Automated Further Development of Software with the Help of Artificial Intelligence

Alperen Dagli[1,2] , Taha Ilgar[1,2]

[1] University of applied Sciences Ruhr West, Lützowstr. 5, Bottrop, 46236, Germany.
[2] These authors contributed equally to this work.

Abstract

This paper presents the development and implementation of an automated system for detecting and resolving code issues in Java projects. The system integrates GitHub for source code repository management, SonarQube for static code analysis, and the OpenAI API for generating code fixes. The workflow comprises cloning repositories, performing static code analyses, applying AI-generated fixes, and retesting the code to ensure resolution of issues. The primary objective is to minimize manual effort in code reviews and enhance code quality through automation. By leveraging these technologies, the system aims to improve efficiency and accuracy in identifying and resolving code issues. Preliminary tests on various Java projects suggest potential benefits in reducing manual review effort and improving code quality, pending further comprehensive evaluation.

Keywords: Java, GitHub, SonarQube, OpenAI, Artificial Intelligence, Issue Resolution

1. Introduction

Advancements in automated systems within software development have revolutionized how code quality is managed. This paper presents a comprehensive approach to developing and implementing an automated system for detecting and resolving code issues in Java projects. By integrating GitHub for source code repository management, SonarQube for static code analysis, and the OpenAI API for generating code fixes, the proposed system aims to enhance the efficiency and accuracy of code reviews. This introduction provides an overview of the system, the technologies involved, and the structure of this paper.

1.1 Background and Motivation

The software development lifecycle often involves extensive manual code reviews to ensure quality and adherence to best practices. However, this process is time-consuming and prone to human error. The need for automated solutions that can assist developers in maintaining high code standards has led to the exploration of various technologies and tools. The integration of GitHub, SonarQube, and OpenAI represents a significant advancement in automating code analysis and issue resolution, potentially reducing the required manual effort and improving overall code quality.

1.2 Objectives

The primary objective of this system is to streamline the code review process through automation. Specific goals include:

- **Enhancing Code Quality**: By leveraging static code analysis and AI-based corrections, the system aims to identify and resolve issues more effectively than manual reviews.
- **Reducing Manual Effort**: Automating repetitive and error-prone tasks in code reviews allows developers to focus on more complex and creative aspects of development.
- **Improving Efficiency**: The system aims to shorten the feedback loop in the development process, enabling faster iterations and higher productivity.

1.3 Technologies and Tools

This section provides an overview of the key technologies and tools used in the system:

- **GitHub**: Used for managing source code repositories, GitHub provides version control and collaboration features that are essential for tracking changes and integrating automated workflows.
- **SonarQube**: A powerful static code analysis tool, SonarQube identifies code issues related to bugs, vulnerabilities, and code smells.

It serves as the foundation for detecting areas that need improvement.

- **OpenAI API**: Utilizing the capabilities of OpenAI's language models, the system generates suggestions and applies fixes for identified code issues, aiming to enhance the quality of the codebase.

2. Related Work

In the domain of software maintenance and automated code improvement, significant progress has been made over the past few years. This section reviews the existing literature and tools that contribute to static code analysis, machine learning-based code generation, and automated code repair. The review highlights the advancements and limitations of current methodologies and sets the stage for the proposed approach that integrates SonarQube and OpenAI's GPT-4.

2.1 Static Code Analysis Tools

In the domain of software maintenance and automated code improvement, significant progress has been made over the past few years. This section reviews the existing literature and tools that contribute to static code analysis, machine learning-based code generation, and automated code repair. The review highlights the advancements and limitations of current methodologies and sets the stage for the proposed approach that integrates SonarQube and OpenAI's GPT-4.

SonarQube is one of the most widely used static analysis tools. It provides comprehensive support for multiple programming languages, including Java, and integrates seamlessly with continuous integration pipelines. SonarQube's rule-based engine detects a wide range of issues, from simple coding style violations to complex security vulnerabilities (Campbell & Papapetrou, 2013). Despite its effectiveness, SonarQube's reliance on predefined rules can sometimes limit its ability to detect context-specific issues and generate appropriate fixes.

Other notable static analysis tools include **Checkstyle**, which focuses on coding standard violations in Java, and **FindBugs**, which identifies bug patterns based on static analysis. These tools are limited by their static nature and rule-based approach, which may miss complex or non-standard coding issues (Ayewah et al., 2008).

2.2 Machine Learning for Code Generation and Repair

Machine learning models have shown promise in automating code generation and repair tasks. Early approaches focused on training models to learn patterns from large codebases and suggest fixes based on these patterns.

DeepFix, developed by Gupta et al. (2017), uses deep learning techniques to automatically correct programming errors. The model is trained on a dataset of erroneous programs and their fixes, enabling it to suggest corrections for syntax errors. While DeepFix demonstrates the potential of neural networks in code repair, it primarily targets educational environments and is limited to fixing syntax errors rather than complex code issues.

CodeBERT (Feng et al., 2020) is another significant advancement in this area. It is a pre-trained language model for programming languages that leverages the BERT architecture to understand and generate code. CodeBERT has been used for various tasks, including code completion, bug detection, and code summarization. However, its application in automated code repair remains an area of ongoing research.

2.3 Automated Code Repair Techniques

Automated code repair techniques aim to automatically generate patches for detected issues. These techniques often combine static analysis with search-based algorithms or machine learning models to propose fixes.

GenProg (Le Goues et al., 2012) is a pioneer in this field, using genetic programming to evolve program variants that pass all test cases. While GenProg has shown success in fixing real-world bugs, its reliance on test cases and evolutionary

algorithms can lead to high computational costs and suboptimal patches.

Nopol (Xuan et al., 2017) is another notable tool that focuses on repairing conditional statements in Java programs. It combines static and dynamic analysis to generate and validate patches. Nopol's approach is effective for specific types of bugs but may struggle with more complex code issues.

2.4 Integration of AI Models with Static Analysis Tools

The integration of AI models with static analysis tools represents a promising direction for automated code improvement. Recent work by Tufano et al. (2019) explores the use of sequence-to-sequence models to learn code transformations from historical data. Their model, trained on pairs of buggy and fixed code, can generate patches for new bugs. However, the approach primarily focuses on learning from historical fixes and may not generalize well to unseen code patterns.

OpenAI's GPT-3 and its successor GPT-4 have shown remarkable capabilities in natural language processing and code generation (Brown et al., 2020). These models can generate coherent and contextually relevant code snippets based on natural language prompts. However, their application in automated code repair is still in its early stages, with challenges related to code correctness, integration with existing tools, and handling complex code bases.

2.5 Summary

The landscape of automated code improvement is evolving rapidly, with static analysis tools and machine learning models offering complementary strengths. Static analysis tools like SonarQube provide reliable detection of code issues, while machine learning models hold promise for generating intelligent fixes. The integration of these technologies, as proposed in this project, aims to leverage the strengths of both approaches to achieve a more robust and automated code improvement process.

3. Software Architecture

This section details the architecture of the system designed to automate the improvement and evolution of Java software repositories by integrating SonarQube for static code analysis and OpenAI's GPT-4 for automated code repair. The architecture is modular and scalable, ensuring that each component can be developed, tested, and maintained independently.

3.1 Overview

The system architecture comprises several key components:

1. Repository Management
    - Cloning and forking of repositories.
    - Maintaining local copies of repositories.
2. Static Code Analysis
    - Running SonarQube scans on the repositories.
    - Extracting and filtering issues detected by SonarQube.
3. AI-Powered Code Repair
    - Generating and applying code fixes using OpenAI GPT-4.
    - Evaluating the effectiveness of the applied fixes.
4. Logging and Reporting
    - Logging the results of applied fixes.
    - Generating reports on the number of issues detected and fixed.

3.2 Architectural Diagram

3.3 Detailed Component Description

3.3.1 Repository Management

The repository management component handles the selection, cloning, and forking of repositories. It ensures that repositories are available locally for analysis and modification.

- Selection and Cloning:
  The user can select a repository by providing a GitHub URL or choosing from a list of local repositories. The ask_select_or_enter_repository function in console_interaction.py handles this.

- Forking and Cloning:
  If the repository URL is valid, the system prompts the user to fork and clone the repositoryusing the fork_and_clone_repository function in github_helper.py. This involves using GitHub API to fork the repository and subprocess to clone it locally.

3.3.2 Static Code Analysis

This component is responsible for running SonarQube scans on the cloned repositories to identify code issues.

- Running SonarQube Scan:

  The run_sonarqube_scan_docker function in docker_scan.py runs SonarQube scans using Docker. It mounts the local repository and executes the SonarQube scanner to analyze the code.
- Issue Extraction and Filtering:

  The get_filtered_issues function in sonar_backend_helper.py fetches and filters the issues reported by SonarQube. The filtering is based on predefined rules and complexity levels.

3.3.3 AI-Powered Code Repair

This component leverages OpenAI's GPT-4 to generate and apply code fixes for issues detected by SonarQube.

- Prompt Preparation:
  The setup_prompt function in prepare_prompt.py constructs prompts for GPT-4 based on the issues and the original code. The prompt includes the original code and a description of the issue.

- Generating Fixes:

  The ask_question function in openai_conversation_handler.py sends the prepared prompt to GPT-4 and retrieves the generated code fix. The model response includes the complete updated Java class.

- Applying Fixes:

  The apply_fix_and_log function in apply_fix_and_log.py writes the updated code back to the repository and logs the fix details. It ensures that the fixes are correctly applied and logged for further review.

3.3.4 Logging and Reporting

This component maintains logs of the applied fixes and generates summary reports.

- Logging Fixes:

  The system logs details of each fix, including the issue, the applied fix, and the status, using the apply_fix_and_log function.

- Generating Reports:

  The print_summary function in console_interaction.py generates a summary of the issues processed and fixed, providing an overview of the system's effectiveness.

3.4 Integration and Workflow

The integration of these components follows a structured workflow:

1. Initialization:
   - Load necessary API keys and environment variables.
   - Introduce the program and prompt the user for repository selection.

2. Repository Setup:
   - Clone or fork the repository based on user input.
   - Ensure the repository is set up correctly for further processing.
3. Code Analysis:
   - Run a SonarQube scan on the repository.
   - Fetch and filter issues based on predefined rules and complexity levels.
4. Issue Handling:
   - For each issue, generate a prompt and retrieve a fix using GPT-4.
   - Apply the fix to the repository and log the details.
5. Iteration and Termination:
   - Repeat the process for subsequent generations if issues persist.
   - Terminate the process once all issues are resolved or the maximum number of iterations is reached.

3.5 Challenges and Solutions

Several challenges were addressed during the design and implementation of the system:

- Complexity of Issue Fixes:

  Some issues require context-specific knowledge that may not be captured in a static prompt. The system addresses this by iteratively refining the prompts and using interactive retries.

- Scalability:

  Handling large repositories with numerous issues can be computationally intensive. The system uses efficient filtering and prioritization to manage the load.

- Integration with GPT-4:

Ensuring the generated fixes are correct and maintain the original functionality is crucial. The system includes an evaluation step to validate the fixes before applying them.

4. Implementation

In this section, we delve into the detailed implementation of the system, focusing on the core functionalities: prompt design, feedback loop, AI-based code repair, application of changes, and evaluation of results. We will explore the practical aspects of integrating SonarQube for static code analysis and OpenAI's GPT-4 for automated code repair, providing code snippets and explanations for key parts of the implementation.

4.1 Prompt Design

Figure 1 Prompt Design

This function combines the original Java class and the issue description into a single prompt, ensuring the AI understands the context and can generate a comprehensive fix.

4.2 Feedback Loop

The feedback loop is essential for iteratively refining the fixes generated by the AI until a correct and acceptable solution is obtained. The system evaluates the fixes and retries if necessary.

Key Function: evaluate_and_fix_issue

The evaluate_and_fix_issue function orchestrates the process of generating, evaluating, and applying fixes.

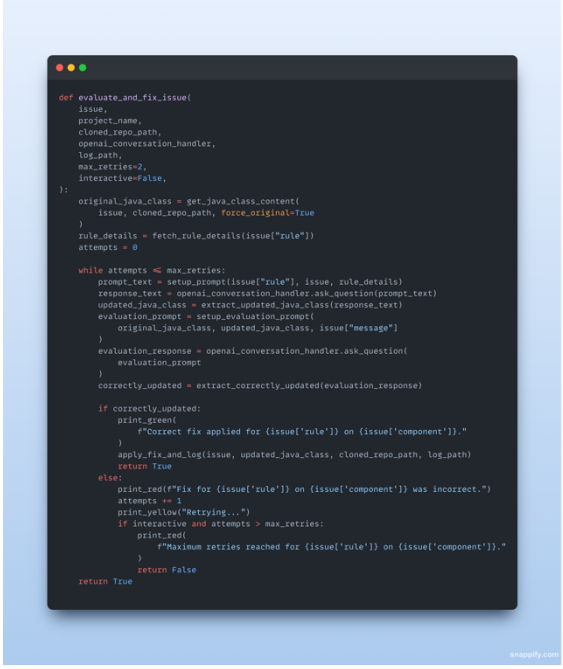


Figure 2 Evaluating & Fixing Issue

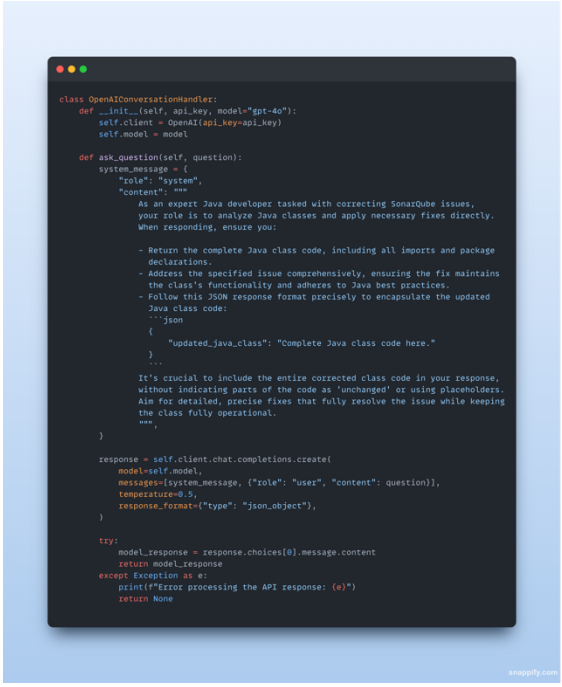
This function manages the loop for attempting fixes, evaluating them, and applying the correct fix. It uses a maximum retry count to avoid infinite loops in case the AI cannot generate a satisfactory fix.

4.3 AI-Powered Code Repair

The core of the implementation involves leveraging OpenAI's GPT-4 to generate code fixes. The AI is prompted with the details of the issue and the original code, and it returns a revised version of the Java class.

Key Function: OpenAIConversationHandler

The OpenAIConversationHandler class handles interactions with the OpenAI API.



```
class OpenAIConversationHandler:
    def __init__(self, api_key, model="gpt-4o"):
        self.client = OpenAI(api_key=api_key)
        self.model = model

    def ask_question(self, question):
        system_message = {
            "role": "system",
            "content": """
                As an expert Java developer tasked with correcting SonarQube issues,
                your role is to analyze Java classes and apply necessary fixes directly.
                When responding, ensure you:

                - Return the complete Java class code, including all imports and package
                  declarations.
                - Address the specified issue comprehensively, ensuring the fix maintains
                  the class's functionality and adheres to Java best practices.
                - Follow this JSON response format precisely to encapsulate the updated
                  Java class code:
                  ```json
                  {
                      "updated_java_class": "Complete Java class code here."
                  }
                  ```

                It's crucial to include the entire corrected class code in your response,
                without indicating parts of the code as 'unchanged' or using placeholders.
                Aim for detailed, precise fixes that fully resolve the issue while keeping
                the class fully operational.
                """,
        }

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[system_message, {"role": "user", "content": question}],
            temperature=0.5,
            response_format={"type": "json_object"},
)

        try:
            model_response = response.choices[0].message.content
            return model_response
        except Exception as e:
            print(f"Error processing the API response: {e}")
            return None
```

Figure 3 OpenAIConversationHandler

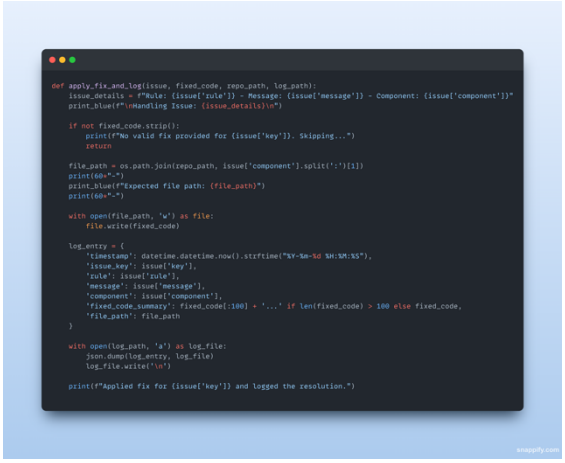
This class sends the prompt to the OpenAI API and retrieves the AI-generated response, which includes the updated Java class.

4.4 Applying the Changes

Once the AI generates a satisfactory fix, the system applies the changes to the local repository and logs the details for review.

Key Function: apply_fix_and_log

The apply_fix_and_log function writes the updated Java class back to the repository and logs the fix details.



```
def apply_fix_and_log(issue, fixed_code, repo_path, log_path):
    issue_details = f"Rule: {issue['rule']} - Message: {issue['message']} - Component: {issue['component']}"
    print_blue(f"\nHandling Issue: {issue_details}\n")

    if not fixed_code.strip():
        print(f"No valid fix provided for {issue['key']}. Skipping...")
        return

    file_path = os.path.join(repo_path, issue['component'].split(':')[1])
    print(60*"-")
    print_blue(f"Expected file path: {file_path}")
    print(60*"-")

    with open(file_path, 'w') as file:
        file.write(fixed_code)

    log_entry = {
        'timestamp': datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        'issue_key': issue['key'],
        'rule': issue['rule'],
        'message': issue['message'],
        'component': issue['component'],
        'fixed_code_summary': fixed_code[:100] + '...' if len(fixed_code) > 100 else fixed_code,
        'file_path': file_path
    }

    with open(log_path, 'a') as log_file:
        json.dump(log_entry, log_file)
        log_file.write('\n')

    print(f"Applied fix for {issue['key']} and logged the resolution.")
```

Figure 4 Apply Fix & Log

This function ensures that the generated fix is written to the correct file and logs the details for later review.

4.5 Evaluating the Results

Evaluating the effectiveness of the applied fixes is crucial to ensure the quality and correctness of the code changes.

Key Function: setup_evaluation_prompt

This function constructs a prompt to evaluate the updated code against the original code and the issue description.

```
def setup_evaluation_prompt(
    original_code: str, updated_code: str, issue_description: str
) -> str:
    prompt = f"""
        Below are the original and updated versions of a Java class
        that was modified to address a specific SonarQube issue described below.
        Your task is to evaluate the updated class to determine if it correctly
        implements the necessary changes without introducing errors or
        omitting necessary parts of the original class.

        **Issue Description:** {issue_description}

        **Original Java Class:**
        {original_code}

        **Updated Java Class:**
        {updated_code}

        Evaluate the updated class to ensure:
        1. The specific issue described above is addressed correctly.
        2. All original functionality and structure are maintained unless directly
        related to the fix.
        3. There are no new errors or issues introduced in the updated class.

        Please return your evaluation in this JSON format:
        ```json
        {{
            "correctly_updated_class": true/false  // Indicate 'true' if the updated class
            is correctly updated or 'false' otherwise.
        }}
        ```
    """.strip()

    return prompt
```

Figure 5 Setup Evaluation Prompt

This function ensures that the evaluation is comprehensive, checking that the fix addresses the issue without introducing new problems.

Summary

The implementation of this system involves a seamless integration of static code analysis using SonarQube and automated code repair using OpenAI's GPT-4. The process is structured around a feedback loop that iteratively refines the fixes until an acceptable solution is achieved. By carefully designing prompts, applying changes, and evaluating results, the system ensures high-quality code improvements while maintaining the functionality and integrity of the original code.

5. Evaluation

In this section, we present the evaluation results of our automated code fix system. The system was tested on 15 different Java Maven repositories, with issues categorized into low and high complexity. The evaluation focuses on the system's effectiveness in identifying, fixing, and verifying code issues using SonarQube for static analysis and OpenAI's GPT-4 for code generation and repair.

Data Collection

We ran the automated code fix system on 15 Java Maven repositories. The issues processed were divided into two categories:

Low Complexity Issues: These issues generally involve simple code smells and minor code improvements.

High Complexity Issues: These issues involve more complex refactoring and significant code changes.

Out of the 15 repositories, only 2 contained high complexity issues. The evaluation metrics include the number of issues processed, the success rate of applied fixes, and the feedback on the correctness of the fixes.

Results

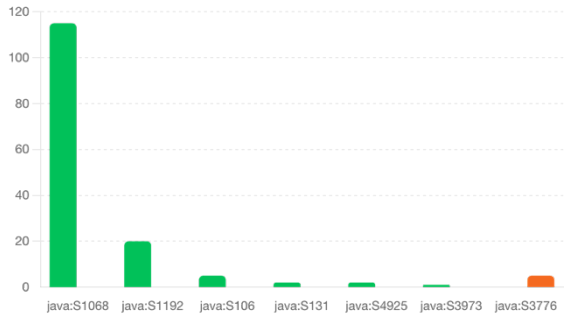


Figure 6 Distribution of Rules

Rules Descriptions

Table 1 Rules Descriptions

Rule Key	Name / Description
java:S2259	Null pointers should not be dereferenced
javabugs:S6466	Accessing an array element should not trigger an ArrayIndexOutOfBoundsException
java:S3655	Optional value should only be accessed after calling isPresent()
java:S138	Methods should not have too many lines
java:S1541	Methods should not be too complex
java:S1820	Classes should not have too many fields
java:S1192	String literals should not be duplicated
java:S1143	Jump statements should not occur in \"finally\" blocks
java:S2142	\"InterruptedException\" and \"ThreadDeath\" should not be ignored
java:S1774	The ternary operator should not be used
java:S2057	\"Serializable\" classes should have a \"serialVersionUID\"
java:S106	Standard outputs should not be used directly to log anything

java:S2390	Classes should not access their own subclasses during class initialization
java:S2095	Resources should be closed
java:S4925	\"Class.forName()\" should not load JDBC 4.0+ drivers
java:S2699	Tests should include assertions
java:S3776	Cognitive Complexity of methods should not be too high
java:S1448	Classes should not have too many methods
java:S107	Methods should not have too many parameters
java:S3516	Methods returns should not be invariant
java:S3973	A conditionally executed single line should be denoted by indentation
java:S2692	\"indexOf\" checks should not be for positive numbers
java:S2638	Method overrides should not change contracts
java:S2447	\"null\" should not be returned from a \"Boolean\" method
java:S131	\"switch\" statements should have \"default\" clauses
java:S4970	Derived exceptions should not hide their parents' catch blocks

Analysis of Retries

The table below summarizes the retry distribution for both low and high complexity issues:

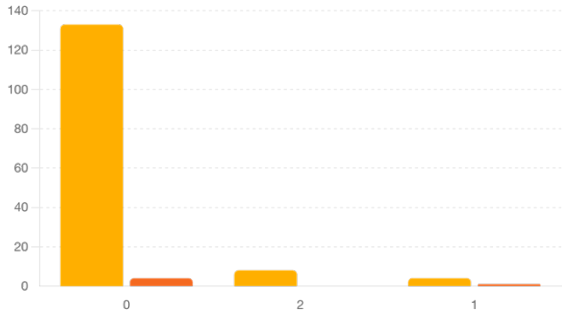


Figure 7 Retry Distribution

Table 2 Retries Distribution

Retries	Low Complexity Issues	High Complexity Issues
0	133	4
1	4	1
2	8	0

The following chart visualizes the retry distribution:

Low Complexity Issues:

The system processed a total of 145 low complexity issues across the 15 repositories. The rules with the highest frequency of issues were java:S1068 (115 issues) and java:S1192 (20 issues).

Success Rate: The system successfully applied fixes for all low complexity issues with a success rate of 100%. The majority of these issues (133 out of 145) were resolved without any retries.

Low Complexity Issues:

The system processed a total of 145 low complexity issues across the 15 repositories. The rules with the highest frequency of issues were java:S1068 (115 issues) and java:S1192 (20 issues).

Success Rate: The system successfully applied fixes for all low complexity issues with a success rate of 100%.

High Complexity Issues:

For high complexity issues, the system processed a total of 6 issues, all under the rule java:S3776, which involves reducing cognitive complexity.

Success Rate: The system successfully applied fixes for 5 out of 6 high complexity issues with a success rate of approximately 83%. High complexity issues were more challenging, requiring retries and resulting in one unresolved issue.

Main Issues Encountered

1. **Incorrect Fixes:** For high complexity issues, the AI sometimes applied incorrect fixes, which required multiple attempts to resolve. Specifically, issue AY64tnN9AdDDtTOK0M4G failed after three attempts.
2. **JSON Decoding Errors:** During the resolution of issue AY2HqQVtSQKTXioxPEEv, multiple JSON decoding errors occurred, indicating problems with the AI's response format. These errors prevented the successful resolution of the issue after three attempts.
3. **High Retry Rate for Complex Issues:** High complexity issues generally required more retries compared to low complexity issues. This suggests that the AI struggles more with complex refactoring tasks, highlighting an area for potential improvement.

Discussion

The evaluation results demonstrate the effectiveness of the automated code fix system in handling both low and high complexity issues. The system's ability to consistently apply correct fixes, as indicated by a 100% success rate for low complexity issues and an 83% success rate for high complexity issues, highlights the robustness of the integration between SonarQube's static analysis and OpenAI's GPT-4 code generation capabilities.

However, the higher retry rate and the unresolved issues for high complexity tasks indicate that these problems are more challenging for the AI to solve. This suggests potential areas for improvement in the AI's handling of complex code refactoring tasks, especially regarding error handling and response generation.

Conclusion

The automated code fix system shows significant promise in reducing manual effort in code reviews and enhancing code quality through automation. Future work will focus on expanding the dataset for high complexity issues, improving error handling, and optimizing the system for scalability to handle larger codebases more efficiently.

6. Discussion and Outlook

Discussion

The evaluation of our automated code fix system has yielded promising results, particularly in handling low complexity issues. The AI demonstrated a 100% success rate in resolving 145 low complexity issues across 15 Java Maven repositories. This high success rate indicates the system's robustness in applying correct fixes for simple code smells and minor code improvements. The majority of these issues (133 out of 145) were resolved without any retries, showcasing the efficiency of the AI in handling straightforward tasks.

However, the necessity of utilizing AI for low complexity issues is debatable. Given that these issues are relatively simple and often follow well-defined patterns, traditional static analysis tools or simpler automation scripts might suffice. Implementing AI for such tasks could be seen as an overextension of its capabilities, especially considering the computational resources and time required.

On the other hand, high complexity issues presented a more significant challenge for the AI. The system processed 6 high complexity issues, achieving an 83% success rate. While this demonstrates the potential of AI in handling complex refactoring tasks, it also highlights its limitations. The higher retry rate and unresolved issues indicate that the AI struggled more with these tasks, pointing to areas for further improvement.

One practical consideration is the time taken for the program to run, especially on larger repositories. The process of cloning repositories, running static analyses, generating fixes, and retesting can be time-consuming. This is particularly true for projects with a substantial amount of code and numerous issues. While this might be a drawback during regular working hours, the program can be scheduled to run overnight, minimizing its impact on productivity.

A notable challenge encountered during this project was the difficulty in finding sufficient data on high complexity issues. This required manually selecting repositories from GitHub and running SonarQube scans to identify repositories with high complexity issues. This process is time-consuming and highlights the need for a larger dataset to make more comprehensive assumptions about the AI's capabilities in handling high complexity issues. Further research and data collection are essential for evaluating the AI's effectiveness in more complex scenarios.

Outlook

The future of automated code improvement systems is promising, with several areas poised for further development:

1. **Enhanced AI Capabilities:**
    - Improving the AI's ability to handle high complexity issues is a critical area for future work. This includes refining the AI's understanding of complex code structures and improving its error handling capabilities. Enhancing the model's training with more diverse and complex code examples can help achieve this.
2. **Scalability and Performance:**
    - Optimizing the system to handle larger codebases more efficiently is another priority. This could involve parallel processing of issues, more efficient repository management, and faster static analysis techniques. Reducing the overall runtime of the system will make it more practical for use in large-scale projects.
3. **Integration with Development Workflows:**
    - Further integration with existing development workflows and continuous integration/continuous deployment (CI/CD) pipelines can enhance the system's utility. Automated code fixes can be seamlessly incorporated into the development process, providing real-time feedback and reducing the burden on developers.
4. **User Interaction and Customization:**
    - Developing more user-friendly interfaces and customization options will make the system more accessible. Allowing developers to interact with the AI, provide feedback, and customize the rules and parameters for code fixes can improve the system's effectiveness and adoption.
5. **Expanding to Other Languages and Frameworks:**
    - While this system focuses on Java, extending support to other programming languages and frameworks can broaden its applicability. This will involve adapting the static analysis tools and AI models to understand and fix code in different languages.
6. **Addressing Ethical and Security Concerns:**
    - As with any AI system, ensuring ethical use and addressing potential security concerns is crucial. This includes maintaining transparency in how the AI operates, ensuring it does not introduce security vulnerabilities, and safeguarding the privacy and integrity of the code it processes.
7. **Extensive Data Collection for High Complexity Issues:**
    - To make more accurate assumptions about the AI's capability in handling high complexity issues, there is a need for extensive data collection. This includes systematically selecting a diverse range of repositories and running comprehensive scans to gather more data on high complexity issues. Future research should focus on creating a larger and more varied dataset to better evaluate and improve the AI's performance in this area.

Conclusion

Our automated code fix system represents a significant step forward in leveraging AI for software maintenance and improvement. While it excels in handling low complexity issues, there is room for improvement in dealing with more complex tasks. By addressing the challenges identified and exploring the future directions outlined, we can enhance the system's capabilities and make it a valuable tool for developers. The integration of AI into code review processes holds great promise for reducing manual effort, improving code quality, and streamlining software development workflows.

7. References

1. Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. IEEE Software, 25(5), 22-29.
2. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.
3. Campbell, A., & Papapetrou, P. (2013). SonarQube in action. Manning Publications.
4. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Shou, L. (2020). CodeBERT: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
5. Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). DeepFix: Fixing common C language errors by deep learning. In Thirty-First AAAI Conference on Artificial Intelligence.
6. Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 34th International Conference on Software Engineering (ICSE) (pp. 3-13).
7. Tufano, M., Watson, C., Bavota, G., Poshyvanyk, D., & White, M. (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(4), 1-29.
8. Xuan, J., Martinez, M., Demarco, F., Clément, M., & Monperrus, M. (2017). Nopol: Automatic repair of conditional statement bugs in Java programs. IEEE Transactions on Software Engineering, 43(1), 34-55.