Improving Code Maintainability and Performance with LLM-Driven Automation

Sunny Bhatt, Ziwei Guo
Department of Computer Science, *Vanderbilt University, Tennessee*
Nashville, TN, USA
{sunny.a.bhatt, ziwei.guo}@vanderbilt.edu

***Abstract*—This paper presents a comprehensive approach to improving software quality and development efficiency by integrating prompt patterns for code complexity analysis, structured code reviews, adaptive documentation, and data analysis. It addresses common challenges in maintaining high-quality code in fast-paced, agile environments, including the difficulties of keeping documentation synchronized with frequent code changes, reducing technical debt, and ensuring code maintainability and performance. By utilizing metrics such as cyclomatic complexity, lines of code, and nesting depth, alongside automated code reviews and data analysis patterns, developers can assess both code and data quality. The adaptive documentation pattern ensures that as code evolves, corresponding documentation is updated automatically, minimizing the risk of outdated information. The inclusion of a data analysis pattern enables a structured approach to inspecting, cleaning, and exploring datasets, enhancing insights and supporting further analysis. This methodology streamlines software development while minimizing manual effort, though some challenges, such as subjectivity in review processes and potential over-documentation, remain. Future work will focus on refining these techniques for wider applicability and automation.**

***Index Terms*—Code Complexity Analysis, Code Review, Adaptive Documentation, Software Quality, Cyclomatic Complexity, Maintainability, Code Optimization, Automated Documentation.**

I. Introduction

In modern software development, maintaining high code quality and ensuring efficient development processes are key challenges, especially in fast-paced environments where frequent changes occur. As software systems grow in complexity, traditional methods of code review, documentation management, and complexity analysis often fall short, leading to issues such as outdated documentation, inconsistent code quality, and technical debt. Developers spend considerable time manually updating documentation, conducting reviews, and refactoring code to keep projects aligned with best practices.

To address these challenges, this paper presents a combined approach that leverages LLM-driven automation to improve code quality, maintainability, and performance. The approach integrates several key patterns, including adaptive documentation, structured code reviews, and code complexity analysis. The adaptive documentation pattern automates the process of keeping documentation synchronized with the codebase, reducing manual effort and ensuring consistency as the code evolves. Structured code reviews provide a standardized method for evaluating code quality using predefined criteria, helping to identify areas of improvement. Additionally, code complexity analysis offers a quantitative evaluation of code structure, highlighting opportunities for optimization in terms of readability, maintainability, and performance.

The structure of this paper is as follows: We begin by exploring single prompt patterns, starting with the adaptive documentation pattern, followed by patterns for code smell detection, code review, and code complexity analysis. Each pattern is discussed in detail, highlighting its motivation, structure, and example implementations. Next, we discuss how these patterns can be combined to create more comprehensive solutions that address multiple aspects of software quality. The paper concludes by considering the practical consequences of implementing these patterns, particularly in terms of scalability, maintainability, and automation efficiency in software development environments.

II. Single Prompt Patterns

A. Adaptive Documentation Pattern

The Adaptive Documentation pattern ensures that documentation evolves alongside the codebase, automatically keeping it accurate and up-to-date. It is critical in agile environments where manual updates to documentation are impractical.

1) Intent and Context: The Adaptive Documentation Pattern ensures that documentation evolves in sync with the codebase, automatically keeping it accurate and up-to-date. This pattern is especially useful in agile development environments, where frequent code changes make manual updates impractical. By automating the process, this pattern reduces the overhead of maintaining documentation while ensuring consistency, preventing it from becoming outdated or inaccurate over time.

2) Motivation: As software systems evolve, keeping documentation aligned with the code can be challenging, especially in fast-paced environments where frequent changes occur. The Adaptive Documentation Pattern automates this process, ensuring that the documentation stays in sync with the codebase. This reduces the manual effort required for maintaining documentation, allowing developers to focus on development tasks while ensuring that the documentation is always up-to-date.

Additionally, to ensure that documentation remains both useful and concise, the pattern incorporates a threshold system that dynamically adjusts the verbosity based on the context.

This system provides concise summaries by default but allows for more detailed documentation when needed, ensuring that only relevant information is included. The pattern further adapts to different environments by offering detailed technical documentation for development and simplified instructions for production, making the documentation flexible and tailored to specific use cases.

3) Structure and Key Ideas: Fundamental contextual statements:

Contextual Statements
When X (code changes are detected), do Y (automatically update the relevant documentation).
If X (new features are added or code is refactored), ensure Y (documentation is modified to reflect these changes).
For X (development environment), provide Y (detailed technical documentation), or for X (production environment), provide Y (simplified instructions).
Whenever X (documentation is updated), do Y (log changes and maintain version history).

TABLE I
CONTEXTUAL STATEMENTS FOR ADAPTIVE DOCUMENTATION PATTERN

To prevent over-documentation, the pattern implements a high-level threshold system that controls the verbosity of the generated documentation. This system ensures that:

Low verbosity is used for concise summaries, ideal for quick overviews or high-level documentation. Higher verbosity is triggered only when necessary, such as during significant code refactoring or for complex technical areas. This keeps the majority of the documentation streamlined while allowing deeper detail when needed. The threshold system helps ensure that documentation remains relevant, concise, and easy to navigate, balancing the need for detail with the importance of clarity.

4) Example Implementation:

> "Whenever updates or changes to the code are detected, automatically generate or modify the documentation. For development environments, provide detailed technical documentation, including function descriptions, parameters, and usage examples. For production environments, simplify the instructions to focus on deployment and usage, ensuring the documentation is concise and easy to follow for operational purposes.
>
> A high-level threshold system will control the verbosity of the documentation, defaulting to concise summaries that can expand into more detailed descriptions as necessary. This system ensures that only essential information is included by default, with additional details available when needed. Additionally, log all documentation changes and maintain a version history for traceability, allowing teams to track the evolution of the documentation and review past versions if required."

5) Consequences: The Adaptive Documentation Pattern automates the process of keeping documentation aligned with the code, ensuring consistency and reducing the likelihood of outdated or inaccurate information. This automation saves time for developers, particularly in large projects where frequent updates make manual documentation impractical. By maintaining a single source of truth, the pattern helps avoid confusion and ensures that all stakeholders have access to the latest and most accurate information.

The pattern also addresses the risk of over-documentation by incorporating a high-level threshold system that controls the verbosity of the documentation. By default, the system generates concise summaries, with the ability to expand into more detailed descriptions when necessary. This ensures that documentation remains clear and focused, providing the right amount of information for the context. Additionally, the pattern includes automatic logging of changes, maintaining a version history that allows teams to track how the documentation evolves over time.

B. Code Smell Detection Pattern

1) Intent and Context: A method to identify and quantify code smells within a codebase. The LLM detects problematic patterns or anti-patterns, assigns a quantitative score based on severity or frequency, lists the identified code smells with brief explanations, and suggests refactoring techniques to improve code quality.

2) Motivation: Code smells can reduce code maintainability and lead to bugs or other issues in production. By detecting and addressing these smells early, developers can ensure that the codebase remains clean and maintainable over time.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Request detection of code smells in the provided code.
Ask for a quantitative score based on the severity or number of code smells found.
Request a list of identified code smells with brief explanations.
Ask for suggestions to refactor the code to eliminate the code smells.

TABLE II
CONTEXTUAL STATEMENTS FOR CODE SMELL DETECTION PATTERN

Assign a quantitative score (e.g., out of 100), deducting points for each code smell based on its severity.

4) Example Implementation:

> "Analyze the provided code and identify potential code smells. Categorize each detected code smell by severity:
>
> - Critical: Major impact on functionality or maintainability.
> - Major: Significant issues that could lead to problems.
> - Minor: Minor issues that affect readability or performance.
>
> For each code smell:
>
> - Provide a brief explanation of the issue.
> - Suggest refactoring strategies.

Assign a score starting at 100 points, deducting:

- 20 points for each Critical code smell.
- 10 points for each Major code smell.
- 5 points for each Minor code smell.

Provide the final score after deductions."

5) Consequences: The Code Smell Detection Pattern offers several benefits but also comes with certain trade-offs. Regular detection and refactoring of code smells significantly improve code quality, ensuring that the codebase remains maintainable and scalable over time. By identifying critical and major issues early, this pattern helps prevent larger problems that could arise during production, thus reducing technical debt. Additionally, the scoring system provides a clear and objective measure of code quality, allowing teams to track improvements over time and prioritize areas that need attention.

However, there are some potential drawbacks. There is a risk that too much focus could be placed on minor issues, diverting attention away from critical or major problems that have a bigger impact on the codebase. Moreover, while the prompt provides clear severity levels for categorizing code smells, the interpretation of what constitutes a critical, major, or minor issue can be subjective and vary depending on the context or the LLM's judgment. Continuously refactoring the code to resolve identified code smells may also lead to overhead, especially in larger or more complex codebases, requiring additional time and effort from developers.

C. Code Review Scoring Pattern

1) Intent and Context: The Code Review Scoring Pattern provides a structured approach to evaluating code quality based on predefined criteria. This pattern ensures that code reviews are thorough and consistent, focusing on critical aspects such as correctness, efficiency, and readability. By applying a scoring system, the review process becomes more objective and measurable, leading to clearer feedback for developers.

2) Motivation: Code reviews play a crucial role in maintaining high standards of code quality and consistency across projects. This pattern introduces predefined criteria to ensure that all reviews are conducted with a focus on essential aspects of code performance and maintainability. Through this approach, developers receive a comprehensive analysis of their code, allowing for targeted improvements.

In addition to assessing specific code elements, the pattern integrates a system context section that allows reviewers to consider the broader architectural implications of the code. By providing high-level design information, developers can ensure that the LLM accounts for the system's overall structure, helping reviewers evaluate the code's role within the larger system. This addition enables the review to capture both detailed code analysis and broader system design considerations.

To further enhance flexibility, the pattern includes an additional concerns criterion, where reviewers can address unique project-specific issues that may not fit neatly into the predefined categories. This ensures that the review covers a wide scope, capturing any potential concerns beyond the standard criteria.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Request a code review for the provided code snippet.
Specify the criteria X, Y, Z for assessment.
Ask for a quantitative score out of N for each criterion.
Include a system context section to input high-level architectural information.
Incorporate an additional concerns criterion.
Ask for an overall score and a brief summary of the findings

TABLE III
CONTEXTUAL STATEMENTS FOR CODE SMELL DETECTION PATTERN

Replace X, Y, Z with appropriate code quality criteria (e.g., correctness, efficiency, readability).Replace N with the maximum score for each criterion.

4) Example Implementation:

"Perform a code review of the provided code snippet. Assess the code based on the following criteria:

- Correctness: Evaluate how accurately the function performs its intended task.
- Efficiency: Assess how optimal the code is in terms of time and space complexity.
- Readability: Determine how easy the code is to understand and maintain.
- System Context: Review high-level system architecture and consider how the code integrates into the overall system.
- Additional Concerns: Address any project-specific issues or unique concerns that may not fall under the predefined criteria.

Provide a quantitative score out of 10 for each criterion, followed by an overall score. Include a brief summary of the strengths and areas for improvement based on both the predefined criteria and any additional concerns raised during the review."

5) Consequences: The Code Review Scoring Pattern ensures consistent and objective evaluations by focusing on predefined criteria. By standardizing the review process, it reduces biases and helps teams evaluate code quality more effectively. The inclusion of a system context section ensures that broader architectural considerations are accounted for during the review, allowing the LLM to assess the code's role within the larger system. This provides a more holistic evaluation that goes beyond the code itself.

Incorporating an additional concerns criterion ensures that reviewers can address unique project-specific issues, allowing for flexibility in the review process. This enhances the comprehensiveness of the review, capturing any concerns that may not fit into the predefined criteria. Together, these elements create a thorough, flexible, and consistent code review process that ensures both technical detail and broader system considerations are covered.

D. Code Complexity Analysis Pattern

1) Intent and Context: This pattern prompts the LLM to analyze the complexity of a given code using specific metrics

such as cyclomatic complexity, nesting depth, and lines of code. The goal is to provide quantitative ratings for each metric and suggest improvements to simplify or optimize the code, thereby enhancing its maintainability and performance.

2) Motivation: Code complexity directly impacts maintainability, readability, and performance. High complexity can lead to issues with debugging, refactoring, and scaling. By analyzing the code's complexity with key metrics, this pattern helps users identify areas that need improvement to achieve cleaner and more efficient code.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Request an analysis of the code's complexity.
Specify the complexity metrics X, Y, Z, such as cyclomatic complexity, nesting depth, and lines of code.
Ask for a quantitative score or rating (0 to 10) for each metric.
Request recommendations to simplify or optimize the code.

TABLE IV
CONTEXTUAL STATEMENTS FOR CODE COMPLEXITY ANALYSIS PATTERN

Replace X, Y, Z with appropriate complexity metrics (e.g., cyclomatic complexity, nesting depth, lines of code).

4) Example Implementation: A sample wording of the *Code Complexity Analysis Pattern* might be:

> Analyze the complexity of the following code using these criteria:
>
> - **Cyclomatic Complexity**: Calculate the number of independent paths through the code. A lower score indicates simpler, more maintainable code.
> - **Nesting Depth**: Evaluate the depth of loops, conditionals, or other nested structures. Deeply nested code is harder to follow and maintain.
> - **Lines of Code (LOC)**: Count the total number of lines of code. Excessive LOC may indicate unnecessarily verbose code.
>
> Provide a quantitative score (0-10) for each metric and recommendations to simplify or optimize the code.

Consider the following code snippet:

```
my_array = [64, 34, 25, 5, 22, 11, 90, 12]

n = len(my_array)
for i in range(n-1):
    min_index = i
    for j in range(i+1, n):
        if my_array[j] < my_array[min_index]:
            min_index = j
    min_value = my_array.pop(min_index)
    my_array.insert(i, min_value)

print("Sorted array:", my_array)
```

5) Consequences: Applying the *Code Complexity Analysis Pattern* provides a clear, structured way to assess the complexity of a codebase using quantitative metrics. The recommendations generated by the LLM can help developers make informed decisions to reduce complexity, such as simplifying control flow, reducing nested structures, or refactoring duplicated code.

This pattern is useful for improving code quality in both small and large projects. However, its effectiveness depends on the quality of the metrics and the context in which they are applied. For example, some complex algorithms may require more in-depth domain knowledge to fully understand the trade-offs between complexity and performance.

E. Behavior Driven Development Prompt Pattern

1) Intent and Context: This pattern prompts the LLM to bridge the communication gap between developers, testers, and non-technical stakeholders by using natural language to describe the expected system behavior. Scenarios written in the "Given, When, Then" format serve as both documentation and executable tests, ensuring that all stakeholders share a clear understanding of the software's behavior.

2) Motivation: In many software projects, miscommunication between business stakeholders and developers can lead to costly misunderstandings and rework. By involving all stakeholders early in the process through behavior-driven scenarios, this pattern ensures alignment between business requirements and the development team's implementation. It encourages collaboration, reduces ambiguity, and helps clarify requirements before development begins.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Use the "Given, When, Then" format to describe system behavior:
Engage developers, testers, and stakeholders in writing scenarios collaboratively to ensure clarity and shared understanding.
Translate scenarios into automated tests, which can be integrated into the development cycle to validate behavior continuously.
Use test results to provide feedback, enabling iterative development and improvement based on business expectations.

TABLE V
CONTEXTUAL STATEMENTS FOR BEHAVIOR DRIVEN DEVELOPMENT PROMPT PATTERN

4) Example Implementation: A sample wording of the *Behavior Driven Development Prompt Pattern* might be:

> Use the 'Given, When, Then' format to describe the expected behavior of the login functionality for an application. Engage developers, testers, and stakeholders in writing these scenarios. Translate these scenarios into automated tests and ensure ongoing feedback to the development team through test results. Example:
>
> **Scenario: Successful Login**
>
> - **Given** the user is on the login page, and they have a valid username and password,
> - **When** they enter the valid credentials and press the login button,

- **Then** they are successfully logged into the dashboard.

This scenario captures the expected behavior in natural language, allowing stakeholders to easily review and understand it while enabling developers to translate it into executable tests.

5) Consequences: Using the *Behavior Driven Development Prompt Pattern* aligns business and technical teams, reducing the risk of misunderstandings and promoting shared ownership of the system's behavior. Scenarios serve as living documentation and executable tests, ensuring that the software meets business requirements at all stages of development. Automated tests derived from the scenarios provide continuous feedback, allowing for early detection of issues and fostering a collaborative, iterative development process.

F. Data Analysis Prompt Pattern

1) Intent and Context: This pattern prompts the LLM to guide users through the process of analyzing a dataset by providing structured steps for data inspection, cleaning, visualization, and exploratory analysis. The output serves as both a data cleaning report and a basis for deeper insights into trends, patterns, and relationships within the data.

2) Motivation: Data analysis is a critical step in understanding a dataset, identifying trends, and preparing it for further analysis or modeling. By following this structured approach, users ensure that the data is clean, insights are drawn effectively, and future analyses are based on sound data. This pattern reduces the risk of overlooking data quality issues and encourages a systematic approach to data exploration.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Data Overview: Start by inspecting the dataset, summarizing key features like data types, number of rows and columns, missing values, and basic descriptive statistics.
Data Cleaning: Handle missing values, identify and treat outliers, normalize or standardize data if necessary, and remove duplicates or inconsistencies.
Visualization: Use Python libraries (e.g., matplotlib or seaborn) to create visualizations such as histograms, bar charts, and correlation heatmaps. Each plot should provide detailed insights.
Exploratory Data Analysis (EDA): Perform a deep dive into the data to identify trends, correlations, and patterns. Highlight group-wise behaviors and any remaining outliers.
Suggestions for Further Analysis: Based on the findings, propose additional analyses like hypothesis testing, predictive modeling, or clustering, as well as recommendations for handling any remaining data issues.

TABLE VI
CONTEXTUAL STATEMENTS FOR DATA ANALYSIS PROMPT PATTERN

4) Example Implementation: A sample wording of the *Data Analysis Prompt Pattern* might be:

> Perform an analysis of the following dataset by completing these steps: 1. Provide a data overview, including the number of rows and columns, missing values, and descriptive statistics. 2. Clean the data by handling missing values, identifying outliers, and normalizing the numeric variables. 3. Visualize the data using histograms, bar charts, and correlation heatmaps. 4. Conduct exploratory data analysis to identify trends and relationships. 5. Suggest further analyses, such as predictive modeling or hypothesis testing, based on the EDA findings.

5) Consequences: Using the *Data Analysis Prompt Pattern* ensures a comprehensive and systematic approach to analyzing a dataset. The data cleaning report provides transparency about how missing values, outliers, and inconsistencies were addressed. Visualizations reveal important insights into the dataset's structure, while exploratory analysis highlights trends, correlations, and patterns. The suggestions for further analysis guide users toward additional insights and more advanced techniques.

III. COMBINED PROMPT PATTERNS

1) Intent and Context: This combined pattern prompts the LLM to analyze the complexity of code using specific metrics such as cyclomatic complexity, nesting depth, and lines of code, while also detecting code smells that may affect code maintainability or performance. The pattern provides both quantitative ratings for complexity and severity-based classification of code smells, along with recommendations for simplifying the code and resolving identified code smells.

2) Motivation: Maintaining clean, understandable, and efficient code requires analyzing both code complexity and the presence of code smells. High complexity makes code harder to maintain and understand, while code smells are often indicative of deeper issues that can degrade performance and lead to bugs. Combining these two assessments provides a more comprehensive evaluation of code quality, allowing developers to optimize both structure and functionality.

3) Structure and Key Ideas: **Fundamental Contextual Statements:**

- Request an analysis of the code's complexity using specified metrics.
- Specify the complexity metrics X, Y, Z to assess (e.g., cyclomatic complexity, nesting depth, lines of code).
- Ask for a quantitative score for each complexity metric.
- Request detection of code smells with severity ratings (Critical, Major, Minor).
- Ask for suggestions to refactor or optimize the code for both complexity reduction and code smell resolution.
- Request an overall score or rating for code quality, factoring in both complexity and code smells.

4) Usage:

- Replace X, Y, Z with desired complexity metrics such as cyclomatic complexity, lines of code, and nesting depth.
- Customize severity levels (Critical, Major, Minor) based on the potential impact of the code smells detected.

A. Code Complexity Analysis with Code Smell Detection Pattern

1) Intent and Context: This pattern is used to identify code complexity and potential code smells within a codebase. The

LLM evaluates code complexity using specified metrics such as Cyclomatic Complexity, Nesting Depth, and Lines of Code (LOC). In addition, the LLM detects problematic code smells, assigns severity levels (Critical, Major, Minor), and provides refactoring suggestions.

2) Motivation: Complex code with numerous code smells can lead to lower maintainability, readability, and scalability. By evaluating both complexity and code smells, the LLM ensures better code quality, resulting in easier debugging and smoother future developments.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Request an analysis of the code's complexity using specified metrics.
Specify the complexity metrics X, Y, Z to assess (e.g., cyclomatic complexity, nesting depth, lines of code).
Ask for a quantitative score for each complexity metric.
Request detection of code smells with severity ratings (Critical, Major, Minor).
Ask for suggestions to refactor or optimize the code for both complexity reduction and code smell resolution.
Request an overall score or rating for code quality, factoring in both complexity and code smells.

TABLE VII
CONTEXTUAL STATEMENTS FOR CODE COMPLEXITY ANALYSIS WITH CODE SMELL DETECTION PATTERN

The LLM assigns a quantitative score for each complexity metric (e.g., out of 10) and a final score for code quality after factoring in code smells. Refactoring suggestions are based on the detected code smells and complexity evaluation.

4) Example Implementation:

> "Please review the following code and perform a complexity analysis. Use the following metrics to assess complexity:
>
> - Cyclomatic Complexity
> - Nesting Depth
> - Lines of Code (LOC)
>
> Assign a complexity score for each metric and provide an overall complexity rating.
> Additionally, identify any code smells and assign severity levels (Critical, Major, Minor). Provide a quantitative score starting at 100, with deductions based on the severity of the code smells.
>
> - Critical: -20 points
> - Major: -10 points
> - Minor: -5 points
>
> After the complexity analysis and code smell detection, give an overall score and suggest refactoring options."

5) Consequences: By combining code complexity analysis with code smell detection, developers are provided with a more complete picture of their code's overall quality. This pattern helps ensure that both the structural aspects of the code and any underlying smells are addressed simultaneously, improving maintainability and readability. The actionable refactoring suggestions provided by the LLM make it easier for developers to tackle both types of issues in one go.

However, as with any automated review system, there is potential subjectivity in how code smells are detected and categorized by the LLM. While the complexity metrics are objective, the interpretation of code smells and their severity may vary. Developers will need to apply their own judgment in determining which suggestions are worth pursuing. Additionally, balancing complexity reduction with the elimination of code smells may require careful prioritization, particularly in larger codebases where refactoring could introduce new issues or disrupt the system.

B. Code Complexity with Code Review with Adaptive Documentation Pattern

1) Intent and Context:: This combined pattern enables the LLM to analyze code complexity, perform a detailed code review, and then automatically update the documentation to reflect changes or findings. This ensures the codebase remains optimized, maintainable, and well-documented in an adaptive manner.

2) Motivation:: Combining these three patterns provides a comprehensive approach to maintaining code quality. The complexity analysis highlights areas where the code structure can be improved, while the code review ensures that functionality, efficiency, and readability are properly evaluated. The adaptive documentation ensures that as the code evolves, the corresponding documentation is automatically updated, minimizing manual effort and reducing the risk of outdated or inaccurate documentation.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Request a complexity analysis using metrics such as cyclomatic complexity, lines of code, and nesting depth.
Request a detailed code review, focusing on correctness, efficiency, and readability.
Ask for a quantitative score for each review criterion and complexity metric.
Generate or update the documentation automatically based on the findings of both the complexity analysis and the code review.
Ensure the documentation reflects both technical aspects (code behavior, complexity) and simplified end-user explanations (where applicable).

TABLE VIII
CONTEXTUAL STATEMENTS FOR CODE COMPLEXITY ANALYSIS WITH CODE SMELL DETECTION PATTERN WITH ADAPTIVE DOCUMENTATION PATTERN

Replace the complexity metrics (e.g., cyclomatic complexity, lines of code) and review criteria (e.g., correctness, efficiency) as needed. Customize the documentation sections that need automatic updates based on the findings.

4) Example Implementation:

> "Analyze the following code using the following steps:
> 1. Perform a complexity analysis based on the following metrics:

- Cyclomatic complexity: Count the independent paths in the code.
- Lines of code: Count the length of each function.
- Nesting depth: Assess the depth of conditional statements.

2. Conduct a code review, focusing on the following criteria:

- Correctness: Does the code perform its intended task accurately?
- Efficiency: How optimized is the code in terms of time and space complexity?
- Readability: How easy is the code to understand and maintain?

3. Generate or update the relevant documentation based on the results of both the complexity analysis and the code review. Ensure the documentation is tailored to both developers and end-users, including:

- Function descriptions.
- Complexity analysis results (e.g., cyclomatic complexity).
- Review-based insights (e.g., suggestions for improvement)."

5) Consequences: This combined approach ensures that the code is thoroughly analyzed, reviewed, and documented. Developers benefit from a quantitative complexity analysis, a structured review of key quality aspects, and the automatic generation or updating of documentation. This not only saves time but also reduces the risk of outdated documentation, which in turn enhances code maintainability and overall quality.

However, there are some trade-offs to consider. Performing complexity analysis, code review, and documentation generation simultaneously may introduce overhead in larger projects, making the process more time and resource intensive. Additionally, automatically updating the documentation based on small changes could lead to over-documentation if not properly managed, resulting in verbose or unnecessary content. Lastly, while the complexity analysis is objective, there is a degree of subjectivity in the review and documentation updates, as these depend on the LLM's interpretation of the code and its context.

C. Code Smell Detection with Behavior Driven Development Pattern

1) Intent and Context: This pattern combines code smell detection with behavior-driven development (BDD) to ensure that code is not only functionally correct but also free of problematic patterns that could reduce its maintainability and scalability. The LLM analyzes the code for both its behavior through BDD scenarios and for any code smells. The detected code smells are categorized by severity, and refactoring suggestions are provided. Additionally, BDD scenarios, written in the "Given, When, Then" format, ensure that all stakeholders understand the system's behavior.

2) Motivation: Ensuring that the code meets business requirements and is free from code smells is critical for delivering high-quality software. Combining BDD and code smell detection enables developers to verify system behavior while maintaining clean, maintainable code. This pattern helps bridge the gap between functional requirements and code quality, ensuring alignment between business objectives and code quality standards.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Use the "Given, When, Then" format to describe expected system behavior.
Engage developers, testers, and stakeholders in writing behavior-driven scenarios collaboratively.
Request detection of code smells in the provided code.
Ask for a quantitative score based on the severity or number of code smells found.
Request suggestions to refactor the code and improve maintainability based on code smell detection.
Translate scenarios into automated tests to validate both code behavior and quality.

TABLE IX
CONTEXTUAL STATEMENTS FOR CODE SMELL DETECTION WITH BEHAVIOR DRIVEN DEVELOPMENT PATTERN

4) Example Implementation:

"Please review the following code using the Behavior Driven Development pattern and identify potential code smells.

Behavior Scenario: Use the "Given, When, Then" format to describe the behavior of the login functionality:

- **Given** the user is on the login page with valid credentials,
- **When** they enter the credentials and press the login button,
- **Then** they should be successfully logged into the dashboard.

Code Smell Detection: Analyze the code for any potential code smells and categorize them by severity:

- Critical: Major impact on functionality or maintainability.
- Major: Significant issues that could lead to problems.
- Minor: Minor issues that affect readability or performance.

For each code smell, provide a brief explanation and suggest refactoring techniques. Assign a score starting at 100, deducting:

- 20 points for each Critical code smell.
- 10 points for each Major code smell.
- 5 points for each Minor code smell.

Provide an overall score based on both the behavior-driven test results and the code smell analysis."

5) Consequences: Combining Behavior Driven Development with Code Smell Detection ensures that code not only behaves as expected according to business requirements but also remains clean and maintainable over time. The BDD scenarios help clarify system behavior for all stakeholders, while the code smell detection process identifies and eliminates problematic code patterns that could impact scalability or performance. This combination of techniques helps developers create high-quality software that is both functionally correct and technically sound.

However, balancing the behavior-driven scenarios with the elimination of code smells can introduce challenges, especially in complex systems where the requirements for behavior and code quality must be managed simultaneously. Developers will need to apply their own judgment when prioritizing refactoring tasks and ensuring that behavior-driven tests remain accurate as the code evolves.

D. Data Analysis with Code Complexity Analysis Pattern

1) Intent and Context: This pattern combines data analysis with code complexity analysis to ensure that both the dataset and code are systematically evaluated. The LLM guides users through data inspection, cleaning, visualization, and exploratory analysis while simultaneously assessing the complexity of the code using metrics such as cyclomatic complexity, nesting depth, and lines of code (LOC). The result is a comprehensive evaluation of both the data and the code quality, with suggestions for optimization and improvement.

2) Motivation: Ensuring that both data and code are of high quality is essential in software development and data-driven projects. Combining data analysis with code complexity analysis ensures that datasets are properly explored and cleaned, while code complexity is assessed and optimized. This holistic approach leads to better performance, maintainability, and clarity in both the data and the underlying codebase.

3) Structure and Key Ideas: Fundamental Contextual Statements:

Contextual Statements
Data Overview: Start by inspecting the dataset, summarizing key features like data types, number of rows and columns, missing values, and basic descriptive statistics.
Data Cleaning: Handle missing values, identify and treat outliers, normalize or standardize data if necessary, and remove duplicates or inconsistencies.
Visualization: Use Python libraries (e.g., matplotlib or seaborn) to create visualizations such as histograms, bar charts, and correlation heatmaps.
Exploratory Data Analysis (EDA): Perform a deep dive into the data to identify trends, correlations, and patterns.
Code Complexity Analysis: Assess the complexity of the code using metrics such as cyclomatic complexity, nesting depth, and lines of code (LOC).
Recommendations for Improvement: Based on both the data analysis and the code complexity analysis, suggest additional analyses and refactoring strategies to optimize both the dataset and the code.

TABLE X
CONTEXTUAL STATEMENTS FOR DATA ANALYSIS WITH CODE COMPLEXITY ANALYSIS PATTERN

4) Example Implementation:

"Please perform a detailed analysis of the following dataset and code:

Data Analysis:

- Inspect the dataset, summarize key features like data types, the number of rows and columns, missing values, and basic descriptive statistics.
- Clean the data by handling missing values, identifying outliers, and normalizing numeric variables if necessary.
- Visualize the data using histograms, bar charts, and correlation heatmaps to uncover insights and trends.
- Perform exploratory data analysis (EDA) to identify trends, correlations, and group-wise behaviors within the data.

Code Complexity Analysis:

- Analyze the complexity of the code using the following metrics:
  - Cyclomatic Complexity: Calculate the number of independent paths through the code. Lower complexity scores indicate simpler, more maintainable code.
  - Nesting Depth: Evaluate the depth of loops, conditionals, or other nested structures. Deeply nested code is harder to follow and maintain.
  - Lines of Code (LOC): Count the total number of lines of code. Excessive LOC may indicate unnecessarily verbose code.
- Provide a quantitative score (0-10) for each metric and recommendations to simplify or optimize the code.

Provide recommendations for further analysis and refactoring strategies to improve both the dataset and the codebase."

5) Consequences: By combining data analysis with code complexity analysis, developers and data scientists gain a holistic view of both the dataset and the codebase, ensuring that both are optimized for performance and maintainability. Data cleaning, visualization, and EDA provide insights into the dataset's structure, while code complexity analysis ensures that the underlying code is efficient and maintainable.

This pattern enables teams to identify issues in the data and code simultaneously, promoting both better data quality and cleaner code. However, balancing data analysis tasks with code complexity improvements may require additional effort, particularly in complex projects where data and code quality are equally critical to success. Developers and data analysts may need to prioritize tasks based on the severity of issues uncovered in both domains.

IV. CONCLUSION

In this paper, we have introduced a combined approach that leverages code complexity analysis, code review, and adaptive

documentation to enhance software quality. By automating the documentation process and integrating quantitative and qualitative assessments of code complexity and functionality, developers can reduce the manual effort required for maintaining high code standards. This approach addresses key challenges in modern software development, such as maintaining consistent documentation and optimizing code for readability, maintainability, and performance.

While the integration of these techniques has clear benefits, such as minimizing technical debt and improving collaboration, it also presents certain challenges, including the potential for over-documentation and subjective interpretation of review findings. Despite these limitations, the proposed approach provides a practical and scalable solution for maintaining code quality in dynamic and fast-paced development environments.

Future work will focus on refining these techniques to further automate the review and documentation processes, ensuring that they remain adaptable and applicable to a wide range of projects and codebases.

V. Limitations

Despite the advantages offered by the proposed patterns, several limitations should be considered in real-world applications:

- **LLM Understanding of Complex Codebases**: Large Language Models (LLMs) often encounter challenges in understanding deeply complex or domain-specific codebases, especially those with intricate business logic, legacy systems, or custom libraries. While LLMs can analyze generic code effectively, domain-specific nuances or specialized algorithms may be missed, leading to suboptimal suggestions. In these cases, developers may need to intervene manually to ensure that recommendations align with the code's specific requirements and context. Furthermore, LLMs might struggle to account for project-specific coding conventions or architectural patterns, necessitating extra oversight to verify that suggested changes are appropriate.
- **Balancing Automation with Developer Oversight**: Although automating code reviews, documentation updates, and code quality assessments saves time and reduces manual effort, there is a risk of over-reliance on these automated systems. Fully automated systems might miss critical edge cases, fail to consider broader architectural concerns, or overlook certain bugs that human reviewers might catch. For example, code reviews conducted purely by LLMs may focus on syntax or logic issues but miss out on important design patterns or project-specific best practices. Therefore, while automation helps accelerate the development process, manual verification and developer oversight are still essential to ensure the accuracy and relevance of the suggested changes, particularly for complex projects.
- **Subjectivity in Code Quality Assessment**: Even though the patterns provide quantitative metrics such as cyclomatic complexity, lines of code, or test coverage, some aspects of code quality remain inherently subjective. Readability, maintainability, and even performance optimizations can vary depending on the project context, developer preferences, or the specific use case. An LLM's interpretation of "clean" or "readable" code might differ from that of the development team, which could result in suggestions that conflict with team standards or create unnecessary churn. Additionally, different projects may prioritize different quality metrics, meaning that a pattern emphasizing one metric (e.g., reducing lines of code) might inadvertently harm another (e.g., clarity or functionality).
- **Context Limitations and Project-Specific Constraints**: LLMs currently lack the ability to fully understand the broader context in which the code is being written, especially when dealing with complex dependencies, environment-specific configurations, or tightly integrated systems. For instance, LLMs may misinterpret or ignore crucial context such as custom-built libraries, external APIs, or microservice architectures that dictate how the code should behave. Without this deep contextual understanding, some automated suggestions—especially related to code refactoring or restructuring—may introduce new issues, such as performance bottlenecks or bugs. In distributed systems, where nuances of parallelism, synchronization, or consistency must be preserved, LLM-generated suggestions may inadvertently break system functionality or introduce deadlocks.
- **Over-Documentation and Potential for Redundancies**: While the adaptive documentation pattern ensures that documentation stays up-to-date, there is a risk of over-documentation or creating redundant information, particularly in projects that evolve quickly. Automated documentation may generate verbose or excessively detailed explanations for minor code changes, which could clutter the documentation and reduce its usefulness. Additionally, without careful oversight, updates triggered by small code modifications may overwhelm users with unnecessary changes in the documentation, making it harder for developers to find important information amidst the noise.
- **Impact on Performance and Scalability**: While the proposed patterns aim to improve maintainability and code quality, the process of continuous complexity analysis, automated code review, and frequent documentation updates could introduce performance overhead in large-scale projects. In larger teams or projects with rapid iteration cycles, constant code assessments and documentation modifications might slow down the workflow or require additional computational resources. This is especially true in environments where large codebases are regularly updated by multiple developers, potentially leading to delays in development pipelines or integration processes.

These limitations highlight the need for a balanced approach when implementing LLM-driven automation in software de-

velopment. While LLMs can significantly enhance productivity and code quality, manual oversight, context-awareness, and developer intervention remain essential to ensure that the suggestions are not only technically correct but also aligned with the broader project goals and constraints. Future research and improvements in LLM capabilities may help address some of these limitations, particularly in areas like context comprehension and minimizing subjectivity in code quality assessments.

REFERENCES

[1] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D. C. Schmidt, "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT," arXiv preprint arXiv:2302.11382, 2023.

[2] J. White, S. Hays, Q. Fu, J. Spencer-Smith, D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design," arXiv preprint arXiv:2303.07839, 2023.