LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead

JUNDA HE, CHRISTOPH TREUDE, and DAVID LO, Singapore Management University, Singapore, Singapore

Integrating Large Language Models (LLMs) into autonomous agents marks a significant shift in the research landscape by offering cognitive abilities that are competitive with human planning and reasoning. This article explores the transformative potential of integrating Large Language Models into Multi-Agent (LMA) systems for addressing complex challenges in software engineering (SE). By leveraging the collaborative and specialized abilities of multiple agents, LMA systems enable autonomous problem-solving, improve robustness, and provide scalable solutions for managing the complexity of real-world software projects. In this article, we conduct a systematic review of recent primary studies to map the current landscape of LMA applications across various stages of the software development lifecycle (SDLC). To illustrate current capabilities and limitations, we perform two case studies to demonstrate the effectiveness of state-of-the-art LMA frameworks. Additionally, we identify critical research gaps and propose a comprehensive research agenda focused on enhancing individual agent capabilities and optimizing agent synergy. Our work outlines a forward-looking vision for developing fully autonomous, scalable, and trustworthy LMA systems, laying the foundation for the evolution of Software Engineering 2.0.

CCS Concepts: • **Software and its engineering → Software development techniques**; **Collaboration in software development**;

Additional Key Words and Phrases: Large Language Models, Autonomous Agents, Multi-Agent Systems, Software Engineering

ACM Reference format:
Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 124 (May 2025), 30 pages.
https://doi.org/10.1145/3712003

1 Introduction

Autonomous agents, defined as intelligent entities that autonomously perform specific tasks through environmental perception, strategic self-planning, and action execution [6, 36, 98], have emerged as a rapidly expanding research field since the 1990s [93]. Despite initial advancements, these early iterations often lack the sophistication of human intelligence [127]. However, the recent advent of **Large Language Models (LLMs)** [65] has marked a turning point. This LLM breakthrough has demonstrated cognitive abilities nearing human levels in planning and reasoning [3, 65], which

Authors' Contact Information: Junda He (corresponding author), Singapore Management University, Singapore, Singapore; e-mail: jundahe@smu.edu.sg; Christoph Treude, Singapore Management University, Singapore, Singapore; e-mail: ctreude@smu.edu.sg; David Lo, Singapore Management University, Singapore, Singapore; e-mail: davidlo@smu.edu.sg.

ACM 1557-7392/2025/5-ART124
https://doi.org/10.1145/3712003

aligns with the expectations for autonomous agents. As a result, there is an increased research interest in integrating LLMs at the core of autonomous agents [90, 134, 148] (for short, we refer to them as *LLM-based* agents in this article).

Nevertheless, the application of singular LLM-based agents encounters limitations, since real-world problems often span multiple domains, requiring expertise from various fields. In response to this challenge, developing ***LLM-Based Multi-Agent (LMA)*** systems represents a pivotal evolution, aiming to boost performance *via* synergistic collaboration. An LMA system harnesses the strengths of multiple specialized agents, each with unique skills and responsibilities. These agents work in concert toward a common goal, engaging in collaborative activities like debate and discussion. These collaborative mechanisms have been proven to be instrumental in encouraging divergent thinking [80], enhancing factuality and reasoning [32], and ensuring thorough validation [146]. As a result, LMA systems hold promise in addressing a wide range of complicated real-world scenarios across various sectors [49, 131, 137], such as **Software Engineering (SE)** [48, 75, 90, 109].

The study of SE focuses on the entire lifecycle of software systems [63], including stages like requirements elicitation [39], development [2], and **Quality Assurance (QA)** [126], among others. This multifaceted discipline requires a broad spectrum of knowledge and skills to effectively tackle its inherent challenges in each stage. Integrating LMA systems into SE introduces numerous benefits:

(1) *Autonomous Problem-Solving*: LMA systems can bring significant autonomy to SE tasks. It is an intuitive approach to divide high-level requirements into sub-tasks and detailed implementation, which mirrors agile and iterative methodologies [68] where tasks are broken down and assigned to specialized teams or individuals. By automating this process, developers are freed to focus on strategic planning, design thinking, and innovation.
(2) *Robustness and Fault Tolerance*: LMA systems address robustness issues through cross-examination in decision-making, akin to code reviews and automated testing frameworks, thus detecting and correcting faults early in the development process. On their own, LLMs may produce unreliable outputs, known as *hallucination* [151, 164], which can lead to bugs or system failure in software development. However, by employing methods like debating, examining, or validating responses from multiple agents, LMA systems ensure convergence on a single, more accurate, and robust solution. This enhances the system's reliability and aligns with best practices in software QA.
(3) *Scalability to Complex Systems*: The growth in complexity of software systems, with increasing lines of code, frameworks, and interdependencies, demands scalable solutions in project management and development practices. LMA systems offer an effective scaling solution by incorporating additional agents for new technologies and reallocating tasks among agents based on evolving project needs. LMA systems ensure that complex projects, which may be overwhelming for individual developers or traditional teams, can be managed effectively through distributed intelligence and collaborative agent frameworks.

Existing research has illuminated the critical roles of these collaborative agents in advancing toward the era of Software Engineering 2.0 [90]. LMA systems are expected to significantly speed up software development, drive innovation, and transform the current SE practices. This article aims to delve deeper into the roles of LMA systems in shaping the future of SE. It spotlights the current progress, emerging challenges, and the road ahead. We provide a systematic review of LMA applications in SE, complemented by two case studies that assess current LMA systems' capabilities and limitations. From this analysis, we identify key research gaps and propose a comprehensive agenda structured in two phases: (1) enhancing individual agent capabilities and (2) optimizing agent collaboration and synergy. This roadmap aims to guide the development of autonomous, scalable, and trustworthy LMA systems, paving the way for the next generation of SE.

To summarize, this study makes the following key contributions:

—We conduct a systematic review of 71 recent primary studies on the application of LMA systems in SE.
—We perform two case studies to illustrate the current capabilities and limitations of LMA systems.
—We identify the key research gaps and propose a structured research agenda that outlines potential future directions and opportunities to advance LMA systems for SE tasks.

2 Preliminary

2.1 Autonomous Agent

An *autonomous agent* is a computational entity designed for independent and effective operation in dynamic environments [98]. Its essential attributes are:

—*Autonomy*: Independently manages its actions and internal state without external controls.
—*Perception*: Detects the changes in the surrounding environment through sensory mechanisms.
—*Intelligence and Goal-Driven*: Aims for specific goals using domain-specific knowledge and problem-solving abilities.
—*Social Ability*: Can interact with humans or other agents, manages social relationships to achieve goals.
—*Learning Capabilities*: Continuously adapts, learns, and integrates new knowledge and experiences.

2.2 LLM-Based Autonomous Agent

Formally speaking, an LLM-based agent can be described by the tuple $\langle L, O, M, P, A, R\rangle$ [24], where:

—L symbolizes the *LLM*, serving as the agent's cognitive core. It is equipped with extensive knowledge, potentially fine-tuned for specific domains, allowing it to make informed decisions based on observations, feedback, and rewards. Typically, an LLM suited for this role is trained on vast corpora of diverse textual data and comprises billions of parameters, such as models like ChatGPT,[1] Claude,[2] and Gemini.[3] These models exhibit strong zero-shot and few-shot learning capabilities, meaning they can generalize well to new tasks with little to no additional training. Interaction with the LLM typically occurs through prompts, which guide its reasoning and responses.
—O stands for the *Objective*, the desired outcome or goal the agent aims to achieve. This defines the agent's focus, driving its strategic planning and task breakdown.
—M represents *Memory*, which holds information on both historical and current states, as well as feedback from external interactions.
—P represents *Perception*, which represents the agent's ability to sense, interpret, and understand its surroundings and inputs. This perception can involve processing structured and unstructured data from various sources such as text, visual inputs, or sensor data. Perception allows the agent to interpret the environment, transforming raw information into meaningful insights that guide decision-making and actions.
—A signifies *Action*, encompassing the range of executions of the agent, from utilizing tools to communicating with other agents.

[1] https://openai.com/chatgpt/
[2] https://claude.ai/
[3] https://gemini.google.com/app

—R refers to *Rethink*, a post-action reflective thinking process that evaluates the results and feedback, along with stored memories. Guided by this insight, the LLM-based agent then takes subsequent actions.

2.3 LLM-Based Multi-Agent Systems

A multi-agent system is a computational framework composed of multiple interacting intelligent agents that interact and collaborate to solve complex problems or achieve goals beyond the capability of any single agent [144]. These agents communicate, coordinate, and share knowledge, often bringing specialized expertise to address tasks across diverse domains.

With the integration of LLMs, *LLM-Based Multi-Agent Systems* have emerged. In this article, we define that an LMA system comprises two primary components: *an orchestration platform* and *LLM-based agents*.

2.3.1 Orchestration Platform. The orchestration platform serves as the core infrastructure that manages interactions and information flow among agents. It facilitates coordination, communication, planning, and learning, ensuring efficient and coherent operation. The orchestration platform defines various key characteristics:

(1) *Coordination Models*: Defines how agents interact, such as cooperative (collaborating toward shared goals) [1], competitive (pursuing individual goals that may conflict) [147], hierarchical (organized with leader-follower relationships) [166], or mixed models.
(2) *Communication Mechanisms*: Determines how the information flows between the agents. It defines the organization of communication channels, including centralized (a central agent facilitates communication [4]), decentralized (agents communicate directly [22]), or hierarchical (information flows through layers of authority [166]). Moreover, it specifies the data exchanged among agents, often in text form. In SE contexts, this may include code snippets, commit messages [169], forum posts [42–44], bug reports [12], or vulnerability reports [55].
(3) *Planning and Learning Styles*: The orchestration platform specifies how planning and learning are conducted within the multi-agent system. It determines how tasks are allocated and coordinated among agents. It includes strategies like *Centralized Planning, Decentralized Execution*—planning is conducted centrally, but agents execute tasks independently, or *Decentralized Planning, Decentralized Execution*—both planning and execution are distributed among agents.

2.3.2 LLM-Based Agents. Each agent may have unique abilities and specialized roles, enhancing the system's ability to handle diverse tasks effectively. Agents can be:

(1) *Predefined or Dynamically Generated*: Agent profiles can be explicitly predefined [48] or dynamically generated by LLMs [134], allowing for flexibility and adaptability.
(2) *Homogeneous or Heterogeneous*: Agents may have identical functions (homogeneous) or diverse functions and expertise (heterogeneous).

Each LLM-based agent can be represented as a node v_i in a graph $G(V, E)$, where edges $e_{i,j} \in E$ represent interactions between agents v_i and v_j.

3 Literature Review

In this section, we review recent studies on LMA systems in SE, organizing these applications across various stages of the software development lifecycle, including requirements engineering,

code generation, QA, and software maintenance. We also examine studies on LMA systems for end-to-end software development, covering multiple SDLC phases rather than isolated stages.

Search Strategy: We conduct a keyword-based search on the DBLP publication database [28] to match article titles. DBLP is a widely used resource in SE surveys [20, 23, 161], which indexes over 7.5 million publications across 1,800 journals and 6,700 academic conferences in computer science.

Our search included two sets of keywords: one set targeting LLM-based Multi-Agent Systems (called [agent words]) and the other focusing on specific SE activities (called [SE words]). Papers may use variations of the same keyword. For example, the term "vulnerability" may appear as "vulnerable" or "vulnerabilities." To address this, we use truncated terms like "vulnerab" to capture all related forms. For LMA systems, we used keywords: *"Agent" OR "LLM" OR "Large Language Model" OR "Collaborat."* To ensure comprehensive coverage of SE activities, we incorporated phase-specific keywords for each stage of the SDLC into our search queries:

(1) *Requirements Engineering*: *requirement, specification, stakeholder*
(2) *Code Generation*: *software, code, coding, program*
(3) *QA*: *bug, fault, defect, fuzz, test, vulnerab, verificat, validat*
(4) *Maintenance*: *debug, repair, review, refactor, patch, maintenanc*

We focus on four key phases of the SDLC: requirements engineering, code generation, QA, and software maintenance. For each phase, the relevant SE keywords are combined using the OR operator to capture all variations. The final search query for each SDLC phase follows the format: [agent words] AND [SE words].

Following the guide of previous work [99, 128, 168], we design the following inclusion and exclusion criteria. In the first phase, we filtered out short papers (exclusion criterion 1) and removed duplicates (exclusion criterion 2). In the second phase, we manually screened each paper's venue, title, and abstract, excluding items such as books, keynote speeches, panel summaries, technical reports, theses, tool demonstrations, editorials, literature reviews, and surveys (exclusion criteria 3 and 4). In the third phase, we conducted a full-text review to further refine relevant studies. Following Section 2.3, we exclude papers that do not describe LMA systems (exclusion criterion 5). Papers that rely solely on LLMs using non-agent-based methods or single-agent approaches are excluded. Furthermore, we focused on LMA systems powered by LLMs with strong planning capabilities, such as ChatGPT and LLaMA, excluding models like CodeBERT and GraphCodeBERT. Since the release of ChatGPT is in November 2022, we limited our review to papers published after this date (exclusion criterion 6). Furthermore, we excluded papers unrelated to SE (exclusion criterion 7) and those that mention LMA systems only in discussions or as future work, without presenting experimental results (exclusion criterion 8). After the third phase, we identified 41 primary studies directly relevant to our research focus. The search process was conducted on November 14th, 2024.

✓ *The paper must be written in English.*
✓ *The paper must have an accessible full text.*
✓ *The paper must adopt LMA techniques to solve SE-related tasks.*
✗ *The paper has less than five pages.*
✗ *Duplicate papers or similar studies authored by the same authors.*
✗ *Books, keynote records, panel summaries, technical reports, theses, tool demos papers, editorials*
✗ *The paper is a literature review or survey.*
✗ *The paper does not utilize LMA systems, e.g., using a single LLM agent.*
✗ *The paper is published before November 2022 (the release date of ChatGPT).*

✗ *The paper does not involve SE-related tasks.*
✗ *The paper lacks experimental results and mentions LMA systems only in future work or discussions.*

Snowballing Search. To expand our review, we conducted both backward and forward snowballing [143] on the relevant papers identified in previous steps. This process involved examining the references cited by the relevant studies as well as publications that have cited these studies. We repeated the snowballing process until reaching a transitive closure fixed point, where no new relevant papers were found, resulting in an additional 30 papers being identified.

3.1 Requirements Engineering

Requirements Engineering [91, 129] focuses on defining and managing software system requirements. This discipline is divided into several key stages to ensure requirements meet quality standards and align with stakeholder needs. These stages include elicitation, modeling, specification, analysis, and validation [25, 46].

Elicitron [9] is an LMA framework that focuses specifically on the elicitation stage. It utilizes LLM-based agents to represent a diverse array of simulated users. These agents engage in simulated product interactions, providing insights into user needs by articulating their actions, observations, and challenges. MARE [59] is an LMA framework that covers multiple phases of requirements engineering, including elicitation, modeling, verification, and specification. It employs five distinct agents, i.e., stakeholder, collector, modeler, checker, and documenter, performing nine actions to help generate high-quality requirements models and specifications. Sami et al. [115] propose another LMA framework to generate, evaluate, and prioritize user stories through a collaborative process involving four agents: product owner, developer, QA, and manager. The produce owner generates user stories and initiates prioritization. The QA agent assesses story quality and identifies risks, while the developer prioritizes based on technical feasibility. Finally, the manager synthesizes these inputs and finalizes prioritization after discussions with all agents

3.2 Code Generation

Code generation [15, 45] has consistently been a longstanding focus of SE research, aiming to automate coding tasks to boost productivity and minimize human error.

A prominent multi-agent setup for code generation typically on role specialization and iterative feedback loops to optimize collaboration among agents. We summarize the common roles identified in the literature, including the *Orchestrator, Programmer, Reviewer, Tester, and Information Retriever*.

The *Orchestrator* acts as the central coordinator, managing high-level planning and ensuring smooth task execution across all agents. Its responsibilities include defining high-level strategic goals, breaking them into actionable sub-tasks, delegating these tasks to the appropriate agents, monitoring progress, and ensuring that workflows align with overall project objectives [17, 56, 60, 75, 76, 106, 155, 158]. For instance, PairCoder [158] features a Navigator agent that interprets natural language descriptions to create high-level plans outlining solutions and key implementation steps. The Driver agent then follows these plans to handle code generation and refinement. The Self-Organized Agents framework [56] employs a hierarchical design, with Mother agents managing high-level abstractions and delegating subtasks to specialized Child agents. In CODES [155], the Orchestrator role is performed by the RepoSketcher, which converts high-level natural language requirements into a repository sketch. This sketch outlines the project structure, including directories, files, and inter-file dependencies. The RepoSketcher then delegates tasks to the FileSketcher and SketchFiller, ensuring the efficient and seamless creation of a complete, functional code repository.

During the implementation phase, the process typically begins with the Programmer, who is responsible for writing the initial version of the code. Once the initial code is produced, roles like the Reviewer and Tester step in to evaluate it, providing constructive feedback on quality, functionality, and adherence to requirements. This feedback initiates an iterative cycle, where the Programmer refines the code or the Debugger resolves identified issues, ensuring that the final code meets the desired standards and performs as expected [21, 31, 69, 72, 81, 88, 96, 102, 132]. For example, INTERVENOR [132] pairs a Code Learner with a Code Teacher. The Code Learner generates the initial code and then compiles it to evaluate its correctness. If issues are identified, the Code Teacher analyzes the bug reports and the buggy code, subsequently providing repair instructions to address the errors. Self-repair [102] and TGen [96] refine code by utilizing feedback obtained from running pre-defined test cases.

When predefined test cases are unavailable, the Tester can generate a variety of test cases, ranging from common scenarios to edge cases. These tests help uncover subtle issues that might otherwise go unnoticed and provide actionable feedback to guide subsequent refinement iterations [53, 54, 56, 117].

Some frameworks employ the Information Retriever to gather relevant information to assist code generation. For instance, Agent4PLC [87] and MapCoder [57] incorporate a Retrieval Agent tasked with sourcing examples of similar problems and extracting related knowledge. This agent provides essential contextual information and references tailored to the user's input, ensuring that solutions are well-informed and adhere to domain-specific best practices. Similarly, CodexGraph [85] employs a translation agent to facilitate interaction with graph databases, which are built using static analysis to extract code symbols and their relationships. By converting user queries into graph query language, this agent enables precise and structured information retrieval, enhancing the capability of LLM-based agents to navigate and utilize code repositories effectively.

Agent Forest [77] adopts a different paradigm instead of role specialization. Instead, it utilizes a sampling-and-voting framework, where multiple agents independently generate candidate outputs. Each output is then evaluated based on its similarity to the others, with a cumulative similarity score calculated for each. The output with the highest score—indicating the greatest consensus among the agents—is selected as the final solution.

3.3 Software QA

In this subsection, we review related work on testing, vulnerability detection, bug detection, and fault localization, with a focus on how LMA systems are being employed to enhance software QA processes.

Testing. Fuzz4All [149] generates testing input for software systems across multiple programming languages. In this framework, a distillation agent reduces user input while a generation agent creates and mutates inputs. AXNav [123] is designed to automate accessibility testing. It interprets natural language test instructions and executes accessibility tests, such as VoiceOver, on iOS devices. AXNav includes a planner agent, an action agent, and an evaluation agent. WhiteFox [150] is a fuzzing framework that tests compiler optimizations. It uses two LLM-based agents: one extracts requirements from source code, and the other generates test programs. Additionally, LMA systems are employed for tasks such as penetration testing [29], user acceptance testing [139], and **Graphical User Interface (GUI)** testing [154].

Vulnerability Detection. GPTLens [51] is an LMA framework for detecting vulnerabilities in smart contracts. The system includes LLM-based agents acting as auditors, each independently identifying vulnerabilities. A critic agent then reviews and ranks these vulnerabilities, filtering out false positives and prioritizing the most critical ones. MuCoLD [94] assigns roles like tester and developer to evaluate code. Through discussions and iterative assessments, the agents reach

a consensus on vulnerability classification. Widyasari et al. [142] introduce a cross-validation technique, where multiple LLM's answer is validated against each other.

Bug Detection. **Intelligent Code Analysis Agent (ICAA)** [35] is used for bug detection in static code analysis. The agents have access to tools like web search, static analysis, and code retrieval tools. A Report Agent generates bug reports, while a False-Positive Pruner Agent refines these reports to reduce false positives. Additionally, ICAA includes Code-Intention Consistency Checking, which ensures the code aligns with the developer's intended functionality by analyzing code comments, documentation, and variable names.

Fault Localization. RCAgent [138] performs root cause analysis in cloud environments by using LLM-based agents to collect system data, analyze logs, and diagnose issues. AgentFL [110] breaks down fault localization into three phases. The Comprehension Agent identifies potential fault areas, the Navigation Agent narrows down the codebase search, and the Confirmation Agent uses debugging tools to validate the faults.

3.4 Software Maintenance

In this subsection, we explore related work on debugging and code review, highlighting how LMA systems contribute to automating and improving software maintenance processes.

Debugging. Debugging involves identifying, locating, and resolving software bugs. Several frameworks, including MASAI [8], MarsCode [86], AutoSD [64], and others [19, 73, 92, 125], follow a structured process consisting of stages like bug reproduction, fault localization, patch generation, and validation. Specialized agents are typically responsible for each stage. FixAgent [70] includes a debugging agent and a program repair agent that work together to iteratively fix code by analyzing both errors and repairs. The system refines fault localization by incorporating repair feedback. The agents also articulate their thought processes, improving context-aware debugging. The MASTER framework [152] employs three specialized agents. The Code Quizzer generates quiz-like questions from buggy code, the Learner proposes solutions, and the Teacher reviews and refines the Learner's responses. AutoCodeOver [165] uses an agent for fault localization *via* spectrum-based methods, collaborating with others to refine patches using program representations like abstract syntax trees. SpecRover [113] extends AutoCodeOver by improving program fixes through iterative searches and specification analysis based on inferred code intent. ACFIX [160] targets access control vulnerabilities in smart contracts, focusing on **Role-Based Access Control (RBAC)**. It mines common RBAC patterns from over 344,000 contracts to guide agents in generating patches. DEI [159] resolves GitHub issues by using a meta-policy to select the best solution, integrating and re-ranking patches generated by different agents for improved issue resolution. SWE-Search [7] consists of three agents: the SWE-Agent for adaptive exploration, the Value Agent paired with a Monte Carlo tree search module for iterative feedback and utility estimation, and the Discriminator Agent for collaborative decision-making through debate. RepoUnderstander [92] constructs a knowledge graph for a full software repository and also uses Monte Carlo tree search to assist in understanding complex dependencies.

Code Review. Rasheed et al. [111] developed an automated code review system that identifies bugs, detects code smells, and provides optimization suggestions to improve code quality and support developer education. This system uses four specialized agents focused on code review, bug detection, code smells, and optimization. Similarly, CodeAgent [124] performs code reviews with sub-tasks such as vulnerability detection, consistency checking, and format verification. A supervisory agent, QA-Checker, ensures the relevance and coherence of interactions between agents during the review process.

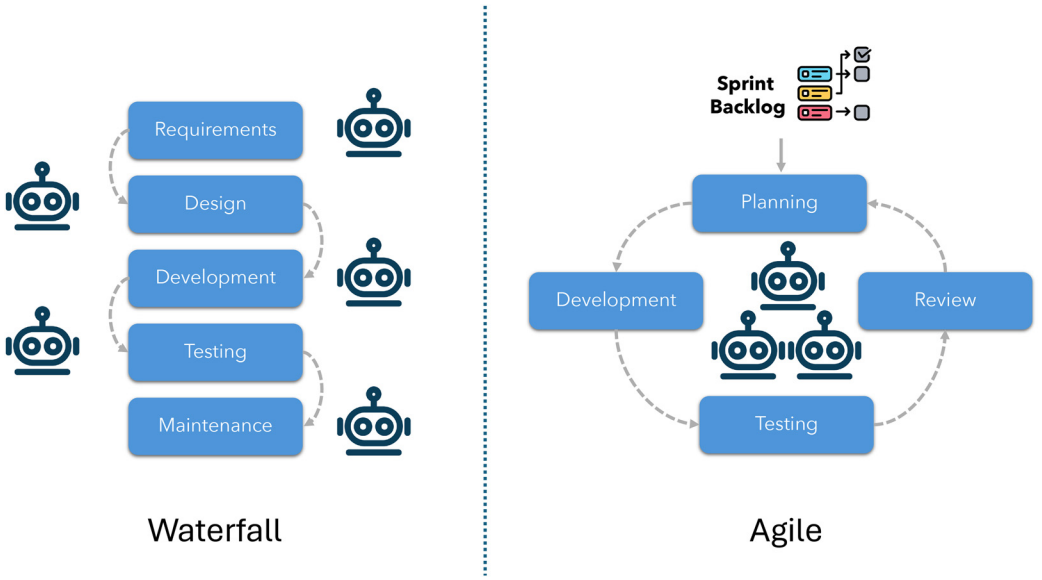

Fig. 1. Multi-agent systems in software development: Waterfall vs. Agile models.

Test Case Maintenance. Lemner et al. [74] propose two multi-agent architectures to predict which test cases need maintenance after source code changes. These agents perform tasks including summarizing code changes, identifying maintenance triggers, and localizing relevant test cases.

3.5 End-to-End Software Development

End-to-end software development encompasses the entire process of creating a software product. While conventional code generation is often limited to producing isolated components such as functions, classes, or modules, end-to-end development starts from high-level software requirements and progresses through design, implementation, testing, and ultimately delivering a fully functional and ready-to-use product.

In practice, developers and stakeholders typically adopt established software process models to guide collaboration, such as Agile [26] and Waterfall [105]. Similarly, the design of LMA systems for end-to-end software development draws inspiration from these software process models. The development process is organized into distinct phases, such as requirements gathering, software design, implementation, and testing. Each phase is managed by specialized agents with domain expertise. Figure 1 illustrates the two software process models.

It is important to note that works such as FlowGen [81] and Self-Collaboration [31] emulate various software process models. However, their experiments focus on generating code segments rather than delivering fully developed software products. As a result, in this article, these approaches are not considered to be designed for true end-to-end software development.

Several works [33, 47, 48, 109, 112, 114, 155, 162] adopt the Waterfall model to automate software development. The Waterfall model used in these multi-agent methods organizes the software development process into distinct, sequential phases, where each stage must be completed before proceeding to the next. The primary phases typically include Requirement Analysis, Architecture Design, Code Development, Testing, and Maintenance. For instance, in MetaGPT [48], the Product Manager agent thoroughly analyzes user requirements. The Architect agent then transforms these requirements into detailed system design components. Subsequently, the Engineer implements

the specified classes and functions as outlined in the design. Finally, the QA Engineer creates and executes test cases to ensure rigorous code quality standards are met. These approaches emphasize a linear and sequential design process, ensuring structured progression and clear accountability at each stage.

AgileCoder [101] and AgileGen [163] adopt Agile process models for software development, emphasizing iterative development by breaking complex tasks into small, manageable increments. AgileCoder [101] assigns Agile roles such as Product Manager and Scrum Master to facilitate sprint-based collaboration and development cycles. AgileGen enhances Agile practices with human-AI collaboration, integrating close user involvement to ensure alignment between requirements and generated code. A notable feature of AgileGen is its use of the Gherkin language to create testable requirements, bridging the gap between user needs and code implementation.

While most methods rely on predefined roles and fixed workflows for software development, a few work [78, 82, 135] investing in dynamic process models. **Think-on-Process (ToP)** [82] introduces a dynamic process generation framework. Since software development processes can vary significantly depending on project requirements, ToP moves beyond the limitations of static, one-size-fits-all workflows to enable more flexible and efficient development practices. Given a software requirement, this framework leverages LLMs to create tailored process instances based on their knowledge of software development. These instances act as blueprints to guide the architecture of the LMA system, adapting to the specific and diverse needs of different projects. Similarly, in MegaAgent [135], agent roles and tasks are not predefined but are generated and planned dynamically based on project requirements. Both ToP and MegaAgent highlight the shift from rigid, static workflows to dynamic, adaptive systems. These frameworks promise more efficient, flexible, and context-aware software development practices, aligning processes with project-specific requirements and complexities.

Additionally, instead of focusing on the process model, several works [107, 108] explore leveraging experiences from past software projects to enhance new software development efforts. Co-Learning [107] enhances agents' software development abilities by utilizing insights gathered from historical communications. This framework fosters cooperative learning between two agent roles—instructor and assistant—by extracting and applying heuristics from their task execution histories. Building on this, Qian et al. [108] propose an iterative experience refinement framework that enables agents to continuously adapt by acquiring, utilizing, and selectively refining experiences from previous tasks, improving agents' effectiveness and collaboration in dynamic software development scenarios.

4 Case Study

To demonstrate the practical effectiveness of LMA systems, we conduct two case studies. Specifically, we utilize the state-of-the-art LMA framework, ChatDev [109], to autonomously develop two classic games: Snake and Tetris. ChatDev structures the software development process into three phases: designing, coding, and testing. ChatDev employs specialized roles, including CEO, CTO, programmer, reviewer, and tester. ChatDev's agents are powered by GPT-3.5-turbo.[4] The temperature setting controls the randomness and creativity of the GPT-3.5's responses. Following the original ChatDev setting, we set the temperature of GPT-3.5-turbo as 0.2.

4.1 Snake Game

For the Snake game, we provide the following prompt to ChatDev to generate the game:

[4]https://platform.openai.com/docs/models/gp#gpt-3-5-turbo

Fig. 2. Screen shots of the Snake game generated by ChatDev.

Snake Game Prompt

"*Design and implement a grid-based snake game displayed on the screen. Initialize the snake with a defined starting position, length, and direction. Enable continuous movement controlled by arrow keys. Introduce food that spawns randomly on the grid, ensuring it does not overlap with the snake. Trigger snake growth when food is consumed, adding a new segment to its body. Implement a game-over condition for boundary or body collisions, displaying a message and providing a restart option. Include a scoring system displayed in the user interface, along with clear instructions.*"

While the first attempt to generate the Snake game was unsuccessful, we resubmitted the same prompt to ChatDev, and the second attempt successfully produced a playable version. ChatDev also generated a detailed manual that included information on dependencies, step-by-step instructions for running the game, and an overview of its features. Figure 2 displays the GUIs of the generated Snake game, showing the starting state, in-game state, and game-over state. The development process was consistently efficient, taking an average of 76 seconds and costing $0.019. Upon playing the game, we confirmed that it fulfilled all the requirements outlined in the prompt.

4.2 Tetris Game

We present the following prompt to ChatDev to guide the generation of the Tetris game:

Tetris Game Prompt

"*Design and implement a Tetris game. Start with a randomly chosen piece dropping from the top. Allow players to control the tetromino using arrow keys for movement (left, right, down) and rotation. Enable automatic downward movement with an adjustable speed. Handle collisions with the boundaries and existing pieces, locking the tetromino in place when it cannot move further. Check for complete rows after each placement and remove them. End the game if new tetrominoes cannot spawn due to a full board, displaying a game over message.*"

During development, ChatDev faced challenges in producing functional gameplay across the first nine attempts. Notice that the same prompt was used for each run. On the tenth attempt, ChatDev successfully produced a Tetris game that met most of the prompt requirements, as shown in Figure 3. The figure illustrates the game's key states: the starting state, in-game states, and game-over state. However, the game still lacks the core functionality to remove completed rows, as demonstrated in the third subplot of Figure 3. Overall, the development process remained efficient, with an average time of 70 seconds and a cost of $0.020 per attempt.

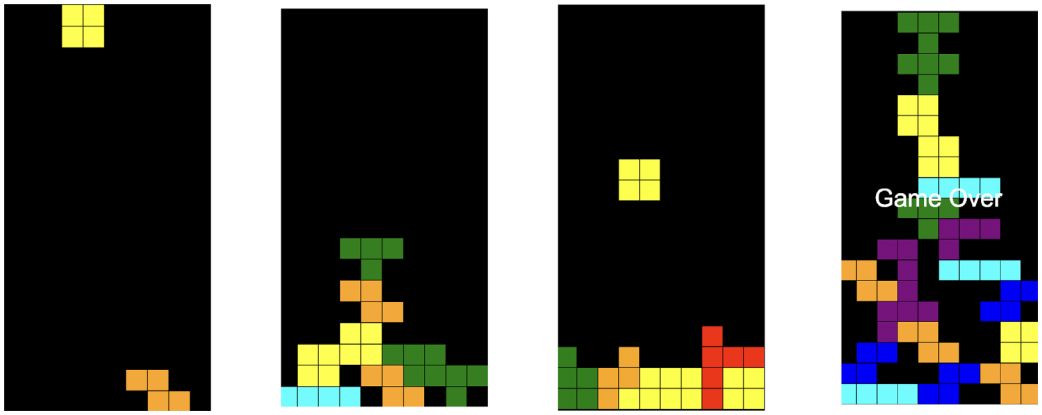

Fig. 3. Screen shots of the Tetris game generated by ChatDev.

Summary of Findings. From our case studies, current LMA systems demonstrate strong performance in reasonably complex tasks like developing a Snake game. The generated Snake game meets all requirements in the prompt within just a few iterations. The process was efficient and cost-effective, with an average completion time of 76 seconds and a cost of $0.019 per attempt. These results emphasize the suitability of LMA systems for moderately complex SE tasks. However, when tasked with more complex challenges like developing a Tetris game, ChatDev successfully generates a playable Tetris game only by the 10th attempt. The game still lacks the core functionality, i.e., removing completed rows. This highlights the limitations of current LMA systems in handling more complex tasks that require deeper logical reasoning and abstraction. Nevertheless, development remains efficient and cost-effective, averaging 70 seconds and $0.020 per run, making the system a promising tool for rapid prototyping.

5 Research Agenda

Previous research has laid the groundwork for the exploration of LMA systems in SE, yet this domain remains in its nascent stages, with many critical challenges awaiting resolution. In this section, we outline our perspective on these challenges and suggest research questions that could advance this burgeoning field. As illustrated in Figure 4, we envision two phases for the development of LMA systems in SE. We discuss each of these phases below and suggest a series of research questions that could form the basis of future research projects.

5.1 Phase 1: Enhancing Individual Agent Capabilities

Indeed, the effectiveness of an LMA system is closely linked to the capabilities of its individual agents. This first phase is dedicated to improving these agents' skills, with a particular focus on adaptability and the acquisition of specialized skills in SE. The potential of individual LLM-based agents in SE is further explored through our initial research questions:

(1) *What SE roles are suitable for LLM-based agents to play and how can their abilities be enhanced to represent these roles?*
(2) *How to design an effective, flexible, and robust prompting language that enhances LLM-based agents' capabilities?*

5.1.1 Refining Role-Playing Capabilities in SE. The role-playing capabilities of LLM-based agents are pivotal within LMA systems [140]. To address the complexity of SE tasks, we need specialized

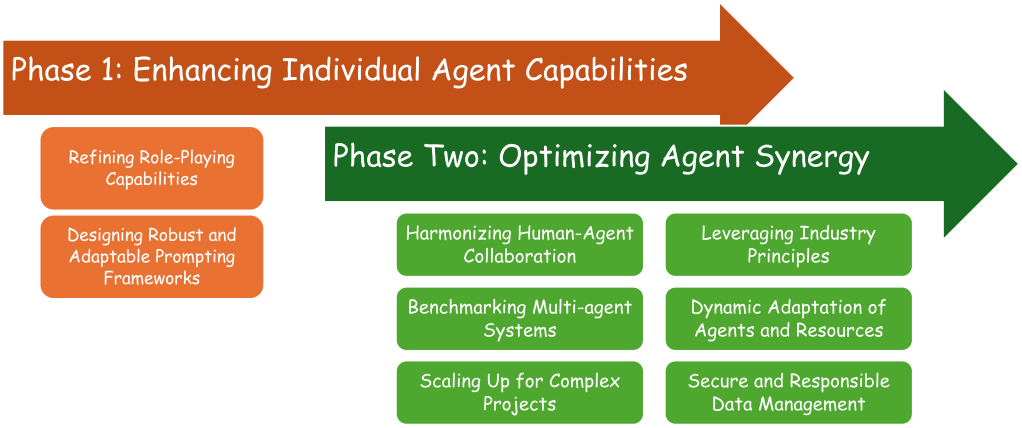

Fig. 4. Research agenda for LLM-based multi-agent systems in SE.

agents capable of adopting diverse roles to tackle intricate challenges throughout the software development lifecycle.

Current State. Existing LMA systems, such as ChatDev [109], MetaGPT [48], and AgileCoder [101], effectively simulate roles like generic software developers and product managers. The agents in these systems rely on general-purpose LLMs such as ChatGPT. Although LLMs like ChatGPT exhibit strong programming skills, they still lack the nuanced expertise required in SE [50]. This limitation hampers their ability to simulate other SE-specific roles. For example, roles involving vulnerability detection or security auditing require a deep understanding of security protocols, threat modeling, and the latest vulnerabilities. However, multiple studies have identified deficiencies in ChatGPT's ability to accurately detect and repair vulnerabilities [18, 37, 120]. This shortcoming underscores the need to integrate domain-specific expertise into LLMs to better support specialized SE roles.

Opportunities. To address this limitation, we propose a structured and actionable three-step approach encompassing the identification, assessment, and enhancement of role-playing abilities, which are:

Step 1: Identifying and prioritizing key SE roles.
Step 2: Assessing LLM-based agents' competencies against role requirements.
Step 3: Enhancing role-playing abilities through targeted training.

The first step focuses on identifying key SE roles, prioritizing those with high industry demand and the potential to substantially boost productivity. This involves:

(1) *Market Analysis*: To begin, we embark on a comprehensive market analysis. It is crucial to assess not only the current trends and needs within the SE sector but also to anticipate future shifts influenced by the integration of LLM-based agents. This analysis involves leveraging various resources such as market reports, job postings, industry forecasts, and technology trend analyses. Platforms like LinkedIn Talent Insights,[5] Gartner reports,[6] and Stack Overflow Developer Surveys[7] may also offer valuable data to inform this assessment. The focus should be on identifying roles that are in high demand and demonstrate rapid growth, especially those

[5] https://www.linkedin.com/products/linkedin-talent-insights/
[6] https://www.gartner.com/en/products/special-reports
[7] https://survey.stackoverflow.co/2024/

requiring specialized skills not typically found among generalist developers. For example, machine learning engineers or cloud architects. Additionally, positions where LLM-based agents could significantly enhance productivity, reduce costs, or accelerate innovation should be evaluated. A key component of this analysis should be determining whether the market has already begun shifting away from recruiting humans for tasks that LLM-based agents can perform, such as routine coding or simple bug fixing. Identifying these trends will help distinguish between roles that are still in demand and those where LLMs have reduced the need for human expertise.

(2) *Stakeholder Engagement*: Engaging comprehensively with a diverse group of stakeholders is essential. This process validates the findings from the market analysis and ensures that the selected roles align with real-world needs. It involves consulting industry professionals who have hands-on experience in the identified roles. This engagement can provide practical insights and challenges associated with these positions. Collaboration with HR departments from leading technology companies is also important. It helps gather perspectives on current hiring trends, skill shortages, and the most sought-after competencies. Additionally, academic experts and researchers can offer forward-thinking views on emerging technologies and methodologies. By incorporating feedback from these various sources, the selection of key roles becomes more robust to reflect both current industry demands and future directions.

(3) *Value Addition Modeling*: The next crucial step is value addition modeling [100], which evaluates the potential advantages that LLM-based agents could bring to each prioritized role. This process involves constructing detailed, data-driven models to analyze key performance indicators such as efficiency improvements, cost reductions, quality enhancements, and the acceleration of innovation resulting from the integration of agents. Pilot projects can be deployed to gather empirical data on these metrics when LLM-based agents are applied to specific tasks. Important factors to consider include the automation of repetitive tasks, the augmentation of human capabilities, and the inclusion of new functionalities that were previously unattainable. It is important to note that the value added by LLM-based agents can differ significantly across different domains; for example, roles in software development may prioritize automation, whereas domains like systems architecture might see more value in LLMs augmenting complex decision-making around resource allocation or performance optimization, where human expertise and contextual understanding remain essential. By quantifying these value propositions, organizations can allocate resources more strategically to roles where LLM-based agents are likely to yield the highest return on investment.

The second step involves understanding the limitations of LLM-based agents relative to the demands of the identified SE roles:

(1) *Competency Mapping*: Competency mapping [66] entails developing comprehensive competency frameworks for each specialized role. These frameworks define the essential skills, knowledge areas, and competencies required, encompassing both technical and soft skills. For instance, technical skills might encompass proficiency in specific programming languages, tools, methodologies, and domain-specific knowledge. For a machine learning engineer, this would include expertise in algorithms, data preprocessing, model training, and tools such as TensorFlow[8] or PyTorch.[9] Soft skills include skills like problem-solving, critical thinking, and collaboration. Clearly outlining these competencies creates a benchmark against which the agents' abilities can be measured.

[8] https://www.tensorflow.org/
[9] https://pytorch.org/

(2) *Performance Evaluation*: The next phase is performance evaluation, which involves designing or selecting tasks that closely replicate the real-world challenges associated with each role. These tasks should be practical and scenario-based to accurately gauge the agents' capabilities. They should assess a wide range of competencies, from technical execution to critical thinking. For example, in evaluating a DevOps engineer, the agent might be tasked with automating a deployment pipeline using tools like Jenkins[10] or Docker,[11] or troubleshooting a continuous integration failure. Such tasks allow for a thorough assessment of both technical and soft skills.
(3) *Gap Analysis*: This step compares the agents' outputs with the expected outcomes for each task. Key areas where the agents underperform—such as misunderstanding domain-specific terminology, neglecting security best practices, or failing to optimize code—are identified and documented. This analysis emphasizes both the agents' strengths and weaknesses, offering valuable insights into recurring patterns of errors or misconceptions.
(4) *Expert Consultation and Iterative Refinement*: To further refine the evaluation process, expert consultation and iterative refinement are essential. By engaging with SE professionals who specialize in the assessed roles, qualitative feedback on the agent's performance can be obtained. These experts provide insights into subtle nuances that may not be captured through quantitative metrics. For instance, while the agent's code may work, it might not follow best practices or address scalability. This feedback helps refine evaluation methods, update competency frameworks, and uncover deeper issues in the agent's understanding.

The final step involves tailoring the LLM-based agents to effectively represent the identified SE roles through specialized training and prompt engineering:

(1) *Curating Specialized Training Data*: At first, this involves creating training datasets that reflect the unique requirements of each specific role. A comprehensive corpus should be built from a variety of sources, including technical documentation such as API guides, technical manuals, and user guides to provide in-depth knowledge of specific technologies. It is also important to incorporate academic and industry research papers, case studies, and whitepapers to capture the latest developments, best practices, and theoretical foundations. Additionally, discussions from forums and software Q&A sites like Stack Overflow,[12] Reddit,[13] and specialized industry forums can provide practical problem-solving approaches and real-world challenges faced by professionals.
(2) *Fine-Tuning the LLM*: After preparing the data, the curated datasets are used to fine-tune the LLM-based agents. Advanced techniques like parameter-efficient fine-tuning [83] are often employed to optimize both efficiency and accuracy.
(3) *Designing Customized Prompts*: A key step is designing prompts tailored to improve the agents' role adaptability. These prompts should clearly define the role, tasks, and goals to ensure the agent understands the requirements. For instance, in a cybersecurity analyst role, the prompt should outline specific security protocols, potential vulnerabilities, and compliance standards. Contextual instructions, including relevant background, constraints, and examples, help the agent grasp task nuances. Creating a library of effective prompts for various scenarios can also serve as reusable templates for future tasks.
(4) *Continuous Learning and Adaptation*: To keep agents aligned with industry developments, continuous adaptation mechanisms are essential. Training data should be regularly updated,

[10] https://www.jenkins.io/
[11] https://www.docker.com/
[12] https://stackoverflow.com/
[13] https://www.reddit.com/

and models may be retrained to incorporate new technologies, best practices, and trends in SE. Monitoring systems can track agent performance over time, enabling proactive adjustments and continuous improvement. Additionally, agents should be guided to consistently reference the latest documentation and standards to ensure their outputs remain relevant and accurate.

While LMA roles may overlap with traditional SE roles, it is important to recognize that they are not necessarily the same, as LMA roles often involve specialized, collaborative tasks suited for agent-based systems. By systematically identifying key roles, assessing agent competencies, and enhancing their capabilities through targeted fine-tuning, we aim to significantly improve the effectiveness of LLM-based agents in specialized SE roles.

5.1.2 Advancing Prompts through Agent-Oriented Programming Paradigms. Effective prompts are crucial for the performance of LLM-based agents. However, creating such prompts is challenging due to the need for a framework that is versatile, effective, and robust across diverse scenarios. Natural language, while flexible, often contains ambiguities and inconsistencies that LLMs may misinterpret. Natural language is inherently designed for human communication, where human communication relies on shared context and intuition that LLMs lack. In contrast, LLMs interpret text based on statistical patterns from large datasets, which may lead to different interpretations than those intended for humans [121, 156]. This highlights the need for a specialized prompting language designed to augment the cognitive functions of LLM-based agents and treats LLMs as the primary audience. Such a language can minimize ambiguities and ensure clear instructions, resulting in more reliable and accurate outputs.

Current State. Multiple prompting frameworks are released to facilitate the usage of LLMs. For example, DSPy [67] and Vieira [79] enable fully automated generation of prompts. AutoGen [145] and LangChain [97] support retrieval-augmented generation [38] and agent-based workflows. However, these frameworks are still human-centered. They often prioritize human readability and developer convenience. As a result, there is a lack of research on a language that treats LLMs as the primary audience for prompts.

Opportunities. **Agent-Oriented Programming (AOP)** [118] offers a promising foundation for this approach. Similar to how Object-Oriented Programming [141] organize objects, AOP treats agents as fundamental units, focusing on their reasoning, objectives, and interactions. An AOP-based prompting language could enable the precise expression of complex tasks and constraints, allowing LLM-based agents to perform their roles with greater efficiency and accuracy. Extending this concept to Multi-Agent-Oriented Programming [13, 14] allows for the creation of systems where multiple LLM-based agents can collaborate, communicate, and adapt to evolving contexts. By explicitly defining agent behaviors, communication patterns, and task hierarchies, we can reduce ambiguity, mitigate hallucinations, and improve task execution in LMA systems.

Furthermore, such a prompting language must be expressive enough to handle diverse and complex tasks, yet simple enough for users to easily adopt. Conversely, overly simplified languages may lack the expressive power needed to represent complex SE workflows. A complex language that introduces a steep learning curve due to their syntax, hinder adoption, especially for users who require simpler interfaces for prompt creation and modification. Balancing functionality and usability will be another key research question to its success.

Additionally, this process may involve tailoring prompts specifically for different LLM models and their versions, as variations in model architectures, training data, and capabilities can affect how they interpret and respond to prompts. What works effectively for one model may not perform as well for another, necessitating careful adjustments. Current prompting languages lack mechanisms to easily adapt prompts across models, requiring manual adjustments and experimentation to achieve consistent performance.

While AOP-based prompting may not be the final solution, it represents an important step toward developing an AI-oriented language with grammar tailored specifically for LLMs. This new approach could further refine communication with LLM-based agents, reducing misinterpretation and significantly enhancing overall performance.

5.2 Phase Two: Optimizing Agent Synergy

In Phase Two, the spotlight turns toward optimizing agent synergy, underscoring the importance of collaboration and how to leverage the diverse strengths of individual agents. This phase delves into both the internal dynamics among agents and the role of external human intervention in enhancing the efficacy of the LMA system. Key research questions guiding this phase include:

(1) *How to best allocate tasks between humans and LLM-based agents?*
(2) *How can we quantify the impact of agent collaboration on overall task performance and outcome quality?*
(3) *How to scale LMA systems for large-scale projects?*
(4) *What industrial organization mechanisms can be applied to LMA systems?*
(5) *What strategies allow LMA systems to dynamically adjust their approach?*
(6) *How to ensure security among private data sharing within LMA systems?*

5.2.1 Human Agent Collaboration. Optimally distributing tasks between humans and LMA agents to leverage their respective strengths is essential. Humans bring unparalleled creativity, critical thinking, ethical judgment, and domain-specific knowledge [95]. In contrast, LLM-based agents excel at rapidly processing large datasets, performing repetitive tasks with high accuracy, and detecting patterns that might elude human observers.

Current State. Several LMA systems incorporate human-in-the-loop designs. For instance, AISD [162] involves human input during requirement analysis and system validation, where users provide feedback on use cases, system designs, and prototypes. Similarly, MARE [59] leverages human assessment to refine generated requirements and specifications. Although these works demonstrate the feasibility of human contributions, key research questions, including optimizing human roles, enhancing feedback mechanisms, and identifying appropriate intervention points, are still underexplored.

Opportunity. Developing role-specific guidelines that outline when and how human intervention should occur is essential. These guidelines should assist in identifying critical decision points where human judgment is indispensable, such as ethical considerations, conflict resolution, ambiguity handling, and creative problem-solving. For example, ethical decisions necessitate human oversight to ensure alignment with societal norms and values, and conflict resolution may require negotiation skills that LLM-based agents lack.

To facilitate seamless collaboration, designing intuitive user-friendly interfaces and interaction protocols is essential [133]. Natural language interfaces and adaptive visualization techniques can make interactions more accessible. These interfaces should efficiently present agent outputs in a digestible format and collect user feedback, while also managing the cognitive load on human collaborators. It is important to note that these interfaces may need to be tailored differently for each human role, as the needs of a project manager, a software developer, and a QA engineer will vary significantly.

Given the complexity of information generated during the agents' workflows [60], designing such interfaces poses challenges. For instance, presenting modifications suggested by an agent at varying levels of abstraction ensures that each stakeholder can engage with the information at the right depth. A project manager might focus on the broader implications, such as the high-level impact on project timelines or deliverables, whereas a developer or architect might drill down into

specific implementation details. Role-specific interfaces will be key to ensuring each stakeholder can effectively collaborate with the agents and extract the necessary information in a manner suited to their specific responsibilities.

Additionally, developing predictive models to determine the optimal human-to-agent ratio across different project types and stages is a fundamental concern. These models must assess factors such as project complexity, time constraints, project priorities, and the specific capabilities and limitations of both human participants and LMA agents. By doing so, tasks can be allocated in a manner that fully harnesses both human ingenuity and agent efficiency throughout the project. Machine learning techniques could also be leveraged to analyze historical project data to predict effective collaboration strategies.

5.2.2 Evaluating the LMA Systems. Current State. Numerous complex benchmarks have been proposed to challenge the capabilities of LLMs in critical aspects of software development, such as code generation [58, 84, 171]. While these benchmarks have advanced the field by providing measurable metrics for individual tasks, their focus on isolated problem-solving reveals limitations as SE projects become more complex. SE is inherently collaborative, with key activities like joint requirements gathering, code integration, and peer reviews playing essential roles in the process. Current benchmarks often overlook these aspects, failing to assess how well LLMs perform in tasks that require cooperation and collective decision-making.

Opportunities. There is a growing need for benchmarks that evaluate the cooperative abilities of LLMs in multi-agent settings, particularly for SE tasks. These benchmarks should simulate real-world collaborative scenarios where LLM agents work together to achieve common development goals.

Such benchmarks should include tasks where agents must:

(1) *Participate in Collaborative Design*: Agents should contribute ideas, propose design solutions, and converge on a unified architecture that balances trade-offs.
(2) *Delegate and Coordinate Tasks*: Effective task division is crucial. Agents should assign responsibilities based on expertise, manage dependencies, and adjust as the project evolves.
(3) *Identify Conflicts and Negotiate*: In collaborative settings, disagreements are inevitable. First, LLMs often struggle to identify conflicts in real time unless explicitly guided to do so [1]. Therefore, agents should be evaluated on their ability to recognize these conflicts—whether in logic, goals, or execution. Moreover, agents should be tested on their ability to handle conflicts constructively. This includes proposing compromises, engaging in constructive negotiation, and ensuring that the team remains aligned with the overarching objectives. Evaluations should focus on the agents' capacity to balance competing priorities, mitigate misunderstandings, and foster consensus, all while maintaining progress toward shared goals.
(4) *Integrate Components and Perform Peer Reviews*: Agents should seamlessly integrate their work, review each other's code for QA, and provide constructive feedback.
(5) *Proactive Clarification Request*: Agents should not assume complete understanding when uncertainty arises. Instead, they should preemptively ask for additional information or clarification to avoid potential errors or misunderstandings. Evaluating agents on this ability ensures they are capable of identifying gaps in their knowledge or instructions and can actively seek out the necessary context or data to complete tasks effectively.

To develop such benchmarks, we need to create realistic project scenarios that require multi-agent collaboration over extended periods. These scenarios should reflect common software development challenges, such as evolving requirements and tight deadlines. Additionally, platforms or sandboxes must be built to provide controlled environments where collaborative interactions between agents

can be observed and measured. These platforms should establish clear interaction rules, including languages, formats, and communication channels, to facilitate effective information exchange.

Most importantly, comprehensive metrics must be developed to assess not just the final output, but also the collaboration process itself. These metrics could measure communication efficiency, ambiguity resolution, conflict management, adherence to best practices, and overall project success.

5.2.3 Scaling Up for Complex Projects. As software projects become more complex, a single LLM-based agent may hit its performance limits. Inspired by the scaling properties of neural models [10], LMA systems can potentially enhance performance by increasing the number of agents within the system. While adding more agents can provide some benefits, handling more complex projects introduces challenges that require more refined solutions.

Current State. Existing LMA systems face significant challenges when scaling up to handle complex software projects. Our case studies illustrate these limitations clearly. For instance, ChatDev was unable to autonomously develop a functional Tetris game. In real-world projects with higher complexity, this limitation becomes even more pronounced.

Opportunities. First, as software projects grow in size and complexity, breaking down high-level requirements into manageable sub-tasks becomes more difficult. It is not just about handling more tasks but also managing the intricate interdependencies between them. A hierarchical task decomposition approach can help, where higher-level agents oversee broader objectives and delegate specific tasks to lower-level agents. This structure streamlines planning and makes global task allocation more efficient.

Second, as the number of agents increases, so does the complexity of communication. Coordinating multiple agents can lead to communication bottlenecks and information overload. Additionally, large-scale software projects challenge the memory capacity of individual agents, making it harder to store and process the extensive information required. Efficient communication protocols and message prioritization are crucial to mitigating these issues. For instance, agents can use summarized updates instead of detailed reports, reducing communication overhead and memory usage. From the outset, the system should be designed with scalability in mind, ensuring that both software and hardware resources can expand efficiently as the number of agents increases.

Moreover, with more agents comes the risk of inconsistencies and conflicts in the shared information. A centralized knowledge repository or shared blackboard system can ensure that all agents have access to consistent, up-to-date information, acting as a single source of truth and minimizing the spread of misinformation. Robust error handling mechanisms should also be implemented to detect and correct issues autonomously before they escalate into significant failures.

Finally, as the number of agents grows, so do the rounds of discussion and decision-making, which can slow down progress. To avoid this, decision-making hierarchies or consensus algorithms can streamline the process. For example, only a subset of agents responsible for a specific module may need to reach a consensus, rather than involving the entire agent network.

5.2.4 Leveraging Industry Principles. As LLM-based agents can closely mimic human developers in SE tasks, they can greatly benefit from adopting established industry principles and management strategies. By emulating organizational frameworks used by successful companies, LMA systems can improve their design and optimization processes. These industrial mechanisms enable LMA systems to remain agile, efficient, and effective, even as project complexities grow.

Current State. As we described in Section 3, numerous works [5, 48, 101, 109] are designed using popular process models like the Waterfall and Agile. For example, ChatDev [109] emulates a traditional Waterfall approach, breaking tasks into distinct phases (e.g., requirement analysis, design, implementation, testing), with agents dedicated to each phase. AgileCoder [101] incorporates the

Agile methodologies, leveraging iterative development, continuous feedback loops, and collaborative sprints.

Opportunities. However, current LMA systems often do not leverage more specialized and modern industry practices, such as Value Stream Mapping, Design Thinking, or Model-Based Systems Engineering. Additionally, frameworks like Domain-Driven Design, Behavior-Driven Development, and Team Topologies remain underutilized. These methodologies emphasize aligning development with business goals, improving user-centric design, and optimizing team structures—key components that could further enhance the efficiency, adaptability, and effectiveness of LMA systems.

Leadership and governance structures from industrial organizations provide valuable insights for designing LMA systems. Project management tools and practices, essential for coordinating large development teams, can be applied to LMA systems to enhance their operational efficiency. Using established project management frameworks, LMA systems can monitor progress, allocate resources, and manage timelines effectively. Agents can dynamically update task boards, report milestones, and adjust workloads in real-time based on project data. This not only improves transparency but also allows for early detection of bottlenecks or delays, ensuring projects stay on track.

Incorporating design patterns and software architecture best practices further strengthens LMA systems [70]. By adhering to these principles, agents can produce well-structured, maintainable code that is scalable and reusable. This reduces technical debt and ensures that the solutions developed by LMA systems are easier to integrate, maintain, and expand in the future.

5.2.5 Dynamic Adaptation. In the context of software development, predicting the optimal configuration for LMA systems at the outset is unrealistic due to the inherent complexity and variability of tasks [71]. The dynamic nature of software requirements and the unpredictable challenges that arise during development necessitate systems that can adapt on the fly [88]. For example, a sudden shift in project requirements or unexpected delays caused by dependencies on external components. Therefore, LMA systems must be capable of dynamically adjusting their scale, strategies, and structures throughout the development process.

Current State. Most existing LMA systems [48, 109] operate with static architectures characterized by fixed agent roles and predefined communication patterns. Recent research efforts [89, 157] have introduced mechanisms for adaptive agent team selection and task-specific collaboration strategies. These methods enable the selection of suitable agent team configurations for specific tasks; however, they still fall short of true dynamic adaptation and lack the capability to adjust to real-time changes. To the best of our knowledge, no previous work addresses the need for on-the-fly adjustments in response to evolving project demands.

Opportunities. To minimize redundant work, LMA systems should continuously evaluate existing solutions [136], identifying reusable elements for new requirements. By learning from each development cycle, the system can recognize patterns of efficiency and inefficiency, enabling it to make informed decisions when handling similar tasks in the future or adapting existing solutions to new requirements.

A key element of dynamic adaptation is the ability to automatically adjust the number of agents involved in a project [41]. This includes not only scaling the number of agents up or down as needed but also generating new agents with new specialized roles to meet emerging task requirements, ensuring both efficiency and responsiveness. Additionally, the system can replicate agents in existing roles to manage increased workloads. Furthermore, LMA systems can generate new agents that come equipped with contextual knowledge of the project—such as its history, current state, and objectives—by accessing shared knowledge bases, project documentation, and recent communications. This allows new agents to integrate smoothly and contribute effectively right from the start, reducing onboarding time and minimizing disruptions.

Another key component is the dynamic redefinition of agent roles [61]. As the project evolves, certain roles may become obsolete while new ones emerge. LMA systems should be capable of reassigning roles to agents or modifying their responsibilities to better align with current project needs. This flexibility enhances the system's ability to adapt to changing requirements and priorities.

Dynamic adaptation also involves the reallocation of memory and computational resources. As agents are added or removed and tasks shift in complexity, the system must efficiently distribute resources to where they are most needed. This may include scaling computational power for agents handling intensive tasks or increasing memory allocation for agents processing large datasets. Effective resource management ensures that the system operates optimally without unnecessary strain on infrastructure.

Finally, the uncertainty of the software development process makes it challenging to define effective termination conditions [119]. Relying solely on predefined criteria may result in infinite loops or premature task completion. To address this, LMA systems must incorporate real-time monitoring and feedback loops to continuously evaluate progress. Machine learning techniques can help predict optimal stopping points by analyzing historical data and current performance metrics, allowing for informed adjustments to task completion criteria as the project evolves.

5.2.6 Privacy and Partial Information. In multi-organizational software development projects, data often reside in silos due to privacy concerns, proprietary restrictions, and regulatory compliance requirements [103]. Each entity may have its own data governance policies and competitive considerations that limit data sharing. This fragmentation poses significant challenges in enabling agents to access necessary information while ensuring that privacy is maintained. Moreover, a lack of transparency in data sources and processes can exacerbate the risk of privacy violations, which may go unnoticed if data handling activities are not fully visible to all parties [27].

Current State. The challenge of ensuring privacy while managing partial information has been extensively studied in the field of computer security. [11, 30]. To the best of our knowledge, existing research has yet to provide a solution to these challenges for LMA systems in SE.

Opportunities. To address these challenges, robust and fine-grained access control mechanisms must be implemented across organizational boundaries. It is essential to prevent unauthorized access while still accommodating the varied data access needs of the system. Traditional models like RBAC [116] and Attribute-Based Access Control [52] may need to be extended to handle the dynamic nature of multi-agent systems effectively. Establishing protocols that allow agents to share insights derived from sensitive data, without exposing the data itself, is critical. Advanced privacy-preserving techniques like Differential Privacy [34], Secure Multi-Party Computation [40], Federated Learning [62], or Homomorphic Encryption [153] can be leveraged to ensure that agents collaborate without compromising data privacy.

Moreover, compliance with data protection laws such as the General Data Protection Regulation [130] in the EU and the California Consumer Privacy Act [104] in the US is crucial. LMA systems should follow privacy-by-design principles, ensuring that data subjects' rights are upheld, and that data processing activities remain transparent and lawful. This includes implementing mechanisms for data minimization, consent management, and honoring the right to be forgotten.

For non-sensitive data, integrated data storage solutions can reduce redundancy, improve data consistency, and increase efficiency. This can be achieved through distributed databases accessible to authorized agents, along with data synchronization mechanisms to ensure agents have up-to-date information in real time. Additionally, using technologies like blockchain [167] and distributed ledgers [122] can enhance transparency, traceability, and tamper-resistance in recording agent transactions and data access events, fostering greater trust among collaborating entities.

6 Discussion

6.1 A Comparison with the Mixture of Experts (MoE) Paradigm

Another paradigm that has recently attracted much attention from both academia and industry is the MoE paradigm [16, 170]. MoE organizes an LLM into multiple specialized components known as "experts." Each expert is designed to focus on specialized tasks. Furthermore, a gating mechanism is employed to dynamically activate the most relevant subset of experts based on the input. While MoE is promising, LMA systems offer several distinct advantages.

One limitation of MoE is its high resource consumption. MoE models contain multiple experts within a single architecture, which makes the total number of parameters rather huge. Furthermore, training MoE is more resource-intensive and time-consuming than standard LLMs. This is mainly due to the complex training process for the gating mechanism. Training the gating mechanism involves optimizing the selection process for the most relevant experts, which adds considerable overhead.

Since specific experts are dynamically activated based on input, MoE can be viewed as a method to learn the internal routing of LLMs. However, there is no interaction and communication between experts in MoE. On the other hand, LMA systems usually are designed to resemble real-world collaborative workflows. Agents in LMA systems can actively communicate with each other, exchange information, and iteratively refine the output based on feedback from other agents. More importantly, LMA systems can also integrate external feedback from tools such as compilers, static analyzers, or testing frameworks. LMA systems also facilitate seamless and continuous human-in-the-loop collaboration, enabling human experts to intervene, validate outputs, and provide guidance at any stage of the process. As a result, we consider LMA systems to be a more appropriate approach to MoE to address the multifaceted challenges of SE.

6.2 Threat to Validity

One potential threat to validity lies in the possibility of inadvertently excluding relevant studies during the literature search and selection process. To mitigate this risk, we conducted a comprehensive search on the DBLP database, ensuring coverage of a broad spectrum of studies, including preprints. Additionally, we enhanced the search process by combining automated querying with forward and backward snowballing, aiming to identify and include all pertinent studies.

7 Conclusion and Future Work

This article explores the evolving role of LMA systems in shaping the future of Software Engineering 2.0 [90]. To support this vision, we first present a systematic review of recent applications of LMA systems across different stages of the software development lifecycle. Our review highlights key advancements in areas such as requirements engineering, code generation, software QA, and maintenance. To further understand the current landscape, we conduct two case studies that illustrate the practical uses and challenges of LMA systems. Based on these insights, we propose a structured research agenda aimed at advancing LMA integration in SE. Future work will focus on addressing critical research questions to enhance LMA capabilities and optimize their synergy with software development processes.

In the immediate term, efforts will be dedicated to enhancing the capabilities of LLM-based agents in representing specialized SE roles. This will involve creating specialized datasets and pre-training tasks that mirror the complex realities of SE tasks. Additionally, there will be a focus on formulating advanced prompting strategies, which can refine the agents' cognitive functions and decision-making skills. Looking toward the longer-term objectives, the emphasis will transition toward optimizing the synergy between agents. The initial step involves examining optimal strategies for

task allocation between humans and LLM-based agents, capitalizing on the unique strengths of both entities. Furthermore, we need to develop scalable methodologies for LMA systems, which would enable them to orchestrate and complete large-scale, multifaceted SE projects efficiently. Moreover, ensuring the privacy and confidentiality of data within LMA systems is also critical. This entails exploring data management and access control mechanisms to protect sensitive information while still enabling the essential exchange of insights among project stakeholders. By systematically exploring these research questions, we aim to drive innovation in LMA systems for SE and create a more cohesive, effective, and flexible LMA-driven development process.

References

[1] Sahar Abdelnabi, Amr Gomaa, Sarath Sivaprasad, Lea Schönherr, and Mario Fritz. 2023. LLM-deliberation: Evaluating LLMs with interactive multi-agent negotiation games. arXiv:2309.17234. Retrieved from https://arxiv.org/abs/2309.17234

[2] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. arXiv:1709.08439. Retrieved from https://arxiv.org/abs/1709.08439

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[4] Saaket Agashe. 2023. *LLM-Coordination: Developing Coordinating Agents with Large Language Models*. University of California, Santa Cruz.

[5] Samar Al-Saqqa, Samer Sawalha, and Hiba AbdelNabi. 2020. Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies* 14, 11 (2020), 246–270.

[6] Stefano V. Albrecht and Peter Stone. 2018. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence* 258 (2018), 66–95.

[7] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. SWE-search: Enhancing software agents with Monte Carlo tree search and iterative refinement. arXiv:2410.20285. Retrieved from https://arxiv.org/abs/2410.20285

[8] Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular architecture for software-engineering AI agents. arXiv:2406.11638. Retrieved from https://arxiv.org/abs/2406.11638

[9] Mohammadmehdi Ataei, Hyunmin Cheong, Daniele Grandi, Ye Wang, Nigel Morris, and Alexander Tessier. 2024. Elicitron: An LLM agent-based simulation framework for design requirements elicitation. arXiv:2404.16045. Retrieved from https://arxiv.org/abs/2404.16045

[10] Yasaman Bahri, Ethan Dyer, Jared Kaplan, Jaehoon Lee, and Utkarsh Sharma. 2024. Explaining neural scaling laws. *Proceedings of the National Academy of Sciences* 121, 27 (2024), e2311878121.

[11] Charles H. Bennett, Gilles Brassard, and Jean-Marc Robert. 1988. Privacy amplification by public discussion. *SIAM Journal on Computing* 17, 2 (1988), 210–229.

[12] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 308–318.

[13] Olivier Boissier, Rafael H. Bordini, Jomi Hubner, and Alessandro Ricci. 2020. *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*. Mit Press.

[14] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. 2009. *Multi-Agent Programming*. Springer.

[15] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. 1996. Automatic code generation from design patterns. *IBM Systems Journal* 35, 2 (1996), 151–171.

[16] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. 2024. A survey on mixture of experts. arXiv:2407.06204. Retrieved from https://arxiv.org/abs/2407.06204

[17] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, et al. 2023. Low-code LLM: Visual programming over LLMs. arXiv:2304.08103. Retrieved from https://arxiv.org/abs/2304.08103

[18] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2024. When ChatGPT meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology* (November 2024). DOI: https://doi.org/10.1145/3702973

[19] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue resolving with multi-agent and task graphs. arXiv:2406.01304. Retrieved from https://arxiv.org/abs/2406.01304

[20] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[21] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *Proceedings of the 12th International Conference on Learning Representations*.

[22] Yongchao Chen, Jacob Arkin, Yang Zhang, Nicholas Roy, and Chuchu Fan. 2023. Scalable multi-robot collaboration with large language models: Centralized or decentralized systems? arXiv:2309.15943. Retrieved from https://arxiv.org/abs/2309.15943

[23] Zhenpeng Chen, Jie M. Zhang, Max Hort, Mark Harman, and Federica Sarro. 2024. Fairness testing: A comprehensive survey and analysis of trends. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–59.

[24] Yuheng Cheng, Ceyao Zhang, Zhengwen Zhang, Xiangrui Meng, Sirui Hong, Wenhao Li, Zihao Wang, Zekai Wang, Feng Yin, Junhua Zhao, et al. 2024. Exploring large language model based intelligent agents: definitions, methods, and prospects. arXiv:2401.03428. Retrieved from https://arxiv.org/abs/2401.03428

[25] Michael G. Christel and Kyo C. Kang. 1992. Issues in requirements elicitation.

[26] David Cohen, Mikael Lindvall, and Patricia Costa. 2004. An introduction to agile methods. *Advances in Computing* 62, 03 (2004), 1–66.

[27] Kate Crawford and Jason Schultz. 2014. Big data and due process: Toward a framework to redress predictive privacy Harms. *Boston College Law Review* 55 (2014), 93.

[28] DBLP computer Science bibliography. 2024. DBLP: Computer Science bibliography. Retrieved November 13, 2024 from https://dblp.org

[29] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2023. Pentestgpt: An llm-empowered automatic penetration testing tool. arXiv:2308.06782. Retrieved from https://arxiv.org/abs/2308.06782

[30] Irit Dinur and Kobbi Nissim. 2003. Revealing information while preserving privacy. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 202–210.

[31] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via ChatGPT. arXiv:2304.07590. Retrieved from https://arxiv.org/abs/2304.07590

[32] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. arXiv:2305.14325. Retrieved from https://arxiv.org/abs/2305.14325

[33] Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, Yifei Wang, Yufan Dang, Weize Chen, and Cheng Yang. 2024. Multi-agent software development through cross-team collaboration. arXiv:2406.08979. Retrieved from https://arxiv.org/abs/2406.08979

[34] Cynthia Dwork. 2006. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*. Springer, 1–12.

[35] Gang Fan, Xiaoheng Xie, Xunjin Zheng, Yinan Liang, and Peng Di. 2023. Static code analysis in the AI era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents. arXiv:2310.08837. Retrieved from https://arxiv.org/abs/2310.08837

[36] Stan Franklin and Art Graesser. 1996. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 21–35.

[37] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *Proceedings of the 2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 632–636.

[38] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. arXiv:2312.10997. Retrieved from https://arxiv.org/abs/2312.10997

[39] Joseph A. Goguen and Charlotte Linde. 1993. Techniques for requirements elicitation. In *Proceedings of the 1993 IEEE International Symposium on Requirements Engineering*. IEEE, 152–164.

[40] Oded Goldreich. 1998. Secure multi-party computation. *Manuscript. Preliminary Version* 78, 110 (1998), 1–108.

[41] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. arXiv:2402.01680. Retrieved from https://arxiv.org/abs/2402.01680

[42] Junda He, Bowen Xu, Zhou Yang, DongGyun Han, Chengran Yang, Jiakun Liu, Zhipeng Zhao, and David Lo. 2024. PTM4Tag+: Tag recommendation of stack overflow posts with pre-trained models. arXiv:2408.02311. Retrieved from https://arxiv.org/abs/2408.02311

[43] Junda He, Bowen Xu, Zhou Yang, DongGyun Han, Chengran Yang, and David Lo. 2022. Ptm4tag: Sharpening tag recommendation of stack overflow posts with pre-trained models. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 1–11.

[44] Junda He, Xin Zhou, Bowen Xu, Ting Zhang, Kisub Kim, Zhou Yang, Ferdian Thung, Ivana Clairine Irsan, and David Lo. 2024. Representation learning for stack overflow posts: How far are we? *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–24.

[45] Jack Herrington. 2003. *Code Generation in Action*. Manning Publications Co.

[46] Ann M. Hickey and Alan M. Davis. 2004. A unified model of requirements elicitation. *Journal of Management Information Systems* 20, 4 (2004), 65–84.

[47] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2023. L2mac: Large language model automatic computer for unbounded code generation. arXiv:2310.02003. Retrieved from https://arxiv.org/abs/2310.02003

[48] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. arXiv:2308.00352. Retrieved from https://arxiv.org/abs/2308.00352

[49] John J. Horton. 2023. *Large Language Models as Simulated Economic Agents: What Can We Learn from Homo Silicus?* Technical Report. National Bureau of Economic Research.

[50] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8, Article 220 (December 2024), 79 pages. DOI: https://doi.org/10.1145/3695988

[51] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *Proceedings of the 2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 297–306.

[52] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. 2015. Attribute-based access control. *Computer* 48, 2 (2015), 85–88.

[53] Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024. Self-evolving multi-agent collaboration networks for software development. arXiv:2410.16946. Retrieved from https://arxiv.org/abs/2410.16946

[54] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv:2312.13010. Retrieved from https://arxiv.org/abs/2312.13010

[55] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–11.

[56] Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization. arXiv:2404.02183. Retrieved from https://arxiv.org/abs/2404.02183

[57] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics* (Volume 1 Long Papers). Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, Bangkok, Thailand, 4912–4944. Retrieved from https://aclanthology.org/2024.acl-long.269

[58] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. arXiv:2403.07974. Retrieved from https://arxiv.org/abs/2403.07974

[59] Dongming Jin, Zhi Jin, Xiaohong Chen, and Chunhui Wang. 2024. MARE: Multi-agents collaboration framework for requirements engineering. arXiv:2405.03256. Retrieved from https://arxiv.org/abs/2405.03256

[60] Martin Josifoski, Lars Klein, Maxime Peyrard, Nicolas Baldwin, Yifei Li, Saibo Geng, Julian Paul Schnitzler, Yuxing Yao, Jiheng Wei, Debjit Paul, et al. 2023. Flows: Building blocks of reasoning and collaborating AI. *a*rXiv:2308.01285. Retrieved from https://arxiv.org/abs/2308.01285

[61] Denis Jouvin and Salima Hassas. 2002. Role delegation as multi-agent oriented dynamic composition. In *Proceedings of Net Object Days (NOD), AgeS Workshop*.

[62] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning* 14, 1–2 (2021), 1–210.

[63] Stephen H. Kan. 2003. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional.

[64] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable automated debugging via large language model-driven scientific debugging. arXiv:2304.02195. Retrieved from https://arxiv.org/abs/2304.02195

[65] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274.

[66] Jaideep Kaur and Vikas Kumar. 2013. Competency mapping: A gap analysis. *International Journal of Education and Research* 1, 1 (2013), 1–9.
[67] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, et al. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. arXiv:2310.03714. Retrieved from https://arxiv.org/abs/2310.03714
[68] Craig Larman. 2004. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional.
[69] Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. INDICT: Code generation with internal dialogues of critiques for both security and helpfulness. arXiv:2407.02518. Retrieved from https://arxiv.org/abs/2407.02518
[70] Cheryl Lee, Chunqiu Steven Xia, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R Lyu. 2024. A unified debugging approach via LLM-based multi-agent synergy. arXiv:2404.17153. Retrieved from https://arxiv.org/abs/2404.17153
[71] Dean Leffingwell and Don Widrig. 2000. *Managing Software Requirements: A Unified Aapproach*. Addison-Wesley Professional.
[72] Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. AutoCoder: Enhancing code large language model with\textsc {AIEV-instruct}. arXiv:2405.14906. Retrieved from https://arxiv.org/abs/2405.14906
[73] Bin Lei, Yuchen Li, Yiming Zeng, Tao Ren, Yi Luo, Tianyu Shi, Zitian Gao, Zeyu Hu, Weitai Kang, and Qiuwu Chen. 2024. Infant agent: A tool-integrated, logic-driven agent with cost-effective API usage. arXiv:2411.01114. Retrieved from https://arxiv.org/abs/2411.01114
[74] Ludvig Lemner, Linnea Wahlgren, Gregory Gay, Nasser Mohammadiha, Jingxiong Liu, and Joakim Wennerberg. 2024. Exploring the integration of large language models in industrial test maintenance processes. arXiv:2409.06416. Retrieved from https://arxiv.org/abs/2409.06416
[75] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2024. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2024), 51991–52008.
[76] Jierui Li, Hung Le, Yinbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. CodeTree: Agent-guided tree search for code generation with large language models. arXiv:2411.04329. Retrieved from https://arxiv.org/abs/2411.04329
[77] Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. 2024. More agents is all you need. arXiv:2402.05120. Retrieved from https://arxiv.org/abs/2402.05120
[78] Yuan Li, Yixuan Zhang, and Lichao Sun. 2023. MetaAgents: Simulating interactions of human behaviors for LLM-based task-oriented coordination via collaborative generative agents. arXiv:2310.06500. Retrieved from https://arxiv.org/abs/2310.06500
[79] Ziyang Li, Jiani Huang, Jason Liu, Felix Zhu, Eric Zhao, William Dodds, Neelay Velingker, Rajeev Alur, and Mayur Naik. 2024. Relational programming with foundational models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 10635–10644.
[80] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. 2023. Encouraging divergent thinking in large language models through multi-agent debate. arXiv:2305.19118. Retrieved from https://arxiv.org/abs/2305.19118
[81] Feng Lin, Dong Jae Kim, et al. 2024. SOEN-101: Code generation by emulating software process models using large language model agents. arXiv:2403.15852. Retrieved from https://arxiv.org/abs/2403.15852
[82] Leilei Lin, Yingming Zhou, Wenlong Chen, and Chen Qian. 2024. Think-on-process: Dynamic process generation for collaborative development of multi-agent system. arXiv:2409.06568. Retrieved from https://arxiv.org/abs/2409.06568
[83] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A. Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.
[84] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024), 21558–21572.
[85] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Wenmeng Zhou, Fei Wang, and Michael Shieh. 2024. CodexGraph: Bridging large language models and code repositories via code graph databases. arXiv:2408.03910. Retrieved from https://arxiv.org/abs/2408.03910
[86] Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024. MarsCode agent: AI-native automated bug fixing. arXiv:2409.00899. Retrieved from https://arxiv.org/abs/2409.00899
[87] Zihan Liu, Ruinan Zeng, Dongxia Wang, Gengyun Peng, Jingyi Wang, Qiang Liu, Peiyu Liu, and Wenhai Wang. 2024. Agents4PLC: Automating closed-loop PLC code generation and verification in industrial control systems using LLM-based agents. arXiv:2410.14209. Retrieved from https://arxiv.org/abs/2410.14209

[88] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2023. Dynamic LLM-agent network: An LLM-agent collaboration framework with agent team optimization. arXiv:2310.02170. Retrieved from https://arxiv.org/abs/2310.02170
[89] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2024. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *Proceedings of the 1st Conference on Language Modeling*.
[90] David Lo. 2023. Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps. arXiv:2309.04142. Retrieved from https://arxiv.org/abs/2309.04142
[91] David Lo. 2024. Requirements engineering for trustworthy human-AI synergy in software engineering 2.0. In *Proceedings of the 2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 3–4.
[92] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? arXiv:2406.01422. Retrieved from https://arxiv.org/abs/2406.01422
[93] Pattie Maes. 1993. Modeling adaptive autonomous agents. *Artificial life* 1, 1_2 (1993), 135–162.
[94] Zhenyu Mao, Jialong Li, Munan Li, and Kenji Tei. 2024. Multi-role consensus through LLMs discussions for vulnerability detection. arXiv:2403.14274. Retrieved from https://arxiv.org/abs/2403.14274
[95] Lina Markauskaite, Rebecca Marrone, Oleksandra Poquet, Simon Knight, Roberto Martinez-Maldonado, Sarah Howard, Jo Tondeur, Maarten De Laat, Simon Buckingham Shum, Dragan Gašević, et al. 2022. Rethinking the entwinement between artificial intelligence and human learning: What capabilities do learners need for a world with AI? *Computers and Education: Artificial Intelligence* 3 (2022), 100056.
[96] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-driven development for code generation. arXiv:2402.13521. Retrieved from https://arxiv.org/abs/2402.13521
[97] Harrison Chase. 2022. LangChain. (2022). Retrieved from https://github.com/langchain-ai/langchain
[98] Alfred R. Mele. 2001. *Autonomous agents: From self-control to autonomy*. Oxford University Press, USA.
[99] Timothy Meline. 2006. Selecting studies for systemic review: Inclusion and exclusion criteria. *Contemporary Issues in Communication Science and Disorders* 33, Spring (2006), 21–27.
[100] Emilia Mendes, Pilar Rodriguez, Vitor Freitas, Simon Baker, and Mohamed Amine Atoui. 2018. Towards improving decision making and estimating the value of decisions in value-based software engineering: The value framework. *Software Quality Journal* 26 (2018), 607–656.
[101] Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. AgileCoder: Dynamic collaborative agents for software development based on agile methodology. arXiv:2406.11912. Retrieved from https://arxiv.org/abs/2406.11912
[102] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *Proceedings of the 12th International Conference on Learning Representations*.
[103] Maria Paasivaara, Sandra Durasiewicz, and Casper Lassenius. 2008. Distributed agile development: Using scrum in a large project. In *Proceedings of the 2008 IEEE International Conference on Global Software Engineering*. IEEE, 87–95.
[104] Stuart L. Pardau. 2018. The California consumer privacy act: Towards a European-style privacy regime in the United States. *Journal of Technology Law & Policy* 23 (2018), 68.
[105] Kai Petersen, Claes Wohlin, and Dejan Baca. 2009. The waterfall model in large-scale development. In *Proceedings of the 10th International Conference on Product-Focused Software Process Improvement (PROFES '09)*. Springer, 386–400.
[106] Huy Nhat Phan, Tien N. Nguyen, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. arXiv:2409.16299. Retrieved from https://arxiv.org/abs/2409.16299
[107] Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, YiFei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, et al. 2024. Experiential co-learning of software-developing agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics* (Volume 1 Long Papers). Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, Bangkok, Thailand, 5628–5640. Retrieved from https://aclanthology.org/2024.acl-long.305
[108] Chen Qian, Jiahao Li, Yufan Dang, Wei Liu, YiFei Wang, Zihao Xie, Weize Chen, Cheng Yang, Yingli Zhang, Zhiyuan Liu, et al. 2024. Iterative experience refinement of software-developing agents. arXiv:2405.04219. Retrieved from https://arxiv.org/abs/2405.04219
[109] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics* (Volume 1 Long Papers). Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, Bangkok, Thailand, 15174–15186. Retrieved from https://aclanthology.org/2024.acl-long.810
[110] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. AgentFL: Scaling LLM-based fault localization to project-level context. arXiv:2403.16362. Retrieved from https://arxiv.org/abs/2403.16362
[111] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. AI-powered code review with LLMs: Early results. arXiv:2404.18496. Retrieved from https://arxiv.org/abs/2404.18496

[112] Zeeshan Rasheed, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. 2024. Codepori: Large scale model for autonomous software development by using multi-agents. arXiv:2402.01411. Retrieved from https://arxiv.org/abs/2402.01411
[113] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. SpecRover: Code intent extraction via LLMs. arXiv:2408.02232. Retrieved from https://arxiv.org/abs/2408.02232
[114] Malik Abdul Sami, Muhammad Waseem, Zeeshan Rasheed, Mika Saari, Kari Systä, and Pekka Abrahamsson. 2024. Experimenting with multi-agent software development: Towards a unified platform. arXiv:2406.05381. Retrieved from https://arxiv.org/abs/2406.05381
[115] Malik Abdul Sami, Muhammad Waseem, Zheying Zhang, Zeeshan Rasheed, Kari Systä, and Pekka Abrahamsson. 2024. AI based multiagent approach for requirements elicitation and analysis. arXiv:2409.00038. Retrieved from https://arxiv.org/abs/2409.00038
[116] Ravi S. Sandhu. 1998. Role-based access control. In *Advances in Computers*. Vol. 46. Elsevier, 237–286.
[117] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024), 8634–8652.
[118] Yoav Shoham. 1993. Agent-oriented programming. *Artificial intelligence* 60, 1 (1993), 51–92.
[119] Preston G. Smith and Guy M. Merritt. 2020. *Proactive Risk Management: Controlling Uncertainty in Product Development*. Productivity Press.
[120] Giriprasad Sridhara, Ranjani H.G, and Sourav Mazumdar. 2023. ChatGPT: A study on its utility for ubiquitous software engineering tasks. arXiv:2305.16837. Retrieved from https://arxiv.org/abs/2305.16837
[121] Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI coders are among us: Rethinking programming language grammar towards efficient code generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1124–1136.
[122] Ali Sunyaev and Ali Sunyaev. 2020. Distributed ledger technology. *Internet Computing: Principles of Distributed Systems and Emerging Internet-based Technologies*. Springer, 265–299.
[123] Maryam Taeb, Amanda Swearngin, Eldon Schoop, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2024. Axnav: Replaying accessibility tests from natural language. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 1–16.
[124] Daniel Tang, Zhenghan Chen, Kisub Kim, Yewei Song, Haoye Tian, Saad Ezzini, Yongfeng Huang, and Jacques Klein Tegawende F. Bissyande. 2024. Collaborative agents for software engineering. arXiv:2402.02172. Retrieved from https://arxiv.org/abs/2402.02172
[125] Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. 2024. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. arXiv:2403.17927. Retrieved from https://arxiv.org/abs/2403.17927
[126] Jeff Tian. 2005. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. John Wiley & Sons.
[127] Rainer Unland. 2015. Software agent systems. In *Industrial Agents*. Elsevier, 3–22.
[128] Raymon Van Dinter, Bedir Tekinerdogan, and Cagatay Catal. 2021. Automation of systematic literature reviews: A systematic literature review. *Information and Software Technology* 136 (2021), 106589.
[129] Axel Van Lamsweerde. 2000. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering*, 5–19.
[130] Paul Voigt and Axel Von dem Bussche. 2017. The EU general data protection regulation (Gdpr). *A Practical Guide*, 1st Ed., Springer International publishing 10, 3152676 (2017), 10–5555.
[131] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. arXiv:2305.16291. Retrieved from https://arxiv.org/abs/2305.16291
[132] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair. In *Proceedings of the Findings of the Association for Computational Linguistics (ACL '24)*, 2081–2107.
[133] Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. 2024. MobileAgentBench: An efficient and user-friendly benchmark for mobile LLM agents. arXiv:2406.08184. Retrieved from https://arxiv.org/abs/2406.08184
[134] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2023. A survey on large language model based autonomous agents. arXiv:2308.11432. Retrieved from https://arxiv.org/abs/2308.11432
[135] Qian Wang, Tianyu Wang, Qinbin Li, Jingsheng Liang, and Bingsheng He. 2024. MegaAgent: A practical framework for autonomous cooperation in large-scale LLM agent systems. arXiv:2408.09955. Retrieved from https://arxiv.org/abs/2408.09955

[136] Siyuan Wang, Zhuohan Long, Zhihao Fan, Zhongyu Wei, and Xuanjing Huang. 2024. Benchmark self-evolving: A multi-agent framework for dynamic LLM evaluation. arXiv:2402.11443. Retrieved from https://arxiv.org/abs/2402.11443

[137] Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. arXiv:2302.01560. Retrieved from https://arxiv.org/abs/2302.01560

[138] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2023. RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. arXiv:2310.16340. Retrieved from https://arxiv.org/abs/2310.16340

[139] Zhitao Wang, Wei Wang, Zirao Li, Long Wang, Can Yi, Xinjie Xu, Luyang Cao, Hanjing Su, Shouzhi Chen, and Jun Zhou. 2024. XUAT-Copilot: Multi-agent collaborative system for automated user acceptance testing with large language model. arXiv:2401.02705. Retrieved from https://arxiv.org/abs/2401.02705

[140] Zekun Moore Wang, Zhongyuan Peng, Haoran Que, Jiaheng Liu, Wangchunshu Zhou, Yuhan Wu, Hongcheng Guo, Ruitong Gan, Zehao Ni, Man Zhang, et al. 2023. Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models. arXiv:2310.00746. Retrieved from https://arxiv.org/abs/2310.00746

[141] Peter Wegner. 1990. Concepts and paradigms of object-oriented programming. *ACM Sigplan OOPS Messenger* 1, 1 (1990), 7–87.

[142] Ratnadira Widyasari, David Lo, and Lizi Liao. 2024. Beyond ChatGPT: Enhancing software quality assurance tasks with diverse LLMs and validation techniques. arXiv:2409.01001. Retrieved from https://arxiv.org/abs/2409.01001

[143] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, 1–10.

[144] Michael Wooldridge. 2009. *An Introduction to Multiagent Systems*. John wiley & sons.

[145] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-Gen LLM applications via multi-agent conversation framework. arXiv:2308.08155. Retrieved from https://arxiv.org/abs/2308.08155

[146] Yiran Wu, Feiran Jia, Shaokun Zhang, Qingyun Wu, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, and Chi Wang. 2023. An empirical study on challenging math problem solving with gpt-4. arXiv:2306.01337. Retrieved from https://arxiv.org/abs/2306.01337

[147] Zengqing Wu, Shuyuan Zheng, Qianying Liu, Xu Han, Brian Inhyuk Kwon, Makoto Onizuka, Shaojie Tang, Run Peng, and Chuan Xiao. 2024. Shall we talk: Exploring spontaneous collaborations of competing LLM agents. arXiv:2402.12327. Retrieved from https://arxiv.org/abs/2402.12327

[148] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. arXiv:2309.07864. Retrieved from https://arxiv.org/abs/2309.07864

[149] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[150] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-Box compiler fuzzing empowered by large language models. arXiv:2310.15991. Retrieved from https://arxiv.org/abs/2310.15991

[151] Chengran Yang, Jiakun Liu, Bowen Xu, Christoph Treude, Yunbo Lyu, Ming Li, and David Lo. 2023. APIDocBooster: An extract-then-abstract framework leveraging large language models for augmenting API documentation. arXiv:2312.10934. Retrieved from https://arxiv.org/abs/2312.10934

[152] Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. 2024. Enhancing the code debugging ability of LLMs via communicative agent based data refinement. arXiv:2408.05006. Retrieved from https://arxiv.org/abs/2408.05006

[153] Xun Yi, Russell Paulet, Elisa Bertino, Xun Yi, Russell Paulet, and Elisa Bertino. 2014. *Homomorphic Encryption*. Springer.

[154] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2024. Intent-driven mobile GUI testing with autonomous large language model agents. In *Proceedings of the 2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 129–139.

[155] Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, et al. 2024. CodeS: Natural language to code repository via multi-layer sketch. arXiv:2403.16443. Retrieved from https://arxiv.org/abs/2403.16443

[156] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 39–51.

[157] Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, and Dawei Cheng. 2024. G-designer: Architecting multi-agent communication topologies via graph neural networks. arXiv:2410.11782. Retrieved from https://arxiv.org/abs/2410.11782

[158] Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. arXiv:2409.05001. Retrieved from https://arxiv.org/abs/2409.05001

[159] Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, et al. 2024. Diversity empowers intelligence: Integrating expertise of software engineering agents. arXiv:2408.07060. Retrieved from https://arxiv.org/abs/2408.07060

[160] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. 2024. Acfix: Guiding LLMs with mined common rbac practices for context-aware repair of access control vulnerabilities in smart contracts. arXiv:2403.06838. Retrieved from https://arxiv.org/abs/2403.06838

[161] Li Zhang, Jia-Hao Tian, Jing Jiang, Yi-Jun Liu, Meng-Yuan Pu, and Tao Yue. 2018. Empirical research in software engineering—a literature survey. *Journal of Computer Science and Technology* 33 (2018), 876–899.

[162] Simiao Zhang, Jiaping Wang, Guoliang Dong, Jun Sun, Yueling Zhang, and Geguang Pu. 2024. Experimenting a new programming practice with LLMs. arXiv:2401.01062. Retrieved from https://arxiv.org/abs/2401.01062

[163] Sai Zhang, Zhenchang Xing, Ronghui Guo, Fangzhou Xu, Lei Chen, Zhaoyuan Zhang, Xiaowang Zhang, Zhiyong Feng, and Zhiqiang Zhuang. 2024. Empowering agile-based generative software development through human-AI teamwork. arXiv:2407.15568. Retrieved from https://arxiv.org/abs/2407.15568

[164] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: A survey on hallucination in large language models. arXiv:2309.01219. Retrieved from https://arxiv.org/abs/2309.01219

[165] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis,* 1592–1604.

[166] Zhonghan Zhao, Kewei Chen, Dongxu Guo, Wenhao Chai, Tian Ye, Yanting Zhang, and Gaoang Wang. 2024. Hierarchical auto-organizing system for open-ended multi-agent navigation. arXiv:2403.08282. Retrieved from https://arxiv.org/abs/2403.08282

[167] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services* 14, 4 (2018), 352–375.

[168] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. arXiv:2404.02525. Retrieved from https://arxiv.org/abs/2404.02525

[169] Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. 2023. CCBERT: Self-supervised code change representation learning. In *Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 182–193.

[170] Tong Zhu, Xiaoye Qu, Daize Dong, Jiacheng Ruan, Jingqi Tong, Conghui He, and Yu Cheng. 2024. Llama-Moe: Building mixture-of-experts from llama with continual pre-training. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing,* 15913–15923.

[171] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. arXiv:2406.15877. Retrieved from https://arxiv.org/abs/2406.15877

Received 2 April 2024; revised 16 December 2024; accepted 18 December 2024