A Multi-Agent Recommendation System for Software Engineering

Juliana Gomes Ribeiro, Joel Machado Pires, Jonas Oliveira Pereira
Computer Institute (IC)
Federal University of Bahia (UFBA)

`{julianagr, joelpires, jonas.pereira}@ufba.br`

1

Abstract. *Recommendation systems have become essential tools in software engineering, assisting developers in various activities throughout the software development lifecycle. In this context, challenges such as team collaboration, deadline estimation, and project management are recurrent and frequently aggravated by variations in development standards, differences in developers' technical skills, and the continuous pursuit of high-quality software that meets real-world demands. This article presents a multi-agent recommendation system for software engineering designed to mitigate the challenges of development and project management. The system supports the entire software engineering process, from requirements gathering to coding, providing intelligent recommendations to human specialists. These specialists can accept or reject the suggestions, dynamically adjusting the direction of the multi-agent system to ensure greater accuracy and alignment with project objectives. Furthermore, the system incorporates a continuous feedback mechanism among the agents, allowing for the constant improvement of recommendation models and increasing the quality of the generated suggestions. With this, the multi-agent recommendation system for software engineering aims to improve user interaction, have greater adherence to the project's real needs, ensure code quality, and create an efficient, collaborative, and reliable environment. By integrating artificial intelligence and human interaction, this system establishes a new paradigm in software development, making it more efficient, structured, and adaptable to market demands.*

Keywords: Software development, multi-agents in software development, LLM in software development.

1. Introduction

In the dynamic field of software engineering, challenges such as collaborative development, schedule estimation, and project management are ever-present. These challenges are compounded by variations in development standards, discrepancies in technical skills among developers, and the perpetual quest for high software quality that aligns with real-world problems. Addressing these issues requires innovative solutions that can enhance collaboration, improve predictability in scheduling, streamline project management, maintain consistent coding practices, bridge skill gaps, ensure software reliability, and accurately capture user needs. Integrating multi-agent systems with Large Language Models (LLMs) offers a promising approach to tackle these challenges head-on.

The application of LLMs has seen exponential growth across various domains, including software engineering and recommender systems. Their ability to process and generate human-like text has opened new avenues for automating complex tasks, enhancing productivity, and improving decision-making processes. LLMs are increasingly being leveraged in software engineering to address long-standing challenges such as requirement ambiguity, code generation, testing, and project management. However, despite their potential, LLMs still face significant limitations, including hallucinations, lack of contextual understanding, and an inability to perform the entire software engineering process autonomously. These challenges highlight the need for a collaborative approach where LLM-powered multiagent systems work alongside human experts to effectively guide the software development lifecycle [Jin et al. 2024, Liu et al. 2024].

Software development is inherently complex, involving multiple stages—from requirements gathering to coding, testing, and deployment—each with its own set of challenges. Human communication gaps, inconsistent development standards, and varying technical skills often lead to inefficiencies and errors. Recent research has explored the use of LLM-based multiagent systems to automate and streamline these processes. For instance, Haolin et al. (2024) propose a framework using transformer-based models like BERT to detect term ambiguities in requirement documents, improving clarity and consistency in interdisciplinary projects [Jin et al. 2024]. Similarly, Junwei et al. (2024) highlight the versatility of LLM-based agents in software engineering, emphasizing their ability to collaborate with humans and external tools to address complex tasks [Liu et al. 2024]. Lin et al. (2024) take this further by emulating software process models using LLM agents to generate artifacts such as requirement documents, code, and test cases. However, their approach lacks mechanisms to ensure code correctness, modularization, and handling complex refactorizations, limiting its practical applicability [Lin et al. 2024].

Other studies, such as Qian et al. (2024), focus on multiagent collaboration frameworks like ChatDev, which integrate LLMs with specialized social roles to develop software solutions. While these frameworks enhance development quality, they often produce low-information-density outputs and struggle with vague requirements, making them more suitable for prototypes than real-world applications [Qian et al. 2024]. Similarly, Jorgensen et al. (2024) explore LLMs for automated test case generation but note that the effectiveness of these models depends heavily on prompt precision and their stochastic nature, which can lead to inconsistent results [Jorgensen et al. 2024]. Sirui et al. (2024) propose MetaGPT, a multiagent system that collaborates with professional development teams to fulfill user specifications. Despite its advantages, MetaGPT faces challenges such as high resource usage and lower productivity metrics compared to other frameworks [Hong et al. 2024].

In the realm of software management, Sami et al. (2024) integrate LLMs like ChatGPT into requirement prioritization processes using methods such as AHP and WSJF. While their approach improves requirement engineering tasks, it does not address time estimation or project scheduling, limiting its overall utility [Sami et al. 2024]. Cinkusz et al. (2024) explore LLM-powered agents in Agile software development, focusing on decision-making and workflow optimization. However, their framework lacks real-world validation and scalability testing, and it does not fully explore multiagent collaboration or communication with human developers [Cinkusz et al. 2025].

Despite these advancements, existing solutions still fall short of addressing the full spectrum of software engineering challenges. LLMs, even in multiagent systems, are prone to hallucinations and struggle to perform the entire software engineering process autonomously. This suggests that LLM-powered systems should not replace human experts but rather work collaboratively with them. To bridge this gap, we propose a recommender system that guides the entire software development process, from requirements gathering to coding. This system provides recommendations to human experts, who can accept or reject them, thereby steering the multiagent system toward the correct direction. By combining the strengths of LLMs with human expertise, this approach aims to enhance the efficiency, accuracy, and reliability of software development processes. This research advances the integration of LLM-powered multiagent systems with human expertise in software engineering by introducing the following key contributions:

- Human-in-the-loop recommender system - A novel framework that combines LLM-based multiagent systems with continuous expert guidance across the software development lifecycle.
- End-to-end process automation with validation mechanisms - System to provide recommendations for all phases of software engineering—requirements gathering, design, coding, testing, and deployment—while embedding validation checkpoints.
- Hallucination mitigation via expert feedback loops - Mechanisms to detect and correct LLM hallucinations by integrating human oversight at critical stages.
- Modularization and code quality enforcement - Automated checks for software standards, cross-file consistency, and refactorization needs during code generation.

By bridging the gap between autonomous LLM agents and human expertise, this work establishes a new paradigm for collaborative, reliable, and scalable software engineering processes.

2. Related Works

Software development presents several challenges due to several issues, such as human communication, development standards, and technical skills. It encompasses the process of management and implementation. The literature has explored the advantages of automating this process through multiagent solutions.

For instance, Haolin et al. (2024) present a comprehensive study on leveraging LLMs in software engineering, focusing on detecting term ambiguities in requirement documents. The study addresses the challenge of identifying terms with different meanings across various domains or interdisciplinary projects. The framework enhances document clarity and consistency by utilizing transformer-based models such as BERT and K-means clustering. Additionally, the work includes a systematic literature review of 117 papers across six software engineering domains, highlighting the growing role of LLM-based agents in the field. This research provides a valuable paradigm for advancing the application of agents empowered with LLMs in requirements engineering [Jin et al. 2024]. Junwei et al. (2024) also present a comprehensive survey on LLM-based agents in software engineering, providing an organized analysis of existing research in the field. It is possible to conclude that LLM-based agents enhance versatility and expertise. The integration of LLM with agent capabilities allows these models to perceive and utilize external resources and tools, significantly extending their applicability in software engineering tasks. Also, there is rapid growth and increasing interest in LLM-based agents within the software engineering domain. The work suggests that LLM-based agents hold promise for addressing complex software engineering challenges through their ability to collaborate with humans and other systems [Liu et al. 2024]

Lin et al. (2024) present an approach to code generation by emulating software process models using LLM agents. The framework demonstrates how agents in each role generate artifacts such as requirement documents, design documents, code, and test cases, with each step building on the output of the previous activity. However, the work acknowledges that there is no guarantee of the correctness of the generated tests, which could potentially lead to nonsensical or incorrect code. The study does not explicitly provide mechanisms for ensuring modularization and other software standards in the generated code. The study does not explicitly address scenarios for handling refactorization needs, such as modifying a module or fixing bugs that require changes across multiple files. In summary, while the work presents a promising approach to emulating software development processes using LLM, it does not fully address issues like ensuring code correctness, modularization, or handling complex refactorizations [Lin et al. 2024].

Qian et al. (2024) show a chat-powered software development framework integrating LLM as autonomous agents with specialized social roles. It focuses on enabling multi-agent collaboration to develop comprehensive software solutions through structured communication patterns and role-based interactions [Qian et al. 2024]. The framework utilizes a structured approach where these specialized agents interact through both natural and programming languages at various stages, guided by mechanisms such as a via chat chain for directing content and communicative dehallucination for managing communication flow. However, while the automated agents enhance development quality by implementing simple logic, they often result in low information density and struggle to grasp task ideas without clear, detailed requirements. Agents may produce basic or static representations without well-defined and detailed software requirements. For example, vague guidelines can lead to simplistic outcomes, such as basic key-value placeholders instead of integrating external databases. Currently, these technologies are more suitable for prototype systems rather than complex real-world applications, limiting their broader adoption and practical impact.

Some works have focused on automating test writing rather than the whole software artifact generation. Jorgensen et al. (2024) explore the application of LLMs for generating test cases tailored to specific curricula, such as C-PAR, D-PAR, C-SEQ, and D-SEQ. The method involves creating detailed prompts that outline high-level specifications and optional target difficulty levels, which are then input into an LLM to generate the desired test cases. The approach allows for the iterative generation of multiple test cases [Jorgensen et al. 2024]. However, this work still has some limitations. The effectiveness of LLM-generated test cases heavily depends on the prompts' precision, where an effective prompt may vary from one LLM to another. The inherent stochastic nature of LLMs can result in different outcomes from the same prompt, which hinders reproducibility. The model's performance is influenced by its familiarity with the problem domain and potential biases present in its training data.

Sirui et al. (2024) proposed an approach for a collaborative process between a multiagent system (MetaGPT) empowered by LLM and a professional development team to fulfill user specifications. The proposed approach employs a system of distinct agents with defined roles, such as the Product Manager, which operates within standardized protocols. The system operation begins with the user providing an input command, such as writing a Python GUI application for drawing image specifications. The product manager (an agent) then generates a product requirement document (PRD), which includes goals, user stories, competitive analysis, and requirement pools. The process is iterative, ensuring that the software developed meets the user's requirements effectively [Hong et al. 2024]. While it presents some advantages, it faces limitations, such as high resource usage. MetaGPT exhibits significantly higher token usage than ChatDev (31,255 vs 19,292 tokens), indicating increased computational costs and resource requirements. The productivity metrics for MetaGPT are notably lower than those of ChatDev (124.3 vs 248.9), suggesting that while it may produce high-quality code, its speed or output rate is efficiently reduced.

Focusing on software management, Sami et al. (2024) explore the integration of LLM such as ChatGPT 3.5 or 4 to enhance software development processes by prioritizing requirements using established methods like Analytic Hierarchy Process (AHP), Weighted Shortest Job First (WSJF). The main focus of this work is the integration of LLM into requirement engineering processes to enhance tasks such as requirement prioritization. The proposed approach facilitates user story generation and prioritization. It also addresses challenges related to consistency and unpredictability. Particularly for tasks such as translating user requirements into structured deliverables like user stories, UML diagrams, API specifications, and test suites [Sami et al. 2024]. However, this work does not address the challenge of estimating the time an employee would spend performing some tasks, nor the total time estimation for the project and scheduling plan suggestion.

Similarly, Cinkusz et al. (2024) explore agents powered by LLM in managing Agile software development tasks. It focuses on how these agents can enhance decision-making processes, optimize workflows, and dynamically adapt to project needs. The study emphasizes the importance of output analysis, particularly through HTML files that visually represent agent interactions. This allows for clearer identification of patterns, decision points, and communication effectiveness [Cinkusz et al. 2025]. Still, the presented approach has some limitations. The experiments were conducted under controlled conditions with limited project types. This restricts the generalizability of the findings to more diverse and complex real-world scenarios. It lacks real-world validation through feedback from actual developers or in live software development environments. It has limited exploration of multi-agent collaboration, not exploring how these agents interact and collaborate. Also, the framework has not been tested for scalability in large enterprises or highly complex projects. The study still lacks the effectiveness of communication between agents and human developers.

The reviewed literature highlights significant progress in leveraging multiagent systems and LLMs to address software engineering challenges, including requirement ambiguity, code generation, test automation, and project management. While these approaches demonstrate the potential of automation in streamlining workflows and enhancing decision-making, critical gaps persist. Existing frameworks often focus on isolated tasks, lack end-to-end process coverage, and struggle with issues such as LLM hallucinations, low-information-density outputs, and insufficient code modularity. Additionally, many solutions exhibit limited scalability, resource inefficiency, or inadequate validation in real-world scenarios, relying heavily on controlled environments or idealized inputs. The absence of robust mechanisms for human-agent collaboration further restricts their practical applicability, as fully autonomous systems risk propagating errors and oversimplifying complex requirements. These limitations underscore the need for a paradigm shift toward integrating human expertise with multiagent systems, ensuring iterative validation, adaptability, and reliability across the entire software development lifecycle.

3. Preproject of Agents

The agents will interact within an environment that is described as follows.

- Partial observable: because the agents can read some parts of the internet, not entirely, and they can partially check the state of the operation system. The agents have access to the whole history of actions performed in the environment, but only a limited part of the feedback is from it.
- Non-deterministic: the agents may not provide some artifacts correctly. For example, the agents can generate the wrong code in a programming language. Then, the result may not be as expected, causing the necessity to replan. Additionally, any modification of the agents over the environment can introduce bugs in the system under development. Finally, the action requested by the agents may not be able to be performed.
- Sequential: if any action fails, the agents must replan.
- Static: the agents only deliberate when the environment is stable, and no agents act over it.
- Discrete: the environment has a limited set of states, and the agents have a limited set of perceptions and actions.
- Multiagent: The system has more than one type and quantity of agents.

The agents of the system are the Project Manager, Developer Specialist, Researcher, Software Engineer, and Operation System Controller. The Project Manager is responsible for interacting with the user expert to propose the system architecture and manage the project, such as defining the requirements and technologies, the schedule, epoch, stories, and tasks. The Developer Specialist is responsible for receiving and developing an artifact's specification. The Researcher can access the internet. For example, the developer may need to refact an artifact to fix a bug, and then the researcher can look for possible solutions online. The Software Specialist is responsible for continuously checking the development process and asking the Developer Specialist to refact some parts of the software or create new artifacts based on the opened tasks. The Operation System Controller agent has limited access to the operation system where the multiagent system is running and can perform a limited set of actions, as requested by other agents. It is responsible for validating and executing the requested action, if valid. The tables 1, 2, 3, 4, and 5 describe the set of performance, the environment, set of actuators and sensors (PEAS) of the agents, Project Manager, Developer Specialist, Researcher, Software Engineer, and Operation System Controller, respectively.

Table 1. PEAS of the Project Manager agent

PEAS	Description
Performance	Provide correct suggestions for system architecture, schedule, roles assignment, and breaking down epochs.
Environment	Access to the database; communicates with Researcher agents, Developer agents, and expert users.
Actuators	Send messages and request writes to the database.
Sensors	Read messages

Table 2. PEAS of the Developer Specialist Agent

PEAS	Description
Performance	Develop software artifacts according to the specifications received, ensuring quality and compliance with requirements.
Environment	Access to the code repository; communicates with Project Manager, Software Engineer, and Researcher agents.
Actuators	Write and modify source code; send messages; request commits to the repository.
Sensors	Read specifications and messages; receive feedback on the developed code.

Table 3. PEAS of the Researcher Agent

PEAS	Description
Performance	Provide research on solutions, libraries, tools, or any relevant information to assist other agents, especially the Developer Specialist.
Environment	Access to the internet; communicates with Project Manager, Developer Specialist, and Software Engineer agents.
Actuators	Send messages; search online databases and resources; provide links or information relevant to the agent's queries.
Sensors	Read and interpret incoming requests for information, as well as feedback from other agents about the research provided.

Table 4. PEAS of the Software Engineer Agent

PEAS	Description
Performance	Ensure the quality of the software development process; provide feedback to developers and specialists; request refactoring or new tasks when necessary to improve the system.
Environment	Communicates with Project Manager, Developer Specialist, Researcher, and Operation System Controller agents.
Actuators	Send messages to other agents to request changes or additional tasks; request code refactoring or improvements; update software components.
Sensors	Receive and interpret feedback from the development process; monitor software quality and progress based on incoming messages from the Developer Specialist and other agents.

Table 5. PEAS of the Operation System Controller Agent

PEAS	Description
Performance	Ensure the stability and correctness of the operation system; validate and execute actions requested by other agents; monitor and manage system resources.
Environment	Limited access to the operation system environment where the multi-agent system runs; communicates with the Project Manager, Developer Specialist, and other agents.
Actuators	Perform actions such as executing commands or modifying system settings as requested by other agents; write and update the system's state based on agent requests.
Sensors	Read system status, monitor resource usage, and receive requests from other agents to perform actions on the operation system.

4. Architectural Decisions

This section outlines the proposed solution, technologies, and its architecture.

4.1. Tecnologies

We used Python, Redis (Remote Dictionary Server), MongoDB, and LLM as the primary technologies for development.

4.1.1. Python

The Python programming language is widely used for software development and machine learning. The language is efficient, high-level, and can run on many platforms. Additionally, it has a large standard library that includes reusable code, especially in the context of machine learning [1].

4.1.2. Remote Dictionary Server

Redis is an open-source, in-memory, key-value NoSQL data structure store primarily used as a cache for applications or a fast-response database. It also has some important features such as Redis Pub/Sub, which is being used in the system. Redis Pub/Sub supports the use of publish and subscribe commands. Users can create high-performance chat and messaging services across their applications and services [2]. The Redis service is utilized to establish a communication channel among all agents.

4.1.3. MongoDB

MongoDB is a free and open-source, non-relational database management system that uses flexible documents instead of tables and rows to process and store various forms of 1 data. It allows the user to easily query and store various data types. The documents are formatted as binary Javascript Object Notation (JSON), which can store various data types and be distributed to different systems. MongoDB is used to persist the history of actions performed in the software development process and to aid in traceability [3].

4.1.4. Large Language Models

LLMs are sophisticated systems that leverage the Transformer architecture. They are trained on massive datasets of text and code sourced from the internet, covering a vast spectrum of topics and linguistic styles. This extensive training enables LLMs to learn intricate statistical relationships and patterns within the data, going beyond simple memorization. As a result, they can perform a variety of tasks, including generating human-like text, translating languages, writing different kinds of creative content, and answering

[1] https://aws.amazon.com/pt/what-is/python/
[2] https://www.ibm.com/br-pt/topics/redis
[3] https://www.ibm.com/br-pt/topics/mongodb

questions. They demonstrate an impressive ability to make associations and exhibit reasoning capabilities based on a given input primarily based on pattern recognition and statistical correlations present in the data used to train.

4.2. System Architecture

Following the architectural decisions, such as the choice of programming language and frameworks, we can define the distribution of modules and their communication.

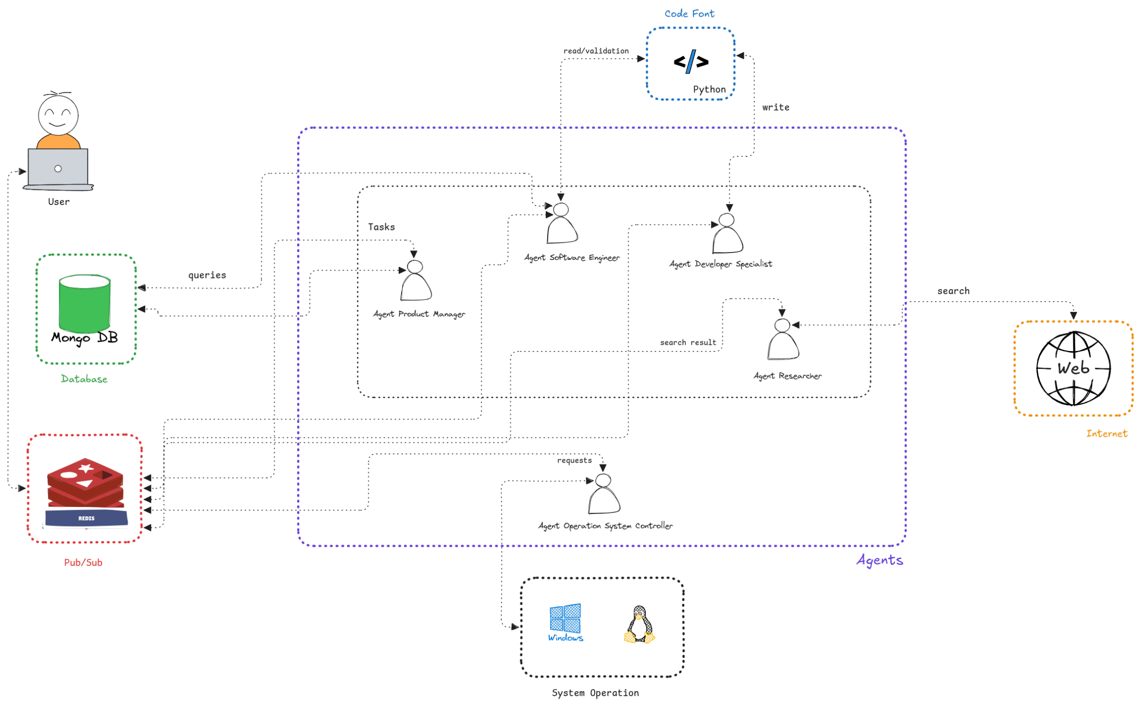


Figure 1. System Architecture. Multiagent System of Recommendation System for Software Engineering.Font: the Author

Figure 4.2 illustrates the interconnections between the modules. In this representation, the system includes a single instance of the Redis service, which acts as a communication bridge among all agents, including the user. It provides a shared channel where agents can send and receive messages. This approach allows for flexibility in refining agent communication as the system evolves rather than defining specific communication rules for each agent. The objective is to enable agents to determine the appropriate recipients necessary to achieve their goals autonomously. For example, the Project Manager agent might need to communicate with both the Software Developer agent and the User, so this agent also has that flexibility. In addition to its effective communication architecture, agents need a memory system to recall past events and reason appropriately.

The MongoDB service provides a structured approach to long-term memory for the agents. This service enables agents to manage the project's database, where messages and project-related entities, such as epics, stories, and tasks, are stored. The Project Manager agent may need to retrieve past messages to better understand the context and assess the current state of the project and other agents. Additionally, it may need to review open, ongoing, and completed stories and tasks to determine whether new tasks or stories should be created, initiated, or completed, as well as to prioritize them accordingly.

Similarly, the Software Engineer agent may need to evaluate whether the project requires refactoring, if a new dependency should be added, or where a new artifact should be created, updated, or removed. These decisions allow it to guide the Developer Specialist and Operating System agents. The Developer Specialist may also need access to the database to review the adopted technologies and coding standards, which the Software Engineer should provide to them.

Finally, the system includes additional actuator interfaces beyond database management, namely *Code Font* and *Internet*. These interfaces enable agents to control the operating system for managing the project's code, including creating, deleting, updating, and executing command-line operations. Additionally, they allow agents to conduct research on the internet.

5. Tropos Model (MOD)

The architecture was modeled using Tropos, a goal-oriented software development methodology, to better represent the reasoning and interactions between different agents in the system. The Tropos framework helps illustrate not only the functional aspects of each agent but also their dependencies, beliefs, and interactions.

The Tropos model is depicted in Figure 5, which illustrates the architecture of the multi-agent system, with each shape representing a fundamental concept:

- **Actors (depicted as circles):** Represent autonomous agents in the system, such as the *Project Manager*, *Software Engineer*, *Developer Specialist*, *Operation System Controller*, and *Research Agent*, each responsible for distinct tasks and interactions.
- **Goals (represented by ovals):** Indicate the high-level objectives that each agent aims to achieve. For instance, the *Software Engineer*'s goal is to Resolve Tasks."
- **Tasks (shown as diamonds):** Define the specific actions agents perform to meet their goals, such as Generate Code", Validate Code" and Perform Research."
- **Resources (displayed as rectangles):** Represent the informational or physical entities required by agents to perform tasks, such as project specifications, feedback, and code artifacts.
- **Dependencies (shown as connections or edges):** Illustrate the relationships between agents, where one agent relies on another to complete certain tasks, share resources or achieve common goals.

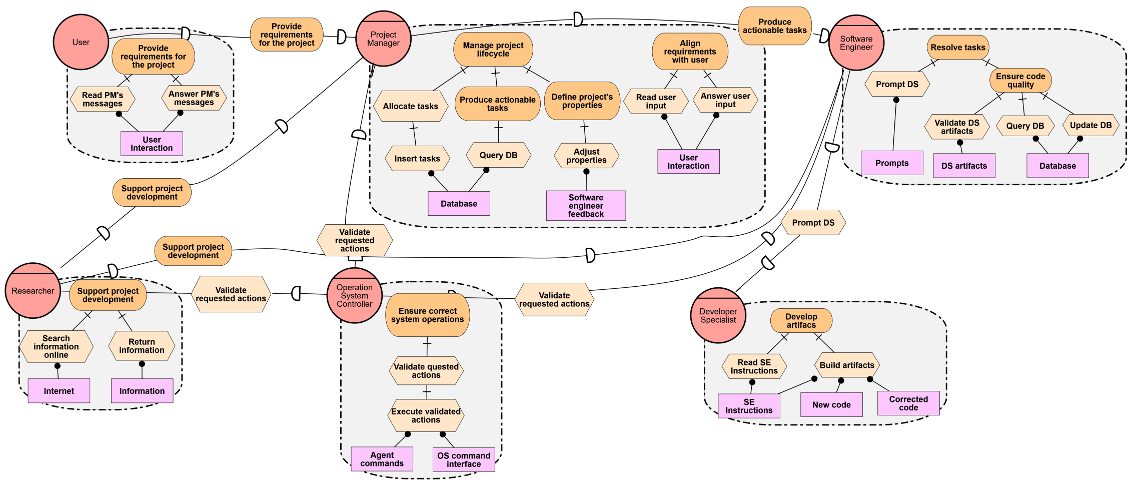


Figure 2. Tropos model representing the interaction between agents in the system.

6. Reasoning Model

In the system, a modeling approach based on predicate logic was adopted to formalize the reasoning process of the agents. This method allows clear representation of both the state of the system and the context in which each agent operates through a set of well-defined predicates.

6.1. Modeling with Predicates

The predicate-based model is built on three key components:

1. **Objects:**
   Objects represent the fundamental entities or events with which the agents interact. For example, a message, a task, or a project specification is modeled as an object. This provides a concrete basis for establishing the state of the system.
2. **Beliefs (Facts):**
   Beliefs are predicates that capture the current state or context of the system. They express conditions such as

   $$\text{message\_received}(m1)$$

   or

   $$\text{task\_id\_present}(m1).$$

   These predicates serve as the foundational knowledge that agents use to make decisions. In essence, they encapsulate the situational context that is essential for reasoning.
3. **Propositions (Rules):**
   Propositions are logical rules — expressed in the form of if–then statements — that define the relationship between the established state (objects and beliefs) and the actions that the agents should execute. For example, a rule might state:

   $$\text{message\_received}(m1) \land \text{sender\_is\_pm}(m1) \to \text{retrieve\_task}(t1).$$

   This rule means that when a message is received and the sender is verified as the Project Manager, the system should retrieve the corresponding task. By defining such propositions, the decision-making process is effectively mapped into formal, testable logical expressions.

6.2. Advantages of Predicate-Based Modeling

This modeling strategy offers several advantages:

- **Clarity:** By breaking down the reasoning process into objects, beliefs, and rules, a transparent model is created where each aspect of the agent's decision-making is explicitly defined.
- **Modularity:** The logical structure allows each agent's behavior to be specified independently. Complex interactions are decomposed into smaller, manageable rules that can be individually analyzed and refined.
- **Flexibility:** Using predicates to represent both state and actions makes it easy to update or extend the model. New conditions, actions, or entire agents can be integrated without disrupting the overall system.

- **Formal Rigor:** The use of predicate logic ensures that the system's behavior is grounded in a formal framework. This not only improves reliability but also facilitates verification and validation of the agents' decision-making processes.

In the following sections, the detailed predicate models for each agent (Project Manager, Software Engineer, Research Agent, and Operation System Agent) are presented, illustrating how state and context are captured through objects and beliefs, and how rules and actions are defined through propositions.

6.3. Project Manager Agent

Objects:

- `message(m).` The incoming message.
- `project_specifications(ps).` The project specifications.
- `user_interaction(ui).` The history of user interactions (e.g., conversations).

Beliefs:

- `message_received(m).` A message has been received.
- `valid_message(m).` The message is valid.
- `sender_is_user(m).` The sender is the user.
- `research_needed(m).` Research is required.

Propositions:

1. **Define Project Specifications and Determine Required Actions:**

   There are two cases:

   (a) If user interaction is available and research is needed, then send a request to the Research Agent:

   $$\text{avaliable\_user\_interaction}(ui) \wedge \text{research\_needed}(pc) \rightarrow \text{send\_request\_to\_research\_agent}(m).$$

   (b) Otherwise, if user interaction is available, the start is authorized, and no research is needed, then define the project specifications and decide on the necessary actions:

   $$\text{avaliable\_user\_interaction}(ui) \wedge \text{start\_authorized}(m) \wedge \neg \text{research\_needed}(pc) \rightarrow \text{define\_projec}$$

2. **Execute Commands:**

   Once the actions are decided, execute the corresponding commands:

   $$\text{decide\_actions}(m) \rightarrow \text{execute\_commands}(m).$$

3. **Update Memory and Respond:**

   After executing the commands, update the agent's memory and send a response:

   $$\text{execute\_commands}(m1) \rightarrow \text{update\_memory\_and\_respond}(m1).$$

6.4. Software Engineer Agent

Objects:

- `message(m1).` — The incoming message.
- `task_id(it1).` — The identifier for the task.
- `task(t1).` — The task retrieved from the database.
- `prompt(p1).` — The prompt generated for further instructions.
- `code_artifact(a1).` — The code artifact produced by the Developer Specialist.
- `feedback(f1).` — Feedback generated after code validation.
- `project_context(pc).` — The contextual information of the project.

Beliefs:

- `message_received(m1).` — A message has been received.
- `sender_is_pm(m1).` — The sender is the Project Manager.
- `sender_is_ds(m1).` — The sender is the Developer Specialist.
- `task_id_present(m1).` — A task ID is present in the message.

Propositions (Rules):

1. **Retrieve Task from the Database:**

   When a message is received that includes a task identifier from the Project Manager, the agent retrieves the corresponding task:

   $$\text{message\_received}(m1) \wedge \text{task\_id\_present}(m1) \wedge \text{sender\_is\_pm}(m1) \rightarrow \text{retrieve\_task}(t1).$$

2. **Generate a Prompt for the Task:**

   After retrieving the task, the agent generates a prompt to instruct further development:

   $$\text{retrieve\_task}(it1) \rightarrow \text{generate\_prompt}(t1).$$

3. **Decide Based on Prompt Generation:**

   If a prompt is successfully generated, the agent sends it to the Developer Specialist; otherwise, it sends an error back to the sender:

   $$\text{generate\_prompt}(t1) \rightarrow \text{send\_prompt\_to\_developer}(p1).$$

   $$\text{generate\_prompt}(t1) \rightarrow \text{send\_error\_to\_sender}(pm1).$$

4. **Validate Code Artifact and Provide Feedback:**

   When a message is received from the Developer Specialist, the agent validates the code artifact. If the artifact is invalid, feedback is sent to the Developer Specialist. If the artifact is valid and no new requirements are detected, it is forwarded to the OS agent. If new requirements are detected, feedback is sent to the Project Manager:

   $$\text{message\_received}(m1) \wedge \text{sender\_is\_ds}(m1) \rightarrow \text{validate\_code\_artifact}(m1).$$

   $$\neg\text{valid\_code\_artifact}(a1) \rightarrow \text{send\_feedback\_to\_ds}(f1).$$

   $$\text{valid\_code\_artifact}(a1) \wedge \neg\text{new\_requirements\_required}(a1) \rightarrow \text{send\_to\_os}(a1).$$

   $$\text{valid\_code\_artifact}(a1) \wedge \text{new\_requirements\_required}(a1) \rightarrow \text{send\_feedback\_to\_pm}(f1).$$

6.5. Developer Specialist Agent

Objects:

- `message(m1).` — The incoming message.
- `feedback(f).` — Feedback to consider when generating code.
- `prompt(p).` — The prompt that guides code generation.

Beliefs:

- `message_received(m).` — A message has been received.
- `task_id(it1).` — The message includes a task identifier.
- `sender_is_se(m).` — The sender of the message is the Software Engineer.
- `includes_prompt(m).` — The message includes a prompt.
- `includes_feedback(m).` — The message includes feedback.

Propositions:

1. **Generate Code from Prompt:**

   When a message is received from the Software Engineer that includes a prompt, the Developer Specialist generates the corresponding artifact code:

   $\text{message\_received}(m) \wedge \text{sender\_is\_se}(m) \wedge \text{includes\_prompt}(m) \rightarrow \text{generate\_artifact\_code}(p).$

2. **Generate Code from Prompt and Feedback:**

   If the message includes feedback, then the Developer Specialist generates the artifact code using both the prompt and the feedback:

   $\text{message\_received}(m) \wedge \text{sender\_is\_se}(m) \wedge \text{includes\_feedback}(m) \rightarrow \text{generate\_artifact\_code}(p, f).$

3. **Notify Error to the Software Engineer:**

   If the artifact code is not generated, the agent sends an error response to the Software Engineer:

   $$\neg \text{artifact\_code\_generated}(D) \rightarrow \text{send\_error\_response\_to\_se}(D).$$

4. **Send Generated Code to the Software Engineer:**

   If the artifact code is successfully generated, it is sent to the Software Engineer:

   $$\text{artifact\_code\_generated}(D) \rightarrow \text{send\_artifact\_code\_to\_se}(D).$$

6.6. Research Agent

Objects:

- `research_request(m).` — Represents the incoming research request.
- `research_result(r).` — Represents the result obtained from the search.

Beliefs (Facts):

- `request_received(m).` — A research request has been received.
- `sender_allowed(m).` — The sender is authorized to request research.

Propositions (Rules):

1. **Perform Search:**

   When a request is received and the sender is allowed, the agent performs a web search:

$$\text{message\_received}(m) \wedge \text{sender\_allowed}(m) \rightarrow \text{perform\_search}(m).$$

2. **Summarize Results or Notify of No Results:**

   If the search yields no results, a "no results" response is sent; otherwise, the results are summarized:

$$\text{search\_results}(m) = \varnothing \rightarrow \text{send\_no\_results\_response}(m).$$

$$\text{search\_results}(m) \neq \varnothing \rightarrow \text{summarize\_results}(r).$$

3. **Send Research Response:**

   Once the results are summarized, the agent sends a research response:

$$\text{summarize\_results}(r) \rightarrow \text{send\_research\_response}(r).$$

6.7. Operation System Agent

Objects:

- `message(m).` — The incoming message.
- `os_action(a).` — The system-level action that is to be executed.

Beliefs (Facts):

- `message_received(m).` — A message has been received.
- `valid_action(i).` — The action specified in the message is valid.

Propositions (Rules):

1. **Extract OS Actions:**

   When a message is received, the agent extracts the system-level actions contained within:

$$\text{message\_received}(m) \rightarrow \text{extract\_os\_actions}(m).$$

2. **Execute OS Actions:**

   The extracted actions are then executed:

$$\text{extract\_os\_actions}(m) \rightarrow \text{execute\_os\_actions}(a).$$

3. **Notify Success:**

   If the OS actions are executed successfully, the agent sends a success response:

$$\text{execute\_os\_actions}(M) \rightarrow \text{send\_success\_response}(M).$$

4. **Notify Failure:**

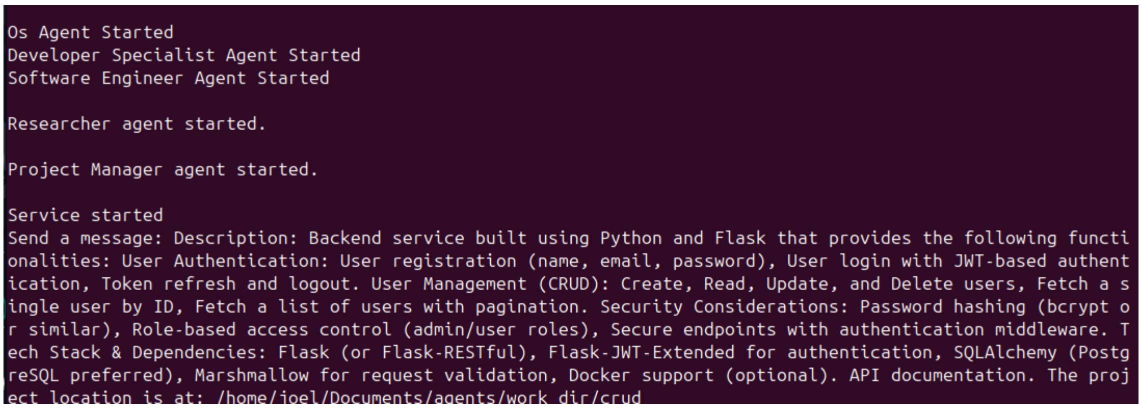
   If the OS actions fail to execute, the agent sends a failure response:

$$\neg\text{execute\_os\_actions}(M) \rightarrow \text{send\_failure\_response}(M).$$

7. Results

The implemented multi-agent system demonstrates a collaborative framework where autonomous agents, each powered by a dedicated Large Language Model (LLM), coordinate to achieve software development tasks. Agents operate independently, with specialized roles, reflecting a modular division of labor. Communication occurs through a Redis-based shared channel, where messages are directed to specific agents, illustrated by ”@” notation, in Figure 7. This figure shows the system being started by the user.

User interaction serves as the system’s trigger, as seen in the command-line initialization of agents and service deployment. Upon startup, agents self-announce their roles (e.g., “Project Manager agent started”), and the workflow aligns with the user’s specifications, such as API documentation and Docker support. The integration of Flask, SQLAlchemy, and security libraries reflects adherence to the defined tech stack, validating the agents’ ability to parse requirements and execute dependency management. Overall, the results illustrate a functional multi-agent ecosystem capable of task delegation, error reporting, and collaborative development, bridging user intent with technical implementation through LLM-driven reasoning.

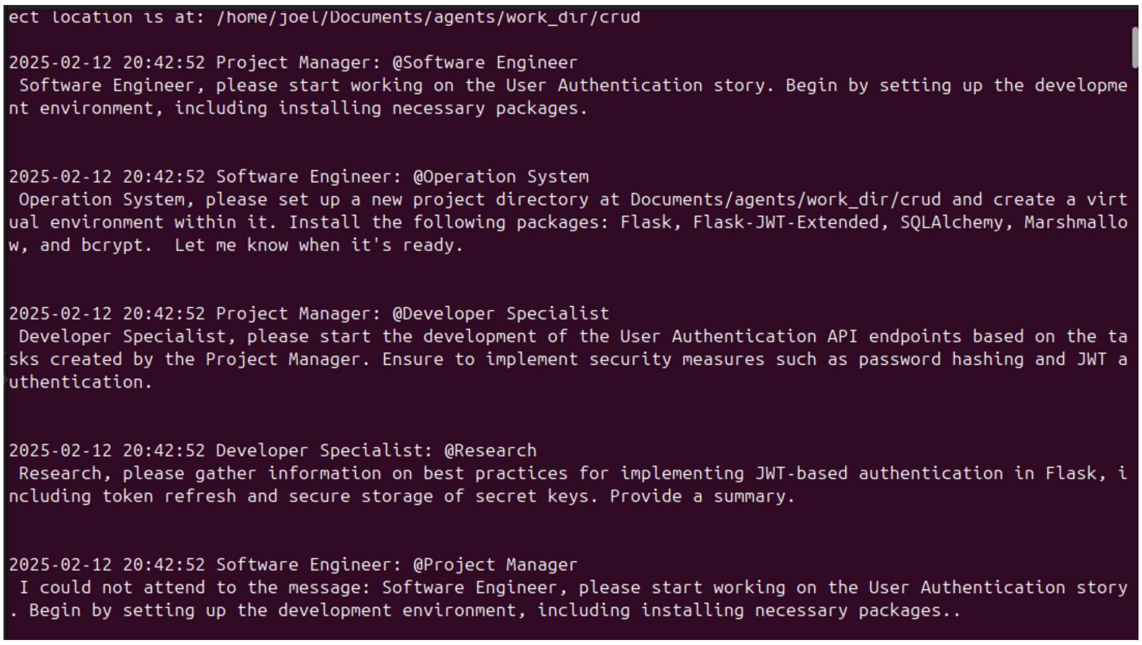


```
Os Agent Started
Developer Specialist Agent Started
Software Engineer Agent Started

Researcher agent started.

Project Manager agent started.

Service started
Send a message: Description: Backend service built using Python and Flask that provides the following functi
onalities: User Authentication: User registration (name, email, password), User login with JWT-based authent
ication, Token refresh and logout. User Management (CRUD): Create, Read, Update, and Delete users, Fetch a s
ingle user by ID, Fetch a list of users with pagination. Security Considerations: Password hashing (bcrypt o
r similar), Role-based access control (admin/user roles), Secure endpoints with authentication middleware. T
ech Stack & Dependencies: Flask (or Flask-RESTful), Flask-JWT-Extended for authentication, SQLAlchemy (Postg
reSQL preferred), Marshmallow for request validation, Docker support (optional). API documentation. The proj
ect location is at: /home/joel/Documents/agents/work_dir/crud
```

Figure 3. Print screen of the terminal when executing the system and giving an input to start the interaction.

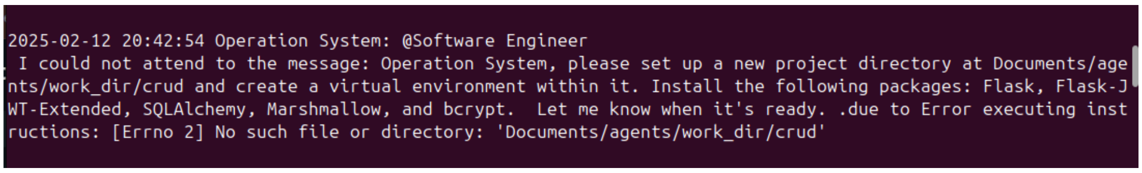
For instance, in Figure 7, the Project Manager initiates tasks by delegating to the Software Engineer (e.g., “@Software Engineer, please start working on the User Authentication story”), who then interacts with the Operation System agent to execute environment setup commands. This structure ensures accountability and clarity in task ownership, as agents respond only to messages explicitly addressed to them.

The logs reveal a dynamic workflow where agents handle both task execution and error resolution, as depicted by Figure 7. When the Operation System agent encounters a missing directory error (“No such file or directory: ’Documents/agents/work_dir/crud’”), it immediately reports the failure, prompting downstream adjustments. Such errors highlight the system’s capacity for self-diagnosis and transparency, enabling iterative problem-solving. Meanwhile, the Developer Specialist and Research agents collaborate on security implementations (e.g., JMT authentication, password hashing), leveraging external knowledge to refine best practices. The project structure further underscores organizational rigor with modular directories for agents, utilities, and configuration files, ensuring scalability.



```
ect location is at: /home/joel/Documents/agents/work_dir/crud

2025-02-12 20:42:52 Project Manager: @Software Engineer
 Software Engineer, please start working on the User Authentication story. Begin by setting up the developme
nt environment, including installing necessary packages.


2025-02-12 20:42:52 Software Engineer: @Operation System
 Operation System, please set up a new project directory at Documents/agents/work_dir/crud and create a virt
ual environment within it. Install the following packages: Flask, Flask-JWT-Extended, SQLAlchemy, Marshmallo
w, and bcrypt.  Let me know when it's ready.


2025-02-12 20:42:52 Project Manager: @Developer Specialist
 Developer Specialist, please start the development of the User Authentication API endpoints based on the ta
sks created by the Project Manager. Ensure to implement security measures such as password hashing and JWT a
uthentication.


2025-02-12 20:42:52 Developer Specialist: @Research
 Research, please gather information on best practices for implementing JWT-based authentication in Flask, i
ncluding token refresh and secure storage of secret keys. Provide a summary.


2025-02-12 20:42:52 Software Engineer: @Project Manager
 I could not attend to the message: Software Engineer, please start working on the User Authentication story
. Begin by setting up the development environment, including installing necessary packages..
```

Figure 4. Print screen with the agent's communication ongoing.



```
2025-02-12 20:42:54 Operation System: @Software Engineer
 I could not attend to the message: Operation System, please set up a new project directory at Documents/age
nts/work_dir/crud and create a virtual environment within it. Install the following packages: Flask, Flask-J
WT-Extended, SQLAlchemy, Marshmallow, and bcrypt.  Let me know when it's ready. .due to Error executing inst
ructions: [Errno 2] No such file or directory: 'Documents/agents/work_dir/crud'
```

Figure 5. Print screen of a case where Operation System agent encountered an error and reported to Software Engineer agent.

8. Conclusion

This work proposes a recommendation system powered by multi-agent systems for semi-autonomous software development, addressing some of the gaps identified in existing solutions. To support this proposal, a literature review was conducted to analyze trends in autonomous software development using multi-agent systems, where LLM is recognized as the primary reasoning approach. LLMs have emerged as powerful models capable of making decisions and understanding contextual environments by capturing logical patterns from vast amounts of training data. Based on this analysis, key limitations in current solutions were identified and mitigated through the proposed system.

This system features an enhanced architecture designed to improve agent communication efficiency, addressing a key limitation in previous solutions. Additionally, the recommendation approach mitigates several challenges faced by current autonomous systems, such as LLM hallucinations, agent decision uncertainty, and limited user interaction. Consequently, the proposed approach yielded a system capable of performing software development in a semi-autonomous manner. This system enables continuous interaction with a software engineering expert (SE) to oversee the development process and generate software artifacts. Operating dynamically, the system employs an agile methodology rather than a waterfall approach, proactively prompting the user for input as needed. It facilitates the creation, modification, and deletion of epics, user stories, and tasks, while autonomously decomposing stories into subtasks and dynamically prioritizing them in consultation with the SE. A key innovation of this work lies in the system's ability to address architectural decisions: when such choices arise, the integrated multi-agent system proposes contextually appropriate solutions to the user. This adaptive suggestion mechanism distinguishes the approach from prior methodologies and underscores its advancement in semi-autonomous software development.

The proposed solution, while demonstrating potential, presents several areas that require refinement to enhance its efficacy and robustness. A primary concern lies in the incomplete implementation of the proposed architecture. Specifically, the agent's actuators exhibit certain limitations, as only a subset of necessary actions has been realized. This constraint not only hinders the agents' ability to achieve their objectives but also introduces potential vulnerabilities, wherein an agent may erroneously perceive its capacity to execute an action despite the absence of a corresponding interface. Furthermore, the memory module remains in a developmental stage, with its implementation incomplete. This deficiency becomes particularly evident upon system restarts, where agents fail to recognize their position within ongoing processes and thus do not request essential updates from the database—a functionality that has yet to be integrated. Additionally, the current framework lacks support for multiple project developments, which could significantly impede scalability and versatility in complex environments. The user interaction mechanism is similarly underdeveloped, relying solely on terminal-based communication. To enhance accessibility and user experience, the implementation of a web-based interface would be advantageous. Another critical issue pertains to the threading architecture within the multiagent system. Although each agent operates in a separate process, the single-threaded implementation for both listening and responding within each process can introduce potential bottlenecks. This is particularly evident during periods of high message throughput, potentially leading to degraded system performance. Lastly, the interaction prompts with the LLM necessitate refinement to foster more precise reasoning outcomes.

Finally, addressing these limitations—completing the architecture implementation, enhancing actuator functionality, developing memory persistence, enabling multi-project support, advancing user interaction mechanisms, optimizing threading processes, and improving LLM interactions—should be prioritized in future research and development efforts.

References

Cinkusz, K., Chudziak, J. A., and Niewiadomska-Szynkiewicz, E. (2025). Cognitive agents powered by large language models for agile software project management. *Electronics*, 14(1).

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. (2024). Metagpt: Meta programming for a multi-agent collaborative framework.

Jin, H., Huang, L., Cai, H., Yan, J., Li, B., and Chen, H. (2024). From llms to llm-based agents for software engineering: A survey of current, challenges and future.

Jorgensen, S., Nadizar, G., Pietropolli, G., Manzoni, L., Medvet, E., O'Reilly, U.-M., and Hemberg, E. (2024). Large language model-based test case generation for gp agents. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '24, page 914–923, New York, NY, USA. Association for Computing Machinery.

Lin, F., Kim, D. J., et al. (2024). When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*.

Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., and Lou, Y. (2024). Large language model-based agents for software engineering: A survey.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., and Sun, M. (2024). ChatDev: Communicative agents for software development. In Ku, L.-W., Martins, A., and Srikumar, V., editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Sami, M. A., Waseem, M., Rasheed, Z., Saari, M., Systä, K., and Abrahamsson, P. (2024). Experimenting with multi-agent software development: Towards a unified platform. *arXiv preprint arXiv:2406.05381*.