CodeAgent: Collaborative Agents for Software Engineering

Daniel Tang[♠] ✉ **Zhenghan Chen**[♣] **Kisub Kim**[▲] ✉ **Yewei Song**[♠] **Haoye Tian**[♠]
Saad Ezzini[♦] **Yongfeng Huang**[★] **Jacques Klein**[♠] **Tegawendé F. Bissyandé**[♠]
[♠]University of Luxmbourg [♣]Microsoft [▲]Singapore Management University
[♦]Lancaster University [★]The Chinese University of Hong Kong.
`xunzhu.tang@uni.lu` `falconlk00@gmail.com`

Abstract

Code review is a heavily collaborative process, which aims at ensuring the overall quality and reliability of software. While it provides massive benefits, the implementation of code review in an organization faces several challenges that make its automation appealing. Automated code review tools have been around for a while and are now improving thanks to the adoption of novel AI models, which help can learn about standard practices and systematically check that the reviewed code adheres to them. Unfortunately, existing methods fall short: they often target a single input-output generative model, which cannot simulate the collaboration interactions in code review to account for various perspectives; they are also sub-performing on various critical code review sub-tasks. In this paper, we advance the state of the art in code review automation by introducing **CodeAgent**, a novel multi-agent-based system for code review. Fundamentally, **CodeAgent** is steered by QA-Checker (short for "Question-Answer Checking"), a supervision agent, designed specifically to ensure that all agents' contributions remain relevant to the initial review question. **CodeAgent** is autonomous, multi-agent, and Large language model-driven. To demonstrate the effectiveness of **CodeAgent**, we performed experiments to assess its capabilities in various tasks including 1) detection of inconsistencies between code changes and commit messages, 2) detection of vulnerability introduction by commits, and 3) validation of adherence to code style. Our website is accessed in `https://code-agent-new.vercel.app/index.html`.

1 Introduction

Code review, as a fundamental activity in software engineering, has been widely studied in the literature [2; 3; 8]. It involves multiple team members, with different expertise and experience, collaborating to check the code along several dimensions, including whether the new code is aligned with existing code in terms of style [14], whether the pull request contents are consistent [43], whether some vulnerability is being injected [4], etc. This review effort is critical for ensuring the stability, quality and readability of the code. Historically, code review has been an intensive manual exercise. Nowadays, however, software projects evolve at a rapid pace with a high frequency of commits and pull requests. For example, in 2022, GitHub, a global platform for software development, witnessed over 3.5 billion contributions, including commits, pull requests, and other forms of contributions [16; 30]. Reviewing these contributions before they are integrated into any code base requires automated tools to ease the workload of code contributors and code reviewers.

✉: Corresponding Authors.

Preprint.

In recent literature, various approaches [35; 32; 34] have been proposed to elevate code review automation, using Transformers [36] for code modeling, code abstraction tools such as src2abs [33] and code transformation engines such as AutoTransform [32]. These state-of-the-art approaches however mainly focused on how to rewrite and adapt the submitted code. Code review, however, is actually a collaborative and iterative process involving multiple stakeholders exchanging on a specific thought process [2] and addressing a wide variety of subtasks such as analyzing the consistency between a code change and the associated commit message (CA) [43], detecting potential vulnerability injection (VA) [4], assessing the consistency of code format (FA) [14], etc.

To address the challenges of complex software collaboration eco-system, researchers have applied agent-based systems across a spectrum of tasks, advancing past the conventional single input-output paradigm [40; 38]. Notably, the advent of multi-agent collaboration stands out as a key innovation, especially in simulating human-like behaviors [10; 24; 27] and harnessing the collective strengths of various agents [7; 22; 17].

In this paper, drawing on the success of agent-based collaboration, we developed an agent-based framework, **CodeAgent**, to simulate the dynamics of a collaborative team engaged in code review processes, incorporating various roles such as code authors, reviewers, and decision-makers. Nonetheless, a notable hurdle within multi-agent systems and Chain-of-Thought (CoT) reasoning is the propensity for conversation topics to drift off course, underlining the necessity for strategies to maintain topic relevance and coherence [20; 6]. The occurrence of drift, often triggered by the model-inspired tangents or the randomness of Large Language Models (LLMs), necessitating the integration of a QA-Checker. This QA-Checker, serving as a crucial supervisory agent within **CodeAgent**, meticulously monitors the flow of conversation, ensuring that questions and responses remain pertinent and on track, thus maintaining the dialogue's intended direction [20; 6]. As an instruction-driven entity, the QA-Checker not only refines queries but also realigns answers to match the original intent, employing a systematic approach grounded in a mathematical framework. This involves leveraging a quality assessment function $\mathcal{Q}$ and the Newton-Raphson optimization method [41] to iteratively guide the conversation towards optimal coherence and relevance.

To evaluate the performance of **CodeAgent**, we employ two distinct sources of data. The first encompasses pre-existing datasets: Trans-Review$_{data}$, AutoTransform$_{data}$, and T5-Review$_{data}$, which have been referenced by leading research in the field. By conducting experiments with these datasets, we aim to compare the code revision capabilities of **CodeAgent** against those identified as state-of-the-art. The second data source comprises a collection of Pull Requests from GitHub, featuring an extensive array of commits, messages, and comments across nine programming languages post-April 2023. These languages include Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. This diverse dataset is specifically curated for testing format analysis. The experimental results indicate that **CodeAgent** outperforms the state-of-the-art results in terms of qualitative metrics for code review, achieving a 41.54 percentage point (pp) increase in hit rate (Rate$_{cr}$ in Table 3) confirming vulnerable issues within merged commits across the nine programming languages. **CodeAgent** also excels in consistency detection, vulnerability identification, and format analysis, outperforming ChatGPT in overall metrics.

In conclusion, we summarize our contributions as follows:

- We build a code review dataset with more than 3,545 real-world commits, commit messages, and corresponding original files for assessing code review model performance in various areas including consistency detection between commits and commit messages, vulnerability detection, code style detection, and code revision.
- 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.
- Experimental results show that **CodeAgent** confirms 104 (23.20% higher) more vulnerable issues. Compared to state-of-the-art, **CodeAgent** improves 5.62pp and 4.00pp of recall and F1-Score, respectively, for the consistency detection between commit and commit message. **CodeAgent** also boosts 15.96pp and 10.45pp of the recall score and F1-Score, respectively, for the format consistency detection. On the code revision task, **CodeAgent** surpasses the state-of-the-art from 29.80pp to 31.60pp of the Edit Progress (EP) metric [44] on average.

2 Tasks and Definition

In this section, we summarize our tasks (❶,❷,❸,and ❹) and definitions (CA,VA,FA) in Table. 1.

Table 1: Tasks and Definitions

Index	Tasks
❶	Semantic consistency detection between commit and commit message
❷	Vulnerability analysis
❸	Format consistency detection
❹	Code revision
Short Term	**Definition**
CA	Consistency analysis between commit and commit message
VA	Vulnerability analysis
FA	Format consistency analysis between commit and original files

3 `CodeAgent`

This section details the methodology behind our innovative **`CodeAgent`** framework. We first describe the defined role cards in Section 3.1 and discuss the pipeline in Section 3.2. Finally, we will discuss the design of the QA-Checker.

3.1 Role Card Definition

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

For each, we defined a role card, which contains: 1) The role name is put on the left-upper corner of each card; 2) The phases of the role involved are put on the right-upper corner of each card; 3) On each role card, we show the role-involved conversation and collaborative roles; 4) We show the intermediate output of the role on the right-hand side of the card; and 5) Finally, we put the corresponding files or content out of conversations on the bottom of the card.

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 (task ❶, ❷, and ❸ in Table 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 Review'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 B.1.

3.2 Pipeline

We specifically consider the following sub-tasks of code review: consistency detection between commit and commit message (CA) [43], vulnerability injection detection (VA) [4], Format consistency detection (FA) [14], and code revision [44]. We defined six characters and four phases for the framework. The roles of characters are demonstrated 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, leveraging 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 by code revision or suggestion feedback to the author; ④ Document, finalizing by synthesizing the opinions of the CEO, CPO, $Coder$ and $Reviewer$ to give out the final comments.

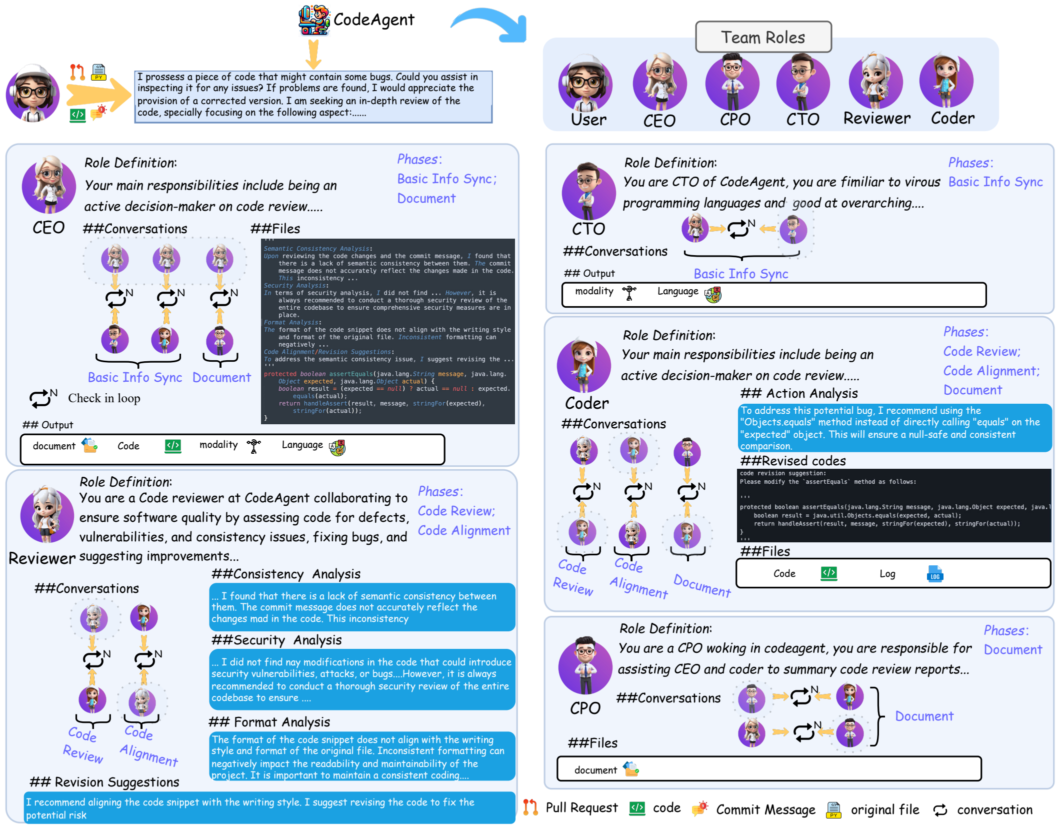


Figure 1: **A Schematic diagram of role data cards of simulated code review team and their conversations within `CodeAgent`.** We have six characters in **`CodeAgent`** across four phases, including "Basic Info Sync", "Code Review", "Code Alignment", and "Document". Code review is a kind of collaboration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

Apart from six defined roles, the proposed architecture of **`CodeAgent`** consists of phase-level and conversation-level components. At the phase level, the waterfall model is used to break down the code review process into four sequential phases. At the conversation level, each phase is further divided into atomic conversations. These atomic conversations involve task-oriented role-playing between two agents, promoting collaborative communication. One agent works as the instructor and the other works as the assistant. Communication follows an instruction-following style, where agents interact to accomplish a specific subtask within each conversation and each conversation is under the supervision of QA-Checker. QA-Checker is used to align the consistency of questions and answers between the instructor and assistant in a conversation to avoid digression. QA-Checker will be introduced in Section 3.3.

Here, we take an example to show the pipeline of **`CodeAgent`**. As shown in Figure 2, **`CodeAgent`** receives the request to do the code review with the submitted commit, commit message, and original files. In the first phase, CEO, CPO, and $Coder$ will cooperate to recognize the modality of the 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 about consistency analysis, vulnerability analysis, format analysis, and suggestions for code revision. Then, in the third phase, according to analysis reports, $Coder$ will align or revise the code if it finds incorrect snippets with the help of $Reviewer$. $Coder$ cooperates with CTO and CEO to summarize the document and codes about the whole code review in the final phase.

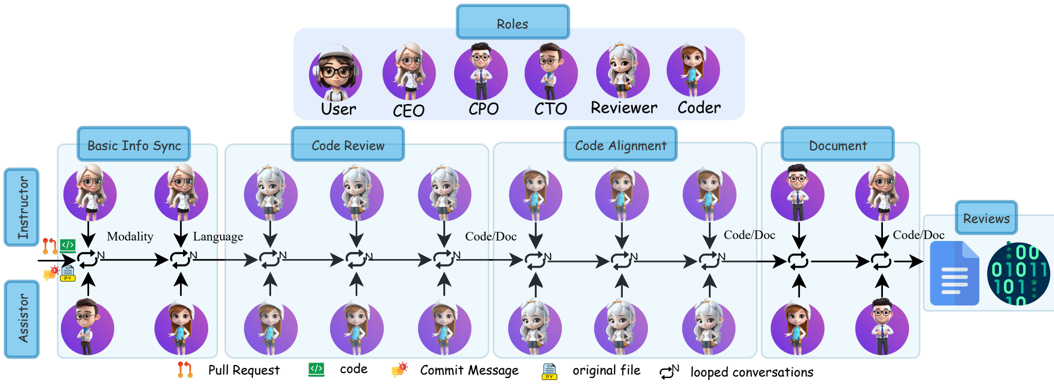


Figure 2: **CodeAgent**'s pipeline/scenario of a full conversation during the code review process among different roles. "Basic Info Sync" demonstrates the basic information confirmation by the CEO, CTO, and Coder; "Code Review" shows the actual code review process; "Code Alignment" illustrates the potential 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.

3.3 Self-Improving CoT with QA Checker

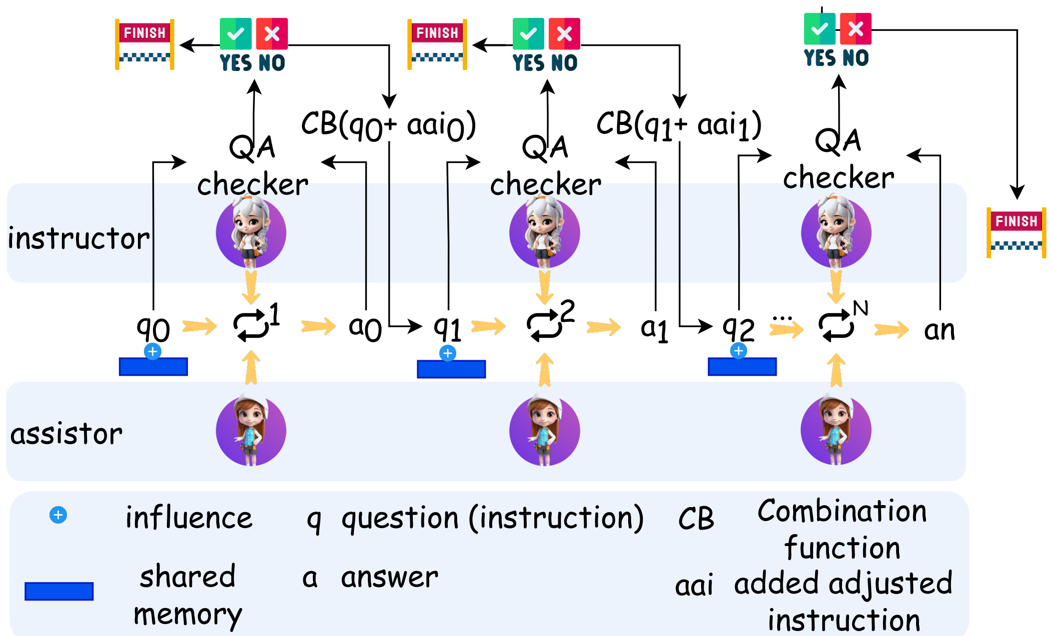


Figure 3: This diagram shows the architecture of our designed Chain-of-Thought (CoT): Question-Answer Checker (QA-Checker).

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, it reaches the maximum dialogue times, otherwise.

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.

Lemma 3.1. *Let $\mathcal{Q}(Q_i, A_i)$ denote the quality assessment function of the QA-Checker for 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 is 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.

4 Experimental Design

We evaluate the performance of **`CodeAgent`** in various qualitative and quantitative experiments in nine programming languages, on four metrics. In this Section, we will discuss experimental settings, including datasets, metrics, and baselines. For more information, please see Appendix B.

4.1 Datasets

For our research, we leverage datasets from prior studies as referenced in the state-of-the-art [44]. Specifically, we utilized Trans-Review$_{data}$, AutoTransform$_{data}$, and T5-Review$_{data}$ to assess the edit progress (EP) metric which is also designed by them [44].

As shown in Table 2, in terms of new data collection, which is called **codeData**, is collected using the GitHub REST API, encompasses over 3,545 commits and 2,933 pull requests from more than 180 projects in 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 E.

Table 2: Comparison of Positive and Negative Samples in CA and FA (CA and FA are defined in Section 2).

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

4.2 Metrics

F1-Score and Recall. We utilized the F1-Score and recall to evaluate our method's effectiveness. 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 [19].

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. A higher EP indicates better efficiency in code generation [9; 11; 44].

Hit Rate (Rate) We also use hit rate to evaluate the rate of confirmed vulnerable issues out of the found issues by approaches.

4.3 State-of-the-Art Tools and Models

Our study evaluates various tools and models for code revision and modeling. **Trans-Review** [35] employs src2abs for code abstraction, effectively reducing vocabulary size. **AutoTransform** [32] uses Byte-Pair Encoding for efficient vocabulary management in pre-review code revisions. **T5-Review** [34] leverages the T5 architecture, emphasizing improvement in code review through pre-training on code and text data. In handling both natural and programming languages, **Code-BERT** [12] adopts a bimodal approach, while **GraphCodeBERT** [13] incorporates code structure into its modeling. **CodeT5** [37], based on the T5 framework, is optimized for identifier type awareness, aiding in generation-based tasks. Additionally, we compare these tools with **ChatGPT** [26] by OpenAI, notable for its human-like text generation capabilities in natural language processing.

5 Experimental Result Analysis

In this Section, we discuss the performance of **`CodeAgent`** on four main experiments: vulnerability analysis (Section 5.1), inconsistency detection between commit and commit message (Section 5.2), format inconsistency detection (Section 5.2), and code revision (Section 5.3). We also discuss the difference in execution time of **`CodeAgent`** across different languages and conduct capabilities analysis between **`CodeAgent`** and other communicative agents in Appendix-Section D.

5.1 Vulnerability Analysis

As shown in Table 1, vulnerability analysis ❷ is one subtask of code reviews. Compared to ❶ and ❸, ❷ is a more complex code review subtask, covering more than 25 different aspects (please see the Appendix-Section F), including buffer overflows, sensitive data exposure, configuration errors, data leakage, etc. This domain necessitates deep technical expertise for accurate data annotation, thereby significantly increasing the time and money cost when labeling manually. Considering the low proportion of commits with vulnerabilities, this paper proposes a proactive verification method for data annotation.

Our approach is twofold. First, we utilize **CodeAgent** to process 3,545 pairs of commits, commit messages and original files across nine languages. These identified data points for potential vulnerabilities were then subjected to manual verification. Second, we employed tools like CodeBERT [12] and ChatGPT to do vulnerability binary detection in the same dataset. The results were then verified for their authenticity.

Table 3: Vulnerable problems (#) found by **CodeAgent** and other approaches. As described in Appendix-Section E, we have 3,545 items to evaluate. Rate$_{cr}$ means the confirmed number divided by the number of findings while Rate$_{ca}$ is the confirmed number divided by the total evaluated number. **CodeAgent** $_{w/o}$ means **CodeAgent** without QA-Checker.

Approach	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	**CodeAgent** $_{w/o}$
Find	1,063	864	671	483	564
Confirm	212	317	345	**449**	413
Rate$_{cr}$	19.94%	36.69%	51.42%	**92.96%**	73.23%
Rate$_{ca}$	5.98%	8.94%	9.73%	**12.67%**	11.65%

Comparison As delineated in Table 3, the deployment of **CodeAgent** successfully identified 483 potential vulnerabilities within a dataset of 3,545 items, with an impressive 449 of these ultimately confirmed as high-risk vulnerabilities, substantiated through a rigorous manual verification process exceeding 120 working hours[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 ChatGPT-3.5 and ChatGPT-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 ChatGPT-3.5, and 51.42% for ChatGPT-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, ChatGPT-3.5, ChatGPT-4.0, and **CodeAgent**, respectively. Evidently, Table 3 not only highlights **CodeAgent**'s high precision in identifying vulnerable commits but also reveals the progressive improvement from ChatGPT-3.5 to ChatGPT-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 ChatGPT and CodeBERT. Further details regarding the importance of the QA-checker can be seen in Appendix-Section C and Section L. Moreover, more experimental results in 9 languages are accessible in Appendix-Section I.

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

[1] The verification process involved meticulous manual examination, extending beyond 120 working hours.

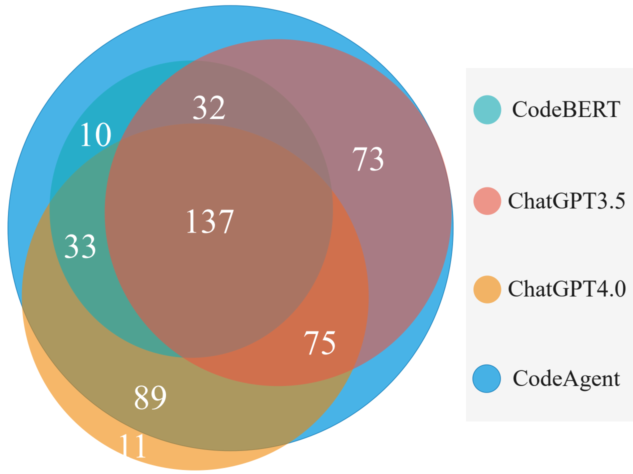


Figure 4: Venn Diagram of CodeBERT, ChatGPT-3.5, ChatGPT-4.0, and **CodeAgent**.

Ablation Study As shown in Table 3, without QA-Checker, **CodeAgent** $_{w/o}$ is less effective in finding vulnerable issues and reduces the hit rate (Rate$_{cr}$ and Rate$_{ca}$) compared with the full version, indicating the importance of QA-Checker in our **CodeAgent**. More detailed information about the ablation study can be found in Appendix-Section L.

Moreover, **CodeAgent** versions 3.5 and 4.0 yielded consistent detection results, but they differ in the quality of explanation when doing the vulnerability analysis of the commit. More details about the difference between **CodeAgent**-3.5 and **CodeAgent**-4.0 are provided in Appendix-Section K.2.

5.2 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 ❶ and ❸. For ❶ and ❸, the dataset we have is shown in Table 2 and more detailed data information is shown in Figure 7 in Appendix.

Consistency Detection Between Commit and Commit Message Our comprehensive study, as illustrated in Table 4, assesses **CodeAgent**'s efficacy in detecting the consistency between commit and commit message, contrasting its performance with other prevalent methods like CodeBERT, ChatGPT-3.5, and ChatGPT-4.0. This evaluation specifically concentrates on merged and closed commits across nine languages, with a keen focus on crucial metrics such as Recall and F1-Score. Notably, **CodeAgent** exhibits a remarkable performance, outstripping 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 5.62pp 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 4.00pp. More comparable details in different languages are shown in Appendix-Section. J.

Format Consistency Detection Between Commit and Original File In our detailed evaluation of format consistency between commits and original files, **CodeAgent**'s performance was benchmarked against established models like CodeBERT and ChatGPT variants across nine different languages. This comparative analysis, presented in Table 5, 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 10.81pp in Recall and 6.94pp 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 average improvement of 15.96pp in Recall and 9.94pp 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 15.96pp in Recall and 10.45pp in F1-Score. These results underscore

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

Merged	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	Imp (pp)
Recall	63.64	80.08	84.27	90.11	5.84
F1	75.00	87.20	90.12	93.89	3.77
Closed	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	Imp (pp)
Recall	64.80	79.05	81.75	87.15	5.40
F1	77.20	87.35	89.10	92.40	3.30
Average	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	Imp (pp)
Recall	64.22	79.57	83.01	88.63	5.62
F1	76.01	87.28	89.61	93.16	4.00

CodeAgent's exceptional capability in accurately detecting format consistency between commits and their original files.

Table 5: Comparison of **CodeAgent** with other methods on merged and closed commits across the 9 languages on **FA task**. 'Imp' represents the improvement.

Merged	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	Imp (pp)
Recall	60.59	60.72	78.53	89.34	10.81
F1	74.14	74.88	87.07	94.01	6.94
Closed	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	Imp (pp)
Recall	69.95	73.61	68.46	89.57	15.96
F1	80.49	84.19	80.16	94.13	9.94
Average	CodeBERT	ChatGPT-3.5	ChatGPT-4.0	**CodeAgent**	Imp (pp)
Recall	65.27	67.17	73.50	89.46	15.96
F1	77.32	79.54	83.62	94.07	10.45

5.3 Code Revision

In this Section, we evaluate the effectiveness of various approaches in bug fixing by comparing their Error Percentage (EP) performance. The methods under consideration include Trans-Review, AutoTransform, T5-Review, CodeBERT, GraphCodeBERT, CodeT5, and **CodeAgent**. As detailed in Table 6, these approaches exhibit varied performance across different datasets. Notably, **CodeAgent** demonstrates a remarkable performance, particularly in the T5-Review dataset, where it achieves the highest EP of 37.6%. This is a significant improvement over other methods, underscoring the effectiveness of **CodeAgent** in handling complex code revision tasks. Additionally, with an average EP of 31.6%, **CodeAgent** consistently outperforms its counterparts, positioning itself as a leading solution in automated code revision. The contribution of **CodeAgent** to the field of automated code revision is noteworthy. Its ability to excel in the T5-Review dataset, a challenging benchmark, indicates a sophisticated understanding and handling of nuanced bugs. Moreover, its overall average performance surpasses that of other state-of-the-art models, highlighting its robustness and reliability. These results suggest that **CodeAgent** outperforms the state-of-the-art approaches in fixing buggy codes on EP metric on average.

Table 6: Experimental Results for the Code Revision of **CodeAgent** and the state-of-the-art works. Bold indicates the best performers.

Approach	**Trans-Review$_{\text{data}}$**	**AutoTransform$_{\text{data}}$**	**T5-Review$_{\text{data}}$**	**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%**

6 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. [15], who addressed code change anticipation, and Siow et al. [29], who introduced CORE for code modification semantics. Hong et al. [18] proposed COMMENTFINDER for comment suggestions, while Tufano et al. [35] and Li et al. [23] developed tools for code review automation using models like T5CR and CodeReviewer, respectively. Recently, Lu et al. [25] 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 [31; 28], focusing on collective thinking, conversation dataset curation [39; 21], and sociological phenomenon exploration [27]. Research by Akata et al. [1] and Cai et al. [5] further explores LLM cooperation and efficiency. However, there remains a gap in integrating these advancements with structured software engineering practices [21; 28], 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 A.

7 Conclusion

This paper introduces **`CodeAgent`**, a novel multi-agent-based framework designed to automate code reviews, leveraging an innovative QA-Checker system to maintain focus and alignment with the review's objectives. **`CodeAgent`** demonstrates promising capabilities in detecting vulnerabilities, ensuring consistency between code changes and commit messages, and improving uniformity of code style. Our **`CodeAgent`** outperforms existing state-of-the-art solutions. By considering the specific characteristics of the code review process and incorporating the human-like conversational framework, **`CodeAgent`** significantly enhances efficiency and accuracy. Finally, we believe that our paper opens a new avenue for future software development collaboration practice and research.

8 Impact Statement

The adoption of our multi-agent-based code review framework promises to revolutionize how software development teams ensure code quality. By automating and enhancing the code review process, developers can allocate more time to creative and value-added tasks, leading to increased productivity and a higher standard of software craftsmanship. We expect that the adaptive learning capabilities of our approach contribute to a sustainable and evolving solution, capable of addressing the dynamic challenges in modern software development and machine learning.

9 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).

References

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

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

[3] Amiangshu Bosu and Jeffrey C Carver. 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, 2013.

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

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

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

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

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

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

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

[11] Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 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, June 2021. Association for Computational Linguistics.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *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, 2020.

[13] 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. 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, 2021.

[14] DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. 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, 2020.

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

[16] Jeremy Holcombe. Key github statistics in 2024 (users, employees, and trends), Ocb 2023.

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

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

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

[20] humanfirst. Prompt drift and chaining, May 2023.

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

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

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

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

[25] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658. IEEE, 2023.

[26] OPENAI. Chatgpt, 2022.

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

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

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

[30] software.com. Pull request frequency, 2023.

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

[32] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering*, pages 237–248, 2022.

[33] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 2019.

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

[35] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[37] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *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, 2021.

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

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

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

[41] Tjalling J Ypma. Historical development of the newton–raphson method. *SIAM review*, 37(4):531–551, 1995.

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

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

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

Contents (Appendix)

A 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. [15] addressed the challenge of anticipating code change positions. Siow et al. [29] introduced CORE, employing multi-level embeddings for code modification semantics and retrieval-based review suggestions. Hong et al. [18] proposed COMMENTFINDER, a retrieval-based method for suggesting comments during code reviews. Tufano et al. [35] designed T5CR with SentencePiece, enabling work with raw source code without abstraction. Li et al. [23] 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. [25] 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. [31] and Qian et al. [28]. 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. [39] and Li et al. [21] have created conversational datasets through role-playing methodologies. Sociological phenomenon investigations, such as Park et al. [27]'s work, involve creating virtual communities with rudimentary language interactions and limited cooperative endeavors. In contrast, Akata et al. [1] scrutinized LLM cooperation through orchestrated repeated games. Collaboration for efficiency, proposed by Cai et al. [5], introduces a model for cost reduction through large models as tool-makers and small models as tool-users. Zhang et al. [42] established a framework for verbal communication and collaboration, enhancing overall efficiency. However, Li et al. [21] and Qian et al. [28], 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.

B Experimental Details

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

B.1 Role Definition

Six roles are defined as shown in Figure 5.

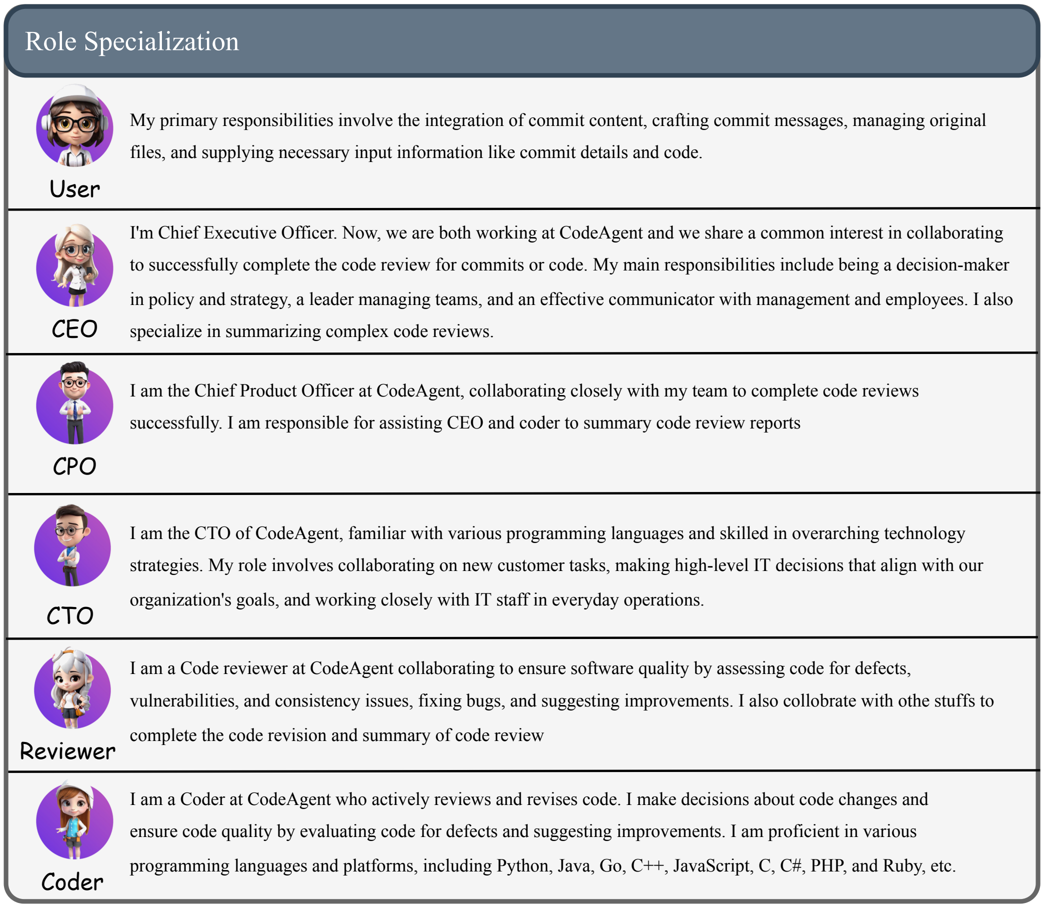


Figure 5: Specialization of six main characters in **CodeAgent**.

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

> 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, I apply advanced algorithms, including Chain-of-Thought reasoning and optimization techniques, to evaluate and guide conversational flow. I am adept at identifying and correcting topic drifts, 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.

B.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.

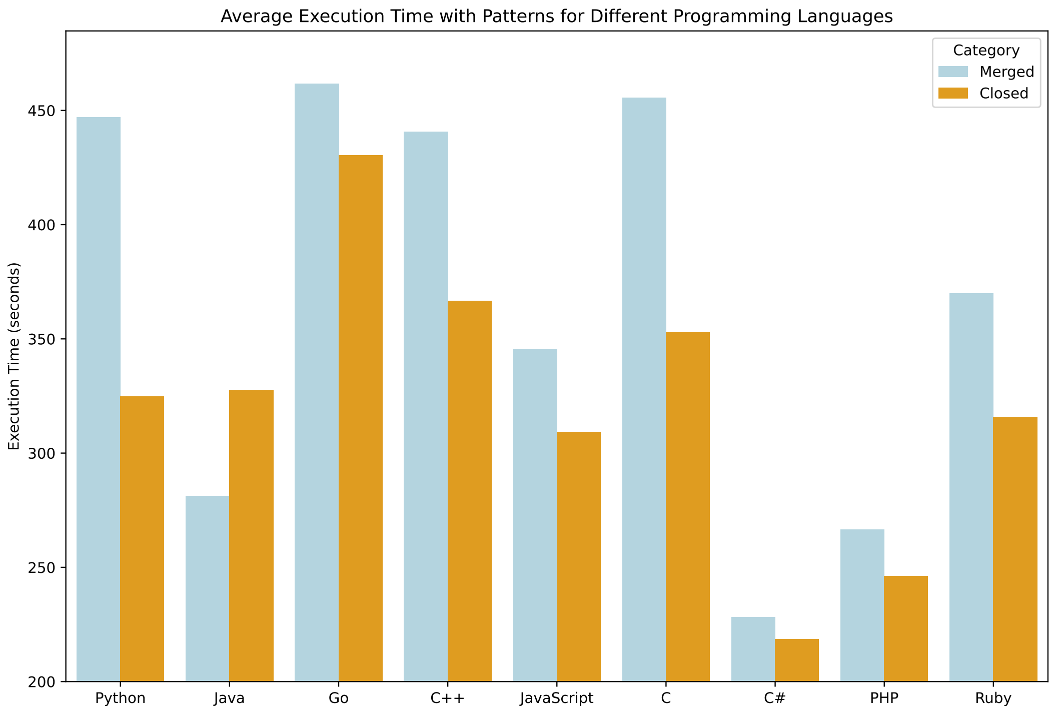


Figure 6: Execution time with **`CodeAgent`** across different language (count unit: second).

C 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 [17], and underscore the importance of the QA-Checker in role conversations.

C.1 Comparison Table

We begin with a comparative overview presented in Table 7.

C.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.

C.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,

Table 7: 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

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.

C.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.

D 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 8, 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 C.

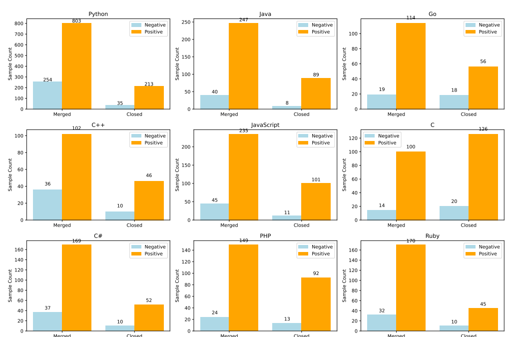
Table 8: Comparison of capabilities for **`CodeAgent`** and other approaches. '✓' indicates the presence of a specific feature in the corresponding framework, '✗ is absence. ChatDev and MetaGPT are two representative multi-agent frameworks, ChatGPT is a kind of single-agent framework, and CodeBert is a representative pre-trained model.

Approaches	Consistency Analysis	Vulnerability Analysis	Format Analysis	Code Revision	COT	QA-Checker
ChatDev [28]	✗	✗	✗	✗	✓	✗
MetaGPT [17]	✗	✗	✗	✗	✓	✗
ChatGPT [26]	✓	✓	✓	✓	✗	✗
CodeBert [12]	✓	✓	✓	✓	✗	✗
`CodeAgent`	✓	✓	✓	✓	✓	✓

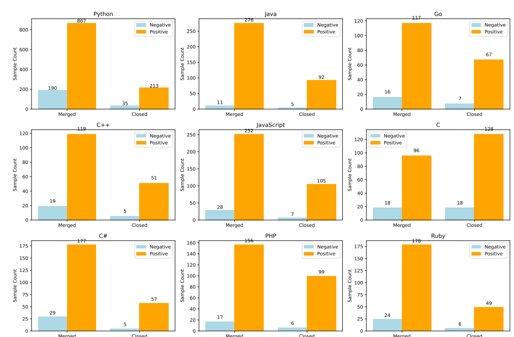
E Dataset

Previous Dataset As shown in [44], 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. [35], 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 Thongtanunam et al. [32] from three Gerrit repositories, comprises only submitted and revised codes without review comments. Lastly, T5-Review$_{\text{data}}$, gathered by Tufano et al. [34] 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.



(a) Positive and negative data of both merged and closed commits across 9 languages on CA task (Table 1).



(b) Positive and negative data of both merged and closed commits across 9 languages on FA task (Table 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'.

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 Table 1)) and FA (format consistency between commit and original (See Table 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.

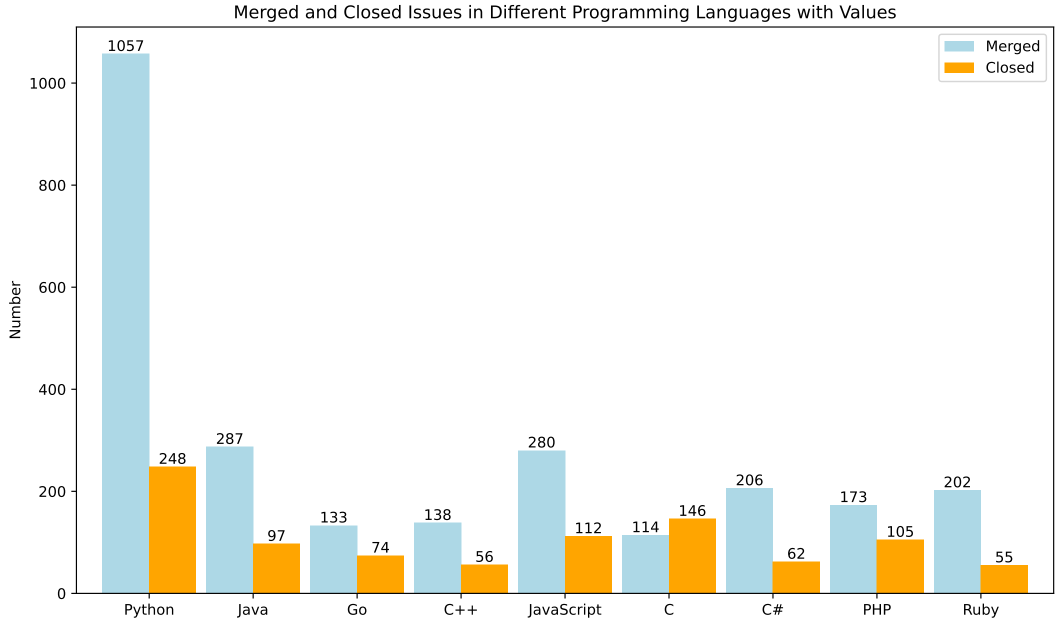


Figure 8: Comparative Visualization of Merged and Closed Commit Counts Across Various Programming Languages

Table 9: 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

F 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.

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.

G Data Leakage Statement

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

H 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: $qScore = \mathfrak{Q}(Q_i, A_i)$
  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

In this algorithm, $\mathfrak{Q}(Q_i, A_i)$ 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.

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

In our comprehensive analysis using **`CodeAgent`**, as detailed in Table 11, 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 11 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.

Table 11: 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%

J 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.

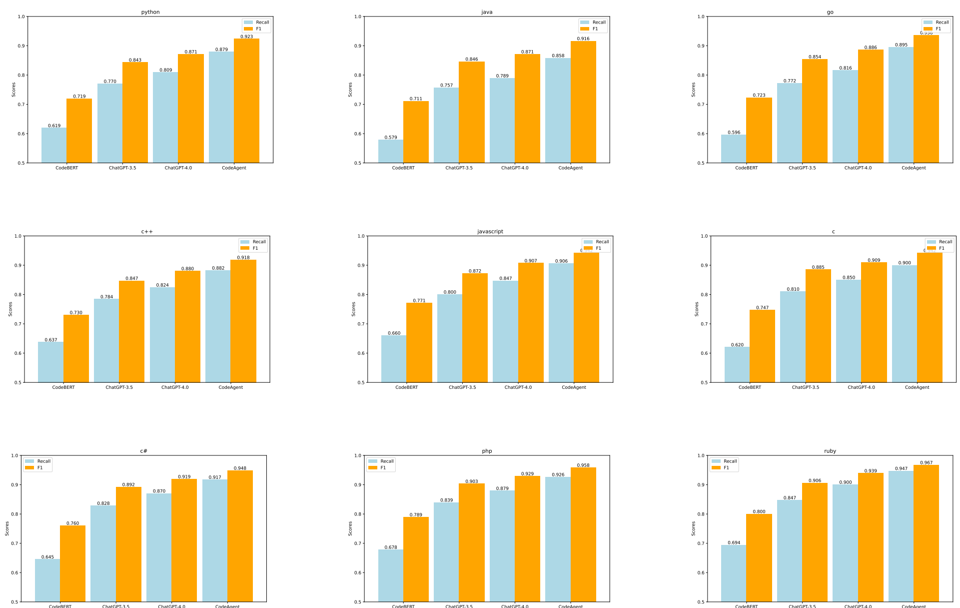


Figure 9: Comparison of models on the **merged** data across 9 languages on **CA task**.

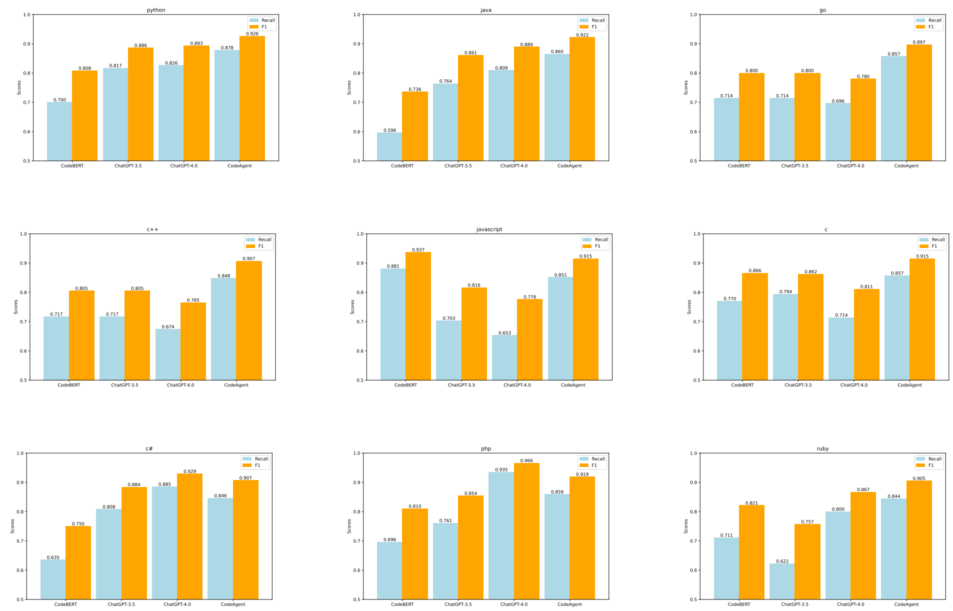


Figure 10: Comparison of models on the **closed** data across 9 languages on **CA task**.

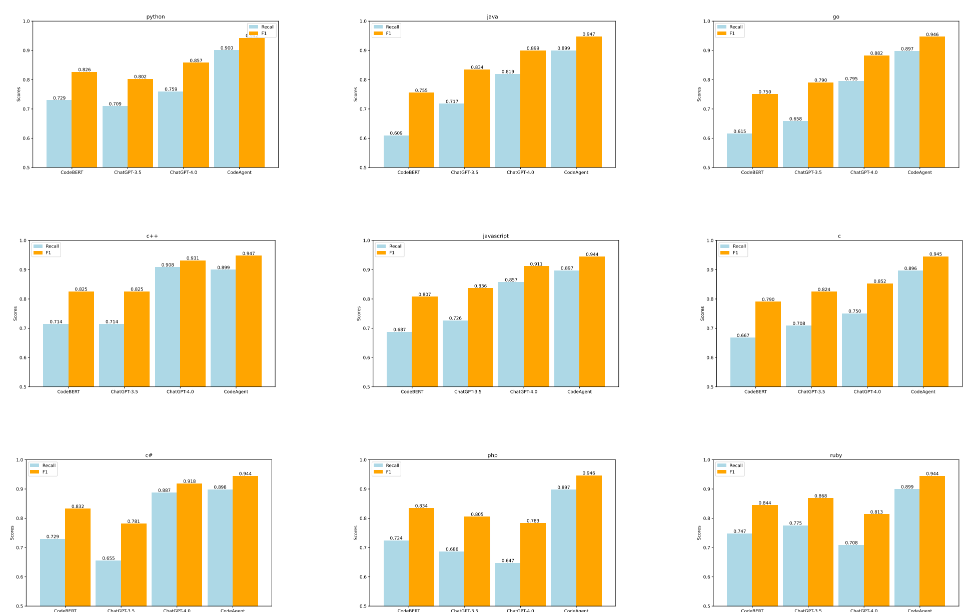


Figure 11: Comparison of models on the **merged** data across 9 languages on **FA task**.

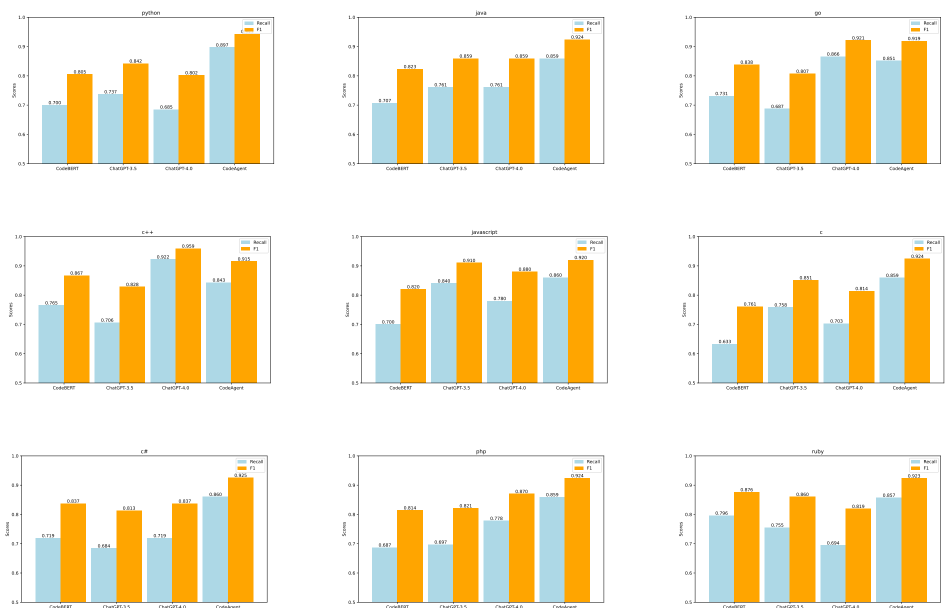


Figure 12: Comparison of models on the **closed** data across 9 languages on **FA task**.

K Case Study

K.1 Performance on 9 languages

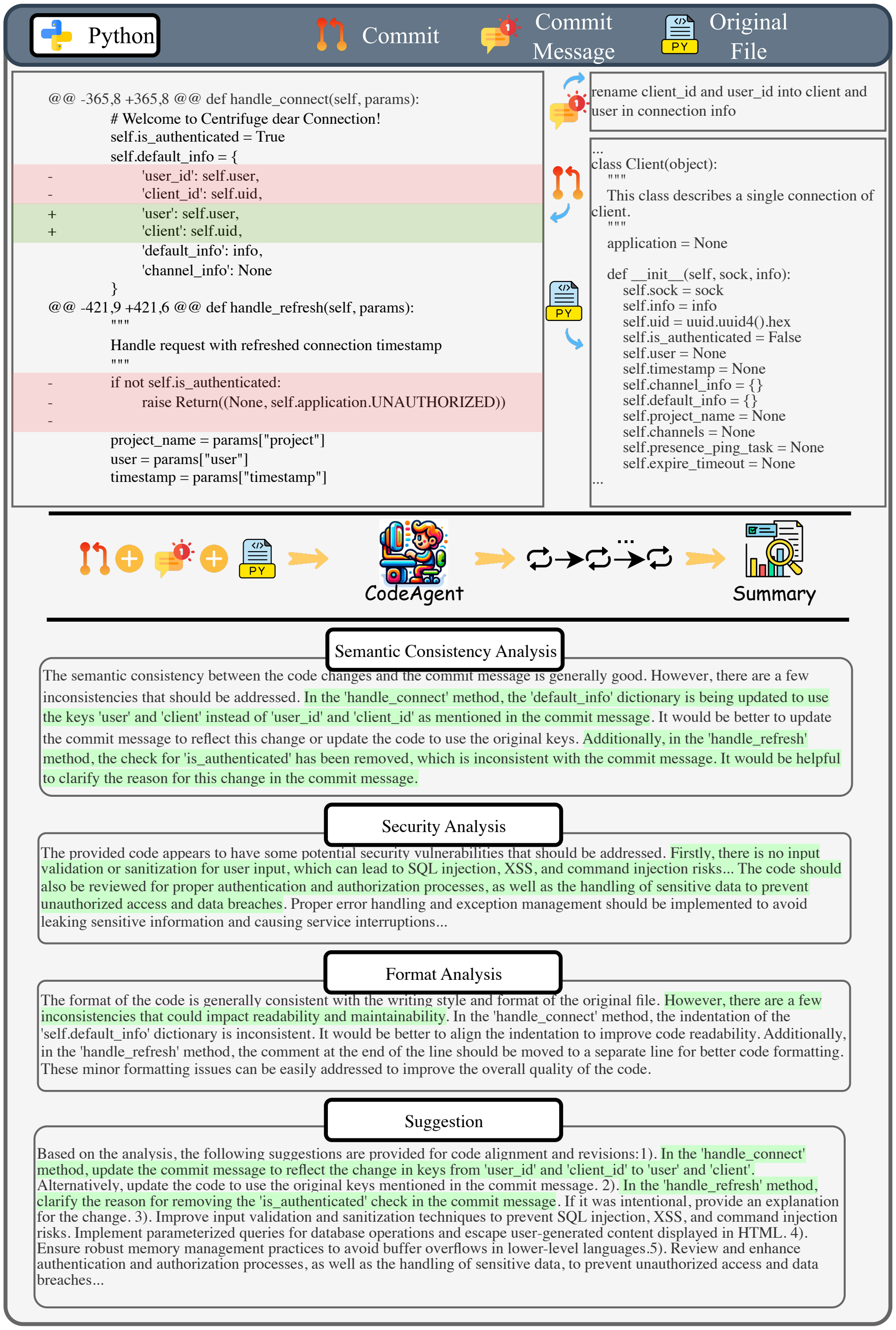


Figure 13: Example in Python project with sha value: “0d3b94bbd7528ed8649cdab4370a62e508bad82c"

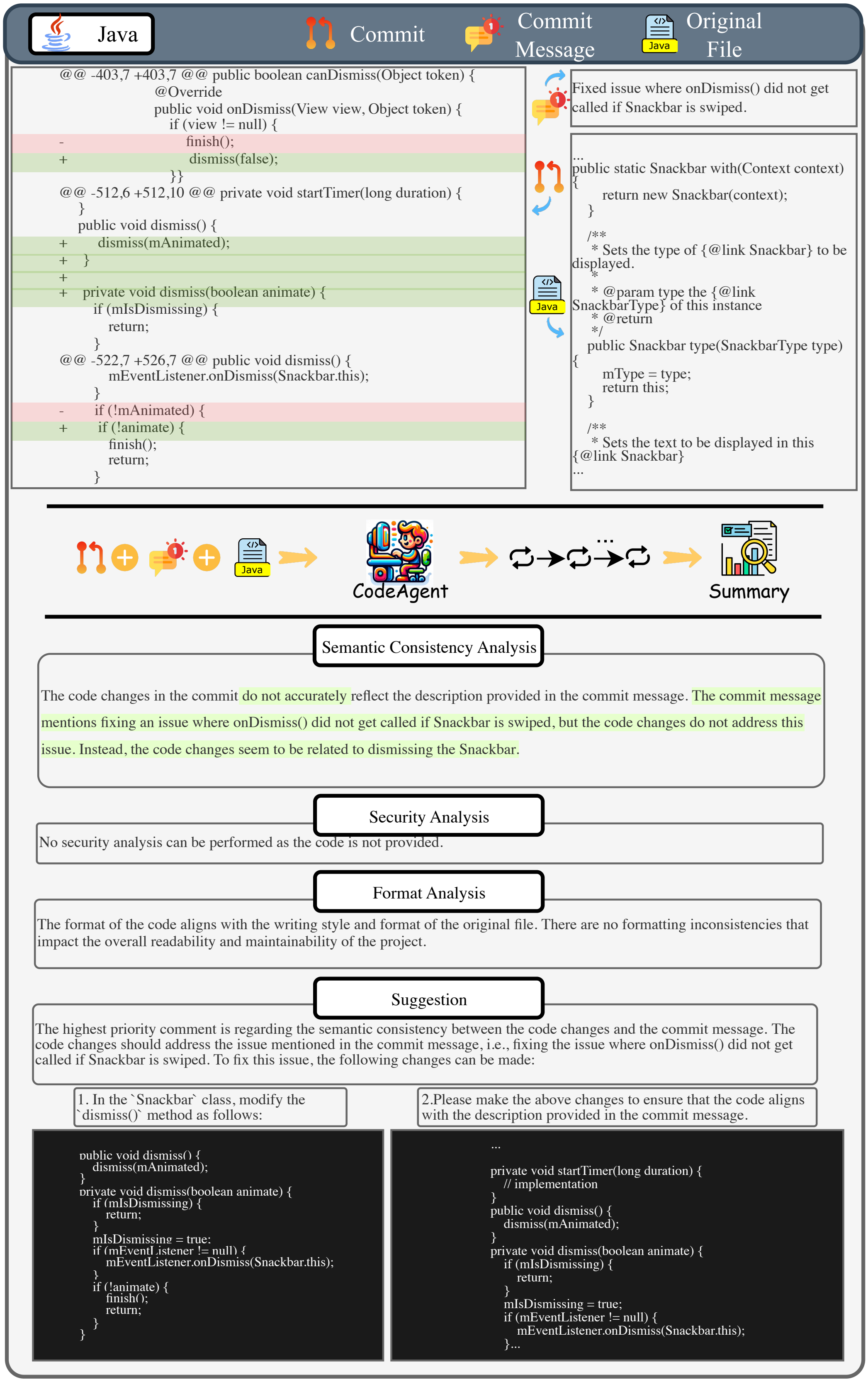


Figure 14: Example in Java project with sha value: “6e3c6d17d943f5ac70b421653eb167e0c34b119f"

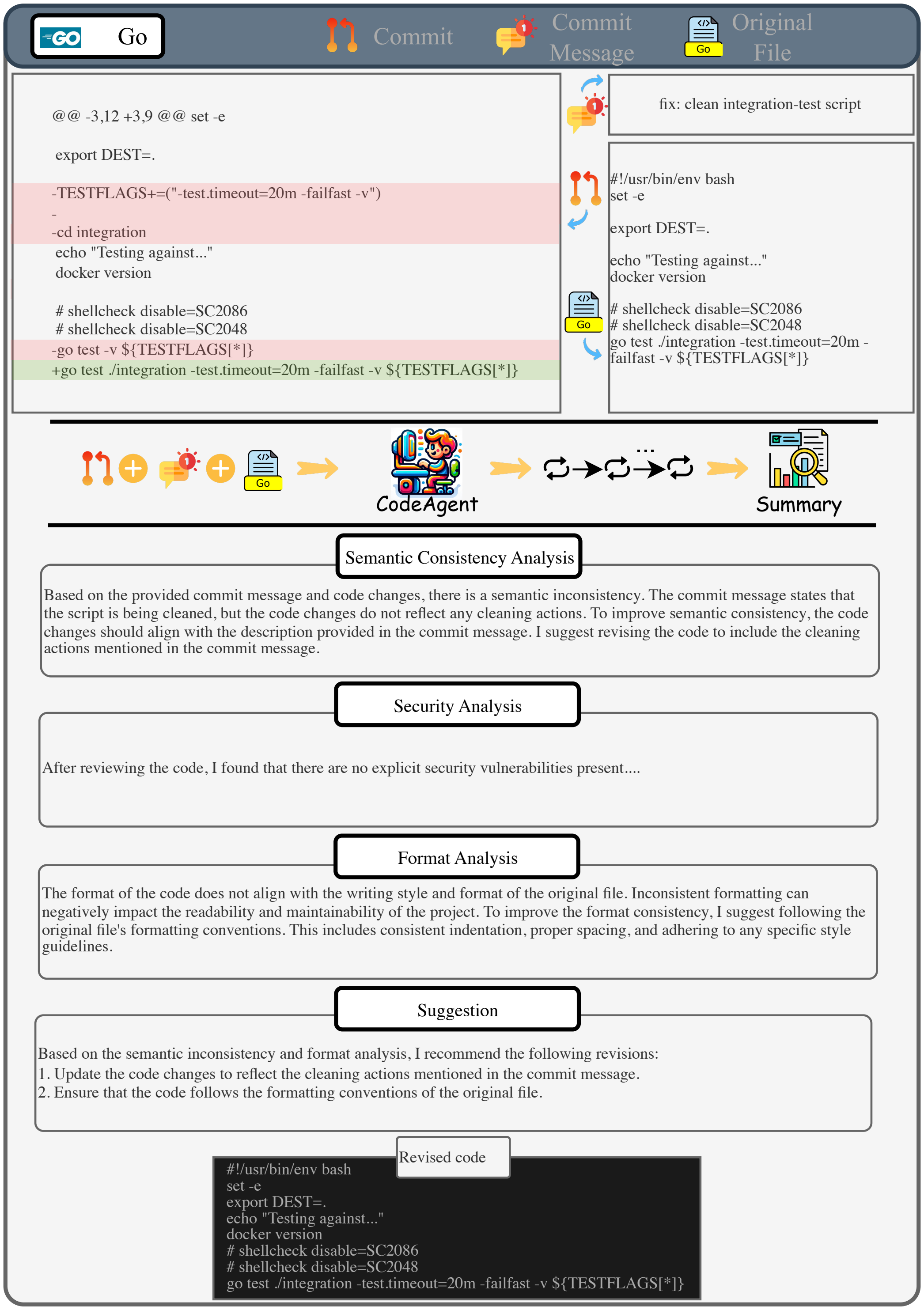


Figure 15: Example in Go project with sha value: “a6b00b097b1a8140bd8b60177247dc830374c8ed"

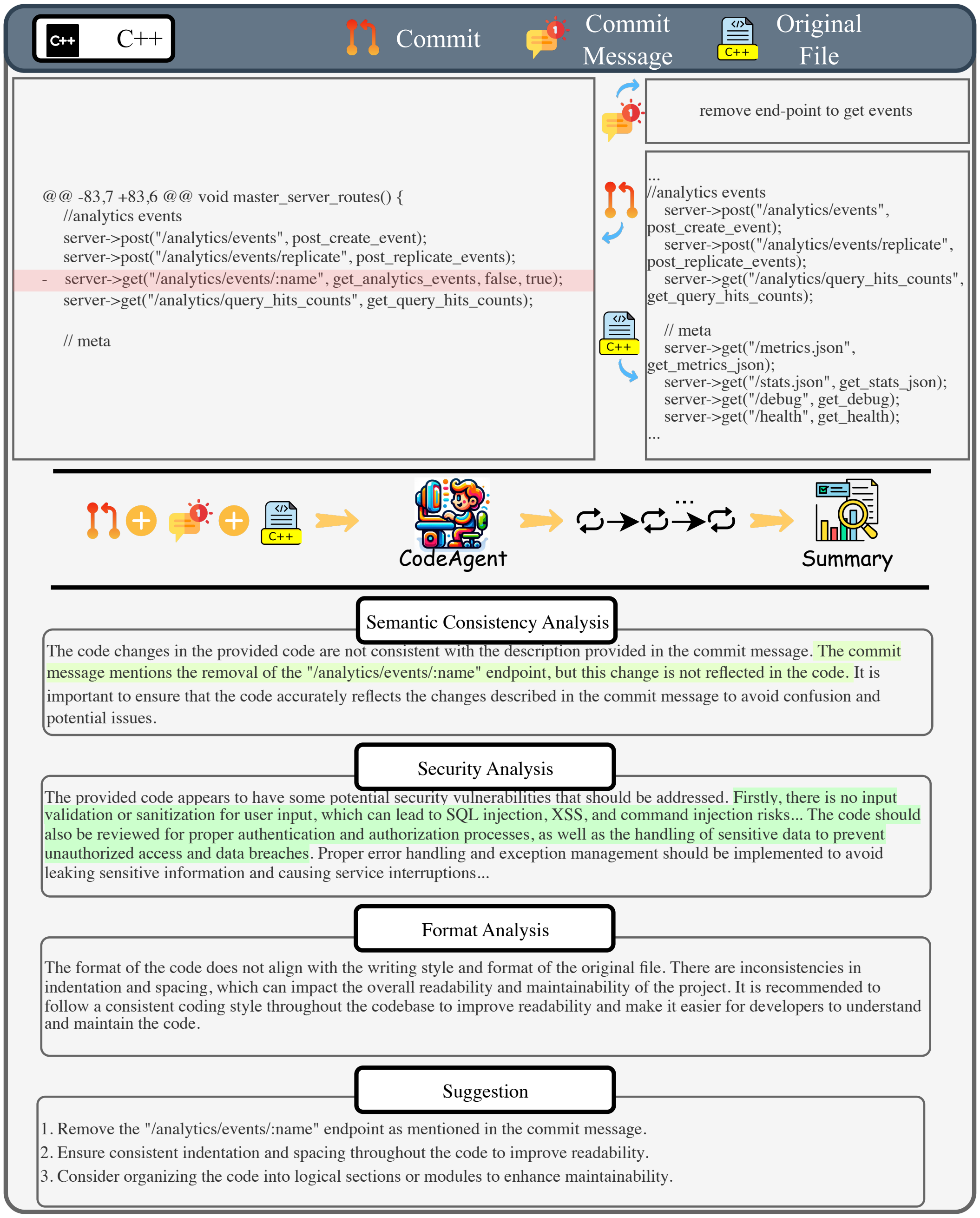


Figure 16: Example in C++ project with sha value: “09e88c7c20347cdf513323fa8c616aba0a1b9ea7"

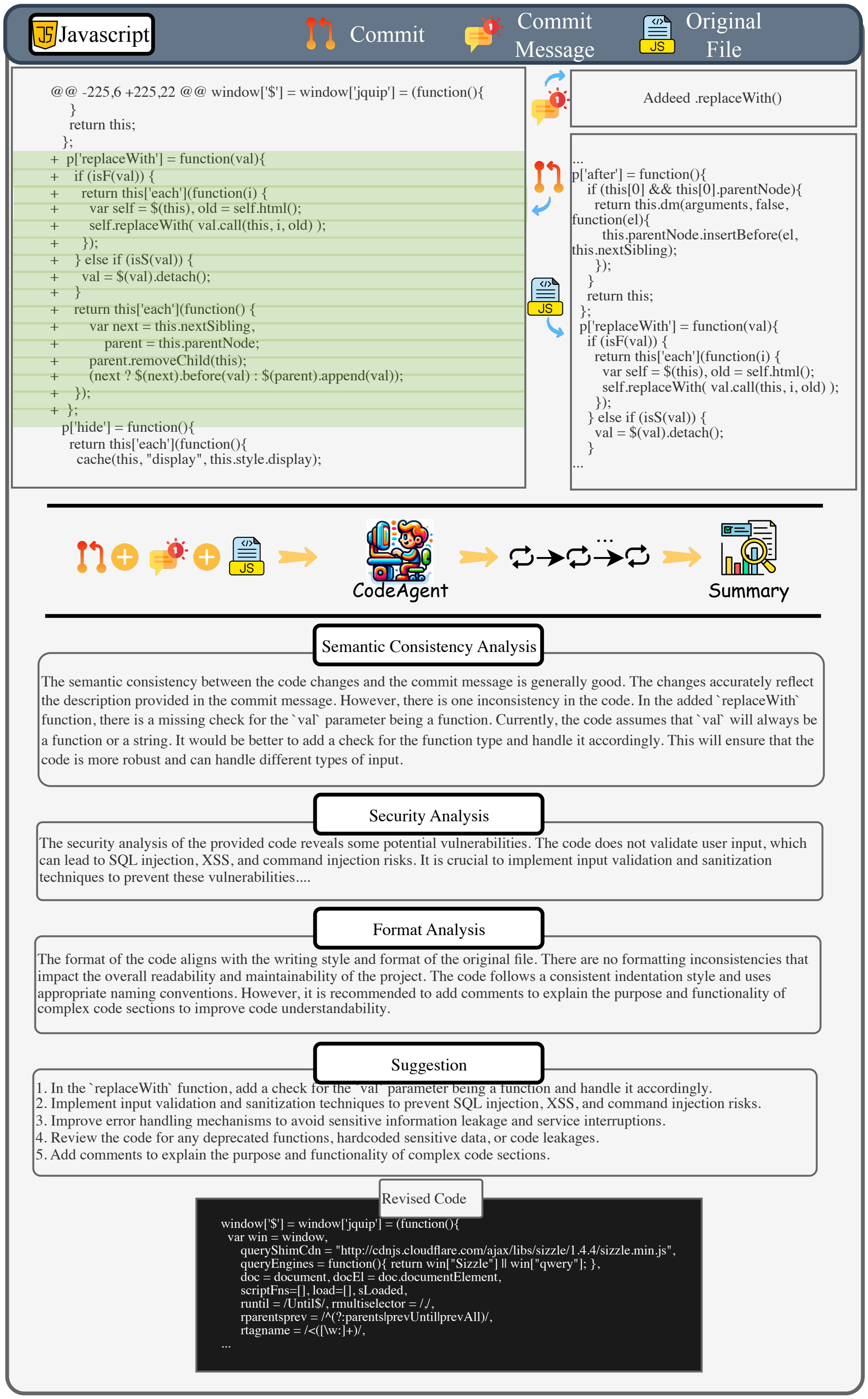


Figure 17: Example in JavaScript project with sha value: “6e9a80d93b470aef4978677ed0b7c9996fddbe20"

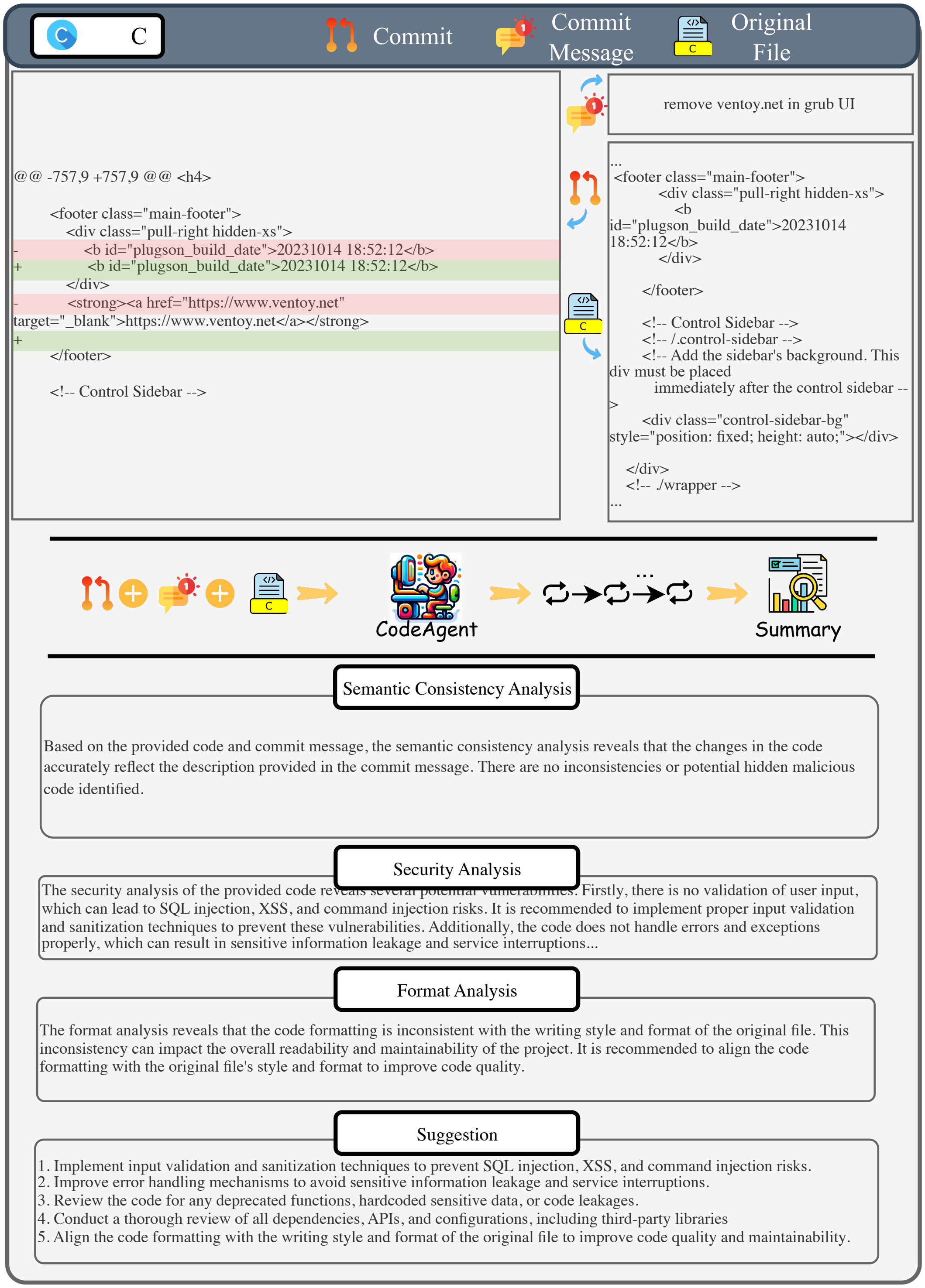


Figure 18: Example in C project with sha value: “5f93cf43dd6f809d7927abb79884285ad77e8a58"

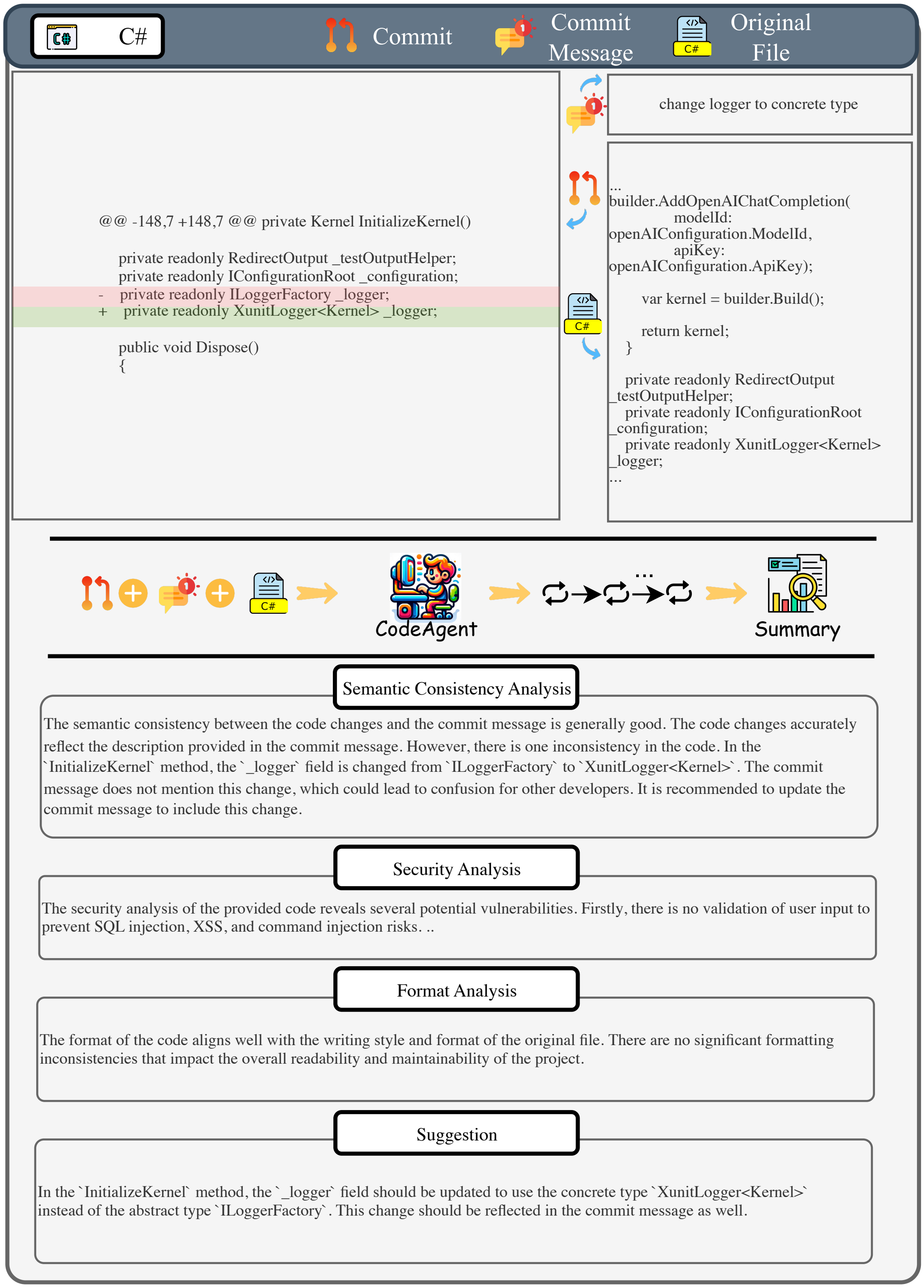


Figure 19: Example in C# project with sha value: “0e231c7a81b318e9eade972f7b877e66128ed67d"

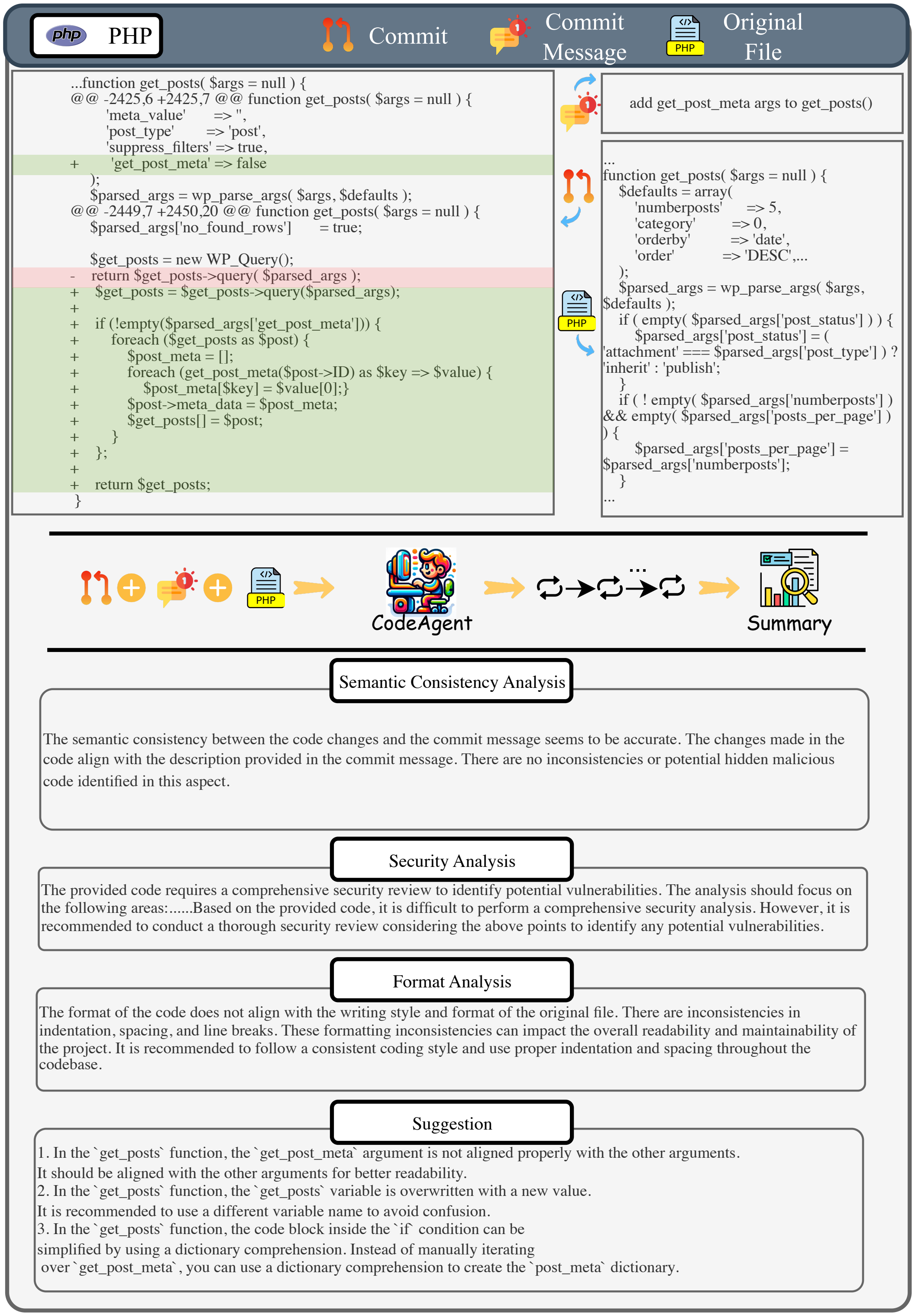


Figure 20: Example in PHP project with sha value: “6679f059b9a0972a90df198471188da652f5c547"

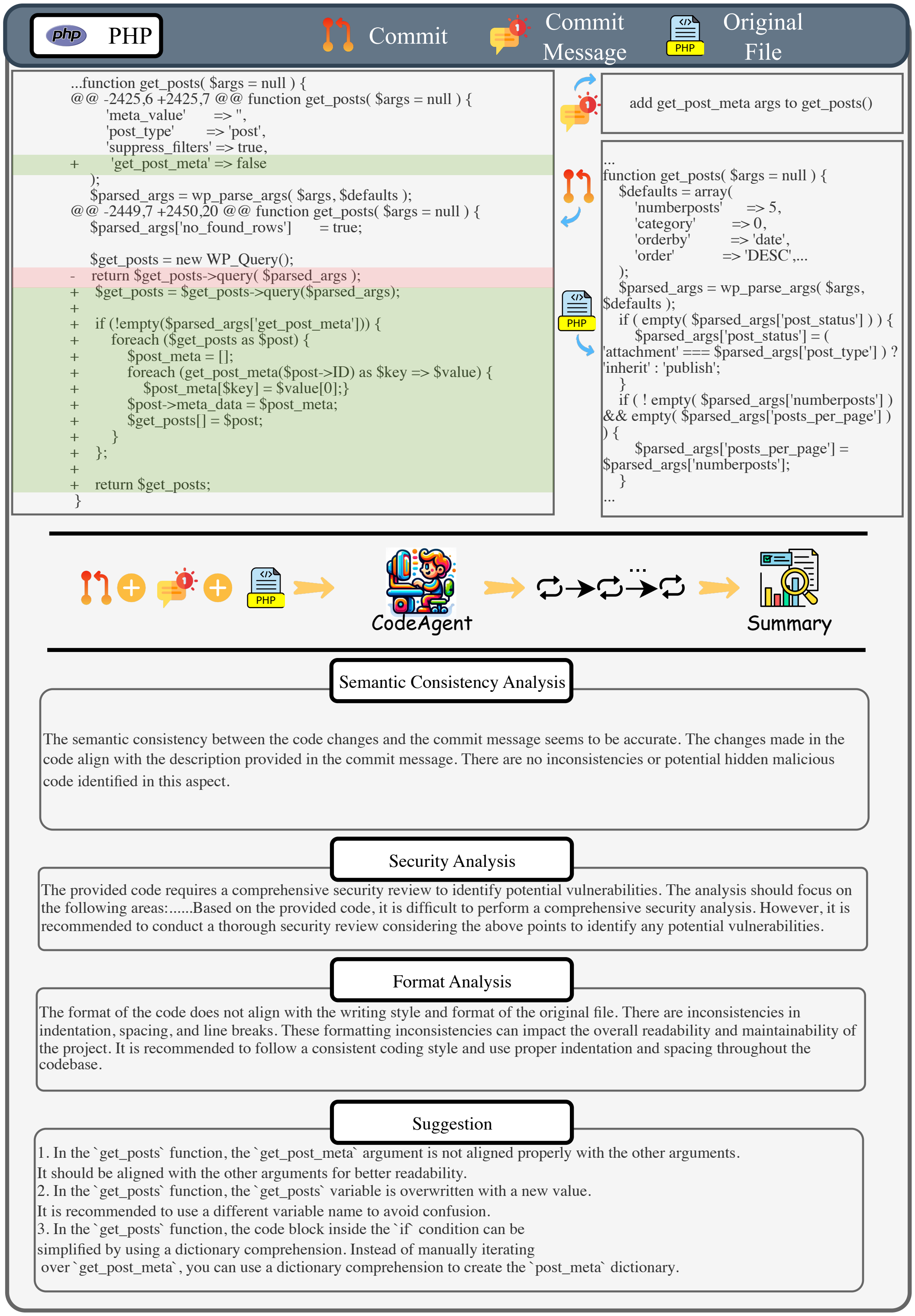


Figure 21: Example in Ruby project with sha value: “584f72e7f4c65066ccbd2183af76bf380b6eb974"

K.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, as **`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`.

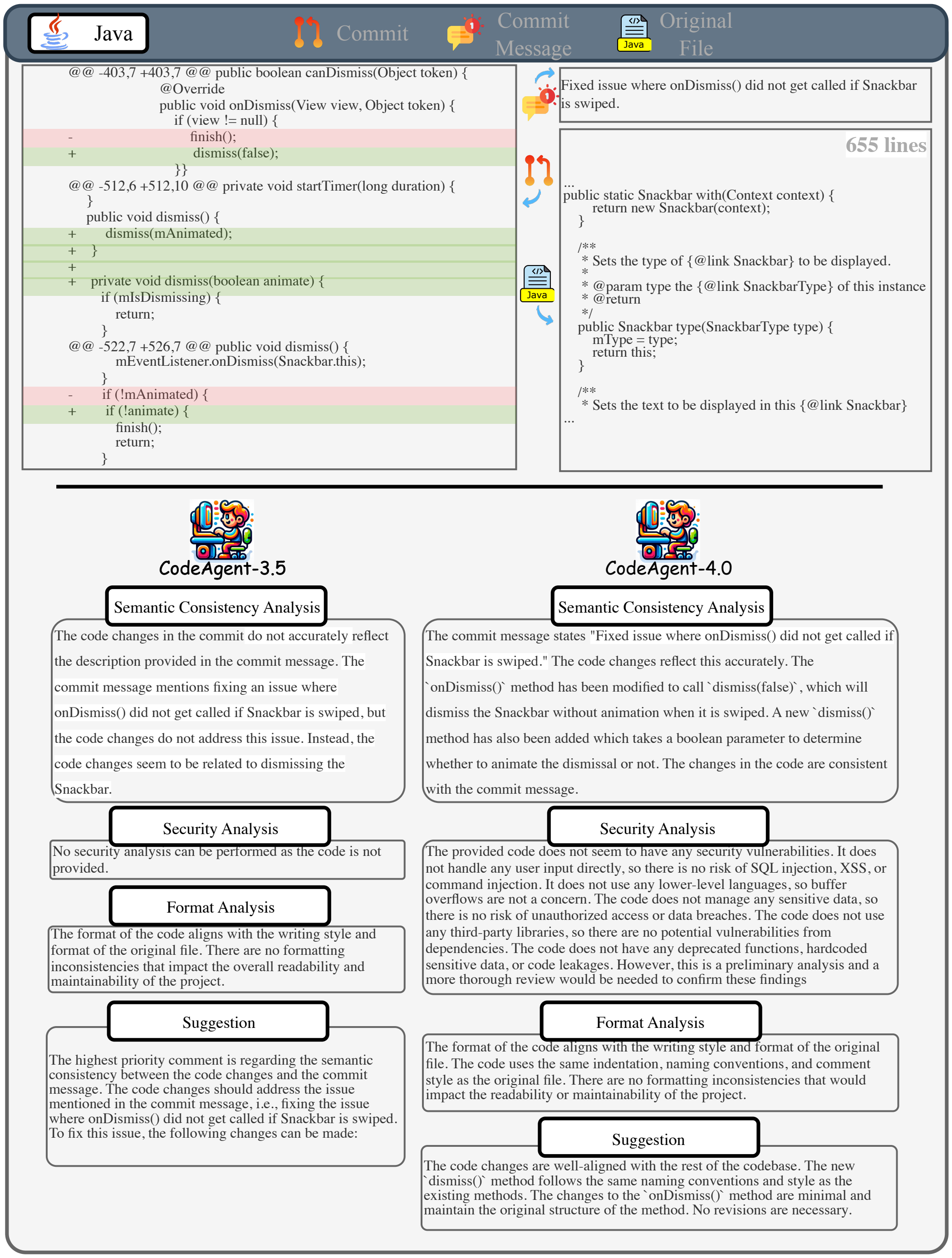


Figure 22: Comparison of code review between **CodeAgent**-3.5 and **CodeAgent**-4.0 on one Java project with sha value: "6e3c6d17d943f5ac70b421653eb167e0c34b119f"

L 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. ChatGPT). As shown in Section 5.1, **CodeAgent** outperforms baselines (a.k.a. CodeBERT, ChatGPT-3.5, ChatGPT-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 12 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 12 with those in Table 11, 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 12 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 12 generally indicate a lower proportion of confirmed vulnerabilities compared to Table 11, reflecting potentially different criteria or efficacy in vulnerability assessment. These variations highlight the impact of QA-Checker in **CodeAgent**.

Table 12: 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%

M 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`

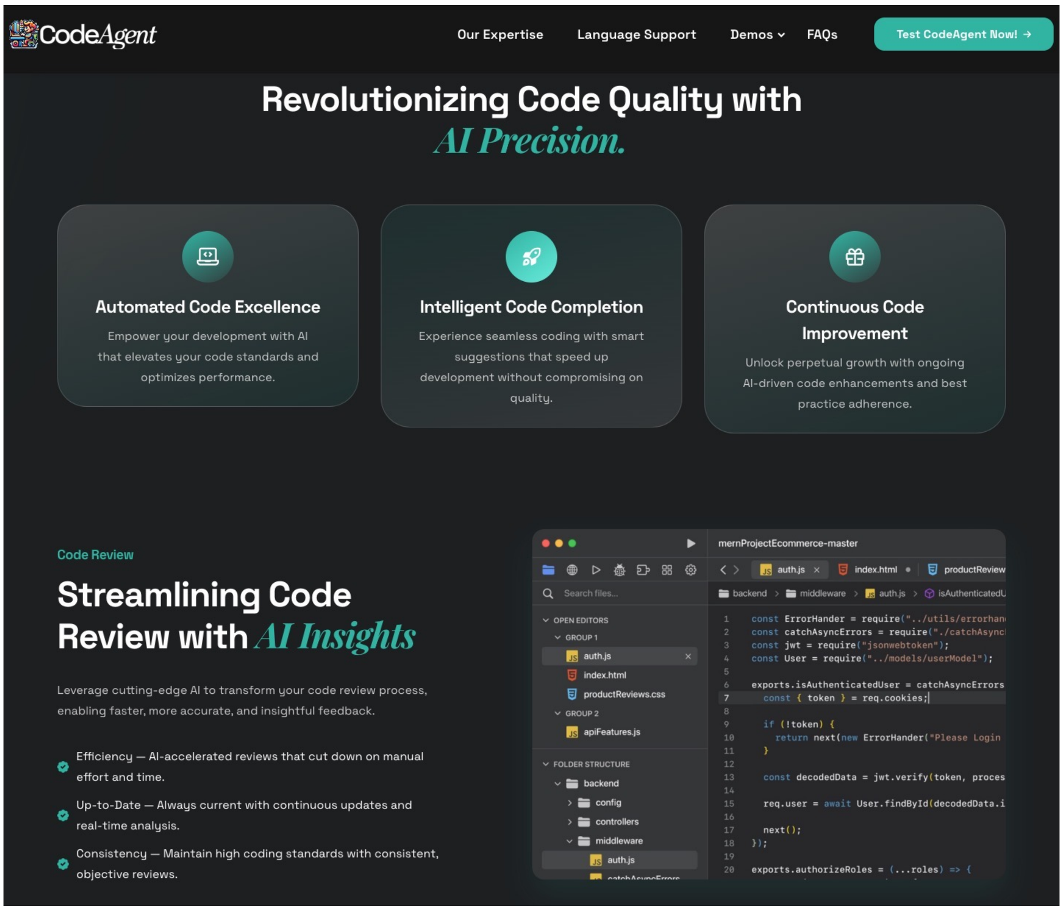


Figure 23: website of **CodeAgent**