IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

API Code Generation Using Multi AI Agent Architecture- A Survey

Siddhant Sinha
Dept. of ECE
RV College of Engineering
Bengaluru, India

Rajani Katiyar
Dept. of ECE
RV College of Engineering
Bengaluru, India

Somesh M
Rakuten India
Bengaluru, India

***Abstract*—This survey explores the landscape of code generation using Multi-AI Agent Architectures, a rapidly evolving paradigm driven by the capabilities of Large Language Models (LLMs). The primary aim is to assess how multi-agent systems comprising specialized LLM-based agentsm can collaboratively automate and enhance the code generation process. We delineate the scope of this survey to include systems that integrate modular agent roles such as planners, coders, testers, and fixers, particularly in the context of RESTful microservice development, algorithmic problem solving, and collaborative task planning. Methodologically, this survey synthesizes insights from empirical evaluations and architecture overviews in recent studies, such as those evaluating ChatGPT's correctness, complexity and security in multi-round code generation tasks, and frameworks that enable iterative design-fix cycles. The results indicate that agent specialization boosts the accuracy of generation, reduces user-tool switching, and enhances debugging through conversational feedback loops. In particular, systems that leverage role-based agents supported by prompt engineering and function-calling mechanisms show marked improvements in developer productivity and code quality. However, limitations persist in handling complex coordination logic and adapting to large-scale projects. The findings underscore the potential of Multi-AI Agent Architectures to transform software engineering workflows, while also highlighting challenges such as hallucination, agent memory limitations, and integration with continuous development pipelines. This survey lays the groundwork for future research into scalable, reliable, and domain-adaptive code generation using collaborative AI systems.**

***Index Terms*—Artificial Intelligence (AI), Large Language Models (LLMs), AI Agents, Code Generation, Multi AI Agent Architecture, Application Programming Interface (API), OpenAPI.**

I. Introduction

The rise of Large Language Models (LLMs) has revolutionized the landscape of software development, particularly in the domain of code generation. Using the linguistic and semantic understanding capabilities of transformer-based architectures, LLMs have demonstrated the ability to generate syntactically valid and semantically meaningful code across multiple programming languages. However, despite the promise of these single agent systems, their limitations, such as hallucination, limited memory, and challenges in handling complex workflows, have led to the emergence of multiagent architectures as a compelling alternative [1]. Code generation, at its core, involves translating high-level natural language descriptions or formal specifications into executable software artifacts. Traditional AI-assisted code generation systems typically operate as monolithic black-box models, which often lack transparency, adaptability, and robustness in dynamic development contexts. In contrast, multi-agent AI architectures distribute responsibilities across multiple specialized agents, such as specification generators, code writers, testers, and fixers, thus mimicking collaborative human development teams and improving overall effectiveness and modularity. Recent empirical studies [1] have evaluated LLMs like ChatGPT using thousands of algorithmic and CWE benchmark problems, revealing strengths in functionally correct code generation for pre-2021 problems [1], but also exposing challenges in multi-round fixing and vulnerability resolution. These findings underscore the importance of structured agent collaboration, particularly in iterative refinement processes where dialog-based debugging, testing, and regeneration play crucial roles. In response to these limitations, multi-agent frameworks now incorporate agents with distinct roles, memory contexts, and tool integration capabilities, such as log analysis, OpenAPI parsing, and containerized testing environments, to significantly reduce the cognitive and operational burden on developers. [1][2]

II. Background and Foundation

Automatic code generation, the process of translating high-level specifications or natural language descriptions into executable software code, represents a pivotal advancement in modern software engineering. Not only enhances productivity and reduces manual effort, it also facilitates rapid prototyping, consistent API implementations, and greater alignment with agile methodologies. Traditional approaches to code generation often relied on rule-based engines and static templates [1], which, while functional, lacked adaptability and semantic depth. The emergence of Large Language Models (LLMs), such as OpenAI's GPT series, has redefined the boundaries of code generation by embedding knowledge of syntax, logic, and domain semantics within scalable transformer-based architectures [2]. Empirical evaluations, such as the large-scale study conducted in ChatGPT, highlight the capabilities and limitations of single-LLM systems in generating correct, complex and secure code in various programming languages [3].

These findings underscore the potential and need for more sophisticated orchestration in code generation workflows.

A. Multi-AI Agent Architectures

In response to the shortcomings of monolithic LLM approaches—such as memory limitations, hallucination, and lack of modular debugging—the concept of multi-AI agent architectures has emerged. In these systems, specialized agents are assigned distinct tasks such as requirement interpretation, OpenAPI specification creation, server code generation, testing, and iterative code fixing. This architectural decomposition emulates human collaborative development processes, enhancing clarity, fault tolerance, and scalability. [5]

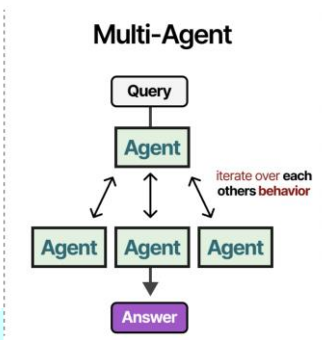

Fig. 1: Multi-AI Agent Architecture

The above figure 1 shows a rudimentary multi agent architecture which can leveraged for various solutions. A proposed a multi-agent LLM-based system that automates RESTful API development by employing distinct agents for OpenAPI generation, code writing, testing, and debugging. The system is modular and leverages function-calling capabilities of GPT-4 agents to execute shell commands, analyze logs, and update code in real-time, thereby reducing context-switching and development time. Their evaluation demonstrates strong alignment with software engineering best practices and highlights productivity gains when compared to manual development workflows. [5]

B. Foundational Technology and Evolution

The development of multi-agent architectures in code generation is rooted in foundational AI principles, including multi-agent systems (MAS), distributed decision-making, and human-in-the-loop computing. Early MAS research emphasized coordination, negotiation, and communication among autonomous entities [5], laying the groundwork for intelligent orchestration in modern applications. The fusion of MAS with LLMs, often referred to as LLM-based Multi-Agent Decision Making (MADM), has proven instrumental in tackling tasks that require distributed reasoning and communication.It is highlighted that MADM frameworks enable LLMs to act cooperatively toward shared goals, benefiting from the language comprehension abilities of each agent. Moreover, such systems demonstrate robust capabilities in real-time code negotiation, debugging, and specification validation through inter-agent dialogue. Historically, frameworks such as ReAct [10] and Reflexion [11] introduced agent-like behaviors within single LLMs, leveraging reasoning and feedback loops. These frameworks inspired the segmentation of responsibilities into dedicated agents, formalizing what are now modular, multi-agent systems. The use of reinforcement learning and role specialization within these systems further enhances decision accuracy and operational efficiency. In contemporary practice, these multi-agent systems are often configured with a director agent that orchestrates tool agents such as:

- A **Specification Agent**, which creates or refines OpenAPI contracts.
- A **Code Generation Agent**, responsible for producing code in frameworks like Express.js.
- A **Test Agent**, which validates functionality using containerized executions.
- An **Error-Fixing Agent**, which parses logs and modifies faulty logic through regeneration.

This agent-based division addresses LLM constraints (e.g., token limits, deterministic variability) and enables seamless plug-and-play customization for different code generation tasks. [5]

C. Importance in Contemporary Research

Multi-agent architectures not only address technical constraints in LLMs but also offer a pathway toward scalable, generalizable AI development systems. Their integration with container technologies (e.g., Docker), testing platforms (e.g., LeetCode), and specification standards (e.g., OpenAPI) represents a convergence of AI, DevOps, and software engineering disciplines [4], [6]. Furthermore, these systems are especially valuable in educational, enterprise, and rapid prototyping settings, where time efficiency, modularity, and iterative refinement are paramount. Industry practitioners acknowledge that the multi-agent approach significantly reduces tool-switching and promotes cleaner codebases, even while highlighting challenges in large-scale customization and performance tuning [5]

III. Architectures of AI Agents

The architecture of AI agents plays a foundational role in shaping the functionality, coordination, and scalability of multi-agent systems. In the domain of code generation, well-defined agent architectures enable the decomposition of complex software engineering tasks into modular, collaborative subtask handled by specialized agents. These agents, endowed with domain-specific capabilities, collectively emulate human-like software development workflows—such as API specification, code synthesis, validation, debugging, and deployment—thus realizing more robust and scalable code generation pipelines. [2]

A. Definition and Role in Multi-AI Systems

An AI agent, in the context of code generation, is an autonomous computational entity equipped with reasoning and language capabilities that enable it to perform specific subtasks in response to user inputs or inter-agent prompts.

In multi-agent architectures, these agents interact in coordinated workflows, often under the supervision of a director or orchestrator agent, to produce high-quality, maintainable, and executable software artifacts. The shift from monolithic LLMs to modular multi-agent architectures addresses key limitations such as token constraints, role ambiguity, and the difficulty of implementing iterative code refinement in single-pass inference systems. The importance of such architectures has been evidenced by projects like Reflexion [11] and ReAct [10], which initially demonstrated agent-like reasoning in single-LLM frameworks. However, subsequent works—such as the modular multi-agent system show that delegating tasks to discrete, role-specific LLM agents significantly improves workflow clarity, system debuggability, and task specialization. [5]

B. Categories of AI Agent Architectures

AI agent architectures used in multi-agent systems for code generation can generally be categorized into the following classes, figure 2 is a representation of how these classes visually are:

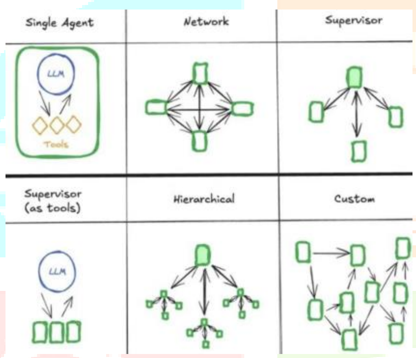

Fig. 2: Classes of AI Agent Architectures

- **Hierarchical Architectures:** These involve a top-down structure in which a central director or coordinator agent delegates subtasks to subordinate agents. For example, the user-facing director agent issues task-specific instructions to subordinate agents such as spec-generator, code-generator, and code-tester, each with a clear role in transforming OpenAPI specs into functional Docker-deployable microservices [5].
- **Pipeline Architectures:** Tasks are processed sequentially by chaining agents together in a fixed order. Each agent transforms its input and passes the result to the next stage. This architecture ensures modularity and fault isolation but is less adaptive to dynamic workflows unless augmented with feedback loops.
- **Decentralized or Peer-to-Peer Architectures:** These allow agents to operate and communicate without centralized control. Coordination emerges through communication protocols or shared memory. Such architectures have been explored in cooperative problem-solving contexts like γ-Bench and DyLAN [2], [3], though they are less common in deterministic tasks like code generation where sequential dependencies are high.
- **Reflective or Self-Improving Architectures:** Inspired by frameworks like Reflexion [11], these agents incorporate feedback and memory modules to refine their own reasoning over time. When incorporated into multi-agent systems, they allow agents to revise outputs based on logs, execution failures, or human feedback

The above figure 2 is a graphical representation of various types of architectures designs.

C. Design Principles of Multi-Agent Architectures

The construction of robust AI agent architectures for code generation relies on several key design principles:

- **Modularity:** Each agent should handle a logically distinct task (e.g., code synthesis, testing, log parsing), allowing independent development, testing, and optimization. It exemplifies this by isolating server code generation, API testing, and error fixing into separate agents with well-defined function-calling interfaces. [5]
- **Task Specialization:** Agents must be tailored to their specific roles. For instance, in the system proposed in [5], the spec-generator focuses exclusively on translating high-level natural language prompts into OpenAPI documents, while the json-cleaner ensures well-structured outputs for parsing and execution.
- **Memory Isolation:** Limiting the memory context of each agent to task-relevant information improves performance and avoids token overflow. This also aligns with constraints in LLMs like GPT-4-turbo, which exhibit optimal performance when confined to narrow contexts.
- **Orchestration and Coordination:** Use of a director agent ensures sequencing, resolves dependencies between tasks, and reduces the risk of context misalignment. For example, the code-tester agent in [5] framework interacts with Docker and curl interfaces to test endpoints and retrieve execution logs before handing over responsibility to the code-fixer agent for refinements [3].
- **Resilience and Debugging:** Logs from container execution and runtime errors are analyzed in iterative cycles by agents like code-fixer, ensuring that faulty code is not just flagged but automatically revised. This introduces semi-autonomous, closed-loop correction cycles into the development workflow. [5]

D. Technological Advancements and Systems

Recent technological milestones have enabled the proliferation of practical multi-agent code generation systems. Key developments include:

- **Function-calling APIs in LLMs:** The ability to execute structured commands via tools like OpenAI's function-calling interface allows agents to trigger file writes, container executions, or log retrievals [3]. This bridges the gap between natural language generation and actual system interaction.

- **Integration with DevOps Toolchains:** Systems like the one in [3] integrate directly with Docker, LeetCode, and logging systems, enabling full-cycle code generation, testing, and debugging within a single environment.
- **Consensus-Driven Agent Interactions:** Projects like DyLAN and SMART-LLM demonstrate how multi-agent LLMs can collectively reason, negotiate, and refine outputs for complex reasoning tasks, including code generation [4], [6].

These advancements validate the architectural importance of AI agent design and coordination in achieving scalable, fault-tolerant, and production-ready code generation workflows.

IV. Datasets and Benchmarks

Datasets and benchmarks form the cornerstone of empirical research in automatic code generation. They provide standardized, reproducible means to train, test, and evaluate the performance of language models and multi-agent systems in generating functionally correct, efficient, and secure code. The diversity, complexity, and labeling quality of these datasets significantly influence the capabilities and generalization of AI-driven code generation systems. [6]

In multi-AI agent architectures, datasets play an expanded role. Not only are they used to pre-train or fine-tune models on code generation tasks, but they also serve to evaluate inter-agent communication, debugging accuracy, and iterative refinement processes. Therefore, selecting appropriate datasets and designing robust benchmarks are critical to advancing the field.[6]

A. Prominent Datasets in Code Generation

Numerous datasets have been curated to support a wide range of code generation scenarios—from algorithmic problem solving to security vulnerability detection. Key examples include:

- **LeetCode Algorithm Problems**: Contains over 2,500 problems in five programming languages. [1] sampled 728 problems to assess ChatGPT's generalization and found higher success rates on pre-2021 data, indicating potential overfitting.
- **Common Weakness Enumeration (CWE) Code Scenarios**: Derived from MITRE's repository, used for vulnerability-focused code generation. [1]. showed over 89% of flaws could be fixed with multi-round ChatGPT interactions, analyzed via CodeQL.
- **HumanEval and MBPP**: HumanEval includes 164 tasks, while MBPP provides 974 Python problems. Both support functional correctness assessment in few-shot settings[6][7]
- **CodeXGLUE and CONCODE**: Large-scale benchmarks for tasks like code completion and NL-to-code translation. CONCODE focuses on Java classes and GitHub documentation[12]
- **APIBank and CodeSearchNet**: APIBank evaluates API-level reasoning. CodeSearchNet spans 2 million code snippets in six languages, aiding model pretraining [4]

B. Benchmarks and Evaluation Metrics

To assess multi-agent LLM systems, both static and execution-based benchmarks are used:

- **Accepted Rate (A. Rate)**: Measures the percent of code submissions passing all test cases [1]
- **Status Rate (SR)**: Classifies outcomes into Accepted, Wrong Answer, Compile Error, Time Limit Exceeded, and Runtime Error, highlighting language-dependent variations.
- **Cyclomatic and Cognitive Complexity**: Evaluates code maintainability. ChatGPT often increases complexity in refined outputs.
- **BLEU, CodeBLEU, and Exact Match (EM)**: Text similarity metrics, with CodeBLEU incorporating syntax and semantics [3]
- **Security and Vulnerability Detection**: Tools like CodeQL assess flaw presence. It shows agent-based repairs enhance code safety. [5]

C. Relevance to Multi-AI Agent Architectures

For multi-agent systems, benchmarks must evaluate not only functional correctness but also:

- **Agent Coordination Accuracy**: Evaluates context and result sharing among agents.
- **Round-wise Fix Success Rate**: Tracks repair efficacy across iterations using dialog history.
- **Tool Invocation Success**: Measures accuracy in using tools like Docker and curl through APIs. [5]

V. Applications of AI Agents

The advent of AI agents, particularly those based on Large Language Models (LLM),has significantly reshaped the landscape of automatic code generation. Within multiagent architectures, these agents are assigned well-defined roles, enabling them to collaborate, coordinate, and complete complex software engineering tasks with improved accuracy and scalability. From code synthesis and specification generation to runtime testing and iterative bug fixing, AI agents are increasingly deployed as modular components in intelligent development pipelines. [5]

A. Code Specification and API Design

One of the most impactful applications of AI agents in code generation is the automated creation of API specifications from user-provided requirements. Through natural language inputs, agents such as spec-generator translate service descriptions into OpenAPI-compliant YAML files. This streamlines the API-first development methodology and enables early validation of interfaces across distributed teams. In a multi-agent framework, this functionality forms the foundation for subsequent service generation stages, ensuring consistency across all modules [5]

B. Server-Side Code Generation

Once the specification is defined, a code-generator agent interprets the API structure and outputs corresponding backend implementations. For example, in [5], the code-generator agent produces JavaScript/Express.js server code organized in a predefined folder structure. A json-cleaner agent ensures the output remains syntactically valid before writing the code to the developer's local environment. This modularization simplifies the deployment process and ensures that the generated code adheres to organizational standards.

C. Error Detection and Debugging via Multi-Round Interaction

A hallmark feature of AI agents in code generation is their ability to participate in iterative refinement cycles. When errors are detected, either through failed test cases or log analysis—agents such as code-tester and code-fixer intervene to correct faults. Liu et al[1] demonstrated that although initial outputs from ChatGPT may be flawed, multiround dialog-based interactions can resolve up to 89% of vulnerabilities. These agents use run-time logs, compiler messages, and execution outcomes to isolate the root causes of errors and autonomously propose or implement fixes. [1]

D. Cross-Language and Multimodal Code Generation

In polyglot development environments, AI agents play a vital role in translating code between languages and adapting templates for multiple stacks. Agents trained on multilingual datasets can generalize solutions and apply design patterns across diverse syntactic domains. This has implications for legacy modernization and integration of heterogeneous systems, especially in enterprise-scale applications.

E. Testing, Execution, and Deployment Orchestration

Beyond code writing, AI agents are increasingly involved in operational workflows. In the system proposed by [5], a code-tester agent interacts with Docker to launch services, validate functionality, and monitor logs - all via LLM-driven command execution. This extends the capability of AI agents from the generation of simple content to real-time orchestration, enabling "full-stack" automation under human oversight.

F. Real-World Implications and Industry Adoption

AI agents embedded in multi-agent frameworks are already demonstrating their value in accelerating development, reducing tool-switching, and improving prototyping agility. According to survey results presented in [3], practitioners highlighted the system's ability to consolidate the development workflow—minimizing reliance on multiple tools such as Postman, IDEs, and Docker CLI—and praised its intuitive debugging interface.

Moreover, industry trends suggest that modular agent-based LLM systems are increasingly favored for domain-specific adaptation, supporting software delivery in healthcare, finance, and embedded systems. The ability to plug in customized agents—for example, ones tuned for specific security protocols or compliance rules—further enhances their appeal.

G. Future Directions

The next phase of research is expected to expand the cognitive capabilities of agents through:

- **Personalization**: Adapting agent behavior based on user preferences or project history.
- **Federated Collaboration**: Enabling agents to share learning across projects while preserving data privacy.
- **Explainability**: Making agent decisions interpretable to human developers for trust and validation.
- **Safety and Security Enforcement**: Integrating formal verification agents to validate code correctness and adherence to best practices.

As these capabilities evolve, AI agents will transition from passive assistants to proactive development collaborators capable of managing entire software lifecycles within orchestrated, intelligent environments. [11]

VI. Evaluation Metrics

Evaluation metrics are critical to objectively assess the performance, reliability, and practical utility of Large Language Models (LLMs) and AI agents in code generation systems. These metrics offer a standardized methodology for comparing various models, architectures, and agent workflows, enabling researchers and practitioners to quantify improvements in functional correctness, generation quality, execution efficiency, and adaptability across tasks.

In multi-agent settings, evaluation goes beyond simple output correctness to include inter-agent coordination, multi-round success rates, and runtime behavior. This broader scope necessitates the application of both traditional NLP-inspired metrics and domain-specific software engineering benchmarks.

A. Conventional Metrics

- **BLEU Score**: Measures n-gram overlap between generated and reference code. While widely used, it may not reflect semantic correctness in code.
- **Exact Match (EM)**: Assesses token-level identity with a reference solution. Effective in constrained tasks like HumanEval, but limited in open-ended generation.
- **CodeBLEU**: Enhances BLEU by incorporating syntax trees, data flow, and semantic similarity for code-specific comparison [8].
- **Accuracy**: Evaluates the proportion of entirely correct outputs against a defined test suite—commonly used in algorithmic challenges like LeetCode.
- **Precision, Recall, and F1-Score**: Applied in classification-oriented tasks such as static bug detection. Precision measures correctness of positive predictions, recall measures completeness, and F1 is their harmonic mean.

B. Execution-Based Evaluation

Execution-based evaluation provides functional validation of generated code in runtime environments:

- **Pass@k**: Estimates the probability that at least one of k generated samples passes all test cases. Useful in stochastic or sampling-based generation [2]
- **Acceptance Rate (A. Rate)**: Represents the proportion of code submissions that pass all test cases. [1] found a 48.14% A. rate difference between pre- and post-2021 LeetCode problems.
- **Status Rate (SR)**: Categorizes outcomes into Accepted (A), Wrong Answer (W.A.), Compile Error (C.E.), Run-time Error (R.E.), and Time Limit Exceeded (T.L.E.), providing deeper performance insights[4].
- **Multi-Round Fix Success Rate**: Tracks the percentage of initially faulty code successfully corrected across multiple agent refinement cycles. In [3] , 25/157 pairs were fixed, with an average of 1.32 rounds per fix.

C. *Complexity and Security Analysis*

- **Cyclomatic and Cognitive Complexity**: Measure logic density and code understandability. Liu et al. noted increased complexity during multi-round fixes, potentially affecting maintainability.
- **Code Vulnerability Rate**: Used in CWE-based evaluations to detect flaws such as buffer overflows or null dereferencing. Tools like CodeQL automate detection. Liu et al. report over 89% of vulnerabilities were resolved via dialog-based agent refinement. [2]

D. *Human-Centric Evaluation*

- **Usability and Developer Satisfaction**: Qualitative assessments based on developer feedback gauge productivity, tool-switching reduction, and debugging ease. Chauhan et al. reported improved prototyping and reduced dependency on external tools like Postman and Docker CLI.

TABLE I: Evaluation Metrics in Code Generation Systems

Metric	Description
BLEU Score	N-gram overlap between generated and reference code
Exact Match (EM)	Token-level match with reference solution
CodeBLEU	BLEU + syntax tree, data flow, and semantic similarity
Accuracy	Percentage of completely correct outputs
Precision	Correctness of positive predictions
Recall	Completeness of predictions
F1 Score	Harmonic mean of Precision and Recall
Pass@k	Probability that at least one of k samples passes all test cases
Acceptance Rate (A. Rate)	Proportion of submissions passing all test cases
Status Rate (SR)	Distribution across A, W.A., C.E., R.E., T.L.E.
Multi-Round Fix Success Rate	% of initially faulty code corrected in multiple rounds (e.g., 25/157 = 15.9%)
Cyclomatic/Cognitive Complexity	Measures of logic density and code understandability
Code Vulnerability Rate	Percentage of code with resolved vulnerabilities (e.g., >89%)

Table 1 is a summary of all the metrics that can be used for checking the code quality generated by an LLM agent to ensure standards.

VII. Challenges and Limitations

Despite the promising advances enabled by multi-AI agent architectures in code generation, several technical, ethical, and practical challenges persist. These challenges affect not only the design and deployment of such systems but also their scalability, reliability, and acceptance in industrial settings. Addressing these issues is essential to ensure that agent-based frameworks can evolve into robust, production-ready solutions.

A. *Technical Limitations*

Computational Complexity: Multi-agent systems often involve several concurrently active LLMs, each responsible for distinct subtasks—such as specification parsing, code generation, testing, and error correction. While this decomposition promotes modularity, it significantly increases computational load. Inference across multiple agents—especially using high-parameter models like GPT-4—leads to elevated latency and resource consumption, making such systems impractical for real-time or edge deployments [4].

Scalability and Coordination Overhead: As the number of agents increases, maintaining coherence, synchronization, and memory sharing becomes more complex. Coordination overhead can result in redundant computations and inconsistent outputs, particularly when agents lack shared context or robust communication protocols. This is especially challenging in decentralized settings.

Accuracy and Error Propagation: Although LLMs demonstrate strong generative ability, they are prone to hallucinations and logical inconsistencies. In multi-agent workflows, an error from one agent (e.g., an incorrect specification) can cascade downstream, compounding failures. While iterative refinement can mitigate some issues, the lack of formal verification limits recovery effectiveness [4]

Tool Integration and Execution Robustness: LLM-driven agents often interact with external environments (e.g., Docker, filesystem, logs). Ensuring secure, deterministic, and reliable interactions with such tools remains a significant challenge. Failures may arise due to API mismatches, permission errors, or execution faults—necessitating robust fail-safes and roll-back strategies.

B. *Ethical and Practical Constraints*

Data Privacy and Information Leakage: LLMs frequently access vast codebases, potentially containing sensitive or proprietary data. In multi-agent systems, data leakage risks increase if sensitive context is inadvertently passed between agents or exposed via logs, especially in regulated domains.

Security and Trust in Generated Code: Generated code may contain vulnerabilities if not thoroughly validated. [1] observed that many code snippets exhibited CWE-class flaws. Although multi-round refinement addressed over 89% of issues, the absence of secure coding guarantees or formal verification remains a critical concern.

Deployment Barriers and User Adoption: Multi-agent frameworks often require complex setup—spanning agent orchestration, tool integration, and environment consistency. These challenges may deter adoption among non-experts or DevOps-constrained teams. As noted in [3], some practitioners expressed concerns about customization burdens for large-scale deployments.

Human Oversight and Explainability: Although LLM agents can operate autonomously, effective deployment requires human-in-the-loop monitoring. The lack of transparency in agent reasoning makes it difficult for developers to validate decisions or debug failures. Enhanced explainability and communication traceability remain open research needs. [8]

C. Future Considerations

To address the aforementioned limitations, future research should explore:

- Efficient LLM compression and distillation techniques for resource-constrained multi-agent applications;
- Robust multi-agent orchestration protocols with contextual memory sharing and conflict resolution;
- Secure function-calling APIs with access control and sandboxing for deterministic tool integration;
- Formal verification layers embedded within agent workflows to validate correctness and security.

The integration of traditional Multi-Agent Decision-Making (MADM) techniques with LLM-based agents is a promising direction, offering hybrid frameworks that combine the resilience of classical systems with the generative flexibility of modern AI [1]

VIII. Future Directions

The fusion of Large Language Models (LLMs) and multi-agent architectures for code generation has demonstrated substantial promise in accelerating and automating software development workflows. Yet, the field remains in its nascent stage, presenting fertile ground for innovation. Future research is poised to enhance the adaptability, transparency, and robustness of such systems, enabling them to handle more complex tasks across broader domains. [7]

A. Enhancing Scalability and Efficiency

As multi-agent systems evolve, addressing scalability without sacrificing performance becomes essential. Current implementations often rely on resource-intensive LLMs that pose computational challenges, particularly when multiple agents operate concurrently. Future systems must incorporate model optimization techniques such as quantization, knowledge distillation, and pruning to allow agent deployment on edge or resource-constrained environments [8].

Moreover, developing dynamic task orchestration frameworks will be critical to minimizing redundant operations and enabling adaptive agent activation based on task complexity. Incorporating scheduling policies and runtime resource monitoring will allow intelligent load balancing and improve throughput in distributed development environments.

B. Agent Specialization and Personalization

Future architectures will support domain-specific agents tailored for verticals such as embedded systems, financial modeling, or safety-critical software. These agents may be fine-tuned on specialized datasets and constrained by domain-specific coding standards.

Personalized agents that adapt to developer preferences, project-specific logic, and coding conventions will improve usability. Integrating feedback loops where agents learn from human corrections will drive adoption and enhance collaboration.

C. Advanced Coordination and Memory Sharing

Improved inter-agent communication is critical for coherent execution. Future systems may use shared memory vectors, messaging protocols, and context synchronization strategies to reduce redundancy and support collaborative refinement. [9]

Multi-agent Reinforcement Learning (MARL) and Decentralized POMDPs may offer formalism for training cooperative agent teams to solve code generation tasks efficiently [10]

D. Human-in-the-Loop and Explainability

Explainable AI (XAI) techniques will be essential to boost transparency and trust. Agents should be able to justify their actions with step-by-step explanations, especially during tasks like debugging or refactoring.

Flexible Human-in-the-Loop (HITL) frameworks will allow developers to dynamically provide feedback, intervene in workflows, and override decisions as needed without resetting pipelines [10].

E. Security and Ethical Safeguards

As LLM-generated code reaches production, ensuring security and ethical alignment is vital. Future systems must integrate automated vulnerability analysis tools (e.g., CodeQL), implement privacy-preserving training methods, and promote safe deployment practices.

Preventing model bias, ensuring fair use, and maintaining transparent governance will be critical to gaining industry and public trust [10].

F. Convergence with Multimodal and Embodied AI

The next generation of systems will integrate multimodal AI agents capable of processing text, diagrams, GUI mockups, and runtime traces. Such capabilities will enhance agents' situational awareness and adaptability.

Systems like CoELA and Co-NavGPT [4] have already demonstrated the value of combining visual and language modalities for embodied agent coordination.

Additionally, autonomous software factories may emerge, in which agents independently plan, generate, test, and deploy services within adaptive CI/CD pipelines, responding to real-time logic and environmental feedback. [12]

G. Research Collaboration and Open Evaluation

To promote replicability and community progress, standardized benchmarks, open datasets, and shared evaluation frameworks are required. Open-source platforms for modular agent development will support collaborative experimentation and accelerate innovation.

IX. IX. Conclusion

This survey has presented a comprehensive analysis of code generation through the lens of multi-AI agent architectures, highlighting the growing role of Large Language Models (LLMs) when embedded in modular, role-specific agent systems. By exploring current methodologies, system architectures, datasets, evaluation metrics, and practical implementations, we underscore the transformative potential of multi-agent paradigms in automating and accelerating software development workflows.

Key findings of this survey reveal that:

- Multi-agent systems decompose complex coding tasks into manageable, coordinated roles such as specification generation, code synthesis, testing, and iterative refinement.
- LLMs, when integrated into agent frameworks, enable dynamic, language-based reasoning and execution across various stages of development.
- Empirical studies confirm that iterative, dialog-driven error correction, collaborative task planning, and memory-contextualized decision-making significantly improve code quality and developer efficiency.
- Evaluation frameworks using acceptance rates, execution status, vulnerability detection, and multi-round fix success rates are essential to benchmark system reliability.

”We provided a brief overview of Multi-Agent Decision-Making. . . Based on these works, we summarized potential future research directions. . . With LLMs, designing MADM frameworks becomes more analogous to modeling the group learning process of animals or even humans” [8]

The implications for current practice include improved productivity, reduction in tool-switching overhead, and democratization of development through conversational interfaces. Additionally, multi-agent systems hold promise for advancing CI/CD pipelines, cross-platform interoperability, and real-time collaboration between developers and AI systems.

Nonetheless, challenges persist in areas such as computational efficiency, coordination overhead, explainability, and security. These gaps call for future research in model compression, multi-modal coordination, secure function invocation, and human-in-the-loop frameworks. Ethical considerations—particularly in data privacy, AI alignment, and transparency—must also be addressed to ensure responsible deployment.

Future directions identified in this paper, such as scalable orchestration frameworks, domain-specific agent fine-tuning, and integration of verification mechanisms, offer a roadmap for next-generation research. Moreover, the convergence of multi-agent AI with embodied systems and secure DevOps environments is likely to expand the scope of applications beyond static code generation into dynamic, end-to-end software engineering.

In conclusion, code generation using multi-AI agent architectures marks a pivotal shift in the intersection of artificial intelligence and software development. By leveraging the specialization, adaptability, and interactivity of agent-based models, this approach lays the groundwork for intelligent, collaborative, and autonomous development systems that are poised to redefine the future of programming.

References

[1] M. W. U. Rahman *et al.*, “Multi-Agent Actor-Critic Generative AI for Query Resolution and Analysis,” *IEEE Transactions on Artificial Intelligence*, 2025.

[2] C. Sun, S. Huang, and D. Pompili, “LLM-Based Multi-Agent Decision-Making: Challenges and Future Directions,” *IEEE Robotics and Automation Letters*, vol. 10, no. 6, pp. 5681–5688, 2025.

[3] Z. Liu *et al.*, “No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT,” *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1548–1584, 2024.

[4] V. S. Jagtap, “Use of Artificial Intelligence in Software Development Life Cycle: A State of the Art Review,” *IJACECT*, vol. 3, no. 3, pp. 398–403, 2015.

[5] S. Chauhan *et al.*, “LLM-Generated Microservice Implementations from RESTful API Definitions,” in *Proc. 20th Int. Conf. Evaluation Novel Approaches Software Engineering (ENASE)*, pp. 161–173, 2025.

[6] J. Cruz-Benito *et al.*, “Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches,” *AI*, vol. 2, no. 1, pp. 1–16, 2021.

[7] H. Alkaissi and S. I. McFarlane, “Artificial hallucinations in ChatGPT: Implications in scientific writing,” *Cureus*, vol. 15, no. 2, 2023.

[8] M. Dudjak and G. Martinovic´, “An API-first methodology for designing a microservice-based backend as a service platform,” *Information Technology and Control*, vol. 49, no. 2, pp. 206–223, 2023.

[9] M. W. U. Rahman *et al.*, “A BERT-based deep learning approach for reputation analysis in social media,” in *Proc. IEEE/ACS 19th Int. Conf. Computer Systems and Applications (AICCSA)*, Abu Dhabi, UAE, pp. 1–8, 2022.

[10] C. Sun, S. Huang, and D. Pompili, “HMAAC: Hierarchical multi-agent actor-critic for aerial search with explicit coordination modeling,” in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 7728–7734, 2023.

[11] S. I. Adam, J. H. Moedjahedy, and J. Maramis, “RESTful web service implementation on unklab information system using JSON Web Token (JWT),” in *Proc. 2nd Int. Conf. Cybernetics and Intelligent System (ICORIS)*, pp. 1–6, 2020.

[12] N. Beaulieu, S. M. Dascalu, and E. Hand, “API-first design: A survey of the state of academia and industry,” in *Proc. 19th Int. Conf. Information Technology-New Generations (ITNG)*, pp. 73–79, 2022.

[13] R. E. Boyatzis, “Transforming Qualitative Information: Thematic Analysis and Code Development,” *Sage*, 1998.

[14] S. Casas, D. Cruz, G. Vidal, and M. Constanzo, “Uses and applications of the OpenAPI/Swagger specification: A systematic mapping of the literature,” in *Proc. 40th Int. Conf. Chilean Computer Science Society (SCCC)*, pp. 1–8, 2021.

[15] S. Chauhan, “Code for service generation and instructions,” 2024.

[16] R. Chen, S. Li, and Z. Li, “From monolith to microservices: A dataflow-driven approach,” in *Proc. 24th Asia-Pacific Software Engineering Conf. (APSEC)*, pp. 466–475, 2017.

[17] X. Chen, Y. Liao, and W. Yu, “Generative AI in higher art education,” in *Proc. 6th Int. Conf. Computer Science and Technologies in Education (CSTE)*, pp. 135–140, 2024.

[18] P. F. Christiano *et al.*, “Deep reinforcement learning from human preferences,” in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[19] B. De, “API Management: An Architect’s Guide to Developing and Managing APIs for Your Organization,” 2nd ed., Apress, Berkeley, CA, 2023.

[20] S. Ding and V. Raman, “Harness the power of generative AI in healthcare with Amazon AI/ML services,” in *Proc. IEEE 12th Int. Conf. Healthcare Informatics (ICHI)*, pp. 490–492, 2024.

[21] M. Dudjak and G. Martinovic´, “An API-first methodology for designing a microservice-based backend as a service platform,” *Information Technology and Control*, vol. 49, no. 2, pp. 206–223, 2020.

[22] R. Chen, S. Li, and Z. Li, “From monolith to microservices: A dataflow-driven approach,” in *Proc. 24th Asia-Pacific Software Engineering Conf. (APSEC)*, pp. 466–475, 2017.

[23] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” *arXiv preprint arXiv:2406.00515*, 2024.

[24] C. C. Jorge, M. L. Tielman, and C. M. Jonker, “Artificial trust as a tool in human-AI teams,” in *Proc. 17th ACM/IEEE Int. Conf. Human-Robot Interaction (HRI)*, pp. 1155–1157, 2022.

[25] K. S. Kaswan, J. S. Dhatterwal, K. Malik, and A. Baliyan, “Generative AI: a review on models and applications,” in *Proc. Int. Conf. Communication, Security and Artificial Intelligence (ICC-SAI)*, pp. 699–704, 2023.

[26] A. Lercher, “Managing API evolution in microservice architecture,” in *Proc. IEEE/ACM 46th Int. Conf. Software Engineering Companion (ICSE-Companion)*, pp. 195–197, 2024.

[27] N. Nguyen and S. Nadi, “An empirical evaluation of GitHub Copilot’s code suggestions,” in *Proc. 19th Int. Conf. Mining Software Repositories*, pp. 1–5, 2022.

[28] L. Olson, “Custom developer GPT for ethical AI solutions,” in *Proc. IEEE/ACM 3rd Int. Conf. AI Engineering – Software Engineering for AI (CAIN)*, pp. 282–283, 2024.

[29] Yashwanth, Y. S., and Rajashree Shettar. "Zero and few short learning using large language models for de-identification of medical records." *IEEE Access* 2024.

[30] Darshan, P., Mandahas, N., MP, P. R., & Rajesh, R. N. (2024, November). Leveraging llm and rag for automated answer script evaluation. In *8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS)* (pp. 1-5). IEEE, 2024.

[31] Bagawan, Sohail, and G. S. Nagaraja. "Develop an On-Device LLM Android Application." *2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS)*. IEEE, 2024.

[32] Meghasree, V., CK Nagendra Guptha, and M. N. Vijayakumar. "Artificial Intelligence Driven e-Commerce Platform for Handicraft Toy Industries." *2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS)*. IEEE, 2024.