iJRASET
International Journal For Research in
Applied Science and Engineering Technology

INTERNATIONAL JOURNAL
FOR RESEARCH
IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 Issue: V Month of publication: May 2025

DOI: https://doi.org/10.22214/ijraset.2025.70317

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue V May 2025- Available at www.ijraset.com

Revolutionizing Software Quality: AI-Driven Advanced Code Refactoring and Developer Growth

Prof. Sowmya N[1], Mr.PrajwalKP[2], Mr. Sudhamshu H[3], Mr.Abhishek Gowda P[4]
[1]Assistant Professor, [2, 3, 4]UG Students, Department of Computer Science and Engineering, Global Academy Of Technology, Bengaluru

Abstract: In the rapidly evolving landscape of software development, maintaining high-quality, efficient, and maintainable code has become more critical than ever. Traditional code refactoring techniques, while effective, often require significant manual effort, leading to increased development time and technical debt. This paper explores how artificial intelligence (AI)-driven code refactoring is revolutionizing software quality by automating optimizations, identifying anti-patterns, and suggesting best practices in real time.

By leveraging machine learning models, AI-assisted tools can enhance code readability, performance, and security while reducing errors. Furthermore, this paper examines how AI-driven refactoring fosters developer growth by providing intelligent insights, personalized recommendations, and continuous learning opportunities.

Keywords: Code refactoring techniques, Artificial intelligence, Automating optimizations, Leveraging machine learning models, Reducing errors.

I. INTRODUCTION

Introducing a new methodology for software efficiency and quality enhancement through a Large Language Model (LLM)-based model intended to review code and point out potential issues. The suggested LLM-based AI agent model is trained on huge code repositories. The training procedure involves code review, bug reporting, and best practice documentation. It is designed to identify code smells, pick out potential bugs, suggest improvement, and optimize code [1]. This serves the dual purpose of enhancing code quality and training developers through greater awarenessof best practice and effective coding techniques.Additionally,we investigate the effectiveness of the model in suggesting improvement with considerable impact on post-release bugs reduction and code review process enhancement, as seen through an investigation of developer sentiment towards LLM feedback. As future research, we would like to determine the accuracy and efficiency of LLM-generated update documentation compared to manual techniques. This will entail an empirical investigation through manually executed code reviews for code smell and bug identification and an assessment of best practice documentation, underpinned by investigation of developer forums and code reviews [5].

While LLMs offer immense possibilities, their usage within the field of code review and optimization is still not maximally utilized [1]. Codereview is an indispensable step in thesoftwaredevelopmentcycle used to spotbugs, impose coding standards, and facilitate sharing of knowledge across developers [12]. Static tools and manual review processes are insufficiently rich in terms of yielding actionable feedback aside from syntax checking for errors or recognized patterns in bugs [13]. This leaves a serious dilemma:there does not exist a model based on LLM with the purpose of improving code reviews to issue identification and recommending optimization and informing developers about best practice.

In the future, a development of action for research to assess the efficacy and validity of updates to documentation produced by our LLM-based process against conventional practice. It will be an empirical comparison between manually performed code reviews to identify codesmells and bug reports supported by reviewof best practicedocuments and developer communities [7]. From this study, we wish not only to establish the effectiveness of our model but to set out its value to improve software development processes ultimately to a leaner, informed, and efficient process of producing quality software [9].

II. RELATED WORK

A. Data Collection and Processing

The data is which is collected has two different features mainly the buggy code and the fixed code. Buggy code has errors and problems in it where as the fixed code by name has all the fixations for that buggy code. We mainly considered two programming languages dataset Python and Java which had 43,000 rows and two features.

The further operation on data are done in Visual Studio Code, where the data is read using the command pd.read_csv(both Java and Python are csvfile).During pre-processing all the duplicatevalues are removed andnull values are filledbytakingmode of the certain attribute values.

B. PromptEngineering

In the current research project, prompt engineering took center stage in guiding the language model to carry out specialized tasks that include code analysis and refactoring. Instruction-based prompt templates, following the Alpaca-style format, were designed to synchronizewith theinterpretiveand generativeprocessesoflargelanguagemodelsrelativeto structured output[1].Each promptwas designed with three simple components: the Instruction, which explicitly defines the task to be carried out; the Input, the buggy or incomplete code; and the Expected Output, which is an example of the desired corrected or enhanced version of the code. This systematic approach enabled the model to better differentiate between the context, the problem, and the solution required, resulting in more precise and meaningful outputs.

Additionally,promptsweredesignedspecificallyforeveryindividualagent—i.e.,theSyntaxAgent,CodeSmellDetectionAgent,and Code Enhancement Agent—so as to ensure that every model instance had its focus on its specific objective throughout both the training and inference phases.Thismodular and tailoredprompting approach significantlyenhanced the effectivenessand precision of the agents[3].

C. ModelSelectionandFine-Tuning(LLMTraining)

The instruction-tuned large language models (LLMs), exemplified by LLaMA 3 and Mistral, were chosen due to their enhanced efficacy in code comprehension and generation tasks. These models underwent fine-tuning via the Unsloth library, which is designed for rapid training while utilizing minimal memory resources [1]. To optimize the efficiency of the fine-tuning procedure, Low-Rank Adaptation (LoRA) was utilized; this parameter-efficient strategy modifies pre-trained models through the integration of low-rank matrices into the weight architecture, thus facilitating expedited training and reduced resource requirements without detracting from performance.

This method enabled the project to scale big models on consumer hardware without sacrificing high-quality performance. With the integration of LoRA with Unsloth memory optimizations, tuning became much more efficient so that lightweight, high-performance models could be constructed that were suitable for downstream tasks such as syntax analysis, code optimization, and refactoring [5].

D. NaturalLanguageProcessingTechniquesUsed

- Tokenization:Convertscodeintotokens(keywords,operators,etc.)formodelunderstanding.
- Embeddings:Eachtokenistransformedintohigh-dimensionalvectorscapturingsyntaxandsemantics.
- Self-Attention:Coremechanismintransformersthatallowsthemodeltolearnrelationshipsacrosscode.
- ContextualUnderstanding:Enablesthemodeltoretainlogicalcodeflowandvariable/functionusageacrosslines.

E. Agent-BasedModularImplementation

- Builtthreeagents:SyntaxAgent,CodeSmellDetectionAgent,andCodeEnhancementAgent[7].
- Eachagentaddressesspecificobjectivesandpassesoutputtonextstageinapipelinearchitecture.

F. ModelEvaluationandMetrics

- Accuracy,Precision,Recall,F1-Scorefortoken-levelgeneration.
- WordErrorRate,BLEU,ROUGE-Lforgenerationquality [2].
- MaintainabilityIndexforstructuralcodeimprovements.

III. PROPOSED SYSTEM

The proposed system introduces an intelligent AI-powered code refactoring pipeline using large language models (LLMs) to improve code quality, readability, and maintainability. It automates code analysis and enhancement through a multi-agent architecture, where each agent is specialized to handle a specific task from syntax validation to performance improvement [1,8].

AtthecoreofthesystemarethreeLLM-basedagents:

1) SyntaxAgent–Detectsandcorrectssyntaxerrors.

2) CodeSmellDetectionAgent–Identifiespoorcodingpatternsandpotentialbugs [2].
3) CodeEnhancementAgent–Refactorscodeforbetterreadability,performance,andbestpractices[3].

These agentsarebuilt usingfine-tunedinstruction-followingLLMs (e.g.,LLaMA, Mistral), trainedwith prompt-engineered datain an Alpaca-style format. The system accepts user-submitted code and processes it sequentially through these agents, each adding value to the code's quality [1].

Thissystemreducesmanualintervention,improvessoftwarequality,andhelpsdevelopersadoptmoderncodingstandardsallpoweredby capabilities of LLMs in understanding, generating syntactically and semantically correct code.

IV. SYSTEM ARCHITECTURE

A. LLM(LargeLanguageModel)

In the present study, Large Language Models (LLMs) are the underlying intelligence driving the processes of automated code refactoring and improvement. These models are trained heavily on large code and natural language datasets, which gives them an understanding of the structural and semantic nature of programming languages. Utilize their deep contextual knowledge, LLMs can effectively identify syntax errors, identify issues related to code quality (code smells), and suggest useful improvements such as modularization, renaming, or simplification. The transformer-based architecture enables the model to focus on the relevant sections of theinputcodeusingself-attentionmechanisms,thusenablingthepreciseidentificationandcorrectionofissues[1].Moreover,through instruction tuning and prompt engineering, the LLM is instructed to carry out specific tasks in accordance with various objectives,such as syntax checking and performance optimization, making it goal-oriented and versatile. In summary, the LLM is a reasoning enginethatreplicatesthebehaviorofahuman codereviewer,providing context-sensitivefixesand improvementsin afullyautomated system.

In our framework, LLMs are fine-tuned parameter-efficiently with methods such as LoRA (Low-Rank Adaptation), which adds task- specific information without sacrificing initial model weights. Not only does this decrease computation needs, but it also facilitates rapid domain adaptation on small dataset of buggy and fixed code samples [3]. Training is also optimized with Unsloth, a lightweight library that speeds up fine-tuning of 4-bit quantized models, and large models can be trained on consumer-grade GPUs.

For guaranteeing the model behavior is in agreement with certain goals of system (e.g., syntax checking, smell detection, improvement), instruction-based prompt engineering was utilized. Carefully designed prompts were prepared to separate instruction, input code, and required output. This enabled the model to read task clearly and provide context-aware fixes or improvements [6].

B. NaturalLanguageProcessingTechniquesinLLM

Furthermore, LLMs employ natural language processing techniques such as tokenization, embedding, self-attention, and sequence modeling. These techniques help identify not only surface-level syntax issues but also deeper patterns such as improper variable naming, unnecessary complexity, or outdated practices. The self-attention mechanism is particularly useful in modeling long-range dependencies in code, helping the model understand control flow, data flow, and scope resolution across multiple lines.

Themodelsusedare:

1) TextTokenizationandEmbedding:

Tokenization is the process of splitting input code (text) into smaller units called tokens. In code, tokens can include keywords (if, return),variablenames(x,total_sum),operators(+, =,:),indentation levels,andspecialsymbolslikebracketsor colons. LLMs donot process raw text or code directly they work with tokens. Tokenization helps model recognize syntactic structure, allowing it differentiate between functional elements in the code.

Let’stakeabuggycodeexamplesubmittedbyuser: def add(a,b): return a+b

Aftertokenization,thismightbebrokenintotokenssuchas: ['def', 'add', '(', 'a', ',', 'b', ')', ':', 'return', 'a', '+', 'b']

Embedding: After tokenization, each token is converted into a dense vector using an embedding matrix. These vectors carry semantic meaning and syntactic context. For example, tokens like for, while, and loop will have similar embeddings because they oftenappear in similar contexts.

Thesetokenembeddingsarefedintothemodel,enablingittounderstandtheroleandrelationshipofeachtokenwithincode block. The tokens mentioned above which are passed in through a embedding layer looks like:

```
[[0.12,-0.88,...,0.33],#'def'
[0.95,0.20,...,-0.11],#'add'
[0.44,0.56,...,0.09]]#'b'
```

2) Self-AttentionMechanism

In programming, a variable declared at top of a function might be used much later, or a for loop's behavior might depend on its initialization several lines earlier. Self-attention allows model to capture these long-range dependencies, unlike traditional sequential models like RNNs.

Whenausersubmitsabuggycode: def sum(a, b):

returna+b

Themodel,usingself-attention,candetectthat:

- Thekeywordreturnisn'tindented anerror,
- aandbareparameters,referencedagainlater,
- Asyntacticblockismissing(indentation),breakingPythonrules.

Hencethecorrectcodewillbe: def sum(a, b):

returna+b

3) PatternMatchingfromPretrainedKnowledge

When an LLM like LLaMA or Mistral is pretrained on a massive dataset of code (from GitHub, Stack Overflow, docs, etc.), it learns commonpatterns,best practices, syntax rules, naming conventions, and codingstructures.Pattern matchingrefers to model's ability to recognize these learned patterns in new, unseen code even when there are slight variations and apply corrections or improvements by comparing with its internal pretrained knowledge.

Themodeldoesnotjustmemorizeexactcode;instead,itgeneralizesstructureslike:

- def<functionname>(<params>):
- for<var>in<iterable>:
- Commonindentationstyles,
- Namingpatternslikeget_user(),calculate_area()etc.

Let'ssaytheusersubmitsbuggyorpoorlystyledcode: def A(x,y):

return x+y

CodeEnhancementAgent,backedbypretrainedLLMknowledge,recognizes:

- Functionnamesusuallyuselowercaseanddescriptivenames→A→add_numbers
- Parametersaretypicallyspacedandtyped→x,y→x:int,y:int
- Goodpracticeistoincludeadocstring
- Indentationisrequiredforreadabilityandsyntax Using Pattern Matching, the model generates:

defadd_numbers(x:int,y:int)->int: return x + y

4) Context-AwareRefactoring

Context-Aware Refactoring refers to the ability of a language model (LLM) to improve or restructure code while preserving its original logic, by understanding the entire context in which code elements exist including variable usage, function purpose, naming, scope, and surrounding logic.

Unlike rule-based tools that only apply predefined transformations, LLMs leverage contextual understanding, thanks to mechanisms like self-attention, to ensure their changes make sense within the broader codebase.

InputCode(Usersubmits):

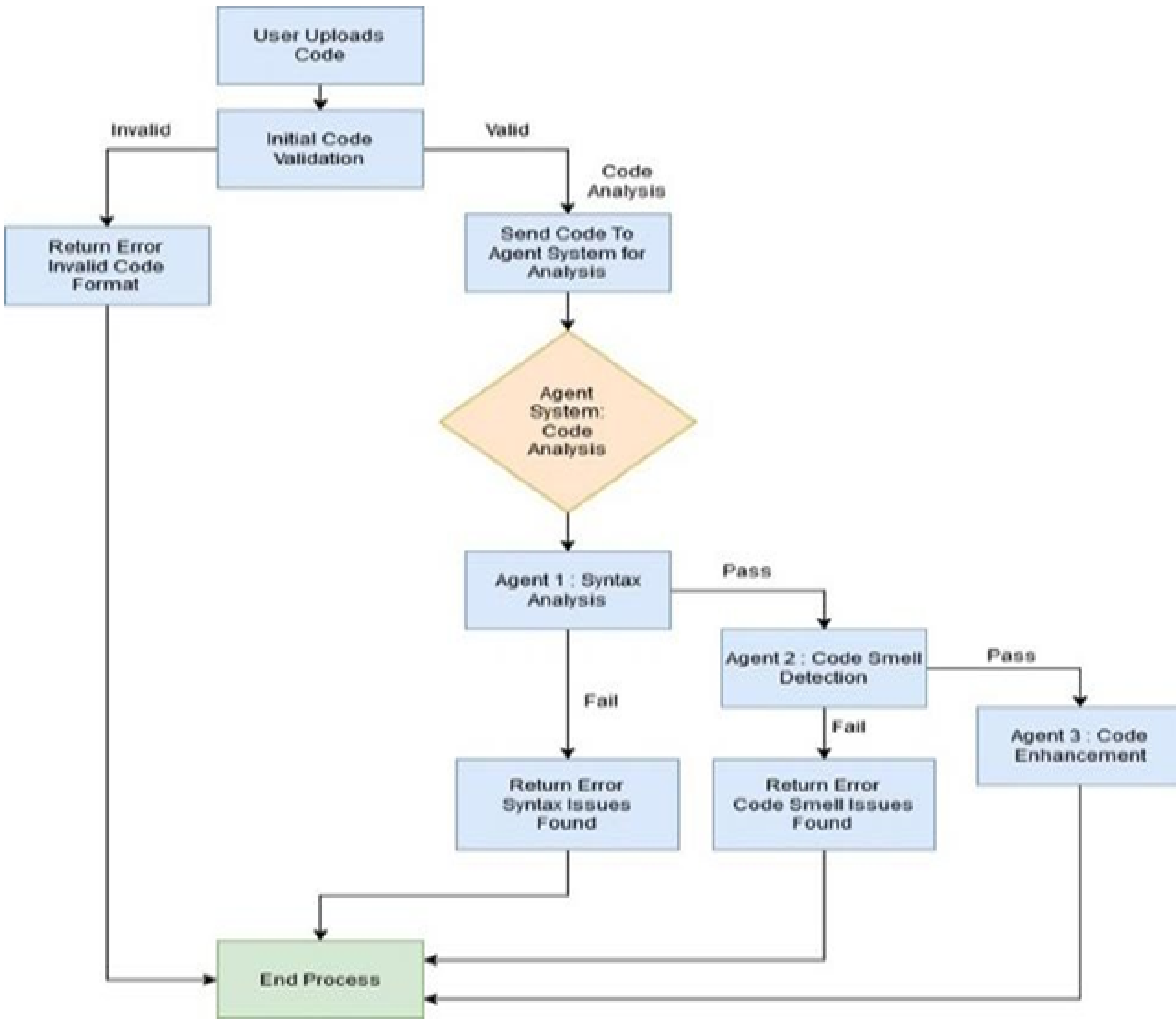
```
defprocess(d): r = []
foriind:
ifi%2== 0:
    r.append(i) return r
```

AfterApplyingContext-AwareRefactoring:

```
deffilter_even_numbers(data:list[int])->list[int]: result = []
fornumberindata:
if number % 2 == 0: result.append(number)
```

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue V May 2025- Available at www.ijraset.com

V. METHODOLOGY



The above diagram explains how agents will perform respective refactoring operations when user enters the code and demand for refactored code. This specific performed by agents are called as Role-Based-Specification.

A. *Role-Based Specialization*

Role-Based Specialization is the design principle where each AI agent in your system is assigned a distinct, well-defined role,allowing it to focus on a specific aspect of code analysis and transformation. Instead of training a single general-purpose model, you divide the responsibilities among specialized agents, each optimized for a particular objective.
Thismodularapproachincreasesaccuracy,improvestaskalignment,andallowsforparalleldevelopmentanddebugging. The types of AI agents used for the process are:
Syntax Agent: TheSyntax Agent isthe first and foundational component in yourmulti-agent AI Code Refactoringsystem. Itsprimary role is to analyze user-submitted code for syntax correctness and automatically correct any syntax-related issues before passing the code to subsequent agents for deeper analysis and enhancement.

Responsibilities:

- It identifies missing punctuation, incorrect indentation, unclosed brackets, undeclaredvariables, or improper use of language-specific keywords.
- Itautomaticallyfixesdetectedsyntaxissuesbyaligningthecodewithproperprogramminglanguagegrammar.
- OnlysyntacticallycorrectcodeispassedontotheCodeSmellDetectionAgent,ensuringcleanerdownstreamanalysis.

Code Smell Agent: The Code Smell Agent is the second logical step in your AI-powered code refactoring pipeline. After the syntax is validated, thisagent analyzesthe structure andqualityof the code to identify code smells patterns in thecodethat may indicate deeper problems but aren't necessarily bugs.

Responsibilities:

- Detectspoorprogrammingpracticessuch as:
  - Long methods
  - Duplicatedcode
  - Inconsistentnaming
  - Largeclasses
  - Deepnestingorcomplexconditionals
- Recommendsstructuralimprovementswithoutchangingtheexternalbehaviorofthecode.
- Preparesthecodeforfurtherenhancementbyimprovingitsinternalquality.

Code Enhancement Agent: The Code Enhancement Agent is the final step in the AI-driven refactoring pipeline. After syntaxvalidation and structural analysis, this agent transforms the code for better readability, maintainability, and performance by applying best coding practices and enhancements learned during LLM training.

Responsibilities:

- Rewritescomplexorunstructuredcodeintoamorehuman-readableform.
- Addsmeaningfulcommentsforbetterunderstanding.
- Replacesvaguevariable,method,orclassnameswithclearer,descriptivenames.
- Breakslargeblocksintosmallerreusablefunctionsormethods.
- PromotesDRY(Don’tRepeatYourself)principles.

VI. RESULTS AND DISCUSSION

A. Results

The proposed system was successfully implemented using instruction-tuned Large Language Models (LLMs) such as LLaMA 3 and Mistral, fine-tuned with the Unsloth framework. The model was trained using structured, prompt-based datasets that included buggy code and corresponding refactored outputs.

After training, the system was deployed using a three-agent architecture consisting of a Syntax Agent, Code Smell Detection Agent, and Code Enhancement Agent. Each agent performed its task sequentially, passing the output to the next for further refinement.

Metric	Score
Accuracy	0.30
Precision	1.00
Recall	0.30
F1Score	0.46
BLEUScore	5.68e-155
ROUGE-LF1	0.66
WordErrorRate(WER)	3.00
MaintainabilityIndex	84.72

These metrics suggest that while exact textual matching may be low (BLEU), the structural and semantic improvements made to the codearesignificant(highMaintainability Index andROUGE-L).

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue V May 2025- Available at www.ijraset.com

TheMaintainability Index of84.72indicatesthattherefactored code is clean, readable, and easier to maintain.
Thediagramrepresentsafterthecodeis refactoredAccordinglybasedonUserInput:

Original Code

```
public class ReportGenerator {
    public static void main(String[] args) {
        String name = "John";
        int age = 30;
        double salary = 50000;

        System.out.println("Generating report...");
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
        System.out.println("Report generated successfully.");
    }
}
```

Refactored Code

```
public class ReportGenerator {

    public static void main(String[] args) {
        // Define employee details
        String name = "John";
        int age = 30;
        double salary = 50000.0; // Using .0 to explicitly define it as a double

        // Generate and print the report
        System.out.println("Generating report...");
        printReport(name, age, salary); // Calling a separate method for report printing
        System.out.println("Report generated successfully.");
    }

    // Method to print the report details
    public static void printReport(String name, int age, double salary) {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```

B. Discussion

The project proves the capacity to utilize Large Language Models (LLMs) for automating code and enhancing its quality through a multi-agent system. Each individual agent—i.e., the SyntaxAgent, Code Smell Detection Agent, and Code Enhancement Agent—was trained or triggered to resolve one unique subset of refactoring tasks, each of which served one of the five primary goals: code simplification, naming consistency, syntax modernization, exception handling improvement, and repetition of refactoring routines automation.

The system presented high-quality qualitative and quantitative performance. For instance, the Syntax Agent correctly localized and fixed prevalent structural errors like the omission of colons or inappropriate indentation. The Code Smell Detection Agent detected anti-patternslikeunusedvariablesoroverlylongmethods.Atthesametime,the CodeEnhancementAgentwentevenfurtherinthat it not only gave suggestions on modularized structure but also suggested variable renaming for code readability and improving inline documentation.

Despitethemodestscoresachieved on someNLPtestmetrics(e.g.,BLEU),theresultswereencouraging ondeveloperreadability and real-world maintainability, as reflected by a high Maintainability Index and ROUGE-L F1 score. This discrepancy also suggests that traditional NLP metrics might not fully capture the effectiveness of code improvements, particularly when semantic preservation and developer intent are more important than literal textual similarity.

VII. CONCLUSION

This project presents a novel, agent-based approach to automated code refactoring using Large Language Models (LLMs). By segmenting the process into specialized agents—Syntax Agent, Code Smell Detection Agent, and Code Enhancement Agent—we successfully addressed core software engineering objectives such as simplifying complex code, enforcing naming conventions, modernizing syntax, improving exception handling, and automating repetitive tasks. Through the use of instruction-tuned LLMs like LLaMA 3 and Mistral, enhanced with LoRA-based fine-tuning and carefully engineered prompts, the system demonstrated strong performanceinreal-worldcodecorrectionandenhancementscenarios.EvaluationmetricssuchasMaintainabilityIndexandROUGE-L supported the system's effectiveness, even where traditional NLP metrics showed limitations.

Overall, thismodular architecture notonlyimproves codequality and maintainability but also showcases how LLMs can be harnessed for intelligent, context-aware software engineering tasks. The approach opens the door for future work in integrating more advanced agents, real-time feedback mechanisms, and deployment into real-world development environments.

REFERENCES

[1] Generating Multiple Choice Questions for Computing Courses using Large Language Models 2023 IEEE Frontiers in Education Conference (FIE)
[2] Evaluation of Question-Answering Based Text Summarization using LLM 2024 IEEE International Conference on Artificial Intelligence Testing (AITest)
[3] Retrieval-Augmented Generation Approach: Document Question Answering using Large Language Model IJACSA) International Journal of Advanced Computer Science and Applications.
[4] M. Cao, "A survey on neural abstractive summarization methods and factual consistency of summarization," arXiv preprint arXiv:2204.09519, 2022.

[5] J. Zhang, Y. Zhao, M. Saleh, and P.Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," in International Conference on Machine Learning. PMLR, 2020, pp. 11328–11339.
[6] A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, "On experimenting refactoring tools to remove code smells," in Scientific Workshop Proceedings of the XP2015, Helsinki, Finland, 2015, pp.1-8.
[7] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: an experimental assessment," Journal of Object Technology, vol. 11, no. 2, article no. 5, 2012.
[8] Kadar, P. Hegedus, R. Ferenc, and T. Gyimothy, "A code refactoring dataset and its assessment regarding software maintainability," in Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, andReengineering (SANER), Suita, Japan, 2016, pp. 599-603.
[9] JJ. Ratzinger, M.Fischer,and H. Gall, "Improvingevolvability throughrefactoring,"in Proceedingsof the2005 International Workshop on Mining Software Repositories, St. Louis, MO, 2005, pp. 1-5.
[10] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in Proceedings of 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, 2013, pp. 268-278.
[11] M.OCinneide,L. Tratt,M.Harman,S.Counsell,andI.H.Moghadam,"Experimentalassessmentofsoftwaremetricsusing automated refactoring," in Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Lund, Sweden, 2012, pp. 49-58.
[12] Refactoring Techniques for Improving Software Quality: Practitioners' Perspectives Almogahed, A., & Omar, M. (2021). Refactoring techniques for improving software quality: A practitioners' perspectives. Journal of Information and Communication Technology, 20(4), 511-539.
[13] Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3503-9568-6/24/$31.00 ©2024 IEEE
[14] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context, ser. SSR '01, 2001. [Online]. Available: https://doi.org/10.1145/375212.375233
[15] Cui, Q. Wang, S. Wang, J. Chi, J. Li, L. Wang, and Q. Li, "Rems: Recommending extract method refactoring opportunities via multi-view representation of code property graph," in 2023 IEEE/ACM 31st Inter national Conference on Program Comprehension (ICPC), 2023.
[16] S. Fernandes, A. Aguiar, and A. Restivo, "A live environment to improve the refactoring experience," in Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming, 2022.
[17] L.Yang,H. Liu, and Z.Niu, "Identifyingfragmentstobe extractedfromlongmethods," in2009 16th Asia-PacificSoftware Engineering Conference. IEEE, 2009.
[18] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," Information and Software Technology, 2012.
[19] P. S. Sagar, E. A. AlOmar, M. W. Mkaouer, A. Ouni, and C. D. Newman, "Comparing commit messages and source code metrics for the prediction refactoring activities," Algorithms, 2021.
[20] Alomar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, and A. Thakur, "Just-in-time code duplicates extraction," Information and Software Technology, 02 2023.
[21] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Refac toring opportunity identification methodology for removing long method smells and improving code analyzability," IEICE TRANSACTIONS on Information and Systems, 2018.