International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 8 Issue 5, Sep-Oct 2024 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

An Autonomous Multi-Agent LLM Framework for Agile Software Development

Manish Sanwal, Ishan Deva

Engineering Department, News Corporation, New York, USA

ABSTRACT

This paper presents a novel approach to simulating the Agile software development process using a multi-agent system based on large language models (LLMs). The system autonomously handles a complete software development pipeline, from user story creation and task planning to code generation, review, and pull request creation. Through collaboration among specialized agents, each performing a distinct role in the Agile lifecycle, the system demonstrates the ability to autonomously complete low to medium-complexity software tasks with minimal human intervention. We evaluate the system's performance across various tasks and provide insights into its limitations and future potential in augmenting real-world software development teams.

KEYWORDS: *Agile software development, automation, code generation, large language models, multi-agent systems*

How to cite this paper: Manish Sanwal | Ishan Deva "An Autonomous Multi-Agent LLM Framework for Agile Software Development" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-8 | Issue-5, October 2024, pp.892-898, URL: www.ijtsrd.com/papers/ijtsrd70467.pdf

IJTSRD70467

I. INTRODUCTION

The rapid advancement of large language models (LLMs) has paved the way for significant breakthroughs in natural language processing, reasoning, and code generation [1, 2]. Initially designed to assist in natural language understanding tasks, recent developments have enabled LLMs to be applied to more complex and structured workflows, including software development [3].

Agile software development is an iterative and flexible methodology that emphasizes adaptive planning, evolutionary development, and continuous improvement. Automating portions of the Agile process could significantly enhance productivity, particularly in handling low to medium-complexity tasks where repetitive actions, such as writing boilerplate code or conducting reviews, can be streamlined [4].

However, most current tools, such as GitHub Copilot, provide code suggestions rather than handling a complete development cycle. In contrast, our system simulates an entire Agile team using multi-agent LLMs, autonomously creating user stories, planning tasks, generating code, reviewing it, and submitting pull requests. This paper focuses on the design, implementation, and evaluation of this system, demonstrating its ability to reduce human intervention in software development.

1.1. Motivation and Challenges

The motivation behind automating Agile workflows stems from the increasing demand for rapid software delivery, driven by the growing complexity of modern software projects and the competitive pressures faced by development teams to deliver features faster and with higher quality [5]. Traditional software development approaches often encounter bottlenecks in scalability, resource management, and consistency, particularly as projects become more intricate.

Automating these workflows presents significant challenges due to the nuanced, context-dependent nature of many Agile tasks [6]. Translating natural language user stories into actionable tasks demands a deep understanding of both the project's technical requirements and its broader business context. Similarly, tasks such as code review require evaluating generated code against existing architectural patterns, style guides, and performance considerations, which cannot be easily reduced to rule-based automation [7].

To address these challenges, we leverage a multi-agent system where each agent specializes in a different phase of the software development process—such as user story creation, task planning, code generation, review, and iteration [9, 10]. This specialization allows the system to handle complex, context-driven decisions more effectively than any single tool or model.

II. RELATED WORK

2.1. LLM-Based Autonomous Agents

The use of LLMs in autonomous agents has been widely explored across various domains, ranging from virtual assistants that handle natural language interactions to sophisticated problem-solving frameworks used in areas such as robotics, logistics, and decision-making systems [8]. Virtual assistants like Alexa and Google Assistant have demonstrated the ability of LLMs to interpret human commands and provide relevant responses.

More complex systems such as AutoGPT [9] and MetaGPT [10] have showcased the potential of leveraging multiple LLMs working collaboratively. These systems distribute tasks among specialized agents, enabling them to tackle multi-step processes that require coordination, logical reasoning, and contextual understanding.

Techniques like Chain-of-Thought prompting [11] and multiple thought chains [6] have been proposed to improve the reasoning capabilities of LLMs, which are essential in complex decision-making tasks. Additionally, methods such as LLM-Debate [12] aim to enhance the reasoning of LLMs through debate mechanisms, potentially leading to more accurate and robust outcomes.

2.2. Software Automation Tools

Advanced automation tools in software development, such as GitHub Copilot, SourceGraph Cody, and TabNine, have transformed how developers interact with code by offering real-time suggestions, autocompletion, and context-aware snippets [3]. These tools leverage cutting-edge large language models to assist in code generation, significantly reducing the time spent on writing repetitive code.

However, despite their usefulness, these tools are inherently limited in scope. While they excel at generating small fragments of code, such as functions or boilerplate templates, they cannot address the broader aspects of the software development lifecycle [7]. They do not provide capabilities for higher-level tasks such as user story creation, task planning, detailed code review, or the integration of generated code into the larger architecture of a project.

III. SYSTEM ARCHITECTURE

Our system is composed of multiple specialized agents, each responsible for a distinct phase in the software development lifecycle. This multi-agent architecture mirrors the roles and responsibilities found in a typical Agile team, where tasks such as user story creation, task planning, code generation, and code review are distributed among different individuals. By assigning these roles to autonomous agents, we create a robust, scalable system that can handle the complexities of software development with minimal human intervention [4].

Each agent is designed to function independently while interacting with other agents in a structured manner to ensure smooth transitions between development stages. The modular design enhances efficiency and allows for independent improvement and updates of individual agents without disrupting the overall workflow. Figure 1 provides an overview of this agent-based system architecture.

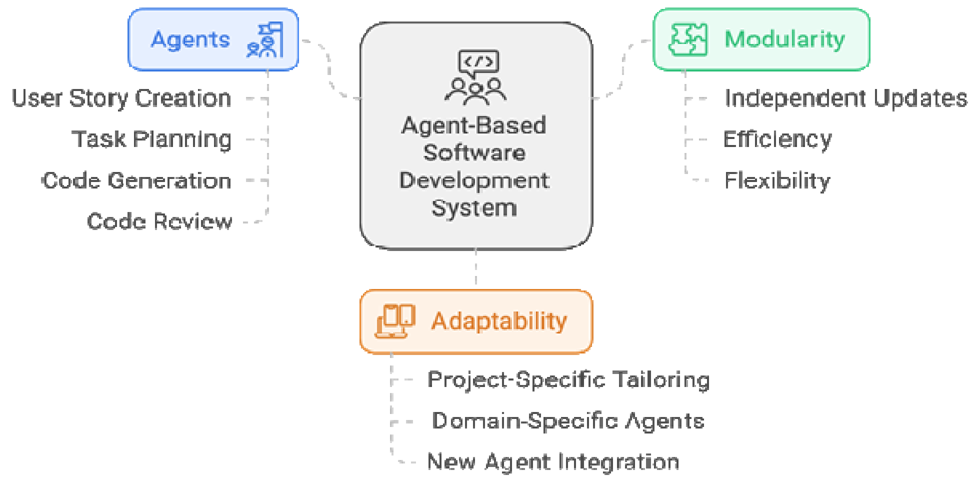

Figure 1: System architecture for the multi-agent LLM system automating Agile software development tasks.

3.1. Workflow Overview

The workflow begins with a high-level task definition provided by a project manager or developer. This task is parsed into a user story by the *User Story Creation Agent*. Subsequent agents, such as the *Planning Agent*, *Code Generation Agent*, and *Code Review Agent*, collaborate to complete the task. The process culminates in the submission of a pull request for human review.

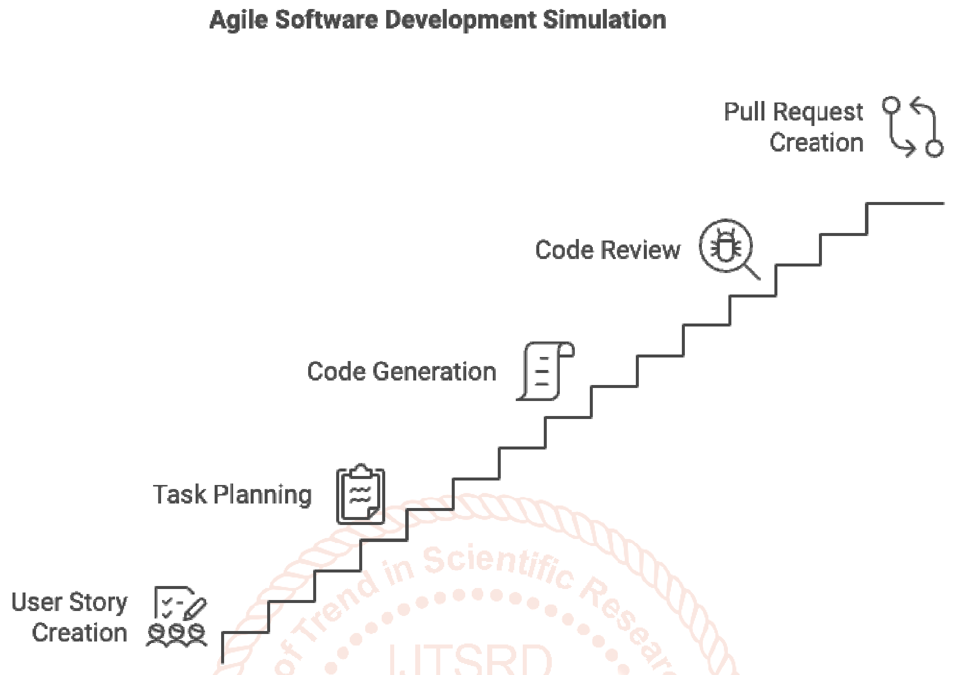

Figure 2: Development workflow of the multi-agent system.

Each agent operates independently but passes context to the next agent in the pipeline. For example, the code generated by the Code Generation Agent is reviewed by the Code Review Agent, which checks it against project standards and suggests improvements.

IV. AGENT DESCRIPTIONS AND IMPLEMENTATION

The detailed prompt designs for each agent are provided in Appendix C.

4.1. User Story Creation Agent

The User Story Creation Agent is designed to translate high-level task definitions into structured, actionable user stories that can be directly consumed by other agents in the development pipeline. Leveraging expertise in the specified domain and programming language, the agent analyzes the issue or task description provided by the user, along with any additional instructions or guidelines.

The agent follows a rigorous process to ensure the creation of user stories that are comprehensive and aligned with the project's goals. The system prompt emphasizes multiple key instructions for each task:

Analysis: Carefully examine the task or issue, breaking it down into smaller components and providing a step-by-step explanation for the solution.

4.1.1. **Clarity and Precision**: Produces precise user stories, avoids unnecessary complexity, and is directly aligned with the given instructions and requirements.

4.1.2. **Acceptance Criteria**: Provides clear acceptance criteria, allowing the development team to understand when a task can be considered complete.

The final output is a JSON response that encapsulates all the elements of a well-structured user story, including detailed acceptance criteria, tasks, and relevant context. The agent follows strict adherence to formatting, as per the defined prompt template, ensuring the response is structured for easy interpretation by other agents or developers.

4.2. Planning Agent

The Planning Agent translates the user story into a detailed, actionable implementation plan that can be directly utilized by the development team. This agent breaks down the user story into precise steps, analyzes the existing codebase, and ensures that all required tasks are clearly defined.

Upon receiving the user story and relevant instructions, the Planning Agent reviews the provided task description, existing codebase, and any additional guidelines. The prompt ensures that the agent takes a methodical approach, focusing on key areas such as:

- **Task Breakdown**: Divides the user story into smaller, manageable tasks.
- **Codebase Analysis**: Examines the existing codebase to understand the current system state, identifying relevant files or modules that need modification or extension.
- **Actionable Steps**: Provides detailed and actionable steps, specifying file paths, the status of each file, and the exact tasks to be completed.
- **Pseudocode and Logic**: Uses pseudocode to illustrate intended logic or structure, aiding developers in understanding the high-level flow and implementation details.

The plan is organized into a strict JSON format, making it easy to pass along to other agents or team members for further processing. Additionally, the plan contains a *Thought Log* section, where the agent can document its thought process or any considerations made while developing the plan.

4.3. Code Generation Agent

The Code Generation Agent translates the planned tasks into fully functional code. Leveraging a structured prompt, it generates well-organized, high-quality code that integrates seamlessly with the existing codebase. The agent provides the following for each file involved:

- **Full File Path**: The exact file location within the project.
- **File Status**: Indicates whether the file is new, being modified, or deleted.
- **File Content**: The complete content of the file, including necessary imports, function definitions, and exports.

Thought Log: Documents the agent's thought process during code generation.

4.4. Code Review Agent

The Code Review Agent ensures that the generated code meets the highest standards of quality, efficiency, and adherence to best practices. It meticulously reviews the generated code by analyzing the new code in the context of the existing codebase, the user story, and the detailed plan. The agent follows a detailed, step-by-step review process, including:

- **Thorough Analysis**: Review the user story, task description, plan, and codebase to understand the context and objectives.
- **Code Quality**: Assesses the code for readability, clarity, and adherence to coding standards.
- **Best Practices**: Check if the code follows best practices for the specific programming language.
- **Functionality**: Verifies that the code fully implements the functionality specified.
- **Integration**: Ensures that the new code integrates seamlessly with the existing codebase.

4.5. Pull Request Creation Agent

The Pull Request Creation Agent generates a well-structured and comprehensive pull request once all code issues have been resolved. It reviews the code changes, user story, and implementation plan to ensure the PR clearly communicates the purpose of the changes, summarizes the modifications, and provides necessary context for reviewers. The PR includes all necessary files, documentation, and testing results, allowing a human reviewer to perform the final review before merging the changes into the repository.

4.6. Implementation Details

All agents were implemented using Anthropic Claude Instant 3.5 running on AWS Bedrock. We also experimented with Google's Gemini 1.5 but did not achieve satisfactory results. We plan to evaluate OpenAI's latest models to assess their potential for enhancing the code generation process.

V. RESULTS

The performance of the system was evaluated across several real-world examples, each varying in complexity. The tasks ranged from simple maintenance operations to medium-complexity development tasks, allowing us to observe how well the system handles different levels of complexity with minimal human intervention. Below, we present a few case studies that highlight the system's effectiveness in handling different types of software tasks.

5.1. Case Study: Maintenance Task—Removing an Obsolete Integration

In this case study, the system was assigned a simple maintenance task involving the removal of an obsolete integration from the project repository. The task required the system to update configuration files by removing specific entries and adjusting the changelog to reflect the changes.

The *User Story Creation Agent* generated a user story outlining the need to remove the obsolete integration, including acceptance criteria such as ensuring no residual code remains and that the application functions correctly without it. The *Planning Agent* created a detailed plan specifying which files needed modification or deletion, including configuration files and documentation. The *Code Generation Agent* executed the plan, removing code references and updating relevant files. The *Code Review Agent* reviewed the changes, confirming that all references to the integration were removed and that the application maintained functionality. Finally, the *Pull Request*

Creation Agent compiled the changes into a pull request with a clear summary and detailed description of the modifications.

The pull request was submitted and, after a brief human review, merged into the main branch without any additional comments or required changes. This demonstrates the system's ability to handle straightforward maintenance tasks efficiently and accurately, autonomously completing all steps from user story creation to pull request submission.

5.2. Case Study: API Endpoint Creation

In another case study, the system was tasked with creating a new API endpoint for a project. This task involved making changes across multiple files, including routes, controllers, and con- figuration settings. Additionally, the system was required to adhere to strict coding guidelines and rules specified in the additional instructions provided.

The *User Story Creation Agent* crafted a user story detailing the requirements of the new endpoint, including the expected inputs, outputs, and error handling. The *Planning Agent* analyzed the existing codebase to identify where new code should be integrated, specifying file paths and outlining the necessary changes. The *Code Generation Agent* wrote the code for the new endpoint, ensuring it adhered to coding standards, included necessary validation, and integrated with existing middleware. The *Code Review Agent* conducted a thorough review, checking for compliance with best practices, potential security vulnerabilities, and consistency with existing code patterns. The *Pull Request Creation Agent* assembled the changes into a pull request, providing a comprehensive description and referencing the original user story.

The task was completed with minimal user interaction, requiring only a final review by a human developer. Minor suggestions were made regarding optimization and additional comments for clarity. After addressing these comments, the code was merged. This case illustrates the system's capability to handle more complex tasks that require integration across multiple components of the application while maintaining adherence to coding standards and project guidelines.

5.3. Case Study: Bug Fix—Thread Concurrency Issue

In this case study, the system was assigned a bug fix related to a threading concurrency issue that was freezing the user interface under certain conditions. The system was required to analyze the codebase, identify the library and version used for UI design, and provide a solution to address the concurrency issue.

The *User Story Creation Agent* outlined the problem based on the bug report, specifying acceptance criteria such as eliminating the UI freeze and ensuring thread safety. The *Planning Agent* examined the codebase to identify the root cause, pinpointing the functions where threading was mishandled. It was also able to correctly identify the UI library in use as well as its version. The *Code Generation Agent* refactored the problematic code sections, implementing proper thread synchronization mechanisms and updating relevant functions.

The *Code Review Agent* evaluated the changes for potential deadlocks, race conditions, and adherence to concurrency best practices. The *Pull Request Creation Agent* created a pull request with detailed explanations of the changes and how they resolve the issue.

The generated pull request was submitted with detailed explanations of the changes, and the solution was approved without any further comments. The human reviewer confirmed that the fix resolved the issue without introducing new problems. This demonstrates the system's ability to diagnose and fix nontrivial bugs that require knowledge of both the codebase and external libraries, handling tasks that involve complex debugging, and an understanding of concurrent programming concepts.

5.4. Case Study: Dependency Management and Documentation

In this final case study, the system was tasked with migrating the dependency management system of a project. For a Python project, the system was required to move dependencies from requirements.txt to the Poetry based system.

The *User Story Creation Agent* produced a user story emphasizing the need to update the dependency management to the newer system and enhance the README with setup instructions. The *Planning Agent* listed all outdated dependencies, planned the migration process, and outlined sections to be added to the documentation. The *Code Generation Agent* updated dependency files (from requirements.txt to pyproject.toml), resolved compatibility issues, generated the .lock files, and wrote comprehensive setup instructions. The *Code Review Agent* verified that updates did not break existing functionality and that the new documentation was clear and accurate. The *Pull Request Creation Agent* compiled the updates into a pull request, highlighting key changes and including notes on testing performed.

The system successfully analyzed the project's structure and dependencies, migrated the dependency management system, and generated a polished README file that included detailed instructions for setting up the project. The human developer was able to follow the updated README file and verify the results, completing the task. This case study illustrates the system's capability to handle tasks that require not only code generation but also broader project management tasks such as documentation and dependency handling, demonstrating versatility in tasks that span both code and documentation updates.

VI. DISCUSSION

6.1. Challenges in High-Complexity Tasks

Although the system performs well on low to medium-complexity tasks, it faces limitations when handling high-complexity tasks. Such tasks often require deeper domain expertise, architectural knowledge, or the ability to reason over complex systems, which current LLMs may not fully possess [5].

High-complexity tasks, such as designing a new microservices architecture or implementing advanced machine learning algorithms, involve abstract thinking and decision making that go beyond pattern recognition. The system may struggle with:

- **Understanding Complex Requirements**: Difficulty in interpreting nuanced specifications or stakeholder expectations.
- **Architectural Decision Making**: Challenges in making high-level design choices that consider scalability, security, and performance tradeoffs.
- **Integrating with Proprietary Systems**: Limited ability to interact with closed source systems or APIs without prior exposure.

To address these challenges, future iterations could enhance agent capabilities by integrating external knowledge bases or domain specific datasets to augment the LLM's understanding [8]. Incorporating fine-tuning techniques tailored to specific industries or tasks could significantly improve performance [2].

6.2. LLMs Cannot Execute or Run Code

LLMs, while proficient in generating code, do not possess the ability to run or test the code they produce. This lack of execution capability limits the system's ability to validate the correctness or functionality of the code autonomously. The generated code must be passed to a human or an external automated testing system for validation.

This limitation impacts the system's ability to:

- **Catch Runtime Errors**: Issues that only manifest during execution may go unnoticed.
- **Optimize Performance**: Without execution, the system cannot profile code to identify bottlenecks.
- **Ensure Security**: Dynamic security vulnerabilities may not be detected without running the code.

Integrating automated code execution or simulation environments may allow the system to perform deeper validations, such as running tests and identifying performance issues before human intervention is required [13].

6.3. Future Directions

While our system already employs retrieval augmented generation (RAG) for code context, there is significant potential for further enhancing its capabilities by integrating more sophisticated context-handling mechanisms and external data sources. Expanding the use of RAG to include broader knowledge bases, documentation repositories, or domain-specific data could provide the additional context needed to solve more complex, specialized tasks [8].

Additionally, specialized fine-tuning for agents based on domain expertise would allow for improved performance in specific industries such as healthcare, finance, or enterprise software [2]. Techniques like Chain-of-Thought prompting [11] and multiple thought chains [6] could be employed to enhance the reasoning capabilities of the agents. Incorporating these methods may enable the agents to handle more complex decision-making processes by simulating deeper reasoning steps.

Furthermore, integrating debate mechanisms among agents [12] could lead to more accurate and robust outcomes. By allowing agents to critique and build upon each other's suggestions, the system may arrive at more optimal solutions, particularly for tasks that require nuanced judgment or have multiple possible approaches.

Integrating real-time execution environments into the system would allow for automated code execution, testing, and validation, enhancing autonomy and reducing the need for manual oversight [3]. This would enable the agents to verify the correctness of the generated code, catch runtime errors, and perform performance profiling without human intervention.

Moreover, exploring the incorporation of automated code analysis tools, such as static analyzers and linters, could improve the quality and security of the generated code. Combining these tools with the agents could help identify potential bugs or vulnerabilities early in the development process, ensuring higher code reliability and maintainability.

Finally, enhancing the agents' ability to handle high-complexity tasks remains a significant area for future work. This could involve integrating more advanced reasoning capabilities, leveraging domain-specific ontologies, or employing hierarchical planning methods to break down complex tasks into manageable subtasks. By addressing these challenges, the system could extend its applicability to a wider range of software development scenarios, including those that require deep domain knowledge and complex architectural decisions.

VII. CONCLUSION

We have introduced a multi-agent LLM system designed to simulate the Agile software development process, automating key tasks such as user story creation, task planning, code generation, and pull request submission. The system has shown great promise in handling low to medium- complexity tasks with minimal human intervention, successfully streamlining the development workflow.

While the system excels in these areas, future work will focus on expanding its capabilities to tackle more complex and domain-specific tasks. Enhancing agent collaboration, incorporating domain-specific knowledge, and integrating real-time code execution environments are among the key improvements that will drive the system toward greater autonomy and flexibility. With these advancements, the system has the potential to further reduce human involvement in soft- ware development while maintaining high standards of quality and efficiency.

ACKNOWLEDGEMENTS

We would like to thank the product and engineering team at News Corp for their support and collaboration in this research.

VIII. REFERENCES

[1] OpenAI, GPT-4 Technical Report, 2023.

[2] H. Touvron et al., LLaMA: Open and Efficient Foundation Language Models, 2023.

[3] M. Chen et al., Evaluating Large Language Models Trained on Code, 2021.

[4] Y. Li et al., Embracing Agile with AI: A Multi-Agent System for Software Development, 2023.

[5] D. Hendrycks et al., Measuring Massive Multitask Language Understanding, 2021.

[6] Wang et al., Self-Consistency Improves Chain of Thought Reasoning in Language Models, 2023.

[7] Liu et al., Improved Code Generation with LLMs, 2023.

[8] W. Zhao et al., A Survey of Large Language Models, 2023.

[9] Significant Gravitas, AutoGPT: An Autonomous GPT-4 Experiment, 2023.

[10] Z. Hong et al., MetaGPT: Meta Programming for Multi-Agent Collaborative Framework, 2023.

[11] J. Wei et al., Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, 2022.

[12] Z. Du et al., Improving LLM's Mathematical Reasoning via Self-Evaluation and Debate, 2023.

[13] K. Cobbe, V. Kosaraju, M. Bavarian, K. Guu, Ł. Kaiser, M. Plappert, and A. Vaswani, "Training verifiers to solve math word problems