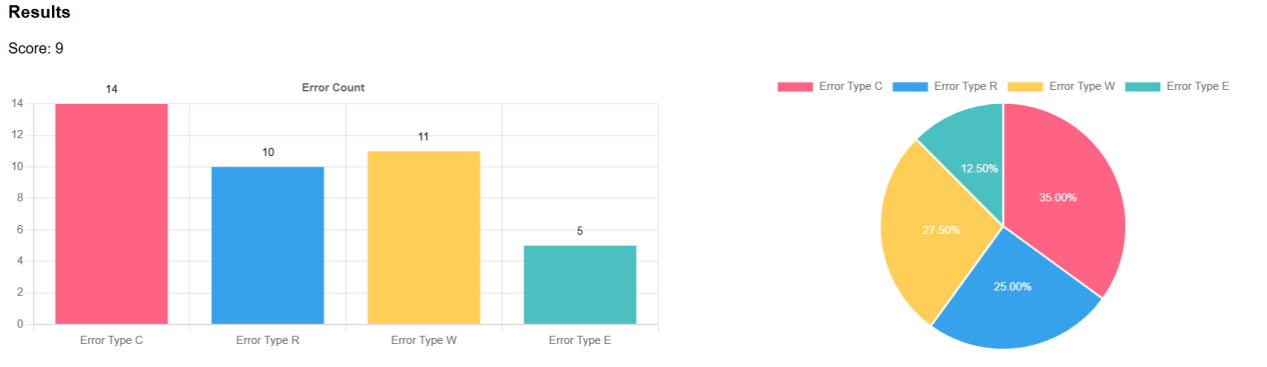
### VI. Evaluation

Our extended Pylint tool aims to solve key problems in code quality analysis, data visualization, and user interaction. It also works to improve the usability and data management features of current tools. This section provides an overview of the challenges encountered during development, the solutions applied, and an evaluation of the improvements made. We also compare our tool with the original tool to highlight the innovations it brings, and where it still needs improvement. By addressing these challenges and introducing new features, our goal is to create a smarter tool that is easy to use and can make it effective for developers.

#### A. Data Visualization Interface Design

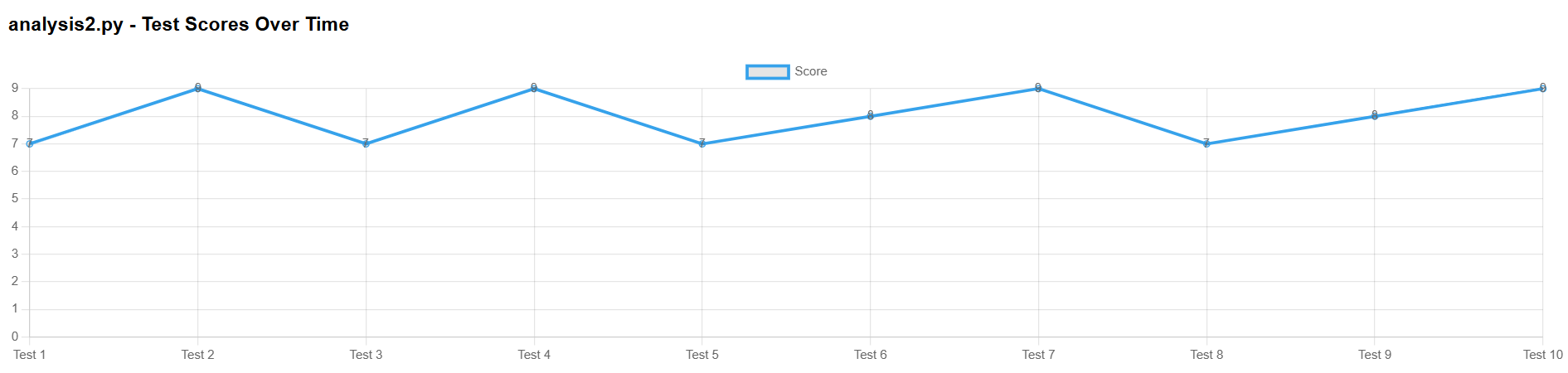
The original Pylint tool could only read input and display output at the command line. We decided to develop the data visualization interface which was one of the critical modules in the development of our tool. However, during the initial design phase, we found that the existing methods of displaying data were insufficient and did not meet user needs. For example, when presenting Common Weakness Enumeration (CWE) issues and code quality scores, traditional text-based or table formats were too complicated, making it difficult for users to fully understand the information. This lack of visual guidance not only influenced the user experience but also reduced efficiency when evaluating code quality.

To solve this issue, we worked closely with front-end designers and referenced established examples of effective visualization interface designs from the industry. We finally improved the data display methods by ‌combining charts like bar graphs and pie charts as shown in Figure 1, making the data presentation more visually appealing and easier to understand.



**Figure 1**: Bar graphs and pie charts showing CWE issues and code quality scores

Other than that, we included interactive chart features into the interface as shown in Figure 2, allowing users to dynamically explore detailed data through clicks or hovers. These interactive tools made it possible for users to explore data from multiple dimensions, improving both clarity and analysis efficiency.

****

**Figure 2**: Data exploration interface with interactive charts

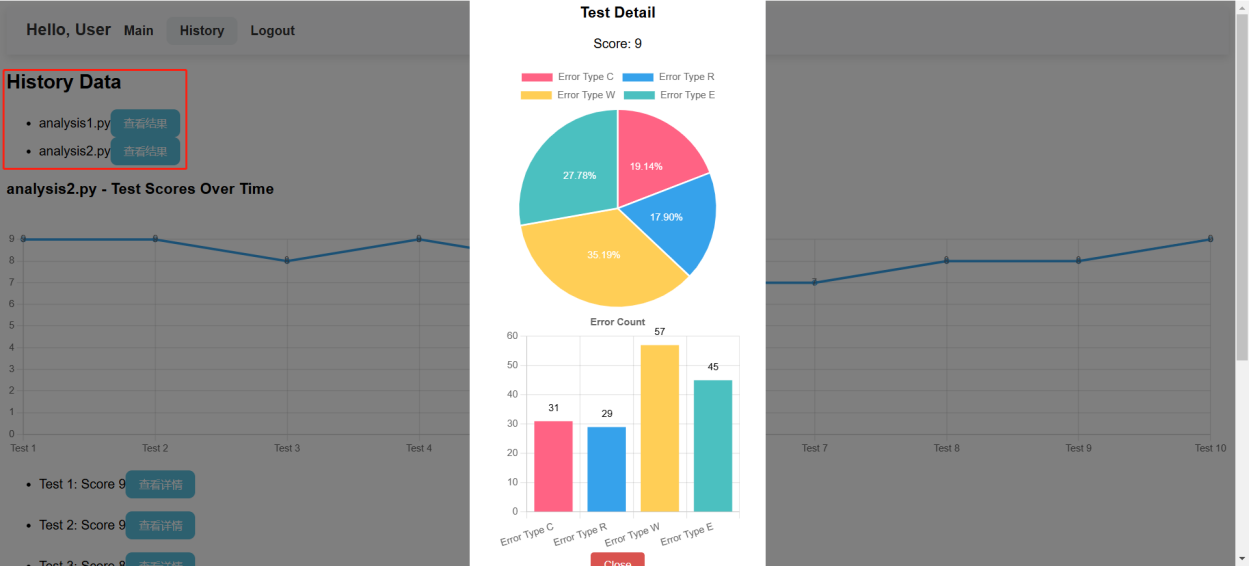
These changes have made the data much easier to understand, allowing users to quickly catch important information. We have created a strong foundation that encourages more people to adopt and rely on the tool in their workflows. This sets the stage for the tool to reach a wider audience and have a bigger impact in the future.

#### B. Data Management

During the development of the tool, we identified that database design was crucial to performance and reliability of the system. However, the initial database structure had several problems, like data redundancy, inconsistent operations, and inefficient management. These problems were particularly tricky in scenarios where multiple users operated simultaneously, as they could lead to errors spreading across the system. Traditional database designs failed to effectively support the demands of complex code quality analysis, which made it harder to maintain in the future.

To deal with these problems, we optimized the database architecture through normalization, reducing data redundancy and improving storage efficiency. Then we implemented data consistency tests to ensure the reliability of update and delete operations. During testing, we verified the relationships between database tables and their integrity to prevent errors from spreading throughout the system.

At the same time, we designed and implemented a user record query module (Figure 3), enabling users to securely access their historical analysis data records. We added access controls to protect user data and optimized the query process to ensure fast responses, even when handling a large amount of data.



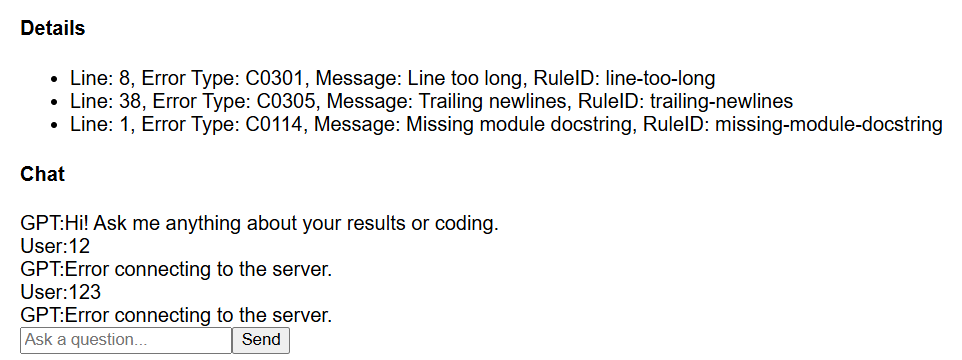
**Figure 3**: Interface design of the user record query module

These steps greatly improved the performance and reliability of the database module, making sure that it meets the standards for data management demanded by the system. This makes the module more effective and dependable for handling complex tasks and maintaining system stability.

#### C. ChatGPT Interaction

The ChatGPT module is one of the core innovations of the tool, designed to provide users with helpful and context-aware suggestions for improving their code. To be honest, the initial interface design we made had some shortcomings. The user feedback inside our group pointed out that it was hard to understand and apply the suggestions because the interface did not clearly show the context for the changes. It was frustrating particularly in scenarios which included multiple rounds of modifications, as it became difficult to track which suggestion belonged to which part of the code. Users felt confused when trying to find and apply one specific suggestion, since the interface failed to make it easy to see what each suggestion was about.

To fix this, we made some major improvements to the interface of this module. First, we added visual associations between suggestions, enabling users to locate the relationship between a suggestion and its corresponding code changes. Then, we provided detailed annotations and guidance for each suggestion, breaking them down step by step, as shown in Figure 4. These explanations showed why the suggestion was made, how to apply it, and what results to expect. This made it much easier for users to understand and follow the recommendations without feeling overwhelmed.



**Figure 4**: Step-by-step guidance for modification suggestions

We managed to make the ChatGPT module much easier and more practical to use by listening to user feedback. Now, the module provides smarter and more helpful support for code analysis. The improved interface clearly connects suggestions to the related parts of the code, making it simple to understand and apply changes. And users can continue to ask questions in the input box and get answers. With step-by-step guidance and detailed explanations, users can easily follow the suggestions without confusion. Users can continue asking questions in the input box and get answers. These updates make the module more efficient, helping users improve their code faster and with less effort.

#### **D. Frontend and Backend Interaction**

Integrating the front-end Vue framework with the back-end Django framework was an important part of system development. During the initial tests, we found that the back-end could not properly receive data sent by the front-end through request.body or request.POST.data, causing some functions to fail. We identified the cause of the issue, the Vue front-end and Django back-end are running on different ports (8080 and 8000), which the browser treats as different origins. Before sending a cross-origin request, the browser sends an OPTIONS preflight request to confirm whether the server allows the request. However, the Django back-end was not properly configured to allow cross-origin access, causing the browser to fail the preflight request.

To fix these problems, we made several improvements to how the front-end and back-end interact. First, in the front-end Vue project, we updated the config/index.js file to set up a proxy server. This made sure that requests were forwarded correctly to the back-end, ensuring the same protocol, domain, and port for both sides, which solved the cross-origin issue. On the back-end, we installed the django-cors-headers module and updated the settings.py file to allow the front-end to access back-end resources. The detailed configuration in the settings.py file is shown in Figure 6.



**Figure 5: Detailed Configuration in the settings.py File**

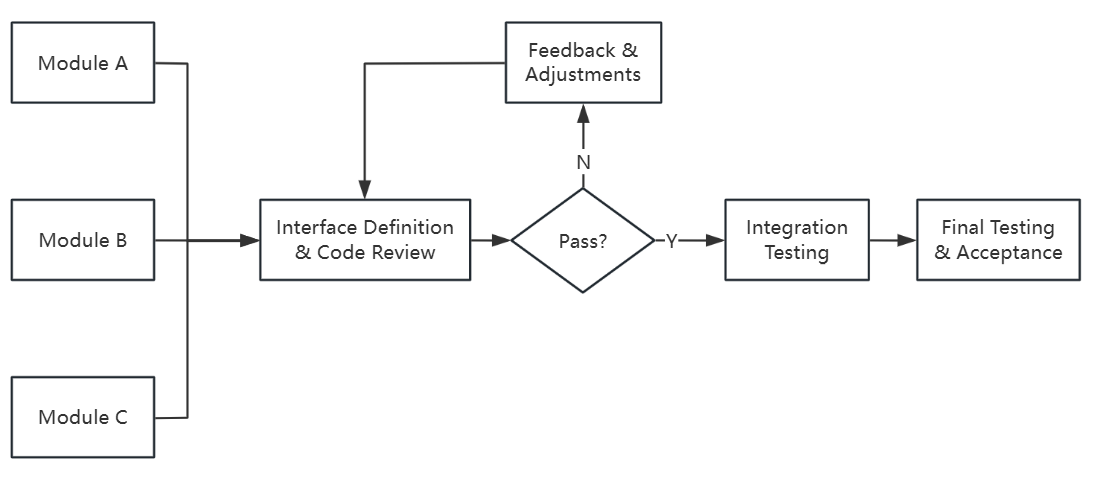
Next, we standardized how data was sent and received by using JSON format. The front-end formatted all request data as JSON and set the Content-Type header to application/json. On the back-end, we used JsonResponse library of Django to process the JSON data correctly. This ensured the back-end could handle all the content sent by the front-end without errors.

Besides, we added features to improve user experience. A loading state management system was implemented so that users would only see fully loaded and updated content, preventing them from interacting with incomplete or loading screens. We also introduced an error feedback system. This system showed users clear error messages when something went wrong and logged detailed error information on the back-end to help developers debug issues faster.

#### E. Cross-Module Coordination and Integration

During the final phase of the project, we encountered significant issues with how modules were connected and how their dependencies worked together. These issues caused inconsistencies during the integration process, making it difficult to ensure that all components worked harmoniously. This slowed down integration testing and made debugging much harder.

To address this challenge, we started holding regular team meetings and introduced code cross-reviews into the development and testing process as shown in Figure 7. In these meetings, team members shared details about their module’s requirements and design, which helped everyone understand how the different modules needed to work together. This made sure that interface designs and implementations were consistent. Apart from that, after development, of each module, we invited developers from other modules for code reviews to identify potential issues.Once the issues are resolved, we hold another meeting for code reviews to see if the modules are ready for integration testing. These steps greatly improved the success of integrating the modules and sped up the testing and delivery process.



**Figure 6**: Development and testing process with code cross-reviews

These efforts greatly improved how we worked together to coordinate and integrate different modules. By improving communication and adding code cross-reviews, we fixed many issues, aligned the modules, and reduced errors. This made integration easier, sped up testing, and made debugging simpler. Our team also gained a better understanding of the overall architecture of the project, which helped to make sure that everyone was on the same page. In the end, we delivered a reliable product faster and set high standards for future projects.

Compared to the original Pylint tool, our extended version has clear advantages in both features and usability. The improved visualization tools make it much easier to understand code analysis results. Instead of relying on long, text-based reports, our tool uses simple and clear charts and graphs, helping users quickly find patterns, spot problems, and understand the overall picture. This makes debugging and improving code faster and less stressful. Besides, the built-in ChatGPT module adds smart suggestions to the process. It does not just point out errors but also explains why they happen and suggests ways to fix them. Whether it is recommending better practices, giving examples, or suggesting improvements, it acts like a helpful assistant. By combining clear visuals with intelligent guidance, our extended Pylint tool not only finds issues but also makes it easier for developers to fix them, improving code quality and saving time.

In summary, this evaluation section comprehensively analyzed the development outcomes and potential improvement areas of our extended Pylint tool. By addressing key issues such as interface design, data management, frontend and backend Interaction, and cross-module coordination, the tool has made great progress in its functionality and user experience. Although the system is still in its early stages and not yet fully mature, the progress achieved so far highlights its strong potential as a powerful code quality analysis tool. With further updates and better demonstrations of its features, this tool has the capability to become a competitive and widely adopted solution in the market.