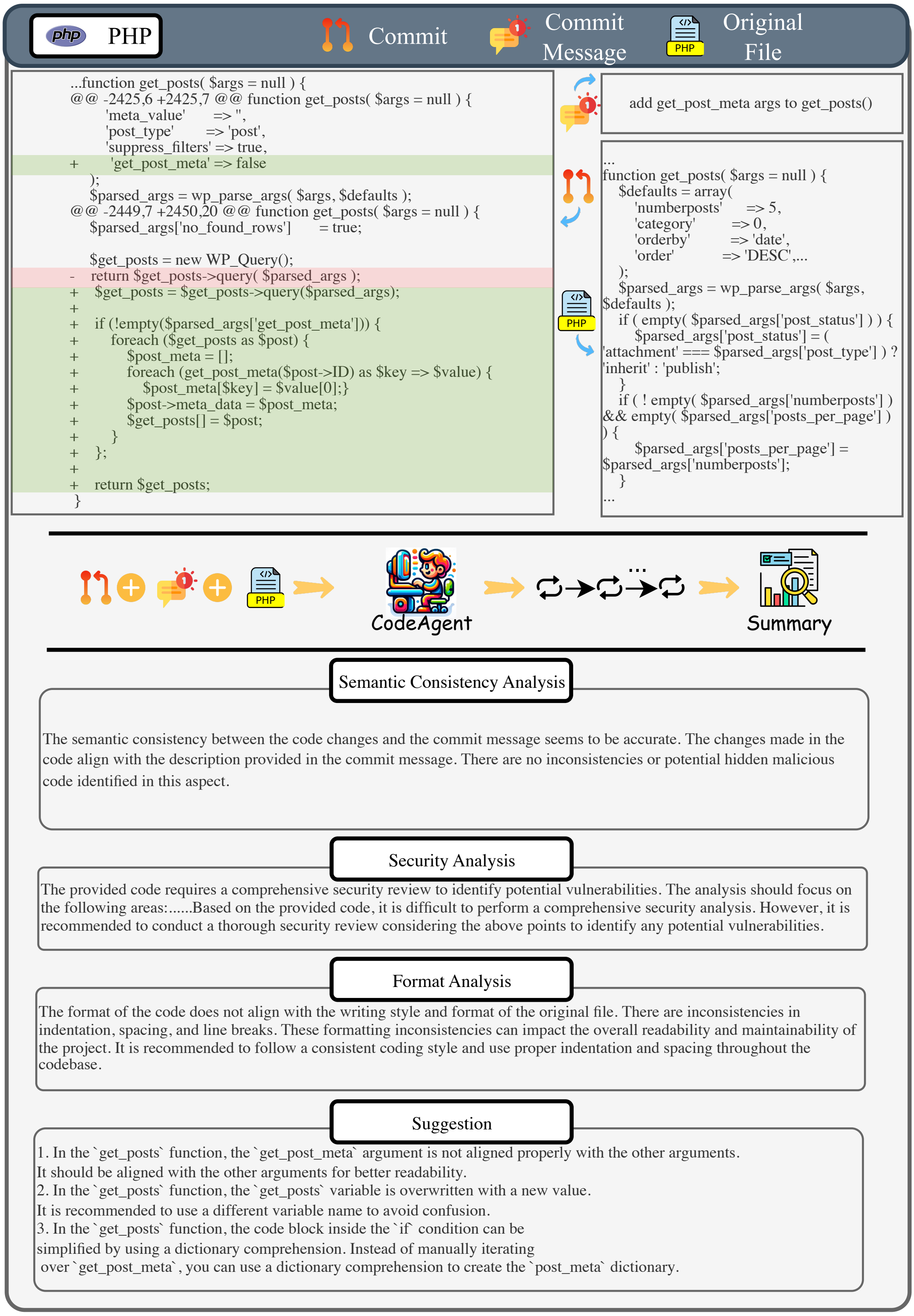
Format Analysis

The format of the code aligns well with the writing style and format of the original file. There are no significant formatting inconsistencies that impact the overall readability and maintainability of the project.

Suggestion

In the `InitializeKernel` method, the `_logger` field should be updated to use the concrete type `XunitLogger<Kernel>` instead of the abstract type `ILoggerFactory`. This change should be reflected in the commit message as well.

Figure 19: Example in C# project with sha value: “0e231c7a81b318e9eade972f7b877e66128ed67d"



```
...function get_posts( $args = null ) {
@@ -2425,6 +2425,7 @@ function get_posts( $args = null ) {
        'meta_value'      => '',
        'post_type'       => 'post',
        'suppress_filters' => true,
+       'get_post_meta' => false
    );
    $parsed_args = wp_parse_args( $args, $defaults );
@@ -2449,7 +2450,20 @@ function get_posts( $args = null ) {
    $parsed_args['no_found_rows']      = true;

    $get_posts = new WP_Query();
-   return $get_posts->query( $parsed_args );
+   $get_posts = $get_posts->query($parsed_args);
+
+   if (!empty($parsed_args['get_post_meta'])) {
+       foreach ($get_posts as $post) {
+           $post_meta = [];
+           foreach (get_post_meta($post->ID) as $key => $value) {
+               $post_meta[$key] = $value[0];}
+           $post->meta_data = $post_meta;
+           $get_posts[] = $post;
+       }
+   };
+
+   return $get_posts;
 }
```

add get_post_meta args to get_posts()

```
...
function get_posts( $args = null ) {
    $defaults = array(
        'numberposts'      => 5,
        'category'        => 0,
        'orderby'         => 'date',
        'order'           => 'DESC',...
    );
    $parsed_args = wp_parse_args( $args,
$defaults );
    if ( empty( $parsed_args['post_status'] ) ) {
        $parsed_args['post_status'] = (
'attachment' === $parsed_args['post_type'] ) ?
'inherit' : 'publish';
    }
    if ( ! empty( $parsed_args['numberposts'] )
&& empty( $parsed_args['posts_per_page'] )
) {
        $parsed_args['posts_per_page'] =
$parsed_args['numberposts'];
    }
...
```

Semantic Consistency Analysis

The semantic consistency between the code changes and the commit message seems to be accurate. The changes made in the code align with the description provided in the commit message. There are no inconsistencies or potential hidden malicious code identified in this aspect.

Security Analysis

The provided code requires a comprehensive security review to identify potential vulnerabilities. The analysis should focus on the following areas:......Based on the provided code, it is difficult to perform a comprehensive security analysis. However, it is recommended to conduct a thorough security review considering the above points to identify any potential vulnerabilities.

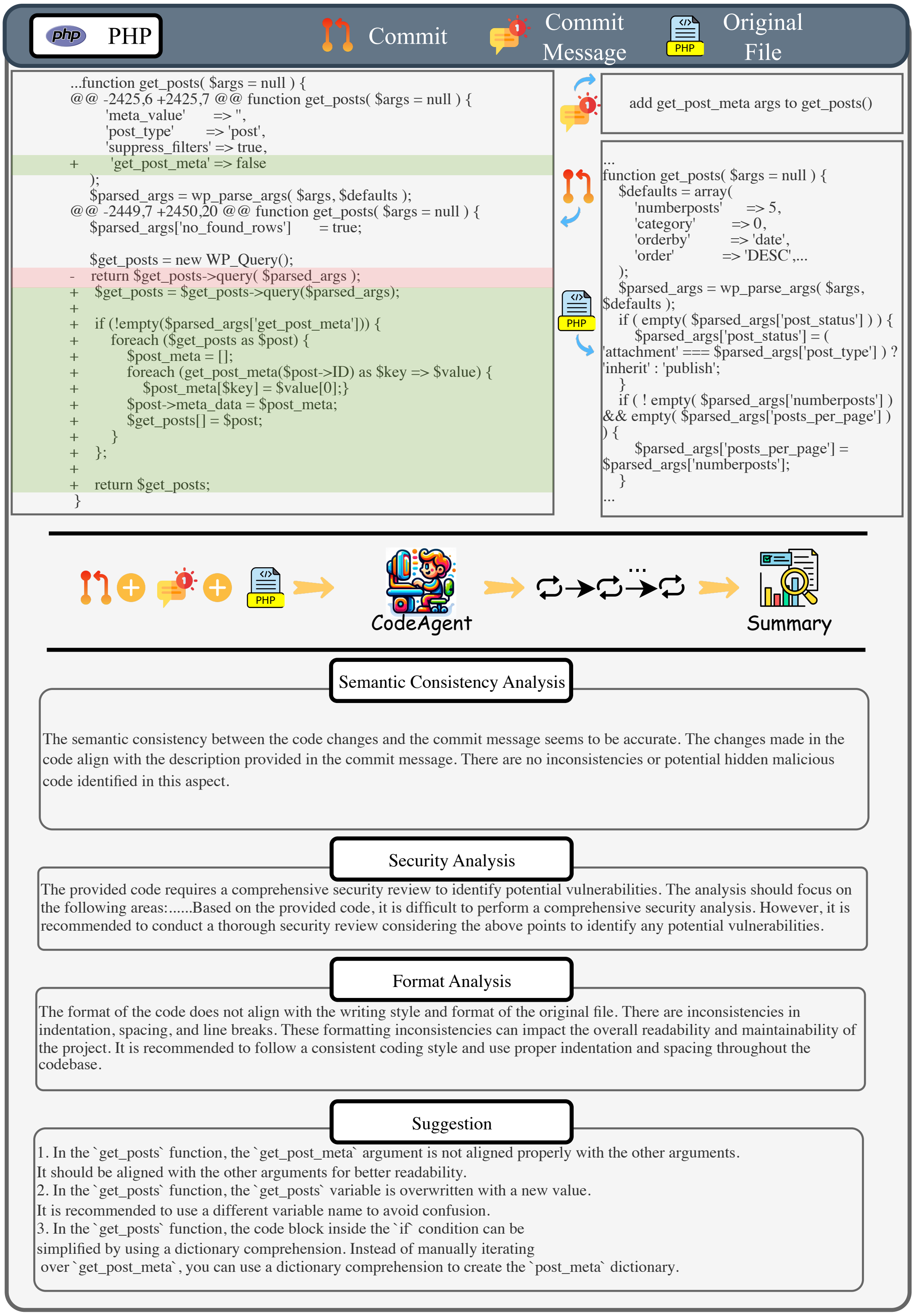
Format Analysis

The format of the code does not align with the writing style and format of the original file. There are inconsistencies in indentation, spacing, and line breaks. These formatting inconsistencies can impact the overall readability and maintainability of the project. It is recommended to follow a consistent coding style and use proper indentation and spacing throughout the codebase.

Suggestion

1. In the `get_posts` function, the `get_post_meta` argument is not aligned properly with the other arguments.
It should be aligned with the other arguments for better readability.
2. In the `get_posts` function, the `get_posts` variable is overwritten with a new value.
It is recommended to use a different variable name to avoid confusion.
3. In the `get_posts` function, the code block inside the `if` condition can be
simplified by using a dictionary comprehension. Instead of manually iterating
over `get_post_meta`, you can use a dictionary comprehension to create the `post_meta` dictionary.

Figure 20: Example in PHP project with sha value: “6679f059b9a0972a90df198471188da652f5c547"



```
...function get_posts( $args = null ) {
@@ -2425,6 +2425,7 @@ function get_posts( $args = null ) {
        'meta_value'      => '',
        'post_type'       => 'post',
        'suppress_filters' => true,
+       'get_post_meta' => false
    );
    $parsed_args = wp_parse_args( $args, $defaults );
@@ -2449,7 +2450,20 @@ function get_posts( $args = null ) {
    $parsed_args['no_found_rows']     = true;

    $get_posts = new WP_Query();
-   return $get_posts->query( $parsed_args );
+   $get_posts = $get_posts->query($parsed_args);
+
+   if (!empty($parsed_args['get_post_meta'])) {
+       foreach ($get_posts as $post) {
+           $post_meta = [];
+           foreach (get_post_meta($post->ID) as $key => $value) {
+               $post_meta[$key] = $value[0];}
+           $post->meta_data = $post_meta;
+           $get_posts[] = $post;
+       }
+   };
+
+   return $get_posts;
 }
```

add get_post_meta args to get_posts()

```
...
function get_posts( $args = null ) {
    $defaults = array(
        'numberposts'     => 5,
        'category'        => 0,
        'orderby'         => 'date',
        'order'           => 'DESC',...
    );
    $parsed_args = wp_parse_args( $args,
$defaults );
    if ( empty( $parsed_args['post_status'] ) ) {
        $parsed_args['post_status'] = (
'attachment' === $parsed_args['post_type'] ) ?
'inherit' : 'publish';
    }
    if ( ! empty( $parsed_args['numberposts'] )
&& empty( $parsed_args['posts_per_page'] )
) {
        $parsed_args['posts_per_page'] =
$parsed_args['numberposts'];
    }
...
```

Semantic Consistency Analysis

The semantic consistency between the code changes and the commit message seems to be accurate. The changes made in the code align with the description provided in the commit message. There are no inconsistencies or potential hidden malicious code identified in this aspect.

Security Analysis

The provided code requires a comprehensive security review to identify potential vulnerabilities. The analysis should focus on the following areas:......Based on the provided code, it is difficult to perform a comprehensive security analysis. However, it is recommended to conduct a thorough security review considering the above points to identify any potential vulnerabilities.

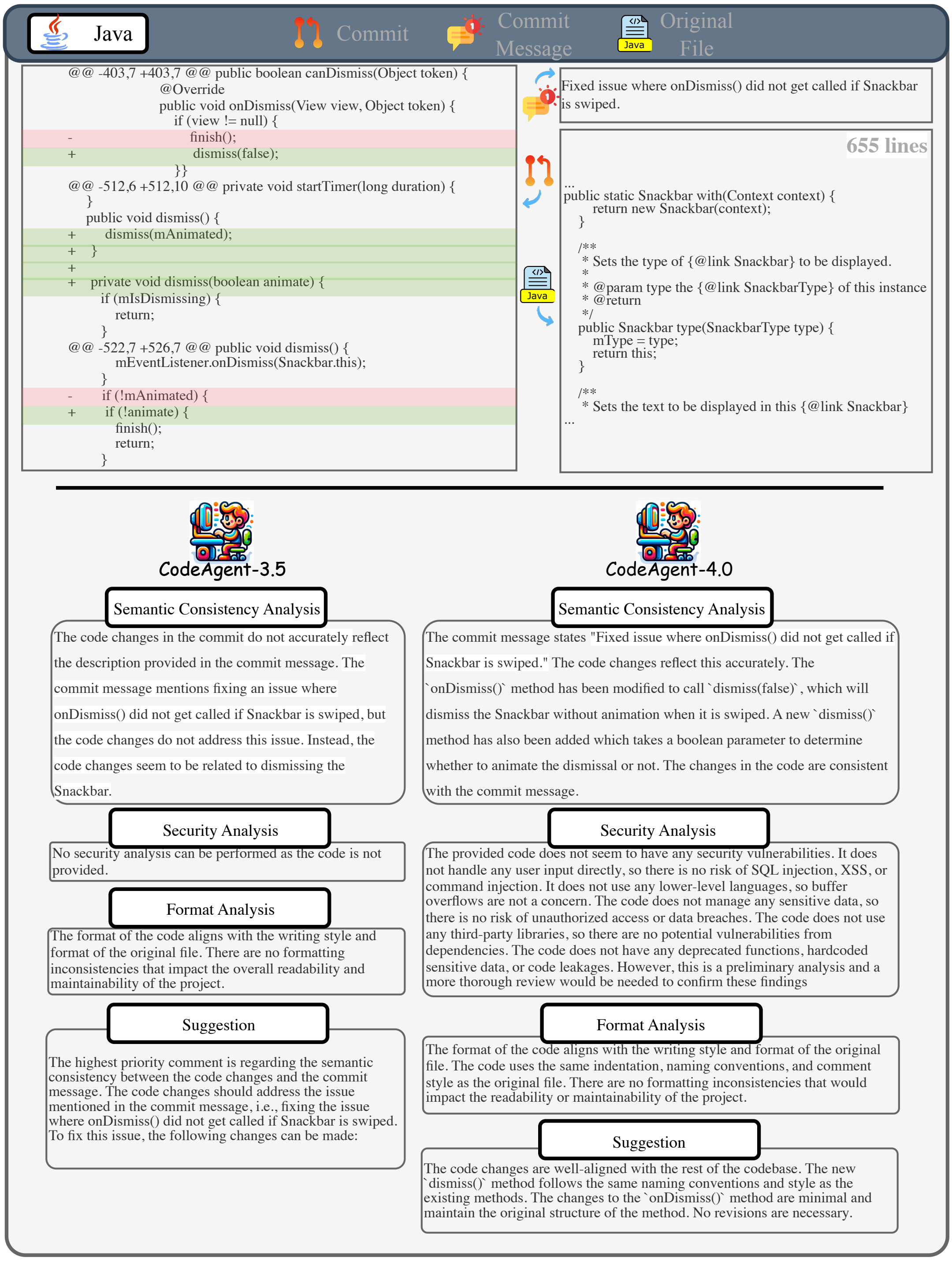
Format Analysis

The format of the code does not align with the writing style and format of the original file. There are inconsistencies in indentation, spacing, and line breaks. These formatting inconsistencies can impact the overall readability and maintainability of the project. It is recommended to follow a consistent coding style and use proper indentation and spacing throughout the codebase.

Suggestion

1. In the `get_posts` function, the `get_post_meta` argument is not aligned properly with the other arguments.
It should be aligned with the other arguments for better readability.
2. In the `get_posts` function, the `get_posts` variable is overwritten with a new value.
It is recommended to use a different variable name to avoid confusion.
3. In the `get_posts` function, the code block inside the `if` condition can be
simplified by using a dictionary comprehension. Instead of manually iterating
over `get_post_meta`, you can use a dictionary comprehension to create the `post_meta` dictionary.

Figure 21: Example in Ruby project with sha value: “584f72e7f4c65066ccbd2183af76bf380b6eb974"



```
@@ -403,7 +403,7 @@ public boolean canDismiss(Object token) {
                    @Override
                    public void onDismiss(View view, Object token) {
                        if (view != null) {
-                           finish();
+                           dismiss(false);
                        }}
@@ -512,6 +512,10 @@ private void startTimer(long duration) {
    }
    public void dismiss() {
+       dismiss(mAnimated);
+   }
+
+   private void dismiss(boolean animate) {
        if (mIsDismissing) {
            return;
        }
@@ -522,7 +526,7 @@ public void dismiss() {
            mEventListener.onDismiss(Snackbar.this);
        }
-       if (!mAnimated) {
+       if (!animate) {
            finish();
            return;
        }
```

Fixed issue where onDismiss() did not get called if Snackbar is swiped.

```
...
public static Snackbar with(Context context) {
        return new Snackbar(context);
    }

    /**
     * Sets the type of {@link Snackbar} to be displayed.
     *
     * @param type the {@link SnackbarType} of this instance
     * @return
     */
    public Snackbar type(SnackbarType type) {
        mType = type;
        return this;
    }

    /**
     * Sets the text to be displayed in this {@link Snackbar}
...
```

CodeAgent-3.5

Semantic Consistency Analysis

The code changes in the commit do not accurately reflect the description provided in the commit message. The commit message mentions fixing an issue where onDismiss() did not get called if Snackbar is swiped, but the code changes do not address this issue. Instead, the code changes seem to be related to dismissing the Snackbar.

Security Analysis

No security analysis can be performed as the code is not provided.

Format Analysis

The format of the code aligns with the writing style and format of the original file. There are no formatting inconsistencies that impact the overall readability and maintainability of the project.

Suggestion

The highest priority comment is regarding the semantic consistency between the code changes and the commit message. The code changes should address the issue mentioned in the commit message, i.e., fixing the issue where onDismiss() did not get called if Snackbar is swiped. To fix this issue, the following changes can be made:

CodeAgent-4.0

Semantic Consistency Analysis

The commit message states "Fixed issue where onDismiss() did not get called if Snackbar is swiped." The code changes reflect this accurately. The `onDismiss()` method has been modified to call `dismiss(false)`, which will dismiss the Snackbar without animation when it is swiped. A new `dismiss()` method has also been added which takes a boolean parameter to determine whether to animate the dismissal or not. The changes in the code are consistent with the commit message.

Security Analysis

The provided code does not seem to have any security vulnerabilities. It does not handle any user input directly, so there is no risk of SQL injection, XSS, or command injection. It does not use any lower-level languages, so buffer overflows are not a concern. The code does not manage any sensitive data, so there is no risk of unauthorized access or data breaches. The code does not use any third-party libraries, so there are no potential vulnerabilities from dependencies. The code does not have any deprecated functions, hardcoded sensitive data, or code leakages. However, this is a preliminary analysis and a more thorough review would be needed to confirm these findings

Format Analysis

The format of the code aligns with the writing style and format of the original file. The code uses the same indentation, naming conventions, and comment style as the original file. There are no formatting inconsistencies that would impact the readability or maintainability of the project.

Suggestion

The code changes are well-aligned with the rest of the codebase. The new `dismiss()` method follows the same naming conventions and style as the existing methods. The changes to the `onDismiss()` method are minimal and maintain the original structure of the method. No revisions are necessary.

Figure 22: Comparison of code review between `CodeAgent`-3.5 and `CodeAgent`-4.0 on one Java project with sha value: "6e3c6d17d943f5ac70b421653eb167e0c34b119f"

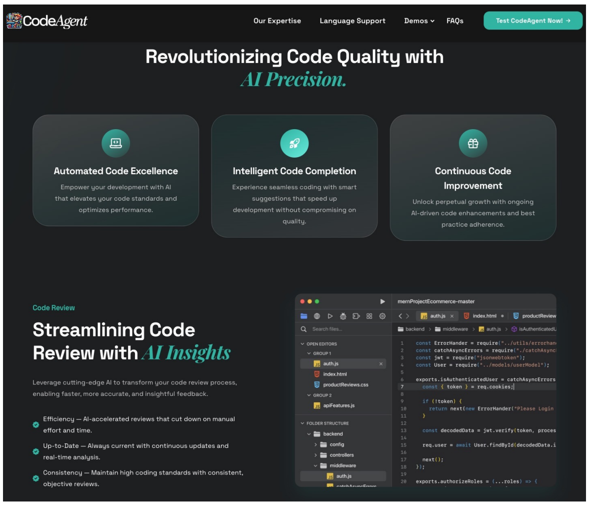
Table 12: Summarizes the average query time and cost per code review for `CodeAgent`-3.5 and `CodeAgent`-4.

Model	Query Time(min)	Cost in USD
`CodeAgent`-3.5	3	0.017
`CodeAgent`-4	5	0.122

O Tool

We develop a website for `CodeAgent`, which is shown in Figure 23, and it is also accessable by visiting following link:

`https://code-agent-new.vercel.app/index.html`



Revolutionizing Code Quality with *AI Precision.*

Automated Code Excellence

Empower your development with AI that elevates your code standards and optimizes performance.

Intelligent Code Completion

Experience seamless coding with smart suggestions that speed up development without compromising on quality.

Continuous Code Improvement

Unlock perpetual growth with ongoing AI-driven code enhancements and best practice adherence.

Code Review

Streamlining Code Review with *AI Insights*

Leverage cutting-edge AI to transform your code review process, enabling faster, more accurate, and insightful feedback.

- Efficiency — AI-accelerated reviews that cut down on manual effort and time.
- Up-to-Date — Always current with continuous updates and real-time analysis.
- Consistency — Maintain high coding standards with consistent, objective reviews.

```
const ErrorHander = require("../utils/errorhan
const catchAsyncErrors = require("./catchAsync
const jwt = require("jsonwebtoken");
const User = require("../models/userModel");

exports.isAuthenticatedUser = catchAsyncErrors
  const { token } = req.cookies;

  if (!token) {
    return next(new ErrorHander("Please Login
  }

  const decodedData = jwt.verify(token, proces

  req.user = await User.findById(decodedData.i

  next();
});

exports.authorizeRoles = (...roles) => {
```

Figure 23: website of `CodeAgent`