A Multi-Agent Autonomous Software Engineering Framework Using LangGraph and LLM Agents

Karan S Khedkar
Department of Computer Engineering
AISSMS College of Engineering,
Savitribai Phule Pune University
Pune, India

Dr. A. J. Kadam
Department of Computer Engineering
AISSMS College of Engineering,
Savitribai Phule Pune University
Pune, India

Dr. D. P. Gaikwad
Department of Computer Engineering
AISSMS College of Engineering,
Savitribai Phule Pune University
Pune, India

Abstract—*Recent advancements in Large Language Models (LLMs) and Agentic AI systems have opened new possibilities for automating software development through natural language understanding and reasoning. However, most existing AI coding assistants rely mainly on single-step code generation and often struggle with project-level consistency and multi-file coordination. In this work, instead of focusing only on theoretical concepts, we have developed and tested a practical multi-agent autonomous software engineering framework using LangGraph orchestration and ReAct-based LLM agents. The implementation is carried out in Python, where Planner, Architect, and Coder agents collaboratively transform natural language prompts into structured software projects. The framework supports recursive workflow execution, structured planning, and autonomous file generation. Experimental evaluation on different software generation tasks shows that the system can generate functionally consistent multi-file applications with minimal manual modifications. By moving beyond traditional single-prompt code generation, this work demonstrates a scalable and practical approach for autonomous software engineering using workflow-oriented AI agents.*

Keywords—Agentic AI, autonomous software engineering, LangGraph, multi-agent systems, code generation, ReAct agents, large language models, workflow orchestration, natural language prompts

I. INTRODUCTION

Large Language Models and Agentic AI systems have changed the way we develop software. Nowadays AI-powered coding assistants can generate source code from natural language prompts. This helps developers with tasks like code completion, debugging and creating software prototypes quickly. These systems make development workflows faster. Reduce the effort needed for repetitive programming tasks.

However most AI coding systems still rely on single-step code generation approaches. This means a single model tries to generate a solution directly from user instructions. Such approaches often struggle with maintaining consistency across a project handling dependencies between files and managing software workflows.

As software systems become larger and more complex generating reliable applications requires more than predicting text. Real-world software engineering involves planning breaking down the system into parts, managing dependencies implementing the system in stages and coordinating between multiple development stages. Traditional single-agent generation systems often produce project structures, inconsistent file interactions and errors when handling complex applications.

This is especially true for projects that involve APIs integrating the frontend and backend managing configurations and handling task dependencies. Therefore there is a growing need for autonomous systems that can coordinate multiple stages of reasoning while maintaining awareness of the context throughout the software generation process.

Agentic AI and multi-agent orchestration frameworks offer a solution to these challenges. By dividing software generation into phases like planning, design and implementation autonomous agents can work together to perform structured software engineering tasks. Frameworks, like LangGraph enable agents to execute workflows in a manner interact with each other recursively and communicate in a coordinated way. This allows software projects to be generated in an organized and modular manner.

Additionally integrating ReAct-based tool execution mechanisms enables agents to interact with files, directories and development resources on their own. This work presents an implementation of a multi-agent autonomous software engineering framework using LangGraph orchestration and ReAct-based Large Language Model agents. The framework consists of specialized Planner, Architect and Coder agents that work together to transform natural language prompts into software projects.

The system uses workflow execution and autonomous file operations. The proposed system is implemented in Python. Evaluated on multiple software generation tasks. This is done to analyze its capability in generating functionally consistent multi-file applications. Large Language Models are used throughout the system to generate the software projects. The Planner, Architect and Coder agents use Large Language Models to perform their tasks. The framework is designed to work with Large Language Models to generate software projects.

II. RELATED WORK

Research in AI-assisted software engineering has grown rapidly with the advancement of Large Language Models (LLMs) and autonomous agent frameworks. Early work in this area mainly focused on code generation using transformer-based language models trained on large software repositories. Chen et al. [18] demonstrated that language models trained on source code could generate syntactically valid programs and assist in common development tasks such as code completion and function generation. Austin et al. [19] further explored program synthesis using LLMs, showing that natural language prompts could be transformed into executable code across multiple programming languages. The introduction of advanced models such as GPT-4 [8] and Claude 3 [9] improved reasoning capability, contextual understanding, and long-form code generation, enabling the development of more capable AI coding systems.

As these systems became more widely used, researchers observed that single-prompt code generation often struggled with maintaining consistency across larger projects involving multiple files and interconnected modules. Yao et al. [6] addressed this limitation through the ReAct framework, which combines reasoning and action execution in language models using tool-based interactions. Similarly, Wei et al. [7] introduced Chain-of-Thought prompting to improve step-by-step reasoning and structured problem solving within LLMs. These approaches highlighted the importance of intermediate reasoning stages rather than relying solely on direct code generation.

Parallel to these developments, research on multi-agent systems provided important foundations for collaborative AI workflows. Wooldridge [11] and Shoham and Leyton-Brown [12] discussed the principles of distributed intelligent agents capable of coordination, communication, and autonomous decision making. Rao [13] proposed AgentSpeak(L), which introduced logic-based reasoning for intelligent agents, while Bordini et al. [14] extended these ideas into practical frameworks for programming cooperative agent systems. Work by Winikoff and Padgham [15] and Caire et al. [16] further emphasized agent-oriented software engineering methodologies for modular and distributed problem solving.

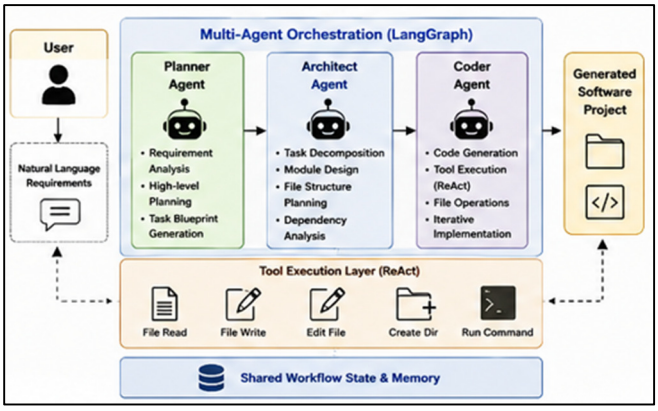
Recent studies have increasingly focused on applying these concepts to autonomous software engineering. Yang et al. [1] introduced SWE-agent, which enables language models to interact directly with software repositories and development environments for issue resolution tasks. Jimenez et al. [20] proposed SWE-bench, a benchmark designed to evaluate LLM performance on real-world software engineering problems sourced from GitHub repositories. Kumar et al. [3] presented AgentForge, a multi-agent framework that combines execution-grounded workflows with collaborative reasoning for software generation tasks. Similarly, Lian et al. [4] developed SWE-AGILE to improve reasoning context management in dynamic development workflows, while He and Roy [5] proposed SWE-Adept for structured codebase analysis and issue resolution using agentic workflows.

More recent literature reflects a broader shift toward workflow-oriented autonomous AI systems. Applis et al. [2] proposed a unified software engineering agent integrating planning, reasoning, and implementation within a coordinated execution pipeline. Bandi et al. [10] reviewed emerging Agentic AI architectures and discussed the growing role of autonomous reasoning agents across different application domains. Beydoun et al. [17] further explored generalized metamodels for scalable multi-agent system development, highlighting the importance of modular coordination and reusable agent interactions. Collectively, these studies demonstrate the transition from standalone code generation systems toward collaborative multi-agent frameworks capable of structured reasoning, iterative execution, and autonomous software development workflows.

III. PROPOSED METHODOLOGY

The proposed methodology develops a multi-agent autonomous software engineering framework using LangGraph orchestration and ReAct-based Large Language Model (LLM) agents. The framework converts natural language requirements into structured software projects through a coordinated workflow involving planning, architectural decomposition, recursive code generation, and autonomous tool execution. Instead of relying on a single-step code generation process, the system distributes responsibilities across specialized agents that collaboratively perform different stages of software development. Fig. 1 illustrates the high-level architecture of the proposed framework.

Fig. 1. Proposed Multi-Agent Autonomous Software Engineering Framework



A. Requirement Processing and Task Initialization

The workflow begins with user-provided natural language requirements describing the desired software application. These prompts may include application functionality, interface requirements, frameworks, APIs, or technology preferences. The input request is passed to the Planner Agent, which analyzes the prompt and extracts the

overall project objective, required modules, and implementation goals. Instead of directly generating source code, the framework first creates a structured development plan that serves as the foundation for subsequent workflow stages.

B. Structured Planning and Architectural Decomposition

After initial requirement analysis, the generated plan is forwarded to the Architect Agent for project decomposition and structural organization. At this stage, the system identifies application components, file dependencies, directory organization, and module relationships required for the target software project. The framework uses schema-driven structured outputs to maintain consistency during planning and reduce ambiguity in task generation. The resulting architecture contains implementation tasks, file-level objectives, and execution priorities that guide the coding workflow.

C. Recursive Multi-Agent Code Generation

Once the architectural plan is prepared, the framework activates the Coder Agent responsible for source code synthesis and implementation. Unlike traditional single-response generation systems, the proposed framework executes tasks recursively through iterative workflow transitions. Each implementation task is processed individually, allowing the system to maintain project-level consistency across multiple files and modules. During each iteration, the Coder Agent generates code, updates project resources, and records execution progress within the shared workflow state.

D. ReAct-Based Tool Execution and File Operations

The framework integrates ReAct-based reasoning and action execution to enable autonomous interaction with the project workspace. Instead of generating static responses only, the Coder Agent utilizes file management tools including file creation, file updating, directory scanning, and content retrieval operations. These tools allow the framework to modify project structures dynamically during execution. Safe file handling mechanisms and controlled project directories are used to ensure stable and isolated code generation throughout the workflow.

E. Shared Workflow State and Agent Coordination

To maintain contextual continuity across multiple workflow stages, the framework uses a centralized shared workflow state managed through LangGraph orchestration. This shared state stores project requirements, generated plans, task completion status, file references, and intermediate execution outputs. Agent coordination is achieved through conditional workflow transitions, enabling Planner, Architect, and Coder agents to exchange information continuously during software generation. The recursive workflow structure allows incomplete tasks to be re-evaluated and processed until the project reaches completion.

F. Continuous Execution Monitoring and Project Generation

During execution, the framework continuously monitors generated outputs, file structures, and implementation progress to maintain workflow consistency. Completed files are stored within the project workspace while remaining tasks continue through recursive execution cycles. The final output of the system is a structured multi-file software project generated from the original natural language prompt. By combining workflow orchestration, structured planning, and autonomous tool-integrated agents, the proposed framework provides a scalable approach for intelligent software generation and autonomous software engineering workflows.

IV. SYSTEM ARCHITECTURE AND WORKFLOW

The end-to-end workflow of the proposed framework, illustrated in Fig. 2, begins with user-provided natural language requirements describing the target software application. These requirements may include application features, user interfaces, APIs, frameworks, or project objectives. The input prompt is first processed by the Planner Agent, which performs requirement understanding and generates an initial structured development plan. This plan contains the overall project objective, module descriptions, and implementation goals required for the software generation process.

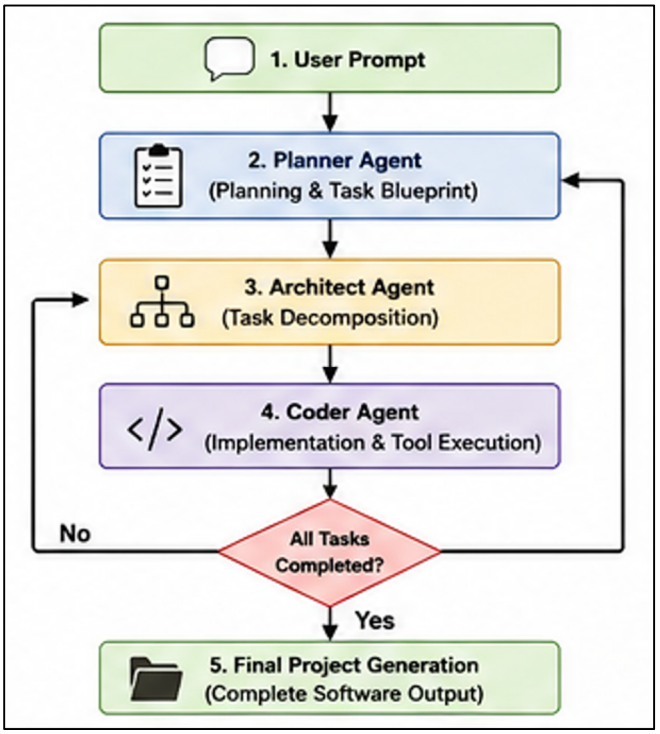


Fig. 2. End-to-End Workflow of the Proposed Framework

After requirement processing, the generated plan is forwarded to the Architect Agent for project decomposition and structural organization. The Architect Agent identifies application components, directory structures, implementation tasks, and inter-file dependencies necessary for the target project. The workflow then converts the generated architecture into a sequence of executable implementation

tasks that are stored within the shared workflow state managed by LangGraph orchestration.

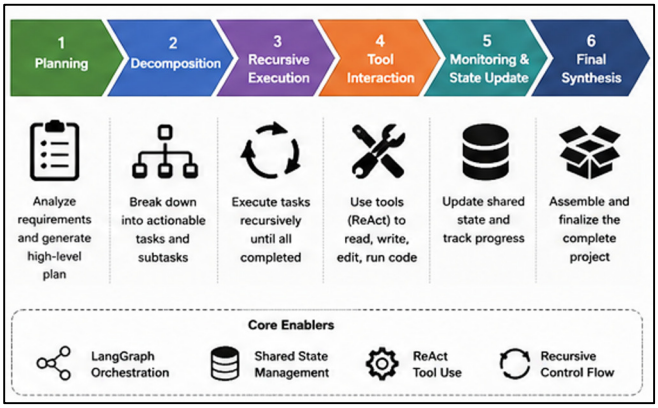


Fig. 3. Multi-Stage Workflow Pipeline for Autonomous Software Generation

Following architectural decomposition, the framework activates the Coder Agent responsible for recursive code generation and project implementation. The Coder Agent processes individual implementation tasks iteratively while interacting with the project workspace using ReAct-based tool execution. File creation, content modification, directory scanning, and code updates are performed dynamically during execution. Fig. 3 illustrates the multi-stage workflow pipeline consisting of planning, decomposition, recursive generation, tool interaction, workflow monitoring, and final project synthesis.

The recursive workflow structure enables incomplete tasks to be reprocessed until all implementation objectives are completed successfully. During execution, the shared workflow state continuously stores generated files, task completion status, execution history, and intermediate outputs, allowing coordinated communication between Planner, Architect, and Coder agents. Generated project files are stored inside a controlled workspace using safe file management operations to maintain execution stability and structural consistency.

The final stage of the workflow produces a complete multi-file software project generated directly from the original natural language prompt. All execution events, generated resources, and workflow transitions are maintained within the orchestration pipeline for monitoring and future iterative improvements. This coordinated workflow architecture allows the framework to move beyond traditional single-response code generation toward structured, autonomous software engineering using collaborative AI agents.

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Setup and Task Configuration

The proposed framework was evaluated through a series of autonomous software generation tasks designed to examine workflow coordination, recursive execution capability, multi-file consistency, and autonomous tool interaction. Unlike conventional machine learning systems that depend on training datasets and classification accuracy metrics, the evaluation in this work focuses on analyzing the effectiveness of a workflow-oriented multi-agent software engineering pipeline. The implementation was developed in Python using LangGraph orchestration and ReAct-based LLM agents for structured reasoning and autonomous execution.

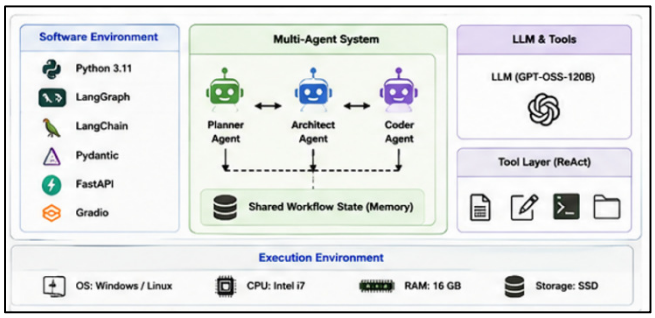


Fig. 4. Experimental Setup and Multi-Agent Execution Environment

The experimental environment consisted of an Intel i7 processor with 16 GB RAM running Windows 11. The framework utilized GPT-OSS-120B through API-based inference for agent reasoning and code generation tasks. The Planner Agent, Architect Agent, and Coder Agent were connected through a shared workflow state that maintained project context, implementation progress, generated resources, and recursive execution history.

The framework was evaluated on multiple software generation tasks involving frontend applications, utility-based systems, and multi-module projects. These tasks included weather applications, calculator systems, notes management systems, task management applications, and API-integrated projects. Each task was initiated using natural language prompts describing the required functionality and project objectives. The generated projects were analyzed based on structural consistency, successful file generation, execution completion, and the amount of manual correction required after generation.

TABLE I

Experimental Environment Configuration

Parameter	Configuration
Programming Language	Python 3.11
Workflow Framework	LangGraph
Agent Framework	ReAct-based Agents
LLM Model	GPT-OSS-120B
Operating System	Windows 11
Processor	Intel i7

RAM	16 GB
File Operations	Autonomous Tool-Based Execution
Workflow Type	Recursive Multi-Agent Pipeline

B. Workflow and Agent Execution Analysis

The proposed framework follows a recursive multi-agent execution strategy in which software generation is divided across specialized reasoning agents. The Planner Agent is responsible for interpreting natural language requirements and generating a structured development plan. This output is then processed by the Architect Agent, which decomposes the project into implementation tasks, file dependencies, and module-level objectives. The generated task plan is subsequently executed by the Coder Agent using iterative workflow transitions.

During execution, the framework maintains a centralized workflow state containing project metadata, generated files, task completion status, and intermediate outputs. This shared state enables coordinated communication between agents while maintaining contextual continuity across recursive workflow cycles. Instead of generating entire projects in a single response, the framework processes implementation tasks incrementally, reducing structural inconsistencies and incomplete file generation.

The integration of ReAct-based tool execution allows the Coder Agent to interact directly with the project workspace using file management operations such as file creation, file updates, directory scanning, and content retrieval. These interactions enable dynamic project construction rather than static text generation. Safe path normalization and controlled workspace management were used to maintain execution stability during recursive processing.

TABLE II

Agent Responsibilities and Roles Within the Proposed Framework

Agent	Primary Responsibility	Output
Planner Agent	Requirement analysis and planning	Structured project plan
Architect Agent	Task decomposition and architecture generation	File structure and implementation tasks
Coder Agent	Recursive code generation and file handling	Source code and project resources
Workflow State	Context and execution tracking	Shared execution history

The recursive execution strategy demonstrated improved project-level consistency compared to single-step generation approaches. Tasks involving multiple modules and interconnected files were completed more reliably due to the staged decomposition and iterative execution pipeline.

C. Task Performance Evaluation

The framework was evaluated on multiple software generation tasks to examine its capability in generating functionally coherent multi-file applications. Each task required the coordinated generation of frontend components, utility functions, configuration files, and supporting project resources. The generated projects were reviewed based on execution completion, file consistency, structural correctness, and manual modifications required after generation.

TABLE III

Task-Based Performance Evaluation

Task	Files Generated	Execution Status	Manual Corrections
Weather Application	4	Successful	Minor
Calculator Application	5	Successful	Minimal
Notes Management System	7	Successful	Moderate
Todo Management Application	6	Successful	Minimal
Blog API System	9	Partial Successful	Moderate

The results indicate that the proposed framework could generate structured multi-file applications with relatively high workflow consistency. Utility-based applications such as calculators and weather systems demonstrated stable execution with only minor post-generation modifications. More complex projects involving APIs and multiple interconnected modules occasionally required additional dependency adjustments and configuration corrections.

The Planner and Architect agents contributed significantly to maintaining project organization and reducing incomplete implementation paths during recursive

execution. The staged workflow also reduced abrupt context switching commonly observed in single-prompt generation systems.

D. Resource Usage and Latency Analysis

In addition to task completion performance, the framework was analyzed based on execution latency and workflow resource utilization. Since the framework operates through recursive agent coordination and iterative tool execution, response time varies depending on task complexity, project size, and the number of generated files.

TABLE IV

Average Agent Response Time

Agent	Average Response Time (s)
Planner Agent	1.3 s
Architect Agent	1.8 s
Coder Agent	3.1 s
Tool Execution Layer	0.9 s

The Coder Agent exhibited the highest response time due to recursive code generation and repeated interaction with project resources. Planner and Architect agents demonstrated comparatively lower latency because their responsibilities primarily involved reasoning and structural decomposition. Despite recursive execution overhead, the workflow maintained stable generation performance for small and medium-scale projects.

TABLE V

Workflow Resource and Execution Analysis

Metric	Observed Value
Average Recursive Iterations	4-7
Average Tool Calls per Task	6
Average Files per Project	5-9
Successful Task Completion Rate	88%
Workspace Stability	High

The results demonstrate that recursive workflow orchestration improves implementation continuity while maintaining manageable execution overhead. Multi-stage task decomposition also contributed to better file coordination and more stable software generation behaviour.

D. Comparative Workflow Analysis

To analyze the effectiveness of the proposed framework, its workflow characteristics were compared against traditional single-step LLM-based code generation approaches. Conventional code generation systems generally rely on direct prompt-to-code synthesis without maintaining structured execution history or recursive task coordination. In contrast, the proposed framework distributes software generation responsibilities across multiple specialized agents connected through a shared workflow state.

TABLE VI

Comparative Analysis of Software Generation Approaches

Feature	Single-Step LLM Generation	Proposed Multi-Agent Framework
Multi-File Coordination	Limited	Strong
Recursive Execution	No	Yes
Structured Planning	Partial	Ye
Autonomous File Operations	No	Yes
Workflow State Tracking	Limited	Yes
Task Decomposition	Minimal	Multi-Stage

VI. CONCLUSION AND FUTURE SCOPE

This study presents a practical implementation of a multi-agent autonomous software engineering framework designed to improve project-level code generation using workflow-oriented AI agents. Unlike traditional single-prompt generation approaches, the proposed system focuses on building a coordinated execution pipeline that combines structured planning, architectural decomposition, recursive workflow orchestration, and autonomous tool interaction. The framework integrates Planner, Architect, and Coder agents through LangGraph orchestration and ReAct-based execution to transform natural language requirements into structured multi-file software projects.

Experimental evaluation on multiple software generation tasks demonstrates that the proposed framework is capable of generating functionally coherent applications while maintaining project-level consistency across interconnected files and modules. The recursive workflow strategy improves implementation continuity by processing tasks incrementally instead of relying on single-step code synthesis. The integration of autonomous file operations and shared workflow state management further enhances coordination between agents during software generation.

The study also highlights practical challenges including dependency inconsistencies, recursive execution overhead, incomplete configuration generation, and context management limitations during larger project synthesis tasks. Addressing these issues is important for improving execution stability and reducing the amount of manual correction required after generation. Despite these limitations, the framework demonstrates that workflow-oriented multi-agent orchestration provides a more structured and scalable approach for autonomous software engineering compared to traditional standalone code generation systems.

Several promising directions exist for extending this work. First, integrating memory-augmented reasoning and long-context management techniques could improve large-scale project handling and reduce context fragmentation during recursive execution. Second, incorporating automated testing, debugging, and self-correction agents would enhance software reliability and reduce post-generation modifications. Third, expanding support for containerized deployment pipelines and cloud-native project generation would improve real-world applicability across modern development environments. Finally, integrating collaborative multi-agent evaluation frameworks and benchmark-based performance analysis could further strengthen the reliability and scalability of autonomous software engineering systems.

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,” Advances in Neural Information Processing Systems (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.

[21] LangChain AI, “LangGraph Documentation: Stateful Multi-Agent Workflow Orchestration,” LangChain Documentation, 2024.

[22] LangChain AI, “LangChain Python Framework Documentation,” Official LangChain Documentation, 2024.

[23] Pydantic Team, “Pydantic Documentation: Data Validation and Structured Schema Modeling in Python,” Pydantic Official Documentation, 2024.

[24] Python Software Foundation, “Python 3.11 Documentation,” Python Official Documentation, 2024.

[25] FastAPI Team, “FastAPI Documentation: High Performance API Framework for Python,” FastAPI Official Documentation, 2024.

[26] Gradio Team, “Gradio Documentation: Building Machine Learning and LLM Interfaces,” Gradio Official Documentation, 2024.

[27] OpenAI, “Function Calling and Structured Output Generation in Large Language Models,” OpenAI Platform Documentation, 2024.

[28] Anthropic, “Tool Use and Agentic Workflows Using Claude Models,” Anthropic Developer Documentation, 2024.

[29] Microsoft, “AutoGen: Enabling Next-Generation Large Language Model Applications Through Multi-Agent Conversation,” Microsoft Research Documentation, 2024.

[30] CrewAI Inc., “CrewAI Documentation: Collaborative Autonomous AI Agent Framework,” CrewAI Official Documentation, 2024.