Engineering Educators Bringing the World Together
2025 ASEE Annual Conference & Exposition
Palais des congrès de Montréal, Montréal, QC • June 22–25, 2025 ASEE
Paper ID #47820

Addressing open-source software complexity using a large language model

Dr. Edward F. Gehringer, North Carolina State University at Raleigh

Dr. Gehringer is a professor in the Departments of Computer Science, and Electrical & Computer Engineering. His research interests include data mining to improve software-engineering practice, and improving assessment through machine learning and natural language processing.

David Mond, North Carolina State University at Raleigh
jack liu

Enhancing Code Quality and Design in Open-Source Projects Using Large Language Models (LLMs)

Abstract

Open-source software (OSS) projects often face challenges in maintainability and design, especially in educational contexts, where large numbers of programmers contribute for a very short period of time. Consequently, a large part of the code base is developed by coders who have little understanding of the rest of the code. This paper explores the potential of Large Language Models (LLMs) to address these challenges by improving code quality, reducing coupling, and enhancing maintainability. Using metrics such as cyclomatic complexity and Halstead metrics, we evaluate LLM-driven refactoring and documentation efforts. Our findings highlight significant improvements in code structure, readability, and design adherence, while also identifying limitations in current LLM capabilities.

1. Introduction

Open-source software (OSS) projects play a pivotal role in software engineering education by offering students real-world coding experience. However, these projects often suffer from poor design and high maintenance costs due to students' limited engagement and adherence to software design principles. Students, constrained by time and struggling to understand the codebase, often structure code poorly and place functionality in the wrong classes, making the codebases harder to interpret and maintain. This study investigates the application of Large Language Models (LLMs), such as GPT-4, in enhancing OSS projects. We aim to evaluate their effectiveness in improving code quality through automated refactoring, identifying code smells, and generating high-quality documentation. This paper presents findings and insights from applying LLMs to the Expertiza [1, 2] codebase, which has incorporated student contributions for nearly 18 years. We hypothesize that LLMs can help students refactor codebases to be more succinct, generate quality code comments, reorganize methods, and move functionality between classes to improve code design. Concepts such as the Single Responsibility Principle (SRP) [9], Don’t Repeat Yourself (DRY) principle [10], and Object-Oriented Programming (OOP) standards are used as baselines for crafting prompts to enhance code quality. Additionally, to enable LLMs to handle complex codebases beyond token limits and circumvent data overload, we have utilized third-party software, open-source tools, and custom-built code-processing LLMs.

2. Background and Related Work

Fang et al. [4] explored the effectiveness of various LLMs in analyzing code inconsistencies and smells. Their findings indicate that OpenAI's GPT-4 outperforms other models, including Meta’s LLaMA family and the open-source StarChat-Beta model, in understanding, refactoring, and modifying code. Notably, GPT-4 excels in recognizing code snippets from open-source software (OSS), which is crucial for our study focusing on improving code design in OSS projects like Expertiza. By recognizing code snippets effectively, GPT-4 can identify areas of inefficiency or poor design and provide actionable suggestions for improvement, directly contributing to better code maintainability and quality. Additionally, GPT-4

demonstrated the highest accuracy and comprehension of both obfuscated and non-obfuscated code among all models tested, achieving 87% accuracy in generating code and summaries based on obfuscated code and 95% accuracy for non-obfuscated code.

These findings are integral to our research, which aims to utilize the most effective publicly available LLM for code design refactoring and suggestions in OSS projects. We selected GPT-4 as the foundational model for evaluating LLM-generated suggestions for enhancing complex OSS code design due to its superior code-comprehension capabilities.

3. Methodology

Our methodology leverages OpenAI GPT-4, alongside tools like Codebuddy [11], the Custom RAG Tool, and Aider, to improve the Expertiza codebase by reducing class coupling and enhancing overall design and readability. Codebuddy provides AI-driven code reviews, pinpointing inefficiencies, redundancies, and potential bugs, which is critical for maintaining a clean and scalable codebase. The Custom RAG Tool (Retrieval-Augmented Generation) uses GPT-4 to retrieve relevant documentation and examples from the codebase, helping us make precise, context-aware improvements aligned with best practices. Aider assists in applying advanced design principles by streamlining implementation and suggesting structural improvements to enhance the project's architecture and scalability. Also, halstead measures and code design standards (parent class relationships, DRY, SRP) are used to ensure code design changes are enhancing the quality of Expertiza. Together, these tools and measures streamline our workflow and ensure the Expertiza project is optimized for maintainability and long-term success.

3.1. Halstead Measure code analysis

This section outlines the metrics we use to quantify code quality and readability. The Halstead Measures, developed by Maurice Halstead in the 1970s, are well-known metrics in the field of computer science for assessing software and codebase complexity [3]. These metrics measure code complexity by counting the occurrences of unique operators (n_1) and operands (n_2), as well as the total number of tokens (operators and operands combined) [3]. In our study, we applied Halstead Measures within the Expertiza code to establish a baseline for software complexity. Additionally, we examined how code refactoring impacts overall complexity.

n_1 = number of distinct operators
n_2 = number of distinct operands
N_1 = total occurrences of operators
N_2 = total occurrences of operands

Halstead Measures estimate programming difficulty D by

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$$

and programming effort E by

$$E = D\left(N_1 + N_2\right) \times \log_2\left(n_1 + n_2\right)$$

Programming time is an estimate of the time it takes to implement or understand fully the given software [3]. This metric is calculated by

$$T = \frac{E}{fS},$$

where f = 60 (seconds per minute) and S = 18 (moments per second). S is Stroud's number, which is supposed to represent elementary discriminations performed by the human brain per second, generally set to 18 [3].

We used these equations and metrics to evaluate whether the LLMs help reduce code complexity by the suggestions they give for refactoring. The results of our experiment will rely on these metrics to validate our hypothesis that LLMs can help improve code quality. To track Halstead measures, an outside OSS tool was used to calculate the measures for the Halstead time, effort, and difficulty. The Mozilla Rust Code Analysis library can calculate these Halstead measure metrics within the codebase for many different languages, but does not have one specifically for the Ruby code in which Expertiza is written [5]. To circumvent this, we developed a Python script that integrates with the rust-code-analysis tool, originally designed for Ruby, enabling us to utilize it effectively within a Python workflow. This allowed us to parse Ruby language tokens and generate the Halstead measure metrics. Additionally, this Python script generated a report of the Halstead measures for each Ruby class or file within Expertiza. As a baseline, we first ran this script on the original Expertiza project to gain complexity data. Figure 1 shows the values we gained from running the tool against the Ruby class `User` in the Expertiza project as an example.

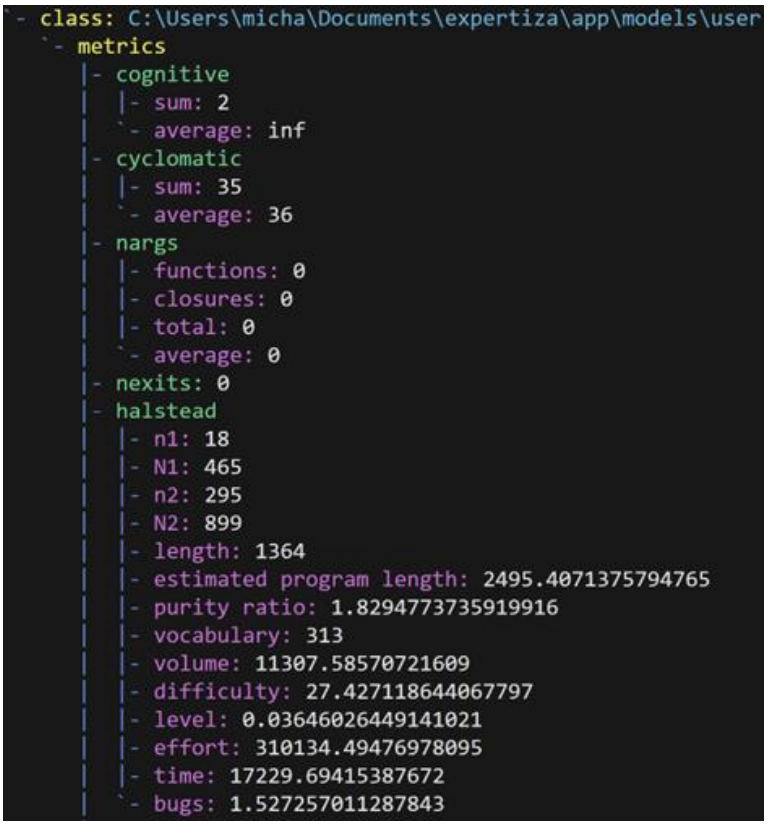


Figure 1. Example of extracted Halstead measure metrics from created Python Script

3.2 OpenAI GPT-4 LLM with additional context tools

OpenAI GPT-4 was selected as the base model for providing suggestions on the Expertiza code, as it excels in understanding code and offering design improvements [4]. A key challenge in our research, however, was GPT-4's token limit, which hinders its ability to process extensive codebases. To address this, we explored tools like Codebuddy, which vectorizes the entire Expertiza codebase as a secondary knowledge base. This approach enables GPT-4 to handle the complex Expertiza codebase more effectively. Similarly, we used OSS Aider, which scans the codebase into a repository map, giving GPT-4 contextual knowledge of Ruby classes, fields, methods, and inputs [6]. For larger repositories, Aider employs a graph-ranking algorithm to minimize token usage, with source files as nodes and dependency links as edges [6]. Aider also selects relevant files to provide focused context for the LLM. Finally, we developed a custom RAG tool that vectorizes each Expertiza code file into embeddings with GraphCodeBERT, paired with FAISS [12] for vector search, to retrieve relevant files based on user prompts. While Codebuddy and Aider were our primary tools due to their proven reliability, our custom tool helped us research how to improve LLM context for code-design suggestions.

3.3 LLM prompting

We utilized a simple prompt shown in Example 1 to improve code quality and design. This general prompt was provided to the context-enhanced LLM and focused on code design principles including the Single Responsibility Principle (SRP) and Don't Repeat Yourself (DRY). For the SRP and DRY principles, a specific prompt is shown in Example 2. We fed selected classes and files from Expertiza into Codebuddy and Aider, specifically targeting those known for being complex and hard to understand, as indicated by high Halstead complexity measures.

Prompt: how to improve complexity and understandability of the file or files given, please give code snippets and suggestions on how to change and what to improve. This includes removing methods that are unused, refactoring method names, moving methods between classes, and following design principles etc.

Example 1. General prompt given to LLM to provide related suggestions about code design of codebase

Prompt: how to improve complexity and understandability of the file or files given, please give code snippets and suggestions on how to change and what to improve. This includes removing methods that are unused, refactoring method names, moving methods between classes, and following [DRY or SRP] design principles etc.

Example 2. Prompt given to LLM to provide DRY and SRP suggestions about code design of codebase

For Object-Oriented Programming standards, we mainly focused on the correct application of subclass relationships (i.e., that they obeyed the Liskov Substitution Principle [8]). To accomplish this we fed parent-child classes in Expertiza into the LLM for context using Codebuddy and Aider and asked the question shown in Example 3.

Prompt: Given these parent-child files and classes, how can we improve code design? Please give suggestions and code snippets on what to change. This includes removing methods that are unused, refactoring method names, moving methods between classes.

Example 3. Prompt given to LLM to provide parent-child related suggestions about code design of codebase

The above prompts aimed at refactoring methods to more appropriate classes and removing duplicate code. Additionally, we addressed complexity reduction by prompting the LLM to suggest improved method names, simplify code, and generate documentation.

3.4 LLM output and testing

After inputting the prompts, the LLM generates outputs in the form of suggestions and code snippets. These suggestions were integrated into the appropriate code classes, and we tested the functionality to ensure it was maintained. Bugs and errors were identified and addressed through RSpec [7] testing files and manual local development testing of Expertiza. Suggestions that introduced errors or bugs were removed and documented, while successful suggestions that improved code design were committed to Git. The Halstead Python script was rerun to verify whether complexity had decreased for Expertiza. Suggestions related to design principles and parent-child relationships were also reviewed and validated.

4. Results

Our research demonstrates that when LLMs are provided with sufficient context about the codebase, they can effectively improve code design, reduce complexity, and generate proper documentation.

4.1 Halstead Measure complexity reduction by LLM suggestions

The Halstead Measure, evaluated after implementing LLM-suggested changes to Expertiza, demonstrated reductions in difficulty, effort, and time. All approved changes adhered to proper code design principles and caused no errors in Expertiza. The Halstead difficulty, which indicates how challenging the code is to read or write, decreased by 9.66. The Halstead time, estimating the time required to implement and fully understand the software, decreased by 2047.5 minutes. This reduction highlights the substantial additional time previously required to implement and comprehend Expertiza. Adhering to principles like method inheritance and minimizing unnecessary operators and operands, the code now reflects improved design and therefore reduced implementation time. Halstead effort, measuring the mental workload needed to reimplement the software, also decreased by 2047.5. These results demonstrate a reduction in code complexity and an improvement in readability for Expertiza, supporting our hypothesis that LLM-generated suggestions for code design can effectively reduce complexity and improve the quality of open-source software (OSS) projects. Table 1 below shows the differences for the entire expertiza system.

Table 1. Table showcasing the changes Halstead measure metrics after LLM code design changes for OSS Expertiza

	Halstead Difficulty	Halstead Effort	Halstead Time
Expertiza	2,592.71	11,457,915.66	636,550.85
Expertiza with LLM Changes	2,583.05	11,421,060.70	634,503.35
Difference	9.66	36,854.96	2,047.50
% Difference	0.37%	0.32%	0.32%

While the table above presents percentage reductions across the entire Expertiza codebase, it's important to note that the LLM was only applied to a small portion of the system, only a few classes were covered.

The differences of 0.37% and 0.32% shown above were obtained by refactoring only three classes, `Response`, `ResponseMap`, and `ResponseViewMap`, which account for just 2.59% of the total lines of code in Expertiza. If these changes (0.32% to 0.37% reduction on 2.59% of the code) could be extrapolated to the rest of the system, that would represent a decrease of 12.4% to 14.3% in Halstead metrics for the entire system. These findings highlight how even limited use of LLMs can lead to measurable improvements, and how broader adoption could enhance code readability and reduce complexity throughout large open-source projects.

4.2 Application of code design principles

The code design principles were incorporated into our prompts so that suggestions provided for Expertiza would ensure adherence to proper coding practices. Our findings indicate that design principles are addressed both when the LLM is given a specific principle to follow and when it is simply tasked with improving code design. Parent-child class relationships in OOP and DRY principles were effectively addressed, as the LLMs suggested changes were successfully applied while maintaining functionality. In Expertiza, a complex parent-child relationship exists for questionnaires, with `Questionnaire` as the parent class and several specialized child classes such as `ReviewQuestionnaire`, `SurveyQuestionnaire`, and `AuthorFeedbackQuestionnaire`. As detailed in Figure 2, when the LLM was prompted to follow the DRY principle and consider parent-child relationships with relevant Ruby files provided as context, it suggested moving the methods `post_initialization`, `get_assessments_for,` and `symbol` into the `Questionnaire` parent class. After manual analysis, we confirmed that this recommendation eliminated code duplication across multiple child classes, following the DRY principle.

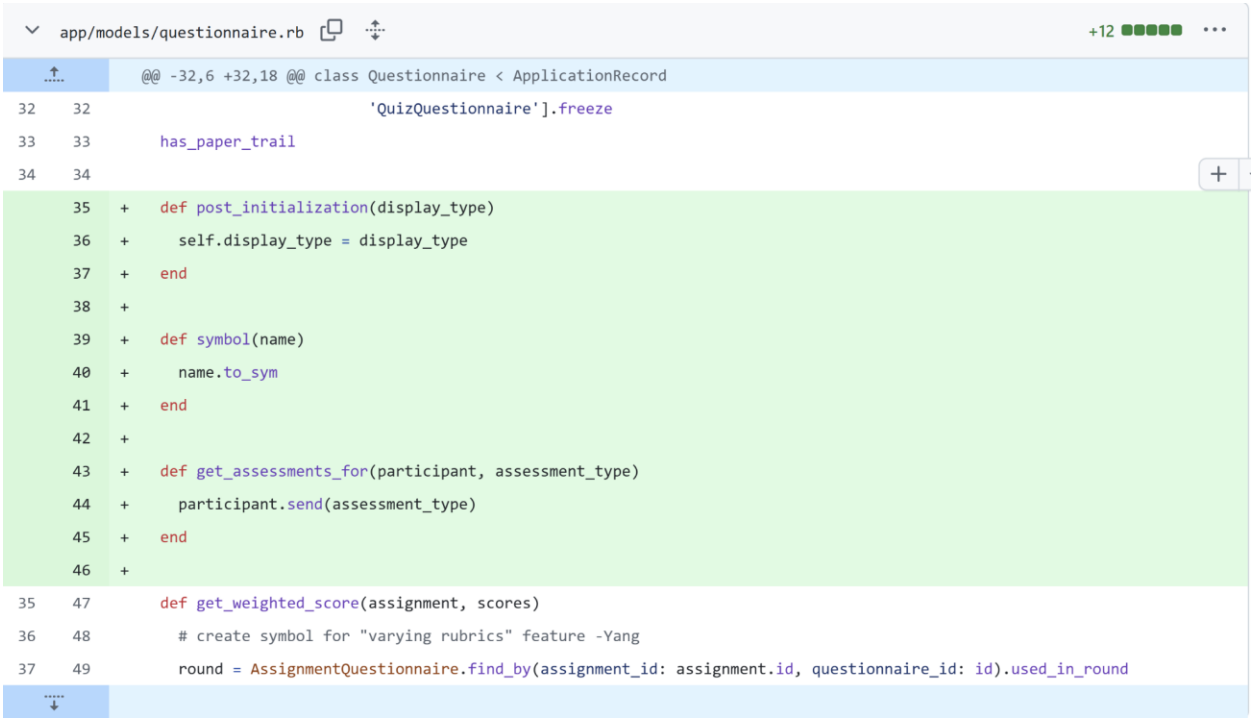


Figure 2. The LLM suggestion to move methods to the parent class Questionnaire to be utilized by multiple child classes

The changes to the inheritance hierarchy are shown in Figure 3. These suggestions successfully aligned with both DRY and LSP principles, enhancing the code's design and maintainability. The LLM was also prompted to make the code follow the Single Responsibility Principle (SRP), leading to findings that

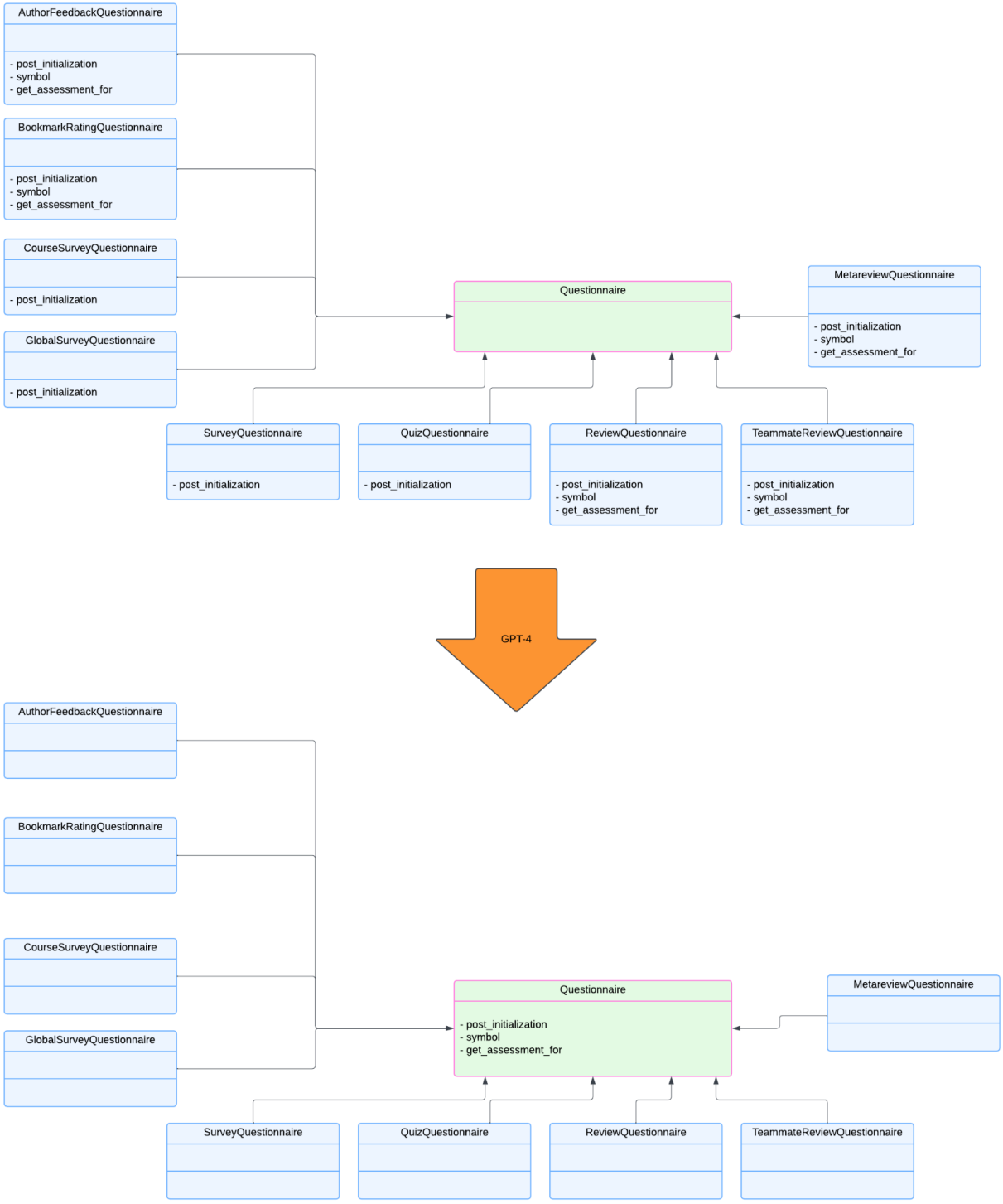


Figure 3. The class hierarchy for questionnaires before and after the suggestions given to improve code design from the LLM

suggested reorganizing several classes within Expertiza to improve readability and ensure that each class adhered to a single responsibility. However, due to limitations in the LLM's ability to maintain full context across the entire source code, some of its recommendations for method relocation were inconsistent. Despite these challenges, our experiments demonstrated that the LLM could effectively apply code design principles and provide actionable suggestions for improving code quality. Our methodology highlights the potential of LLMs to enhance code design and maintainability through adherence to established principles.

The Questionnaire hierarchy in Expertiza plays a key role in enabling feedback collection and evaluation across various contexts, such as surveys, peer reviews, and assessments. Originally, the structure of the questionnaire classes involved repetitive implementations of similar methods across multiple child classes, leading to inefficiencies and potential maintenance issues. The LLM analyzed this structure and recommended centralizing common methods into the parent class, streamlining the functionality and making the system more cohesive.

Before the change, each child class individually implemented overlapping functionalities, making it difficult to maintain consistency when updates were needed. The LLM suggested a new structure where shared logic was moved to the parent class, ensuring that all child classes could inherit and utilize the methods without duplication. This approach not only reduced redundancy but also made the system more extensible, allowing for easier addition of new questionnaire types in the future. By reorganizing the methods, the LLM improved the overall clarity and maintainability of the code while preserving the unique functionality of each child class.

5. Discussion

This study shows that LLMs like GPT-4 can help improve code quality through intelligent, context-aware suggestions. While traditional tools such as linters, static analyzers, and peer reviews still play an important role, LLMs offer more flexible and interactive support for writing better code. They respond to natural language input, understand code structure, and adapt their suggestions to the developer’s goals. LLMs give contextual guidance, meaning they don’t just check syntax or style. They look at how the code fits together across files. This makes them useful for improving not just formatting, but logic and structure too. In our research, the LLM analyzed parts of the Expertiza codebase and recommended specific changes that helped with readability and simplicity. One example is shown in Figure 4, where the LLM added a helpful comment about how the Course object is used across multiple classes in the system. This kind of high-level understanding is difficult for rule-based tools to achieve [14].

Compared to static analysis tools like SonarQube or ESLint, which are designed to enforce predefined rules, LLMs provide personalized support. They adjust their suggestions based on the current code and user input. This makes them more adaptable for different coding styles or project requirements. As shown in Figure 5, the LLM identified and removed an unused method from a file. It was able to recognize that the method served no purpose in context and suggested removing it, which reduced clutter and improved maintainability [15]. Another advantage of LLMs is their ability to help developers learn better coding practices while writing code. They can act like mentors by explaining why a certain change is helpful or what principle it supports. For example, they may suggest breaking large functions into smaller ones,

replacing nested logic with cleaner alternatives, or naming variables more clearly. These kinds of suggestions not only improve the current code but also teach better habits over time [14].

We measured the impact of these suggestions using Halstead metrics. After applying the LLM's changes, we saw improvements in difficulty, effort, and time. This suggests the code became easier to read, maintain, and understand, which is a key goal in software quality. While LLMs don't replace automated testing or peer reviews, they are a strong addition to the toolkit, especially when used together with traditional methods [16]. Ultimately, LLMs give developers a more interactive and dynamic way to reinforce good coding practices. They go beyond syntax checks and allow for continuous learning and refactoring. Tools like GPT-4, when combined with static analyzers or automated tests, can make the development process faster, cleaner, and more understandable, especially in large or complex codebases [15].

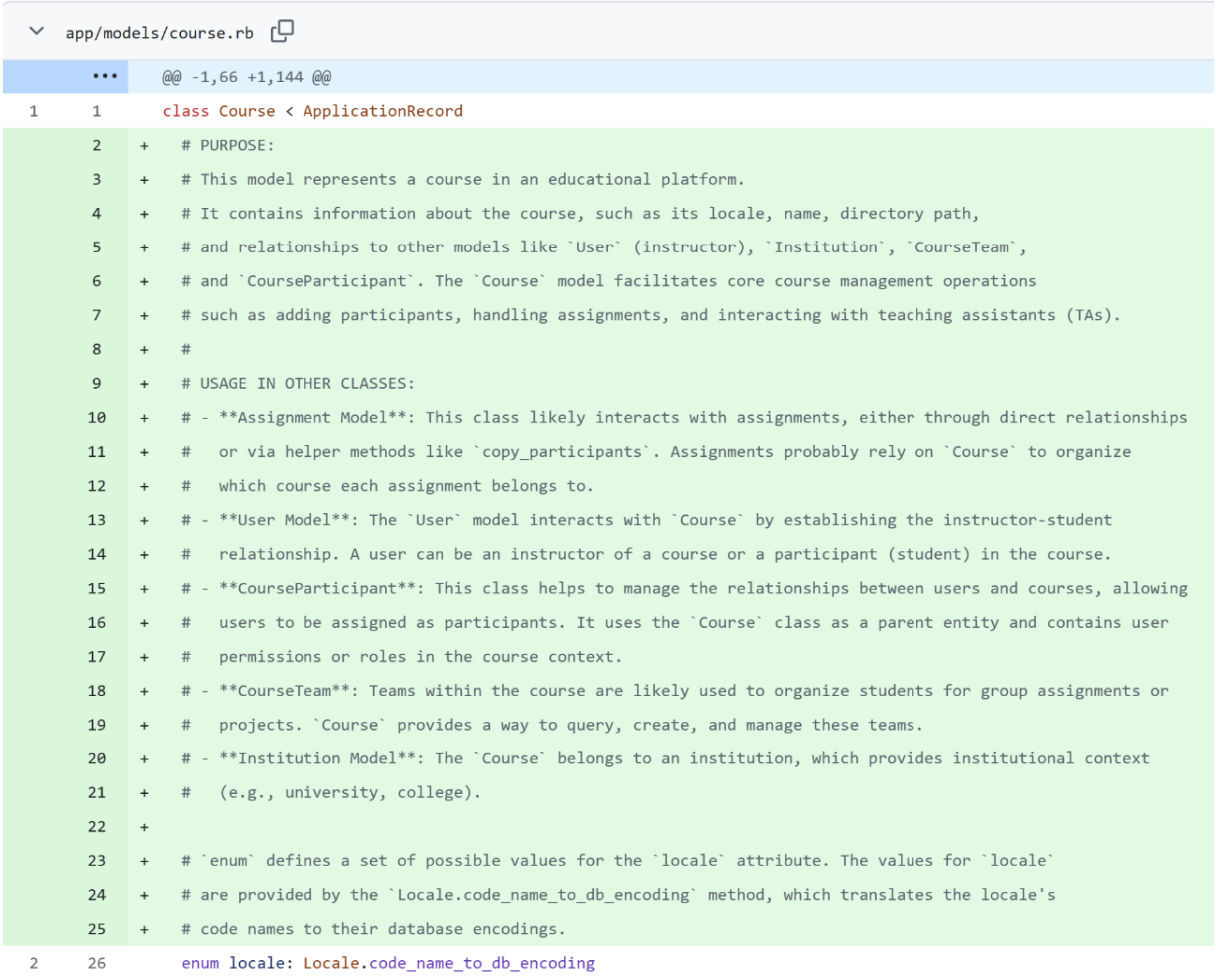
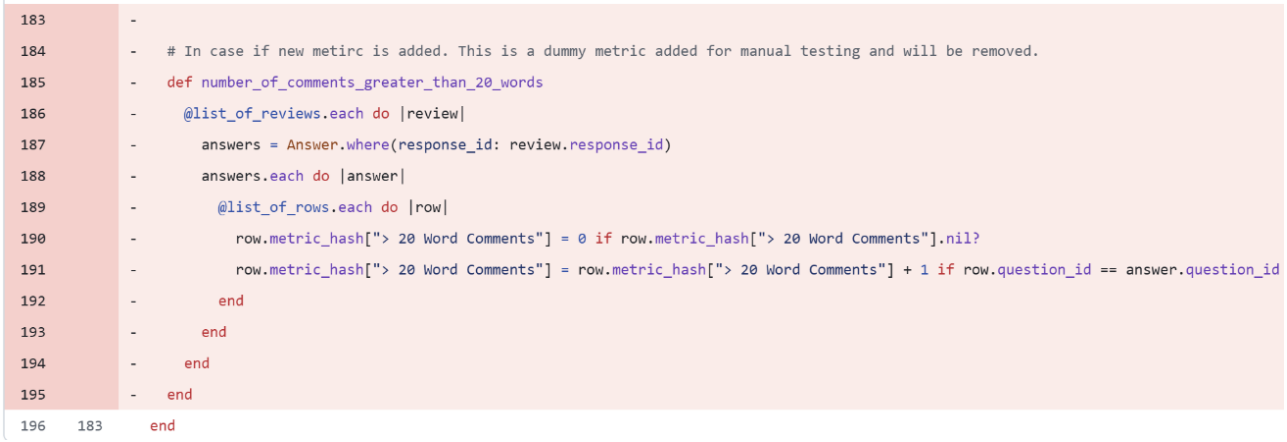


Figure 4. The LLM code comment addition that showcases connection to other classes and Course object use



```
183 -
184 -   # In case if new metirc is added. This is a dummy metric added for manual testing and will be removed.
185 -   def number_of_comments_greater_than_20_words
186 -     @list_of_reviews.each do |review|
187 -       answers = Answer.where(response_id: review.response_id)
188 -       answers.each do |answer|
189 -         @list_of_rows.each do |row|
190 -           row.metric_hash["> 20 Word Comments"] = 0 if row.metric_hash["> 20 Word Comments"].nil?
191 -           row.metric_hash["> 20 Word Comments"] = row.metric_hash["> 20 Word Comments"] + 1 if row.question_id == answer.question_id
192 -         end
193 -       end
194 -     end
195 -   end
196 183   end
```

Figure 5. The LLM code suggestion that removed unused code from an independent class

Our findings demonstrate the potential of LLMs to transform OSS projects by automating refactoring and improving code design. However, current limitations, such as token constraints and inconsistent context handling, highlight the need for more specialized tools or training methods.

6. Conclusion

This study highlights the transformative potential of Large Language Models (LLMs), like GPT-4, in improving the quality and maintainability of open-source software (OSS) projects. By applying these models to the Expertiza codebase, we demonstrated their effectiveness in enhancing code design, reducing complexity, and generating meaningful documentation. Our results show a tangible reduction in software complexity, as measured by Halstead metrics, with clear decreases in difficulty, effort, and implementation time. These improvements reflect the ability of LLMs to suggest meaningful refactoring that adheres to established code design principles such as DRY, Single Responsibility, and object-oriented parent-child hierarchies.

The LLM's suggestions, such as consolidating methods in parent classes for inheritance and removing redundant code, improved maintainability and readability while maintaining full functionality. When given context, LLMs provide detailed documentation and comments, and identify code smells effectively. However, challenges remain, particularly when the LLM lacks full context of the entire codebase. While generally accurate and beneficial, some inconsistencies in recommendations highlight the importance of comprehensive context and combining automated insights with manual verification.

Overall, our findings affirm that LLMs are valuable tools for OSS development. They improve code quality, reduce complexity, and enhance documentation. By integrating these tools into software development workflows, teams can streamline refactoring processes, adhere to design principles, and foster more robust and maintainable software. This research paves the way for broader adoption of LLMs in OSS projects, contributing to a future where AI and human developers collaborate seamlessly to advance software innovation.

7. Future Work

For future work, one priority is addressing the limitations of the LLM when it comes to understanding the full context of a codebase. Currently, the model struggles with larger and more interconnected tasks, where the relationships between classes or components play a crucial role. To improve this, we plan to explore ways to provide the LLM with more detailed and relevant input, ensuring it has a clearer picture of the code's structure. This could involve breaking tasks into smaller steps that better align with the model's strengths or providing additional metadata about the code. By focusing on these improvements, we aim to make the LLM more reliable for handling real-world scenarios, especially in projects where the scope goes beyond isolated functions or methods.

Another key area of future work involves testing the LLM on more complex and larger-scale refactoring tasks. This includes scenarios where the codebase exceeds 1,000 lines and involves multiple tightly coupled components. These tests will allow us to assess not only the LLM's ability to make accurate suggestions but also how it scales with increasing complexity. Additionally, we plan to evaluate the model’s performance on specific challenges like reducing class coupling or improving readability across interconnected files. By tackling these challenges, we hope to uncover strategies for making the LLM a more effective tool for developers working on projects of all sizes. These efforts will help bridge the gap between theoretical capabilities and practical applications.

8. References

[1] Edward F. Gehringer, Luke M. Ehresman, Susan G. Conger, and Prasad A. Wagle, “Reusable learning objects through peer review: The Expertiza approach,” *Innovate—Journal of Online Education* 3:6 (August/September 2007).

[2] Edward F. Gehringer, “Expertiza: information management for collaborative learning,” in *Monitoring and Assessment in Online Collaborative Environments: Emergent Computational Technologies for E-Learning Support*, A. A. Juan Perez [ed.], IGI Global Press, 2009.

[3] T. Hariprasad, G. Vidhyagaran, K. Seenu and C. Thirumalai, "Software complexity analysis using Halstead metrics," 2017 International Conference on Trends in Electronics and Informatics (ICEI), Tirunelveli, India, 2017, pp. 1109-1113, doi: 10.1109/ICOEI.2017.8300883.

[4] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun, "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?," arXiv preprint arXiv:2310.12357, 2023.

[5] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, "rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, p. 100635, 2020. [Online]. Available: https://doi.org/10.1016/j.softx.2020.100635 (https://github.com/mozilla/rust-code-analysis.)

[6] Aider, "Aider: AI Pair Programming in Your Terminal," GitHub repository, 2024. [Online]. Available: https://github.com/Aider-AI/aider. [Accessed: Oct. 27, 2024]

[7] RubyGuides, "The Definitive RSpec Tutorial With Examples," 2018.

[8] B. Liskov and J. Wing, "A Behavioral Notion of Subtyping," ACM Transactions on Programming Languages and Systems, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[9] R. C. Martin, "The Single Responsibility Principle," Clean Coder Blog, May 2014.

[10] A. Hunt and D. Thomas, The Pragmatic Programmer: From Journeyman to Master, Addison-Wesley, 1999.

[11] CodeBuddy, "CodeBuddy: Collaborate and Share Code," CodeBuddy Platform, 2025. [Online]. Available: https://www.codebuddy.com.

[12] Facebook AI, "FAISS: A library for efficient similarity search and clustering of dense vectors," GitHub repository, 2025. [Online]. Available: https://github.com/facebookresearch/faiss.

[13] Expertiza, "Expertiza: A peer-review system for education," GitHub repository, 2025. [Online]. Available: https://github.com/expertiza/expertiza.

[14] LLM-Powered Code Review: Top Benefits & Key Advantages, https://api4.ai/blog/llm-powered-code-review-top-benefits-amp-key-advantages

[15] Jaoua, I., Sghaier, O. B., & Sahraoui, H. (2025). Combining Large Language Models with Static Analyzers for Code Review Generation. arXiv preprint arXiv:2502.06633.

[16] Wadhwa, N., Pradhan, J., Sonwane, A., Sahu, S. P., Natarajan, N., Kanade, A., ... & Rajamani, S. (2024). Core: Resolving code quality issues using llms. Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE), 789-811.