A Multi-Agent Autonomous Software Engineering Framework Using LLM-Based Agents

AnilKumar J Kadam[1], Karan S Khedkar[2]
[1]*Department of Computer Engineering, All India Shri Shivaji Memorial Society's College of Engineering (AISSMS COE), affiliated with Savitribai Phule Pune University, Pune, India*
[2]*Department of Engineering, ME Artificial Intelligence and Data Science AISSMS College of Engineering, affiliated with Savitribai Phule Pune University Pune, India*

Abstract: ***As software applications continue to grow in complexity, developers often spend significant time on planning, structuring, and writing repetitive code components. Existing AI coding assistants mainly provide autocomplete suggestions or generate small code snippets, but they usually lack the ability to handle complete software development workflows in an organized and autonomous manner. To overcome these limitations, this work presents a Multi-Agent Autonomous Software Engineering Framework that converts user requirements written in natural language into structured software projects. The framework uses multiple AI agents, where each agent is assigned a specific responsibility such as project planning, architecture generation, and source-code development. A workflow-based orchestration approach is used to coordinate communication between agents and manage the execution sequence. The system also supports structured task generation, recursive execution flow, and tool-based file operations for creating and managing project files automatically. The proposed framework is implemented using LangGraph, ReAct-based agents, and large language models to support intelligent reasoning during software generation. Initial testing performed on different software generation tasks such as CRUD applications, API-based systems, and utility projects shows that the framework can generate organized project structures and complete implementation steps with consistent performance. The study demonstrates how multi-agent AI systems can support the development of autonomous and scalable software engineering solutions beyond traditional code suggestion tools.***

Keywords: **Autonomous software engineering, multi-agent systems, code generation, large language models, LangGraph, ReAct agents, AI-based coding systems, workflow orchestration, intelligent software development, autonomous coding framework, software automation, agentic AI systems**

I. INTRODUCTION

Software development has evolved rapidly over the past decade with the increasing demand for scalable applications, faster deployment cycles, and continuous integration of new technologies. Modern software systems often require developers to manage large codebases, maintain architectural consistency, and implement features within limited development timelines. Although recent advances in artificial intelligence have introduced AI-assisted coding tools capable of generating code snippets and providing programming suggestions, most existing systems are still limited to autocomplete-style assistance and lack the capability to autonomously manage complete software engineering workflows. As software projects continue to grow in complexity, the need for intelligent and workflow-driven development systems has become increasingly important.

Large Language Models (LLMs) have demonstrated strong capabilities in natural language understanding, reasoning, and source-code generation. Research efforts such as OpenAI Codex, SWE-agent, and other autonomous coding frameworks have shown that language models can assist in solving programming tasks, understanding repositories, and generating implementation logic from textual instructions. However, many current approaches still operate as isolated coding assistants and often struggle with long-term planning, dependency management, multi-file coordination, and iterative execution. In practical software engineering environments, generating functional applications requires more than producing individual code snippets. It involves structured

planning, architectural decomposition, contextual reasoning, and controlled interaction with development tools and project resources.

In this context, multi-agent AI systems provide a promising direction for autonomous software engineering. By distributing responsibilities across specialized agents such as planning, architecture generation, and implementation, these systems can manage software development tasks in a more organized and scalable manner. Agent-oriented workflows combined with tool-using language models enable autonomous interaction with files, execution pipelines, and project structures, allowing AI systems to move beyond passive assistance toward active software generation. Frameworks such as LangGraph and ReAct-based agents further support iterative reasoning and stateful workflow execution, making them suitable for handling complex development pipelines.

Moreover, modern software projects often involve interconnected modules, multiple dependencies, and continuous modifications across different components. Traditional single-step generation methods may produce inconsistent implementations or incomplete project structures when handling such complexity. An effective autonomous software engineering framework must therefore be capable of maintaining contextual continuity, managing execution flow, coordinating between multiple development stages, and adapting dynamically to evolving project requirements. By integrating structured outputs, workflow orchestration, and autonomous tool interaction, intelligent multi-agent systems can improve development efficiency and support scalable software generation processes. As AI-driven software engineering continues to advance, such approaches have the potential to redefine how applications are designed, developed, and maintained in modern computing environments.

II. RELATED WORK

Research in AI-assisted software development has increased rapidly with the advancement of Large Language Models (LLMs) and intelligent agent-based systems. Earlier coding assistants mainly focused on autocomplete suggestions and template-based code generation, providing limited support for larger software engineering workflows. These systems improved developer productivity but lacked contextual understanding, long-term reasoning, and autonomous execution capabilities. As software systems became more complex, researchers started exploring more intelligent approaches capable of handling multi-step software engineering tasks. OpenAI Codex [1] demonstrated that transformer-based language models could generate source code directly from natural language instructions, opening new directions for AI-driven software development. Similar studies by Chen et al. [18] and Austin et al. [19] further explored code generation and program synthesis using large language models, highlighting their ability to solve programming tasks across multiple programming languages.

To improve reasoning capability in AI systems, researchers introduced approaches combining intermediate reasoning with task execution. Wei et al. [7] proposed Chain-of-Thought prompting, which enabled language models to generate reasoning steps before producing final outputs. Later, Yao et al. [6] introduced the ReAct framework, combining reasoning and action-based execution within a unified workflow. This approach allowed language models to interact dynamically with external tools while solving complex tasks. ReAct-based systems became particularly important for autonomous software engineering because they enabled AI agents to read files, modify project resources, and execute iterative development operations during code generation workflows.

Recent studies have also focused on autonomous software engineering frameworks capable of repository-level reasoning and multi-step task execution. Yang et al. [1] proposed SWE-agent, a framework designed to resolve real-world software engineering issues through autonomous repository interaction and iterative reasoning. Kumar et al. [3] introduced AgentForge, a multi-agent execution framework where multiple specialized agents collaborate to perform software development tasks. Similarly, Lian et al. [4] proposed SWE-AGILE, which focused on dynamic context management and structured reasoning for large-scale software engineering workflows. He and Roy [5] further explored autonomous repository analysis and issue resolution using agentic frameworks capable of

dependency-aware code modification and structured debugging processes.

Parallel research in multi-agent systems and agent-oriented software engineering also contributed significantly to the development of collaborative AI workflows. Wooldridge [11] discussed the principles of multi-agent coordination and distributed task execution, while Shoham and Leyton-Brown [12] explored algorithmic and logical foundations of multi-agent systems. Rao [13] introduced AgentSpeak-based architectures for intelligent autonomous agents, and Bordini et al. [14] further extended these concepts for practical multi-agent software engineering environments. These foundational studies later influenced modern LLM-based orchestration systems where different AI agents perform specialized tasks such as planning, implementation, testing, and debugging.

Researchers have additionally focused on evaluating the real-world capability of autonomous coding systems. Jimenez et al. [20] introduced SWE-bench, a benchmark designed to evaluate whether language models can resolve practical GitHub software issues. Their findings highlighted several challenges including long-context reasoning, repository understanding, and maintaining consistency across large codebases. Recent reviews on Agentic AI systems by Bandi et al. [10] also emphasized the growing importance of workflow orchestration, memory handling, tool integration, and self-correcting execution loops for scalable autonomous software engineering systems.

Overall, existing studies show a clear transition from traditional AI coding assistants toward workflow-driven autonomous software engineering frameworks. The integration of multi-agent orchestration, structured reasoning, and tool-based execution has emerged as a promising direction for developing intelligent systems capable of handling complex software development workflows with reduced human intervention.

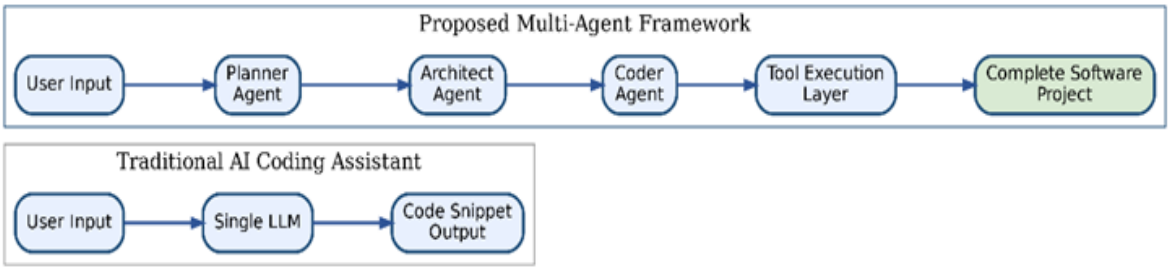

Figure 1. Comparison Between Traditional Coding Assistants and Proposed Multi-Agent Framework

III. METHODOLOGY

The proposed framework uses a multi-agent workflow architecture to automate different stages of the software development process. The system converts software requirements written in natural language into structured project plans, implementation tasks, and executable source code. Instead of depending on a single AI model for all operations, the framework distributes responsibilities among multiple specialized agents responsible for planning, architectural task generation, and code implementation. The workflow is managed using LangGraph orchestration and ReAct-based reasoning agents to support iterative execution and autonomous interaction with project resources.

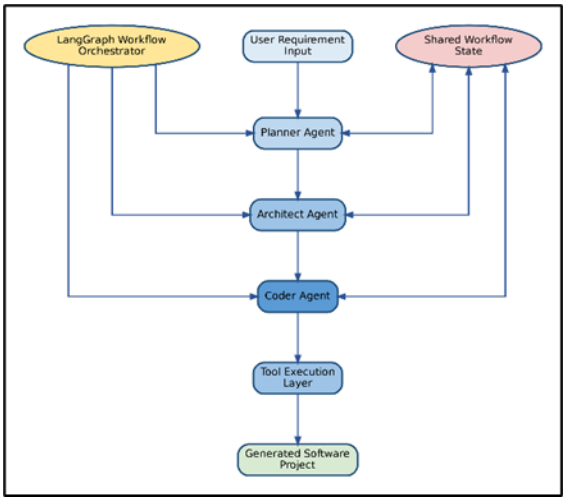

Figure 2. Framework Architecture

1. Requirement Collection and User Input

The first stage begins with collecting software requirements from the user in natural language format. These requirements may describe application functionality, preferred technologies, project objectives, or expected features. The system accepts high-level development instructions such as creating web applications, APIs, CRUD systems, or utility-based software projects. Gathering complete requirement information at this stage is important because the quality of generated architecture and implementation depends heavily on the clarity of the input instructions.

2. Workflow Initialization and State Management

After receiving the user request, the framework initializes a centralized workflow state that stores project-related information throughout the execution process. This shared state acts as a communication layer between different agents participating in the workflow. Important details such as project plans, implementation tasks, generated files, execution status, and intermediate outputs are continuously updated during execution. Maintaining a centralized state ensures consistency between planning, architecture generation, and coding stages.

3. Project Planning and Structured Task Generation

The planning stage is responsible for converting the user requirement into a structured software development plan. A dedicated Planner Agent analyzes the input prompt and identifies important project details such as application type, technology stack, features, modules, and required files. Structured outputs generated using schema-based validation help maintain consistency in the planning process. This stage transforms unstructured natural language instructions into machine-readable engineering information that can be processed reliably during later stages of execution.

4. Architecture Design and Dependency Analysis

Once the initial project plan is created, the Architect Agent generates detailed implementation tasks for the software project. The agent divides the application into smaller development units and identifies dependencies between modules, files, and components. The architecture stage also defines implementation order, integration flow, and file-level responsibilities to ensure organized project generation. This structured decomposition helps the system handle larger projects more effectively compared to single-step generation approaches.

5. Autonomous Code Generation Through Tool-Using Agents

After architecture generation, the Coder Agent begins the implementation phase. The framework uses ReAct-based reasoning agents capable of interacting with external tools during execution. Instead of generating static text outputs, the coding agent can read project files, write source code, inspect directory structures, and update implementations dynamically. Tool-based interaction enables the framework to maintain contextual continuity while generating code across multiple files and interconnected modules. The generated files are automatically stored within a controlled project workspace.

6. Iterative Execution and Recursive Workflow Handling

The framework supports iterative execution using recursive workflow transitions. After completing an implementation task, the workflow checks whether additional tasks remain pending. If unfinished tasks exist, execution continues automatically until all project components are generated. This recursive execution mechanism allows the system to process software development in multiple stages rather than relying on a single response generation cycle. Such iterative handling improves organization and enables better coordination between project modules.

7. File Management and Controlled Resource Handling

The system includes autonomous file management operations for handling project resources safely during execution. Dedicated tool functions are used for reading files, writing implementations, and inspecting project structures. File operations are performed inside a controlled workspace to prevent unauthorized access or unintended modification of external resources. This controlled execution environment improves the reliability and safety of the autonomous development process.

8. Multi-Agent Coordination and Workflow Orchestration

The final stage of the methodology involves coordination between all participating agents through

workflow orchestration. LangGraph manages execution flow between the Planner Agent, Architect Agent, and Coder Agent while maintaining execution state throughout the development lifecycle. The workflow-based architecture enables structured communication, organized task execution, and controlled transitions between different software engineering stages. By combining multi-agent reasoning, structured outputs, and tool-based execution, the proposed framework provides a scalable approach for autonomous software generation and intelligent software engineering workflows.

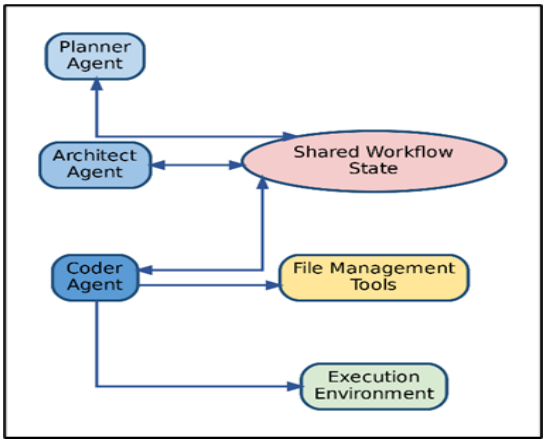

Figure 3. Inter-Agent Communication Through Shared Workflow State

IV. SYSTEM WORKFLOW AND MATHEMATICAL FORMULATION

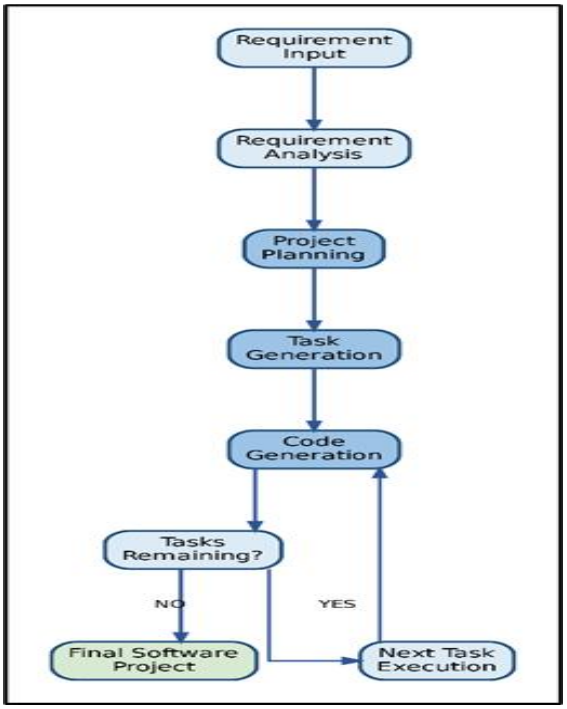

Figure 4. Recursive Multi-Agent Workflow Execution Process

A. Phase 1: Requirement Processing and Planning (Input Stage)

Step 1. User Requirement Collection

The workflow begins with collecting software requirements from the user in natural language form. These requirements may describe application functionality, preferred technology stack, expected modules, APIs, frontend features, authentication systems, or database requirements. The input prompt acts as the primary source of project information and provides the foundation for all later stages of autonomous software generation.

Step 2. Workflow Initialization and State Aggregation

Once the requirement is received, the framework initializes a centralized workflow state for maintaining execution information throughout the software generation process. The state stores project plans, generated tasks, implementation status, file structures, and intermediate outputs. This shared execution layer enables communication between different agents participating in the workflow and helps maintain consistency across development stages.

Step 3. Structured Planning and Task Preparation

The next stage converts the natural language requirement into a structured software engineering plan. The Planner Agent analyzes the input and extracts important project details such as application type, features, modules, technologies, and required files. Structured outputs generated through schema validation help maintain consistent formatting and reliable task representation. This stage transforms user instructions into machine-readable development information suitable for autonomous execution.

B. Phase 2: Architecture Generation and Autonomous Development (Core Execution Stage)

Step 4. Multi-Agent Architecture and Task Decomposition

After generating the project plan, the Architect Agent divides the software project into smaller implementation tasks. The framework identifies

dependencies between modules, implementation order, integration flow, and file-level responsibilities. Instead of generating the entire application in a single step, the workflow processes software development incrementally through smaller organized tasks. This improves scalability and reduces inconsistency during multi-file code generation.

Step 5. Autonomous Code Generation Through ReAct Agents

At this stage, the Coder Agent performs source-code implementation using ReAct-based reasoning and tool interaction. The agent is capable of reading project files, generating code, updating implementations, and inspecting directory structures dynamically during execution. Tool-based interaction enables the system to maintain contextual continuity while handling interconnected project modules. The generated source code is automatically written into project files inside a controlled execution workspace.

Step 6. Recursive Workflow Execution and Task Continuation

The framework supports recursive execution where the workflow continuously checks whether additional implementation tasks remain pending. After completing one task, execution automatically transitions to the next development stage until all project components are generated successfully. This iterative execution process allows the framework to manage larger software projects more effectively compared to traditional single-response generation systems.

C. Phase 2: Output Handling and Controlled File Management (Output Stage)

Step 7. File Management and Resource Handling

The generated source code and project files are stored using autonomous file management operations. Dedicated tool functions handle reading, writing, and directory inspection during execution. File operations are restricted to a controlled workspace environment to prevent unauthorized modification of external resources. This controlled handling improves both reliability and execution safety during autonomous software generation.

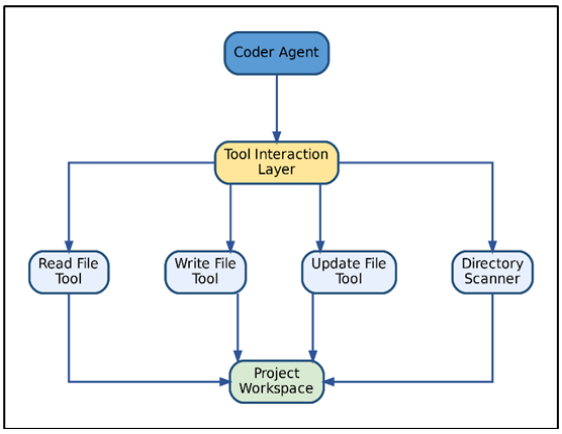

Figure 5. Autonomous Tool Interaction and File Management Layer

Step 8. Workflow Coordination and Continuous Execution

The final stage manages communication and coordination between all participating agents. LangGraph orchestration controls execution flow between the Planner Agent, Architect Agent, and Coder Agent while preserving workflow state across different stages of development. The system continuously updates execution progress and generated outputs during runtime. By combining multi-agent reasoning, structured task execution, and tool-based interactions, the framework supports scalable and organized autonomous software engineering workflows.

Mathematical Formulation of the Proposed Framework

1. User Requirement Representation

The user requirement is represented as an input prompt:

$$P = \{r1, r2, r3, \dots, rn\}$$

Where:

- P = complete project requirement
- r_n = individual requirement or feature instruction

2. Structured Task Generation Function

The planning agent converts the requirement into structured implementation tasks:

$$T = f(P)$$

Where:

- P = user project requirement
- f = planning function generated by the Planner Agent
- T = structured task set

3. Multi-Agent Workflow Transition

The workflow execution between agents is represented as:

$$A_i \rightarrow A_{i+1}$$

Where:

- A_i = current executing agent
- A_{i+1} = next agent in the workflow pipeline

This represents transitions between:

- Planner Agent
- Architect Agent
- Coder Agent

4. Code Generation Function

The autonomous code generation process is represented as:

$$C = g(T, F)$$

Where:

- C = generated source code
- T = implementation task
- F = existing project files and context
- `g = code generation function performed by the Coder Agent`

5. Recursive Execution Condition

The workflow continues until all tasks are completed:

$$W = \begin{cases} Continue, & T_{remaining} > 0 \\ Stop, & T_{remaining} = 0 \end{cases}$$

Where:

- W = workflow execution state
- T_remaining = number of pending implementation tasks

6. File Operation Representation

The autonomous file management process is represented as:

$$F_{new} = F_{old} + \Delta C$$

Where:

- F_old = existing project file
- ΔC = newly generated code changes
- F_new = updated project file after execution

Autonomous Software Generation Workflow

The framework follows a structured execution pipeline beginning with natural language requirement collection and ending with complete software project generation. The workflow first processes the user requirement and converts it into a structured project plan. The Architect Agent then decomposes the project into smaller implementation tasks while identifying dependencies between files and modules. After task generation, the ReAct-based Coder Agent autonomously performs file operations and generates source code incrementally through iterative execution cycles.

The generated project structure may include frontend files, backend APIs, database configurations, utility modules, and routing components depending on the user requirement. Since execution is managed recursively, the workflow continues until all project tasks are completed successfully. This iterative multi-agent pipeline enables the framework to support organized and scalable software generation workflows for modern application development environments.

V. CONCLUSION

This study presents a multi-agent autonomous software engineering framework designed to automate different stages of the software development process using Large Language Models, workflow orchestration, and tool-based execution. The proposed framework converts natural language software requirements into structured project plans, implementation tasks, and executable source code through coordinated interaction between specialized agents responsible for planning, architecture generation, and code implementation. By combining LangGraph-based workflow management with ReAct-driven reasoning agents, the system supports iterative execution and organized software generation across multiple files and interconnected modules.

The framework demonstrates how autonomous AI systems can move beyond traditional code suggestion tools toward more structured software engineering workflows. Instead of generating isolated code snippets, the proposed approach focuses on project-level execution involving task decomposition, dependency-aware planning, contextual coordination, and autonomous file handling. The use of structured outputs and centralized workflow state management further improves consistency during software generation and allows communication between different stages of execution.

The study also highlights several important challenges in autonomous software engineering, including project consistency, multi-file coordination, execution management, and contextual reasoning across larger applications. Existing AI coding assistants often struggle when handling complete software development workflows involving architecture planning and interconnected project structures. The proposed multi-agent approach addresses some of these limitations by distributing responsibilities among specialized agents and supporting recursive execution throughout the development lifecycle.

Overall, the framework demonstrates the growing potential of multi-agent AI systems in modern software engineering environments. The integration of reasoning agents, workflow orchestration, and autonomous tool interaction provides a scalable direction for intelligent software generation systems capable of supporting complex development tasks with reduced manual effort. As research in autonomous software engineering continues to evolve, such frameworks may play an increasingly important role in future AI-driven application development workflows.

VI. FUTURE SCOPE

1. The proposed autonomous software engineering framework provides several opportunities for future improvement and expansion. One important direction is the integration of automated testing and validation mechanisms within the workflow. Currently, the framework focuses mainly on project planning and source-code generation. Future versions can include dedicated testing agents capable of executing generated applications, identifying runtime or syntax errors, and automatically correcting implementation issues through self-repair workflows.
2. Another possible enhancement involves improving long-context understanding and repository-level memory management. As software projects grow larger, maintaining consistency across multiple modules and files becomes increasingly difficult. Future systems can incorporate vector-based memory retrieval, repository summarization, and semantic code indexing to improve contextual awareness during long execution workflows.
3. The framework can also be extended by introducing additional specialized agents responsible for tasks such as debugging, security analysis, refactoring, documentation generation, and deployment automation. A larger multi-agent ecosystem would allow the workflow to support more advanced software engineering operations beyond code generation alone. Such extensions could improve scalability and make the framework more suitable for real-world software development environments.
4. Another promising area for future work is integration with modern development platforms and DevOps pipelines. The framework can be connected with version control systems, CI/CD workflows, containerized deployment environments, and cloud-based development platforms. This would allow autonomous agents

to participate directly in software delivery pipelines and support continuous software engineering processes.

5. Future research may also focus on improving execution efficiency and reducing resource consumption during autonomous development workflows. Optimization techniques such as task parallelization, context compression, caching, and adaptive workflow execution can improve scalability for larger projects. In addition, explainable AI techniques may help developers better understand how autonomous agents make planning and implementation decisions during software generation.
6. Finally, future autonomous software engineering systems may evolve toward collaborative human-AI development environments where developers and intelligent agents work together during the software lifecycle. Instead of replacing developers, such systems can function as intelligent engineering assistants capable of accelerating development, reducing repetitive work, and supporting more efficient application design and implementation processes.

REFERENCES

[1] Yang, J., Jimenez, C. E., Wettig, A., et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," arXiv preprint arXiv:2405.15793, 2024.

[2] Applis, L., et al., "Unified Software Engineering Agent as AI Software Engineer," Proceedings of ICSE 2026, 2026.

[3] Kumar, R., Ali, W., Ahmed, J., et al., "AgentForge: Execution-Grounded Multi-Agent LLM Framework for Autonomous Software Engineering," arXiv preprint arXiv:2604.13120, 2026.

[4] Lian, S., Liu, J., Chen, Y., et al., "SWE-AGILE: A Software Agent Framework for Efficiently Managing Dynamic Reasoning Context," arXiv preprint arXiv:2604.11716, 2026.

[5] He, K., and Roy, K., "SWE-Adept: An LLM-Based Agentic Framework for Deep Codebase Analysis and Structured Issue Resolution," arXiv preprint arXiv:2603.01327, 2026.

[6] Yao, S., Zhao, J., Yu, D., et al., "ReAct: Synergizing Reasoning and Acting in Language Models," arXiv preprint arXiv:2210.03629, 2023.

[7] Wei, J., Wang, X., Schuurmans, D., et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," NeurIPS, 2022.

[8] OpenAI, "GPT-4 Technical Report," arXiv preprint arXiv:2303.08774, 2023.

[9] Anthropic, "Claude 3 Technical Report," Anthropic Research, 2024.

[10] Bandi, A., Kongari, B., Naguru, R., et al., "The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges," Future Internet, vol. 17, no. 2, 2025.

[11] Wooldridge, M., "An Introduction to MultiAgent Systems," John Wiley & Sons, 2009.

[12] Shoham, Y., and Leyton-Brown, K., "Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations," Cambridge University Press, 2009.

[13] Rao, A. S., "AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language," Journal of Logic and Computation, vol. 8, no. 3, pp. 293–343, 1998.

[14] Bordini, R. H., Hübner, J. F., and Wooldridge, M., "Programming Multi-Agent Systems in AgentSpeak Using Jason," Wiley Series in Agent Technology, 2007.

[15] Winikoff, M., and Padgham, L., "Agent-Oriented Software Engineering," Springer, 2004.

[16] Caire, G., Coulier, W., Garijo, F., et al., "Agent-Oriented Software Engineering II," Springer Berlin Heidelberg, 2002.

[17] Beydoun, G., Low, G., Henderson-Sellers, B., et al., "FAML: A Generic Metamodel for MAS Development," IEEE Transactions on Software Engineering, vol. 35, no. 6, pp. 841–863, 2009.

[18] Chen, M., Tworek, J., Jun, H., et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.

[19] Austin, J., Odena, A., Nye, M., et al., "Program Synthesis with Large Language Models," arXiv preprint arXiv:2108.07732, 2021.

[20] Jimenez, C. E., Yang, J., Wettig, A., et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" arXiv preprint arXiv:2310.06770, 2023.