Enhancing Code Quality with LLM in Software Static Analysis

Niu Zhi[1,2], Dong Luming[1,2]

(1. State Key Laboratory of Mobile Network and Mobile Multimedia Technology, Shenzhen 518055, China;
2. ZTE Corporation, Shenzhen 518057, China)

DOI: 10.12142/ZTECOM.202601009

https://kns.cnki.net/kcms/detail/34.1294.TN.20240725.1454.002.html, published online July 25, 2024

Manuscript received: 2024-04-18

Abstract: In the modern era of ubiquitous and highly interconnected information technology, cybersecurity threats stemming from software code vulnerabilities have become increasingly severe, posing significant risks to the confidentiality, integrity, and availability of modern information systems. To enhance software code quality, enterprises often integrate static code analysis tools into Continuous Integration (CI) pipelines. However, the high rates of false positives and false negatives remain a challenge. The advent of large language models (LLMs), such as ChatGPT, presents a new opportunity to address these challenges. In this paper, we propose AI-SCDF, a framework that utilizes the custom-built Nebula-Coder AI model for detecting and fixing code security issues in real time during the developer's personal build process. We construct a static code checking rule knowledge base through summarizing and classifying Common Weakness Enumeration (CWE) code security problems identified by security and quality assurance teams. The rule knowledge base is combined with CodeFuse-processed code contexts to serve as input for an AI code security detection microservice, which assists in identifying code quality and security issues. If any abnormalities are detected, they are addressed by an AI code security patching microservice, which alerts the developer and requests confirmation before committing the code into the repository. Experimental results show that our approach effectively improves code quality. We also develop a VSCode plugin for code alert detection and fix based on LLMs, which facilitates test shift-left and lowers the risk of software development.

Keywords: software static analysis; LLM; CWE; knowledge base

Citation (Format 1): Niu Z, Dong L M. Enhancing code quality with LLM in software static analysis [J]. ZTE Communications, 2026, 24(1): 65 – 71. DOI: 10.12142/ZTECOM.202601009

Citation (Format 2): Z. Niu and L. M. Dong. "Enhancing code quality with LLM in software static analysis," *ZTE Communications*, vol. 24, no. 1, pp. 65 – 71, Mar. 2026. doi: 10.12142/ZTECOM.202601009.

1 Introduction

Static Application Security Testing (SAST) is a technique that analyzes source code to detect software vulnerabilities. The general architecture of SAST involves three stages: preprocessing, analysis, and detection. In the first stage, SAST tools use compilers to preprocess source code and generate intermediate files. Based on these files, they then perform lexical analysis and generate an abstract syntax tree (AST). In the second stage, SAST tools analyze the AST to conduct data flow analysis, control flow analysis, dependency analysis, interval analysis, and pointer analysis, thereby obtaining mathematical representations of the program. Finally, in the third stage, SAST tools solve constraints using these mathematical expressions and well-known vulnerability patterns from a database to achieve high-precision detection of code security issues.

State-of-the-art SAST tools, such as Coverity[1], Klocwork[2], Infer[3], SonarQube[4], and CodeQL[5], play a crucial role in code security assessment. Therefore, many enterprises must choose appropriate SAST tools for their Continuous Integration (CI) process. However, for commercial software, achieving high-level static code analysis often requires compiling source code. This adds time overhead to enterprise CI builds. Conversely, open-source tools are mostly rule-based and thus prone to generating false warnings. This increases the cost of eliminating alerts and reduces software productivity.

With the development of machine learning (ML), researchers have begun incorporating it into SAST to detect vulnerabilities in source code. Harer, Kim, et al. proposed a data-driven ML-based approach for automated software vulnerability detection[6]. Li, Zou, et al. developed VulDeePecker, a deep learning-based vulnerability detection system for detecting vulnerabilities in software code[7 – 8]. Wang, Liu, et al. utilized Deep Belief Networks (DBNs) to automatically learn code semantic features from AST to predict code defects[9]. Li, He et al. extracted markings from program ASTs and applied convolutional neural networks to detect software defects[10]. Experimental results demonstrate that pre-trained ML models significantly improve the effectiveness of detecting and fixing program vulnerabilities.

Since OpenAI released GPT-3[11] in 2020, large language models (LLMs) have entered a phase of rapid development. Early and prominent examples include Google's PaLM2[12], Microsoft's Copilot[13], and Meta's open-source LLaMa[14]. These LLMs are widely used in the software development lifecycle (SDLC), laying the foundation for subsequent advances in AI-assisted programming. The superior ability of LLMs to understand complex code makes them capable of providing infinite possibilities in improving code defect detection and self-fix. Berabi, Gronskiy, et al. fine-tuned GPT-4 with data on code contexts, defect reports, and code snippets to achieve automatic software vulnerability fix[15]. Wadhwa and Prdadhan used a pair of LLMs to improve code quality through software code quality improvement and to rank the improvements, respectively[16]. Li, Hao, et al. developed the LLift fully automatic framework by combining static analysis with LLMs to detect use-before-initialization defects[17].

These findings show that combining LLMs (such as GPT-4) with static analysis can significantly improve code quality and provide new ideas for automated code fix. However, these studies also highlight the limitations of LLMs' ability to understand and handle complex code logic, which requires further research and practice.

In light of this, we propose AI-SCDF, a framework that utilizes the Nebula-Coder LLM developed by ZTE to detect and fix code changes during the developers' personal build process. AI-SCDF does not require compiling the source code for defect detection. Instead, it analyzes the contextual information of the code to facilitate LLM learning and reasoning. This significantly enhances code checking efficiency while overcoming the limitations of commercial static code analysis tools. Moreover, it addresses the high false alarm rate issue prevalent in open-source static code analysis tools. First, we summarize the Common Weakness Enumeration (CWE) code security issues based on rules from security and quality teams to build a static code checking rule knowledge base. Second, this knowledge base, combined with processed code context, serves as input to an AI code flaw detection microservice, which intercepts potentially flaw code. Once potential flaw code is detected, we employ the AI code flaw fix microservice to rectify it, and feedback is provided to the developer to confirm the improvement in coding quality and prevent flaw code from being committed to the code repository. We mainly use the Juliet Test Suite[18] to evaluate and verify the flaw detection and fix capabilities of AI-SCDF.

The main contributions of the paper are summarized as follows:

1) Static analysis checker knowledge base: We developed a powerful static analysis checker knowledge base system that covers CWE vulnerability types for various programming languages and includes a large number of violating cases and fixing samples. These cases are derived from open-source code and real-world closed-source code flaws found during continuous integration. The sample data serve as input for LLM learning.

2) Code context analysis: We utilized control flow analysis tools to extract the context of code changed by developers. This enables LLMs to better understand software structure while avoiding the high token cost of processing entire codebases.

3) LLM utilization: We developed AI-SCDF, a code flaw detection and fix framework based on LLMs, and integrated it into personal build processes to improve code quality and reduce software development risks.

4) Results: Through testing on the Juliet Java 1.3 test set and analyzing metrics including true positives (TP), true negatives (TN), false positives (FP), false negatives (FN), accuracy, precision, and recall, we found that AI-SCDF effectively improved code quality and reduced the cost of alert fixing.

2 Overview

Our objective is to develop a fully automated framework that detects and fixes defects during the developer's personal build process to improve code quality and reduce the labor costs of the R&D team. To this end, our approach leverages the context of changed code and a static analysis checker knowledge base to construct prompts that guide LLMs in code flaw detection. Subsequently, information about any detected defective code is combined with these detection prompts to form code flaw fix prompts, which then guide the LLMs to perform automatic code fix.

Consider the Java code snippet shown in Fig. 1, where the resource objects named pstmt and rs are not properly closed. This may cause resource leakage until the resources are exhausted, leading to software quality issues. In actual operations, such resources are typically managed using a try-catch-finally block to ensure they are properly closed.

As shown in Fig. 2, the AI-SCDF workflow involves the following stages:

1) Static analysis checker knowledge base: The knowledge base includes checkers, checker descriptions, severity levels, code flaw violations and corresponding fixed examples for each checker across different programming languages, as well as custom severity ratings. This data is derived from coding best practices, coding standards, commercial and open-source static analysis tools, along with a large number of data sources and cases. By integrating CWE information, a unified static analysis checker knowledge base is formed. The data in this knowledge base are used by subsequent AI microservices to

```
@@ -0,0 +1,11 @@
+ public  List<String> runQuery(Connection conn,String kb) throws SQLException{
+    List<String> examples = new ArrayList<String>();
+    PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM knowledge WHERE kb = ?");
+    pstmt.setString(1,kb);
+    ResultSet rs = pstmt.executeQuery();
+    while(rs.next()){
+      examples.add(rs.getString("example"));
+    }
+    return examples;
+
+  }
```

Figure 1. Flaw code of resource leak

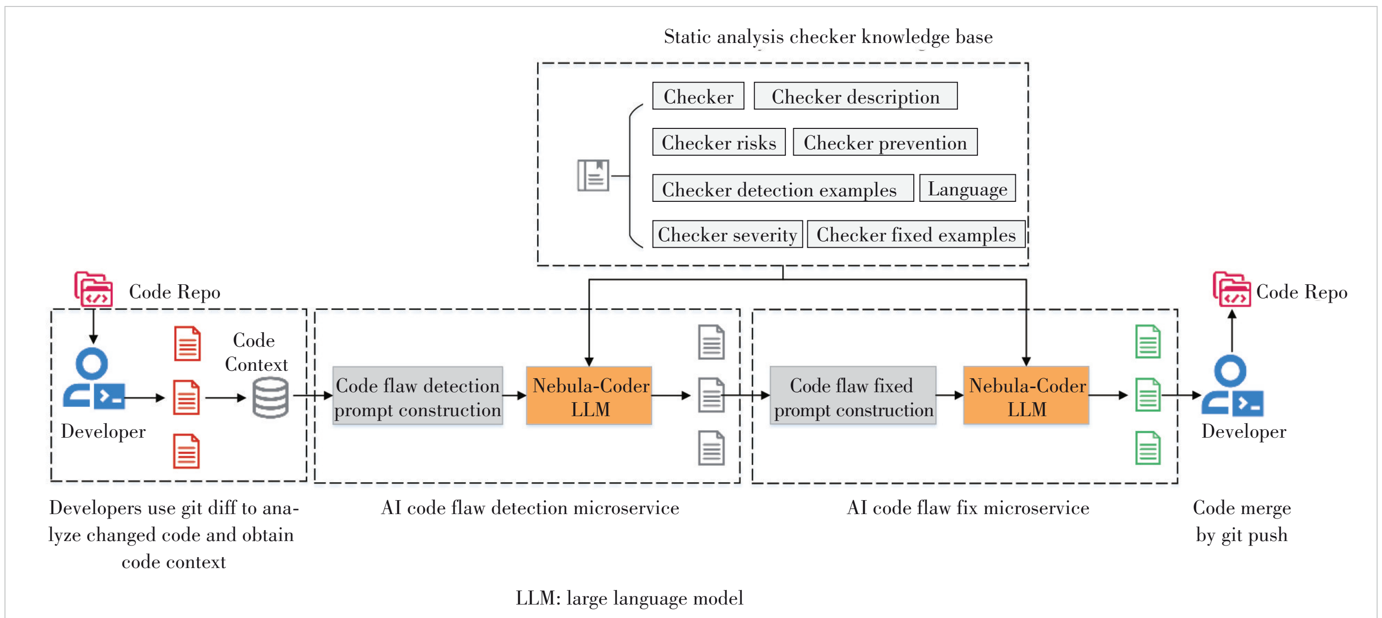

Figure 2. Overview of the AI-SCDF framework and its core workflow

construct prompts, particularly for generating examples of violations and fixes. This process enhances the learning capability of LLMs and reduces the likelihood of generating hallucinations.

2) Code change context analysis: Using git commands and program analysis methods, the context of the changed code is obtained, including all functional code segments in its control flow. This decreases the number of tokens in the LLM prompts, thereby reducing computational costs.

3) AI code flaw detection microservice: Prompts are automatically constructed based on the context of the changed code and the detailed checkers in the static analysis checker knowledge base. The LLM API is then invoked to detect flaws in the changed code. These prompts include code snippets of changed functions or methods, relevant code context, descriptions of static checking rules, and examples of violations. This microservice is also effective in identifying false alarms reported by commercial code checking tools, thereby improving developers' efficiency in addressing alerts.

4) AI code flaw fix microservice: Prompts are automatically constructed based on the context of the changed code, the identified defective code, and the detailed rules in the static analysis checker knowledge base. The LLM API is then invoked to fix defective code. These prompts include code snippets of the changed functions or methods, relevant code context, the flaw code, descriptions of static checking rules, and fix cases.

3 Design

This section describes the detailed design of AI-SCDF based on its architecture and workflow.

3.1 Static Analysis Checker Knowledge Base

The design of the static analysis checker knowledge base involves the following steps. First, CWE-IDs are extracted from the CWE Information Library. CWE is an open-source vulnerability classification system that provides a common language and numeric identifiers for software weaknesses. By parsing the CWE Information Library, we obtain CWE-IDs for various programming languages. Next, these CWE-IDs are fused with data crawled from network resources related to code flaws. These resources include checker descriptions from commercial tools, Common Vulnerabilities and Exposures (CVE) cases related to code, coding standards, the Secure Coding Practices published by the Open Web Application Security Project (OWASP) and publicly available code flaw datasets for machine learning (e.g., from GitHub). These documents may exist in different formats (e.g., HTML, PDF, Word), which are not convenient for data fusion and aggregation. Therefore, all data are converted into a unified format, such as JSON or XML, for subsequent storage and querying. The normalized data are then stored in the Knowledge Base Storage Service, which manages all static analysis checker knowledge, including checking rules, checker descriptions, risk levels, code flaw violations, and corresponding fix cases, as well as custom severity ratings for each checker across different programming languages. The entire process is illustrated in Fig. 3.

The format of a data record in the static analysis knowledge base is shown in Fig. 4. Data are stored using Markdown syntax, as this format is more user-friendly for LLMs when processing prompts that follow popular Markdown styles.

3.2 Code Change Context Analysis

The code change context analysis is designed to obtain the

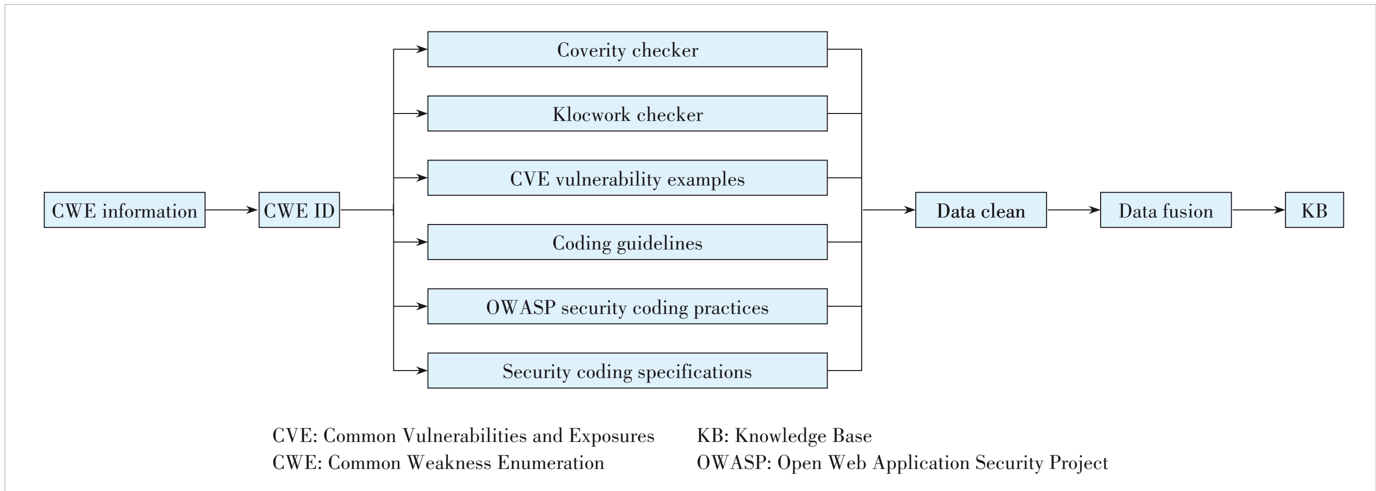

Figure 3. Flow chart of static analysis checker knowledge base

````
Knowledge Base Example
# Checker
  CWE89_SQL_Injection_SV.SQL
# Checker Description
  This Checker detects the situation when unvalidated or tainted data is used directly in an SQL query string.
# Language
  Java
# Risks
  SQL injections put data in the database at risk. Since unvalidated user input is being used in the SQL
  statement, an attacker can inject any SQL statement they wish to execute. This includes deleting, updating
  or creating data. It may also be possible to retrieve sensitive data from the database with this type of
  vulnerability. If the command is used for authentication, it will lead to unauthorized access.
# Prevention
  Prevent SQL injection flaws by validating any and all input from outside the application (user input, file input,
  system parameters, etc.). Validation should include length and content. Typically only alphanumeric
  characters are needed (i.e., A-Za-z, 0-9). Any other accepted characters should be escaped.
# Detect Examples
 ```java
  public ResultSet getUserData(ServletRequest req, Connection con) throws SQLException {
      String accountNumber = req.getParameter("accountNumber");
      String query = "SELECT * FROM user_data WHERE userid = '" + accountNumber + "'";//POTENTIAL FLAW
      Statement statement = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_READ_ONLY);
      ResultSet results = statement.executeQuery(query);
      return results;
    }
  ```
# Fixed Examples
  ```java
  public ResultSet getUserData(ServletRequest req, Connection con) throws SQLException {
   String accountNumber = req.getParameter("accountNumber");
   String query = "SELECT * FROM user_data WHERE userid = ?";
   PreparedStatement statement = con.prepareStatement(query);
   statement.setString(1, accountNumber);
   ResultSet results = statement.executeQuery();
   return results;
 }
  ```
````

Figure 4. Example data record illustrating the Markdown format of the static analysis checker knowledge base

contextual information of changed code for LLM learning. Its workflow is shown in Fig. 5. First, we obtain the information about code changes using git diff operations and parse it with a scripting language to extract the absolute file paths and the corresponding source code of the changed functions. Next, we trigger a call graph analysis tool on the local repository to analyze the call chains related to the changed code, and obtain the call chain related to the changed code based on the file paths and function names obtained in the previous step. As shown in this figure (fun_1->fun_8->fun->fun_9->fun_11->fun_12), each function in this call chain is then parsed to obtain the contextual code information relevant to the changed code.

3.3 AI Code Flaw Detection Microservice

The AI code flaw detection microservice constructs detection prompts based on the code context, the changed code functions, and the static analysis checker knowledge base. The format of the AI code flaw detection prompt template is shown in Fig. 6. These prompts include the checker name, checker description, risk ratings, examples of violations, code context, and changed code information. After the prompts are constructed, the LLM API is invoked for analysis. We employ the Nebula-Coder LLM, developed by ZTE using its proprietary AI technology. This model is a fine-tuned version of a foundation model to meet the needs of the code flaw detection service. The output of the LLM model determines whether the changed code violates the current checking rules. This microservice is designed to enhance code quality and proactively identify and rectify potential code flaws, thereby improving software development efficiency and quality.

3.4 AI Code Flaw Fix Microservice

The AI code flaw detection microservice produces a boolean output after modification: a value of 1 indicates that flaws exist in the changed code, while 0 signifies no flaws. If no flaws are detected (0), the fix process is skipped, and the entire AI-SCDF call process ends. If flaws are detected (1), the AI code flaw fix microservice is invoked to rectify them. The prompts for the code flaw fix service are also constructed using the same context and knowledge base as before. The fix prompt template (Fig. 7) includes the checker description, prevention guidance, fixed examples, buggy code, and code context. Finally, the fixed code with the related flaw is output for developers to confirm acceptance. Since LLMs may exhibit hallucination, it is necessary to have the fixed code reaffirmed by developers or quality assurance personnel. If accepted, the fixed code can be directly merged into the code repository.

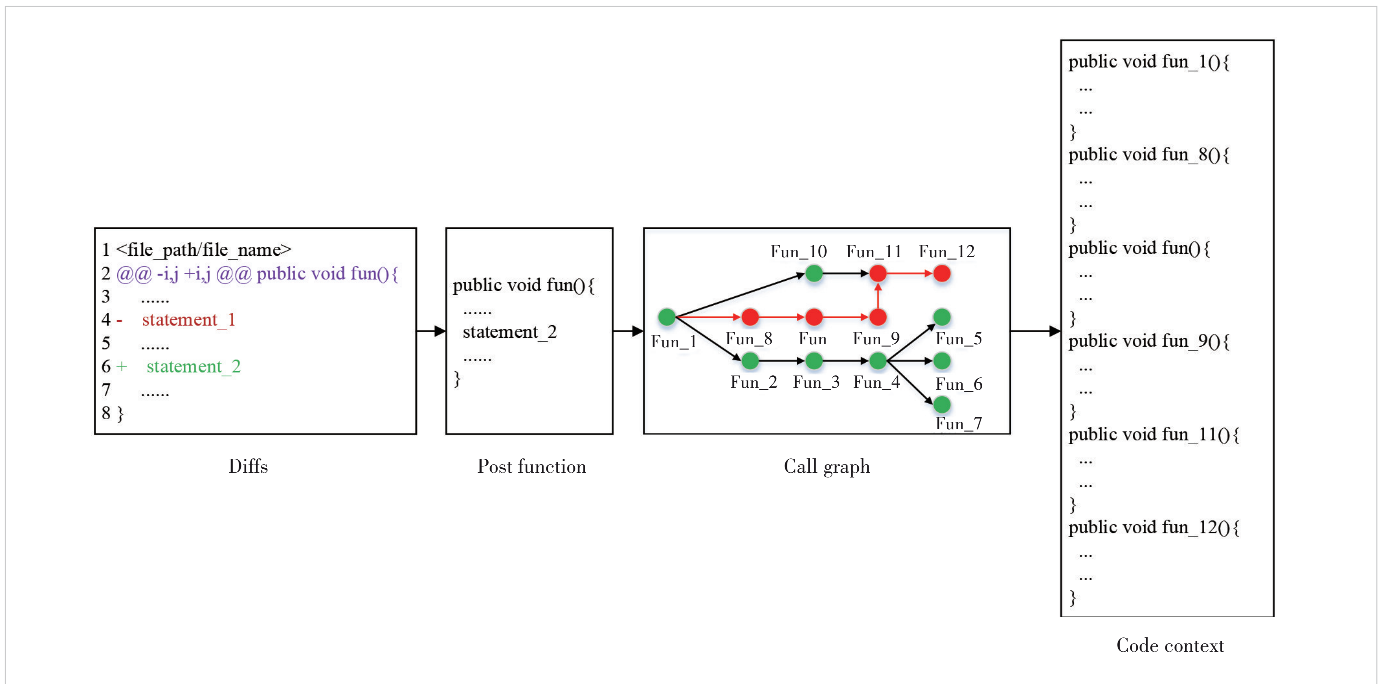

Figure 5. Workflow of code change context analysis

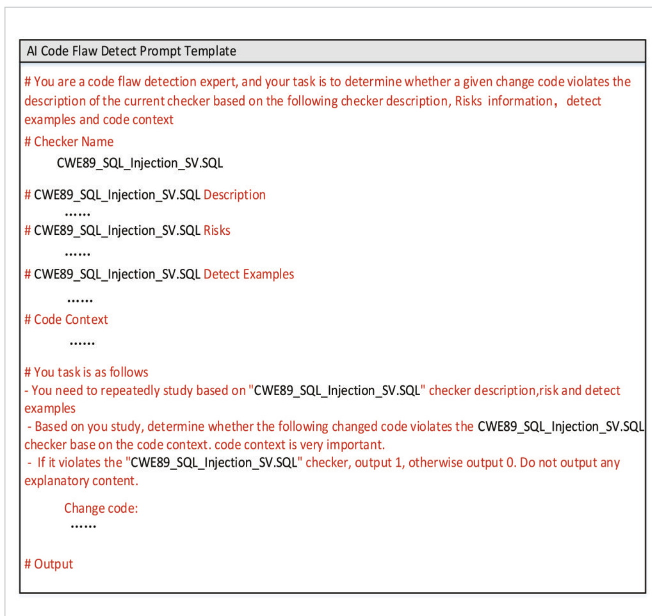

Figure 6. AI code flaw detection prompt template

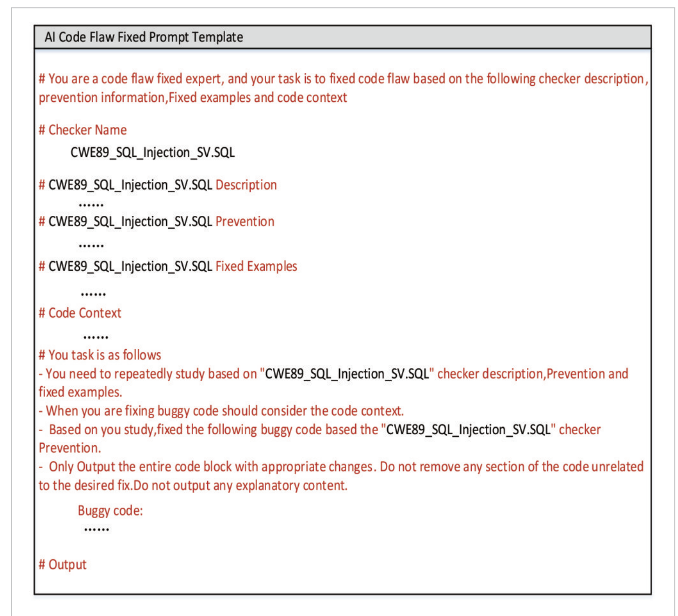

Figure 7. AI code flaw fixed prompt template

3.5 Nebula-Coder LLM

The Nebula-Coder LLM is dedicated to assisting developers in requirement analysis, product design, coding, testing, and deployment. It employs a vast amount of high-quality domain data, knowledge accumulation, and over 100 million technical documents and one trillion tokens of code corpus from ZTE's extensive experience in the telecommunications field. The model is pre-trained incrementally using parallel training frameworks. Since its launch in April 2023, the daily active users of Nebula-Coder have reached 120 000, with a code acceptance rate ranging from 40% to 45%, resulting in a 30% increase in coding efficiency and a 10% overall improvement in development efficiency. AI-SCDF utilizes the Nebula-Coder LLM for code defect detection and fix, enabling seamless integration with the public API provided by Nebula-Coder LLM.

4 Experimental Setup and Evaluation

The experimental dataset was the Juliet Java version 1.3 test suite, created by the NSA's Center for Assured Software (CAS). Juliet covers 112 different CWE code defect cases and is often used to evaluate the effectiveness of static code checking tools. Each test case includes its related code flaws and corresponding fixed examples. We employed the Nebula-Coder LLM, which was fine-tuned from a foundation model to meet the needs of experimental verification.

First, we deployed the AI code flaw detection microservice to detect 10 representative CWE categories: CWE-23, CWE-78, CWE-89, CWE-315, CWE-325, CWE-190, CWE-398, CWE-382, CWE-760, and CWE-369. The model performance was evaluated using TP, FN, TN, FP, Accuracy, Precision and Recall as evaluation metrics. The experimental results are shown in Table 1. In this table, TP denotes cases where a defective code sample is correctly identified as flawed, TN represents correct identification of a non-defective code sample, FP indicates non-defective samples incorrectly flagged as flawed, and FN represents defective samples incorrectly classified as non-flawed. Accuracy is the proportion of correct samples predicted by the model, calculated as (TP+TN)/(TP+TN+FP+FN). Precision indicates the correctness of positive sample predictions, computed as TP/(TP + FP). Recall measures the coverage rate of positive sample predictions, calculated as TP/(TP + FN). The experimental results indicate that AI-SCDF achieves higher accuracy in detecting standard coding defect categories, such as CWE-315, CWE-325, CWE-398, CWE-382, and CWE-760, with accuracy rates of 98.4%, 91.2%, 90.5%, 100%, and 100%, respectively. This is because these standard code flaw categories require only attention to coding standards and simple handling, making them easier for LLMs to handle. However, for more complex code flaws like CWE-369, detailed contextual analysis of code snippets is required. This places greater demands on the LLM's learning and reasoning capabilities, resulting in lower accuracy. Further tuning of the LLM is needed to improve the accuracy of code defect detection.

Subsequently, we rectified the code flaw alerted by the AI code flaw detection microservice to verify its defect fix capability. The results are shown in Table 2. In this table, Accept means that the developer confirmed both the presence of a code flaw and its correct fixation, Reject indicates either no code flaw was found or the code flaw was not corrected properly, and Acceptance Rate represents the proportion of accepted fixes among all detected flaws. As shown in Table 2, the AI code flaw fix microservice performs well for standard code defect categories (CWE-315, CWE-325, CWE-398, CWE-382, CWE-760). However, for complex alerts like CWE-369, the LLM's hallucination or reasoning limitations adversely affect the fix capability, resulting in lower fix rates. Future work will focus on optimizing the LLM to improve the code flaw rectification capability and ensure code security.

Table 1. Experimental results of AI code flaw detection microservice on Juliet Java

CWE-ID	CWE Description	Testcase Path	Flaw Sum	Not Flaw Sum	TP	FN	TN	FP	Accuracy/%	Precision/%	Recall/%
CWE-23	Relative path traversal	Java/src/testcases/CWE23_Relative_Path_Traversal	444	276	378	66	202	74	80.6	83.6	85.1
CWE-78	OS command injection	Java/src/testcases/CWE78_OS_Command_Injection	444	276	363	81	171	105	74.2	77.6	81.8
CWE-89	SQL injection	Java/src/testcases/CWE89_SQL_Injection/s01	592	384	451	141	281	103	75	81.4	76.2
CWE-315	Plaintext storage in cookie	Java/src/testcases/CWE315_Plaintext_Storage_in_Cookie	37	24	37	0	23	1	98.4	97.4	100
CWE-325	Missing required cryptographic step	Java/src/testcases/CWE325_Missing_Required_Cryptographic_Step	34	0	31	3	0	0	91.2	100	91.2
CWE-190	Integer overflow	Java/src/testcases/CWE190_Integer_Overflow/s01	592	384	416	176	303	81	73.4	83.7	70.3
CWE-398	Poor code quality	Java/src/testcases/CWE398_Poor_Code_Quality	137	0	124	13	0	0	90.5	100	90.5
CWE-382	Use of system exit	Java/src/testcases/CWE382_Use_of_System_Exit	34	0	34	0	0	0	100	100	100
CWE-760	Predictable salt one way hash	Java/src/testcases/CWE760_Predictable_Salt_One_Way_Hash	17	0	17	0	0	0	100	100	100
CWE-369	Divide by zero	Java/src/testcases/CWE369_Divide_by_Zero/s01	592	384	434	158	276	108	72.3	80.1	73.3

CWE: Common Weakness Enumeration FN: false negative FP: false positive SQL: Structured Query Language TN: true negative TP: true positive

Table 2. Accuracy of the AI code flaw fix microservice for Juliet Java

CWE-ID	CWE Description	Testcase Path	Confirm Flaw	Accept	Reject	Accepted Ratio
CWE-23	Relative path traversal	Java/src/testcases/CWE23_Relative_Path_Traversal	452	271	181	60.0%
CWE-78	OS command injection	Java/src/testcases/CWE78_OS_Command_Injection	468	243	225	51.9%
CWE-89	SQL injection	Java/src/testcases/CWE89_SQL_Injection/s01	554	316	238	57.0%
CWE-315	Plaintext storage in cookie	Java/src/testcases/CWE315_Plaintext_Storage_in_Cookie	38	38	0	100%
CWE-325	Missing required cryptographic step	Java/src/testcases/CWE325_Missing_Required_Cryptographic_Step	31	31	0	100%
CWE-190	Integer overflow	Java/src/testcases/CWE190_Integer_Overflow/s01	497	239	258	48.1%
CWE-398	Poor code quality	Java/src/testcases/CWE398_Poor_Code_Quality	124	120	4	96.7%
CWE-382	Use of system exit	Java/src/testcases/CWE382_Use_of_System_Exit	34	34	0	100%
CWE-760	Predictable salt one way hash	Java/src/testcases/CWE760_Predictable_Salt_One_Way_Hash	17	17	0	100%
CWE-369	Divide by zero	Java/src/testcases/CWE369_Divide_by_Zero/s01	542	233	219	43.0%

CWE: Common Weakness Enumeration

5 Conclusions

This paper introduces the design and implementation of AI-SCDF, a tool designed to enhance software code quality. Through the construction of a static analysis checker knowledge base and the integration of code context, we develop effective LLM prompts to facilitate code flaw detection and fix. The experimental results indicate that our tool has effective flaw detection and rectification capabilities for standard code defect categories such as CWE-315, CWE-325, CWE-398, CWE-382, and CWE-760. However, our tool still faces challenges in detecting and rectifying more complex code defects. Furthermore, due to inherent limitations of LLM technology, its accuracy of code defect rectification still needs to be improved. Future work will focus on collecting high-quality corpora to further fine-tune the LLM model, aiming to better meet the needs of code defect detection and rectification.

References

[1] Coverity static analysis [EB/OL]. [2024-04-12]. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
[2] Klocwork static analysis [EB/OL]. [2024-04-12]. https://help. klocwork. com/current/en-us/concepts/checkersintro.htm
[3] Infer static analyzer [EB/OL]. [2024-04-12]. https://fbinfer.com
[4] SonarQube [EB/OL]. [2024-04-12]. https://docs.sonarqube.org/latest
[5] CodeQL website [EB/OL]. [2024-04-12]. https://codeql.github.com
[6] Harer J A, Kim L Y, Russell R L, et al. Automated software vulnerability detection with machine learning [PP/OL]. ArXiv (2018-02-14) [2024-04-12]. https://arxiv.org/abs/1803.04497
[7] Li Z, Zou D Q, Xu S H, et al. SySeVR: a framework for using deep learning to detect software vulnerabilities [J]. IEEE transactions on dependable and secure computing, 2022, 19(4): 2244 – 2258. DOI: 10.1109/tdsc.2021.3051525
[8] Li Z, Zou D Q, Xu S H, et al. VulDeePecker: a deep learning-based system for vulnerability detection [C]//Proc. 2018 Network and Distributed System Security Symposium. Internet Society, 2018. DOI: 10.14722/ndss.2018.23158
[9] Wang S, Liu T Y, Tan L. Automatically learning semantic features for defect prediction [C]//Proc. 38th International Conference on Software Engineering. ACM, 2016: 297 – 308. DOI: 10.1145/2884781.2884804
[10] Li J, He P J, Zhu J M, et al. Software defect prediction via convolutional neural network [C]//Proc. IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2017: 318 – 328. DOI: 10.1109/QRS.2017.42
[11] GPT-3 [EB/OL]. [2024-04-12]. https://openai.com/blog/gpt-3-apps
[12] PaLM2 [EB/OL]. [2024-04-12]. https://ai.google/discover/palm2
[13] Copilot [EB/OL]. [2024-04-12]. https://www. microsoft. com/en-us/microsoft-copilot
[14] Llama [EB/OL]. [2024-04-12]. https://llama.meta.com
[15] Berabi B, Gronskiy A, Raychev V, et al. DeepCode AI Fix: fixing security vulnerabilities with large language models [PP/OL]. ArXiv (2024-02-19) [2024-04-12]. https://arxiv.org/abs/2402.13291
[16] Wadhwa N, Pradhan J, Sonwane A, et al.Frustrated with code quality issues? LLMs can help! [PP/OL]. ArXiv (2023-09-22)[2024-04-12]. https://arxiv.org/abs/2309.12938
[17] Li H N, Hao Y, Zhai Y Z, et al. Enhancing static analysis for practical bug detection: an LLM-integrated approach [J]. Proc. ACM on programming languages, 2024, 8(OOPSLA1): 474 – 499. DOI: 10.1145/3649828
[18] Juliet Java [EB/OL]. [2024-04-12]. https://samate. nist. gov/SARD/test-suites/111

Biographies

Niu Zhi (niu.zhi@zte.com.cn) received his master's degree in control engineering from Chongqing University, China. He is currently working at ZTE Corporation. His research interests include distributed systems, formal verification, and software reliability.

Dong Luming received his master's degree in control theory and control engineering from Huazhong University of Science and Technology, China. He is currently working at ZTE Corporation. His research interests include distributed systems, formal verification, software reliability, and innovative security technologies for wireless communications.