NEXT-GENERATION SOFTWARE ENGINEERING: A MULTI-AGENT SYSTEM FOR END-TO-END SDLC AUTOMATION USING LARGE LANGUAGE MODEL

Mr.Periyasamy.T
Department of Information Technology
Sri Manakula Vinayagar Engineering College
Puducherry,India
periyasamy2204@gmail.com

Anburaj J
Department of Information Technology
Sri Manakula Vinayagar Engineering College
Puducherry, India
anbu48088@gmail.com

Fyzal K
Department of Information Technology
Sri Manakula Vinayagar Engineering College
Puducherry India
fyzalfaz623@gmail.com

Jayakrishnan B
Department of Information Technology
Sri Manakula Vinayagar Engineering College
Puducherry India
krish.infowork@gmail.com

Prasanth S
Department of Information Technology
Sri Manakula Vinayagar Engineering College
Puducherry India
prasanth500981@gmail.com

ABSTRACT

Software development lifecycle automation refers to the orchestration of multiple AI agents to autonomously handle complex development tasks, which are commonly required for building and deploying modern applications. The proposed system represents a significant leap forward in the ongoing battle against software development complexity, addressing a critical challenge inherent in existing approaches. This challenge lies in the inability of current systems to coordinate multiple specialized agents across planning, coding, testing, and deployment phases while maintaining context coherence and enabling human oversight. The proposed system offers a groundbreaking solution to combat development inefficiency by leveraging the LangGraph workflow orchestration algorithm, addressing a critical limitation in existing approaches. Unlike conventional single-agent methods, our system integrates a multi-agent architecture with specialized roles, reducing development time while maintaining code quality and accuracy. This innovative approach tailored for end-to-end software automation has shown remarkable performance improvements in rigorous comparative testing, surpassing current systems in task coordination and completion efficiency. The integration of Human-in-the-Loop checkpoints enhances the system's robustness, providing proactive intervention capabilities against the dynamic nature of autonomous decision-making errors. Overall, our pioneering system not only enhances automation measures against development complexity but also introduces a more streamlined and responsive solution to combat the evolving demands of software engineering workflows. This advancement signifies a significant stride forward in AI-assisted development, offering an effective framework against the continuously changing landscape of modern software development challenges.

Keywords : LangGraph, Multi-Agent Orchestration, Human-in-the-Loop (HITL), SDLC Automation

I. INTRODUCTION

Software development lifecycle complexity refers to the growing challenge of managing multiple interconnected phases required for building, testing, and deploying modern applications. Development teams may struggle with coordination inefficiencies or face bottlenecks when handling sequential tasks across planning, coding, review, and deployment stages. This complexity can take various forms, such as context switching overhead, communication gaps, or delayed feedback loops, aiming to slow down delivery timelines and reduce overall productivity. The development team initiates a seemingly straightforward project, and the complexity compounds, leading to potential delays, quality issues, or project failure. To combat software development complexity, advanced automation techniques, such as multi-agent orchestration and graph-

based workflow execution, are increasingly employed to coordinate and streamline tasks within these projects, enhancing overall development efficiency.

A. MULTI-AGENT SYSTEMS

Multi-agent systems, a paradigm in artificial intelligence, refer to a category of software architectures intentionally designed to coordinate multiple autonomous agents to accomplish complex tasks that exceed the capabilities of individual agents. This umbrella term encompasses a wide range of agent configurations created with the intent of collaborating, delegating, or distributing workloads across specialized components. Multi-agent systems can manifest in various forms, including hierarchical agents, peer-to-peer agents, supervisor-worker patterns, swarm intelligence, and federated learning architectures. Development teams deploy multi-agent systems through orchestration frameworks, often using workflow graphs, message passing, or shared state management to coordinate and synchronize the behavior of multiple agents. Once configured, multi-agent systems can execute a variety of collaborative activities, such as parallel task processing, specialized role delegation, or providing coordinated responses to complex queries. Implementing multi-agent systems involves employing robust orchestration measures, including state management, agent registries, and communication protocols, to coordinate, monitor, and optimize the performance of these collaborative architectures on software development workflows.

II. RELATED WORK

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

In modern software engineering, the growing complexity of systems and development workflows has intensified the need for intelligent automation across the software development lifecycle (SDLC). This paper presents a comprehensive study of Large Language Model (LLM)-based Multi-Agent (LMA) systems as an emerging paradigm for addressing complex software engineering challenges. The authors conduct a systematic review of recent primary studies to analyze how collaborative LLM-based agents are applied across key SDLC phases, including requirements engineering, code generation, quality assurance, maintenance, and end-to-end software development. To assess practical feasibility, two case studies using the state-of-the-art ChatDev framework are performed, demonstrating the strengths and limitations of current LMA systems in autonomously developing software applications. The study highlights that while LMA systems show strong performance in moderately complex tasks with low cost and rapid execution, they still struggle with deeper reasoning and scalability for highly complex projects. Based on these findings, the authors identify critical research gaps and propose a two-phase research agenda focused on enhancing individual agent capabilities and optimizing agent collaboration and synergy. Overall, this work establishes a structured foundation and forward-looking vision for advancing LLM-based multi-agent systems toward scalable, autonomous, and trustworthy Software Engineering 2.0.

[2] Guidelines for Future Agile Methodologies and Architecture Reconciliation for Software-Intensive Systems

In software-intensive systems, achieving a balance between rapid development and sound architectural design remains a persistent challenge. Agile methodologies emphasize speed, adaptability, and minimal documentation, while software architecture requires deliberate planning and documentation to ensure system quality, maintainability, and long-term evolution. This paper investigates the long-standing tension between agility and architecture and argues that this perceived conflict represents a false dichotomy. Through an extensive literature review grounded in General Systems Theory, the authors analyze the historical evolution of both agile and architecture-centric approaches and examine prior attempts to reconcile them. Based on this analysis, the study proposes a structured set of guidelines to support future agile methodologies while preserving essential architectural practices. These guidelines emphasize lightweight architectural documentation, stakeholder collaboration, incremental architectural planning, and continuous evaluation of architectural quality. The paper concludes that software architecture is not an impediment to agility but rather an enabler, providing stability and guidance in highly dynamic development environments. This work offers practical direction for organizations seeking to integrate agile practices with architectural rigor in modern software-intensive systems**.**

[3] Transpiler-Based Architecture Design Model for Back-End Layers in Software Development

In software development, supporting multiple programming languages and deployment platforms often leads to increased development cost, maintenance complexity, and architectural rigidity. This paper introduces a transpiler-based software architecture design model that centralizes back-end development around a single transpilable source code, which can be automatically transformed into multiple equivalent back-end implementations across different programming languages and execution platforms. The proposed model integrates transpilers as a core architectural element and defines a layered design encompassing development, preparation, execution (nest implementations), and presentation stages. It also incorporates automated code generation, native

compilation, proxy generation, and cross-cutting components such as security, configuration, and logging. To validate the feasibility of the approach, an empirical experiment is conducted through the implementation of a collaborative to-do list application, demonstrating reduced duplication effort and improved technological flexibility. The results indicate that the proposed model enables language interoperability, extends software lifespan, and reduces long-term dependency on specific technologies, while introducing manageable complexity in build and deployment processes. This work contributes a novel architectural paradigm for designing multi-programming-language back-end systems, addressing scalability, maintainability, and adaptability challenges in modern software ecosystems.

[4] ALMAS: An Autonomous LLM-Based Multi-Agent Software Engineering Framework

The increasing adoption of Large Language Models (LLMs) in software engineering has led to significant progress in tasks such as code generation, testing, and maintenance; however, most existing solutions remain fragmented and focus narrowly on isolated coding activities. This paper introduces ALMAS, an autonomous LLM-based multi-agent software engineering framework designed to support the entire software development lifecycle (SDLC) in alignment with agile development practices. ALMAS organizes specialized agents according to real-world agile roles, including sprint planning, development, code review, and supervision, enabling coordinated end-to-end automation of software projects. A key contribution of the framework is its context-aware design, which employs dynamic code summarization and a Meta-RAG retrieval strategy to mitigate LLM context window limitations and attention degradation in large codebases. Furthermore, ALMAS incorporates cost-efficient resource allocation by routing tasks to appropriately sized LLMs based on task complexity. The framework supports both autonomous execution and interactive collaboration with human developers, integrating seamlessly with common SDLC tools such as Jira and Bitbucket. A case study demonstrates ALMAS's ability to generate a complete application and subsequently augment it with new features, highlighting its effectiveness in planning, implementation, verification, and iterative development. Overall, ALMAS represents a significant step toward integrated, scalable, and cost-effective AI-driven software engineering ecosystems.

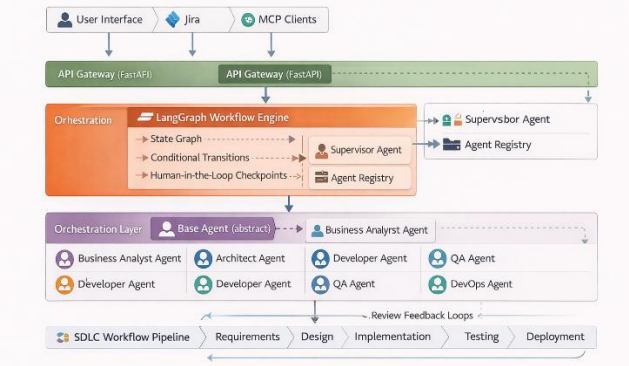
[5] Fully Autonomous Programming Using Iterative Multi-Agent Debugging with Large Language Models

Program synthesis using Large Language Models (LLMs) often suffers from the near-miss syndrome, where generated code appears correct but fails unit tests due to minor logical or syntactic errors. This paper proposes SEIDR (Synthesize, Execute, Instruct, Debug, and Rank), a fully autonomous multi-agent framework that integrates LLM-based code generation with iterative debugging and evolutionary search principles. SEIDR decomposes program synthesis into specialized agents that generate candidate solutions, execute them against validation tests, produce natural-language debugging instructions, repair faulty programs, and rank candidates using selection strategies such as tournament and lexicase selection. Extensive experiments on the PSB2 and HumanEval-X benchmarks in both Python and C++ demonstrate that SEIDR consistently outperforms standalone LLM generation and traditional genetic programming approaches. The results show that balancing program repair and replacement is crucial for overcoming near-miss failures, with moderate tree-arity configurations yielding the best performance. Overall, the study establishes SEIDR as an effective and scalable approach for fully autonomous programming, advancing the reliability and generalizability of LLM-based software synthesis.

III. PROPOSED METHOD

In the proposed system for automating software development lifecycle, LangGraph workflow orchestration and Multi-Agent coordination algorithms play a pivotal role in the execution process. LangGraph and Multi-Agent architectures are specialized orchestration frameworks designed to effectively capture and manage sequential development tasks, making them particularly well-suited for workflows involving phase dependencies and state management. By incorporating these advanced algorithms into the system, the model gains the ability to learn and adapt to the dynamic patterns characteristic of software development workflows over time. LangGraph, with its graph-based connections, can retain information from previous workflow steps, allowing the model to capture long-range dependencies within the sequential development phases. Multi-Agent networks, on the other hand, excel in delegating and coordinating tasks across specialized roles, making them effective in recognizing and handling complex requirements indicative of diverse development needs. By leveraging the strengths of LangGraph and Multi-Agent orchestration, the proposed system enhances its ability to accurately coordinate and execute development tasks, thereby bolstering automation efficiency against evolving challenges in the software engineering landscape.

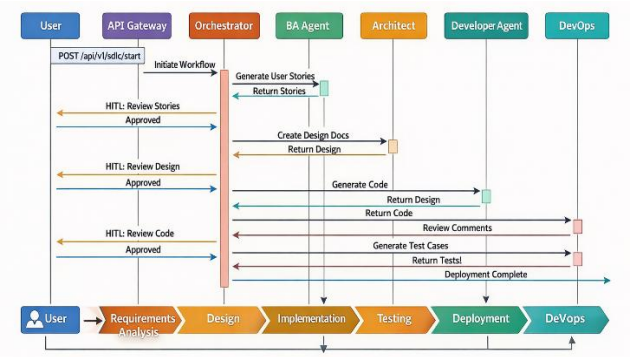
ARCHITECTURE DIAGRAM



The architecture diagram illustrates the structural components and flow of information within the backend processing system. The API Gateway receives incoming requests through FastAPI and routes them to specialized endpoints for SDLC operations, Jira integration, and MCP protocols, serving as the primary entry point for all external communications. The Orchestration Layer manages workflow execution through two parallel mechanisms: LangGraph state graphs handle sequential workflow transitions with conditional edges and Human-in-the-Loop checkpoints enabling human oversight at critical decision points, while the Multi-Agent Orchestrator coordinates specialized agents through a supervisor agent pattern and centralized agent registry for dynamic agent discovery and load balancing. The Specialized Agents Layer contains domain-specific agents including Business Analyst for requirements gathering and user story generation, Architect for design document creation and technical decisions, Developer for code generation and implementation, QA for test case creation and quality assurance, and DevOps for deployment automation, all inheriting from a common Base Agent abstract class that provides standardized communication protocols and task execution interfaces. The SDLC Workflow processes sequential development phases from requirements through deployment with review checkpoints at each stage ensuring quality gates, where each phase can loop back for revisions based on feedback before proceeding to the next stage. This visual representation provides a comprehensive overview of the backend processing design and operation, aiding in understanding its functionality, dependencies, and relationships between different components.

A. INPUT PROCESSING

The input for this project is sourced from user-provided requirements submitted through the API Gateway or user interface components. Specifically, the system accepts project names and requirement specifications as the foundation for initiating the automated SDLC workflow. This input, provided by end users or integrated systems, comprises natural language descriptions of software features, functionalities, and constraints that define the scope of the development project. The inclusion of structured requirement formats is crucial for the system to effectively parse and transform user intent into actionable development artifacts. The input is expected to encompass variations in project complexity, domain terminology, and technical specifications, encapsulating the diverse nature of real-world software requirements. Leveraging this user-provided input is fundamental for driving the multi-agent orchestration process, ensuring that the generated outputs align with user expectations and project objectives. The commitment to accepting flexible input formats underscores the project's dedication to creating an accessible and user-friendly SDLC automation solution.



The workflow visually depicts the transformation of user requirements through multiple processing stages within the automated system. The input flows through the BA Agent for requirement analysis and user story generation, then proceeds to the Architect Agent for design document creation. Subsequently, the Developer Agent generates implementation code, followed by the QA Agent producing test cases and validation reports. Finally, the DevOps Agent handles deployment configurations. This sequential processing ensures that each stage builds upon the validated outputs of the previous phase. Understanding this transformation pipeline informs the system's ability to maintain traceability from initial requirements to final deployment artifacts, thereby ensuring consistency and quality throughout the automated development lifecycle.

B. REQUEST PROCESSING

The request processing phase is a crucial step in preparing the raw input from user-submitted requirements for effective use in the multi-agent orchestration system. This phase involves several key procedures to enhance the quality and suitability of the input data. Initial steps typically include request validation, which addresses any inconsistencies, missing fields, or irregularities in the submitted requirements. Subsequently, the input may be subjected to normalization or standardization to ensure

uniformity in the format of requirement specifications. Schema validation techniques are applied to capture essential characteristics and ensure proper structure indicative of valid project requirements. Additionally, the request is routed through the API Gateway to appropriate workflow handlers based on endpoint specifications. This request processing phase is pivotal for mitigating potential errors, reducing malformed inputs, and optimizing the data for consumption by the orchestration layer. The meticulous handling of input during this stage significantly contributes to the overall effectiveness and reliability of the SDLC automation system.

C. REQUIREMENT PARSING

In the process of requirement parsing, redundant specifications are identified and consolidated to enhance the efficiency and accuracy of the workflow execution. Redundant requirements, when present in the input, can skew the results by duplicating effort across agents or introducing unnecessary complexity in the generated artifacts. By consolidating redundancies, the requirement parsing stage ensures that each unique feature or functionality is appropriately represented and addressed during the SDLC phases. This helps in streamlining the workflow and reducing computational overhead during subsequent stages of artifact generation. Additionally, parsing requirements helps in improving the interpretability of user intent, as it ensures that only relevant and distinct specifications contribute to the final deliverables. Overall, the parsing and normalization of requirements plays a crucial role in producing reliable and meaningful outputs from the system, facilitating more accurate development processes across all SDLC phases.

D. WORKFLOW ORCHESTRATION

The LangGraph workflow orchestration algorithm is a specialized form of state-based graph execution that captures phase dependencies in sequential software development workflows. Unlike traditional linear pipelines, LangGraph employs conditional edge mechanisms to dynamically route execution paths based on review outcomes and workflow state. At each workflow node, the orchestrator receives the current state and outputs from previous phases, and then determines the next execution step using conditional routers and Human-in-the-Loop checkpoints. This design enables effective management of complex development patterns such as requirements gathering, code generation, testing, and deployment automation.

Through its graph-based architecture, LangGraph preserves contextual information across long execution chains, addressing the context loss problem commonly observed in conventional automation pipelines. As a result, relevant artifacts and decisions are continuously propagated across workflow stages, making the approach well-suited for end-to-end SDLC automation.In a LangGraph workflow node, the computation of the next workflow state at phase t involves multiple coordinated components, including a state updater, condition evaluator, feedback handler, and candidate next node selector. These components collectively govern how execution proceeds through the workflow graph:

- The state updater determines how new outputs are incorporated into the existing workflow state.
- The condition evaluator analyzes review status to select the appropriate execution path.
- The feedback handler integrates revision requests into the workflow.
- The candidate next node represents the potential subsequent workflow phase.

The workflow state transition can be expressed as:

$$S_{t+1} = U_t(S_t, O_t)$$

Where:

- S_t represents the current workflow state containing all accumulated artifacts
- O_t represents the output from the current phase execution
- U_t represents the state update function

The conditional routing decision is determined by:

$$N_{t+1} = \begin{cases} \text{next\_phase} & \text{if } C_t(S_t) = \text{approved} \\ \text{revise\_phase} & \text{if } C_t(S_t) = \text{feedback} \end{cases}$$

Here , Ct is the condition evaluation function that inspects the review status stored within the workflow state

These equations govern execution flow within a LangGraph-based workflow, enabling iterative refinement cycles and preserving contextual continuity across extended development phases. Consequently, LangGraph effectively addresses coordination challenges present in traditional automation approaches and enhances the reliability of AI-driven SDLC orchestration.

IV RESULT AND DISCUSSION

Performance metrics are essential tools for assessing the effectiveness of multi-agent orchestration systems. Workflow Completion Rate, the most intuitive metric, provides an overall measure of system reliability by quantifying the proportion of successfully completed SDLC workflows. However, in scenarios with complex requirement specifications, completion rate alone may not adequately capture the system's performance. Agent Efficiency and Phase Success Rate offer more detailed insights by focusing on specific workflow stages. Agent Efficiency measures the proportion of phases completed without revision requests, emphasizing the system's ability to generate high-quality artifacts on the first attempt. Phase

Success Rate quantifies the proportion of approved phases among all phase executions, highlighting the system's ability to meet stakeholder expectations. Response Latency guides optimization of agent coordination by measuring the time between request submission and artifact generation. Evaluating the system using a combination of these metrics provides a comprehensive understanding of performance and suitability for enterprise SDLC automation.

A. WORKFLOW COMPLETION RATE

Workflow Completion Rate measures the proportion of successfully completed SDLC workflows among the total number of workflows initiated by users. A high completion rate indicates effective agent coordination and reliable progression through all SDLC phases.

$$\text{Workflow Completion Rate} = \frac{\text{Successfully Completed Workflows}}{\text{Total Initiated Workflows}} \times 100\%$$

Completion Rate is useful for evaluating end-to-end system reliability. However, it should be interpreted along with other metrics to account for variations in requirement complexity and phase criticality.

Metric	Description	Target
End-to-End Completion	Workflows reaching deployment phase	> 85%
Phase Completion	Individual phase success rate	> 95%
First-Pass Approval	Phases approved without revision	> 70%

B. ARTIFACT GENERATION QUALITY

Artifact Generation Quality evaluates the standard of outputs produced by specialized agents. It examines artifacts based on completeness, consistency, and adherence to predefined templates and best practices. Requirements documents are evaluated for clarity and coverage. Architecture designs are assessed for scalability and maintainability. Generated code is analyzed for correctness and adherence to coding standards. Test cases are measured for coverage and effectiveness. Deployment configurations are checked for reliability and security compliance. This multi-dimensional evaluation ensures that generated artifacts meet technical and stakeholder expectations.

C. AGENT PERFORMANCE METRICS

Agent Efficiency, Response Quality, and Artifact Completeness Score are used to evaluate the performance of individual agents. Agent Efficiency measures the proportion of phase executions completed without Human-in-the-Loop intervention.

$$\text{Agent Efficiency} = \frac{\text{Phases Completed Without HITL}}{\text{Total Phases Executed}} \times 100\%$$

A higher value indicates that the agent can independently generate acceptable artifacts. Response Quality measures the proportion of phase submissions approved on the first attempt.

$$\text{Response Quality} = \frac{\text{First-Pass Approvals}}{\text{Total Phase Submissions}} \times 100\%$$

High Response Quality indicates strong requirement understanding and minimal need for revisions. Artifact Completeness Score provides a combined measure of efficiency and quality.

$$\text{Completeness Score} = \frac{2 \times \text{Efficiency} \times \text{Quality}}{\text{Efficiency} + \text{Quality}}$$

This score reflects the balance between autonomous operation and output quality.

Agent	Efficiency	Quality	Completeness Score
Business Analyst	82%	88%	84.9%
Architect	75%	85%	79.7%
Developer	70%	78%	73.8%
QA Engineer	85%	90%	87.4%
DevOps Engineer	88%	92%	89.9%

Together, these metrics provide actionable insights into each agent’s strengths and areas for improvement.

D. RESPONSE LATENCY ANALYSIS

Response Latency evaluates the real-time performance of the orchestration system by measuring the time between request submission and artifact generation.

Phase	Average Latency	P95 Latency	P99 Latency
Requirements Analysis	2.3 s	4.1 s	6.2 s
Architecture Design	3.8 s	6.5 s	9.1 s
Code Generation	5.2 s	8.7 s	12.4 s
Test Generation	2.8 s	4.9 s	7.3 s

Phase	Average Latency	P95 Latency	P99 Latency
Deployment Planning	1.9 s	3.2 s	4.8 s

The results indicate consistent performance across workflow phases, with code generation exhibiting higher latency due to computational complexity. Human-in-the-Loop delays are excluded from these values.

CONCLUSION

In conclusion, the integration of the LangGraph-based multi-agent orchestration framework within the proposed system for automating SDLC workflows signifies a significant advancement in software development automation. By harnessing LangGraph's specialized capabilities in managing state-based graph execution and coordinating specialized agents, the system gains a heightened ability to navigate complex development phases from requirements analysis through deployment. This strategic integration not only enhances the system's ability to maintain context across workflow phases but also strengthens development productivity through intelligent agent coordination and Human-in-the-Loop checkpoints. Ultimately, the utilization of multi-agent orchestration represents a proactive and adaptive approach to accelerating software delivery, offering a sophisticated solution to the persistent challenges faced in managing end-to-end development lifecycles. As such, the incorporation of LangGraph with specialized agents (Business Analyst, Architect, Developer, QA Engineer, and DevOps Engineer) marks a pivotal advancement in SDLC automation, paving the way for more effective and reliable software development processes in enterprise environments.

REFERENCE

[1] J. He, C. Treude, and D. Lo, “LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead,” arXiv preprint, 2024.

[2] J. Shore and S. Warden, “Guidelines for future agile methodologies and architecture reconciliation for software-intensive systems,” Electronics, vol. 12, no. 15, 2023.

[3] A. M. Al-Zoubi et al., “Transpiler-based architecture design model for back-end layers in software development,” Applied Sciences, vol. 13, no. 20, 2023.

[4] V. Tawosi, K. Ramani, S. Alamir, and X. Liu, “ALMAS: An autonomous LLM-based multi-agent software engineering framework,” arXiv preprint arXiv:2510.03463, 2025.

[5] A. Grishina, V. Liventsev, A. Härmä, and L. Moonen, “Fully autonomous programming using iterative multi-agent debugging with large language models,” ACM Transactions on Evolutionary Learning and Optimization, vol. 5, no. 1, 2025.

[6] Konrad Cinkusz et al., “Cognitive agents supported by large language models for Agile project management in software engineering,” Electronics, vol. 14, no. 87, pp. 1–22, 2025.

[7] D. Russo, “Understanding the complexity of adopting generative AI in software engineering,” ACM Transactions on Software Engineering and Methodology, vol. 33, no. 5, article 135, pp. 1–50, June 2024.

[8] A. Liesenfeld and M. Dingemanse, “Rethinking open source generative AI: open-washing and the EU AI Act,” in Proc. ACM Conf. Fairness, Accountability, and Transparency (FAccT ’24), Rio de Janeiro, Brazil, Jun. 2024, pp. 1–14.

[9] E. S. Grant et al., “Integrating artificial intelligence into software development education,” in Proc. IEEE 6th Eurasia Conf. IoT, Communication and Engineering, Yunlin, Taiwan, Nov. 2024, Engineering Proceedings, vol. 92, no. 26, pp. 1–8, Apr. 2025.

[10] D. Shin et al., “Transforming design implications into cards using generative AI,” in Proc. CHI Conf. Human Factors in Computing Systems (CHI ’24), Honolulu, HI, USA, May 2024, pp. 1–15.

[11] M. Alenezi and M. Akour, “AI-driven innovations in software engineering: current practices and future research directions,” Applied Sciences, vol. 15, no. 3, p. 1344, 2025.

[12] P. Kokol, “Artificial intelligence in software engineering: a synthetic review of recent literature,” Information, vol. 15, no. 6, p. 354, 2024.

[13] G. D. De Siano et al., “Code translation with large language models and human-in-the-loop evaluation,” Information and Software Technology, vol. 172, p. 107785, 2025.

[14] G. Bhandari et al., “Applying code language models to generate vulnerability fixes,” Information and Software Technology, vol. 172, p. 107786, 2025.

[15] V. Muttillo et al., “Synthetic trace generation of modeling operations for intelligent modeling assistants using large language models,” Information and Software Technology, vol. 172, p. 107806, 2025.

[16] L. Banh, F. Holldack, and G. Strobel, “Copiloting the future: generative AI’s impact on software engineering,” Information and Software Technology, vol. 172, p. 107751, 2025.

[17] V. Jackson et al., “Generative AI and creativity in software development: setting a research agenda,” ACM

Transactions on Software Engineering and Methodology, vol. 34, no. 5, article 133, May 2025.

[18] J. Sheard, P. Denny, A. Hellas, J. Leinonen, L. Malmi, and Simon, “Instructor perspectives on AI-powered code generation tools: insights from a multi-institutional study,” in Proc. 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024), vol. 1, Portland, OR, USA, Mar. 2024, pp. 20–23.

[19] D. Cambaz and X. Zhang, “AI-based code generation models in programming education: a systematic literature review,” in Proc. 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024), vol. 1, Portland, OR, USA, Mar. 2024, pp. 20–23.

[20] M. A. Haque, “Large language models: a game-changer for software engineers?,” Journal of Technology Benchmarking, vol. 2025, no. 100204, pp. 1–15, May 2025.