International Journal of Science and Research Archive
eISSN: 2582-8185
Cross Ref DOI: 10.30574/ijsra
Journal homepage: https://ijsra.net/

(RESEARCH ARTICLE)

Check for updates

A Conceptual Framework for the Evolution of Application Development: From Manual Coding to Agentic Development

Noor Romy Rahwani *

Study Program of Accounting Information Systems, Politeknik Negeri Banjarmasin, Indonesia.

International Journal of Science and Research Archive, 2026, 20(01), 200–210

Publication history: Received on 28 May 2026; revised on 04 July 2026; accepted on 06 July 2026

Article DOI: https://doi.org/10.30574/ijsra.2026.20.1.1458

Abstract

Purpose: This study proposes a conceptual framework to explain the evolution of application development from traditional manual coding to the emerging era of agentic development. While previous studies have primarily examined manual programming, low-code development, AI-assisted coding, or agentic software engineering as separate topics, this study integrates these approaches into a unified evolutionary perspective.

Design: This study employs a conceptual research design based on an integrative literature review. Relevant studies on software development paradigms, software reuse, low-code/no-code platforms, generative AI, vibe coding, and agentic software engineering were systematically analyzed and synthesized to develop a conceptual framework describing the evolution of application development. The proposed framework was subsequently analyzed through three analytical dimensions: development abstraction, developer role transformation, and degree of automation.

Findings: The study identifies five dominant application development paradigms: Manual Coding, Framework-based Development, Low-Code Development, AI-assisted Development, and Agentic Development. The findings indicate that application development has progressively evolved toward higher levels of development abstraction, increasing software engineering automation, and continuous transformation of developer roles from software implementers to AI collaborators and orchestrators. The study further argues that these paradigms should be viewed as complementary and evolutionary rather than mutually exclusive, as contemporary software development frequently combines multiple paradigms within a single project.

Originality/Value: This study contributes a unified conceptual framework that integrates previously fragmented software development paradigms into a coherent evolutionary model. The framework provides a theoretical foundation for future research in software engineering and offers practical implications for redesigning application development curricula and preparing developers for increasingly AI-driven and agentic software engineering environments.

Keywords: Application Development Evolution; Development Abstraction; Software Development Paradigms; Low-Code Development; AI-assisted Development; Agentic Development; Developer Role Transformation

1. Introduction

The rapid advancement of artificial intelligence (AI), particularly large language models (LLMs), has significantly changed the way software applications are developed [1]. Activities that previously required extensive manual programming can now be assisted by AI-powered coding tools capable of generating source code, explaining programming concepts, identifying programming errors, and automating repetitive development tasks through natural language interaction. More recently, AI agents have further expanded these capabilities by autonomously planning

* Corresponding author: Noor Romy Rahwani

development tasks, interacting with external tools, executing workflows, and iteratively refining software with limited human intervention[2].

However, these recent developments represent only the latest phase in the continuous evolution of application development. Over the past several decades, application development has undergone significant transformations driven by the need to improve developer productivity, reduce programming complexity, and enhance software quality. Early software applications were developed primarily through manual coding, requiring developers to write every line of source code and manage all aspects of software implementation. An example of this approach is the inventory application developed by Rahwani and Nugraha using Visual Basic 2015, in which the application was manually programmed without relying on modern frameworks or AI-assisted development tools [3].

As software systems became increasingly complex, framework-based development [4] emerged to promote code reuse, standardized software architectures, and improved maintainability while reducing repetitive programming tasks. More recently, the rapid adoption of low-code and no-code platforms has further simplified application development by enabling developers to build applications with substantially less manual programming. An example of this approach is the application development project reported by Rahwani [5][6] , illustrating how repetitive coding tasks can be reduced through low-code development.

The emergence of generative AI has accelerated this evolutionary process. AI-assisted development, often referred to as vibe coding, enables developers to describe application requirements using natural language while AI generates corresponding source code. More recently, agentic development has shifted software engineering beyond code generation by allowing AI to plan, coordinate, and execute multiple development tasks toward a specified goal. Consequently, the role of developers is gradually changing from manually writing software to supervising, validating, and orchestrating AI-driven development processes.

Existing studies have extensively investigated framework-based development, low-code/no-code platforms, AI-assisted programming, and AI agents. However, these technologies are generally discussed as independent innovations rather than as successive stages in the broader evolution of application development. As a result, there is limited conceptual understanding of how these paradigms collectively represent increasing levels of development abstraction, changes in the role of developers, and the growing degree of development automation.

Therefore, this paper proposes a conceptual framework for understanding the evolution of application development. Figure 1 illustrates the proposed conceptual framework, which analyzes application development evolution through three analytical dimensions: development abstraction [7], developer role transformation [8], and the degree of automation [9]. These dimensions provide a conceptual lens for examining the progression of application development across five paradigms: Manual Coding [3], Framework-based Development [4], Low-Code Development [10], AI-assisted Development (Vibe Coding) [11], and Agentic Development [2].

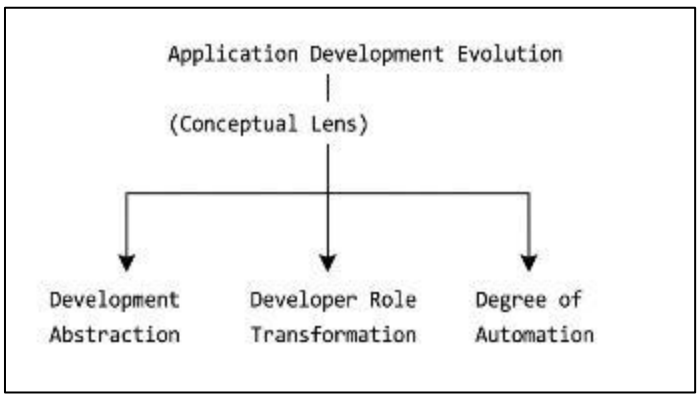

Figure 1 Proposed conceptual framework of application development evolution.

Rather than viewing these paradigms as isolated technological advances, this study argues that they represent successive paradigms in the evolution of application development. The proposed framework is expected to provide a conceptual foundation for future research on software engineering practices and application development education in the era of artificial intelligence.

2. Research Method

2.1. Research Design

This study adopts a conceptual research approach to develop a framework explaining the evolution of application development. Rather than conducting empirical testing or a systematic literature review, the study synthesizes existing literature to identify major paradigm shifts in application development and to organize them into a unified conceptual framework. Conceptual research is particularly appropriate for developing theoretical perspectives by integrating knowledge from multiple streams of literature and proposing new ways of understanding emerging phenomena.

2.2. Literature Identification

The conceptual framework was developed through a structured review of the literature related to software engineering, application development, low-code/no-code development, AI-assisted programming, and agentic software engineering. Relevant publications were identified primarily through Google Scholar and major academic digital libraries, including IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, and Scopus-indexed journals.

The literature search employed combinations of keywords such as manual coding, framework-based development, software abstraction, low-code development, no-code development, AI-assisted software development, vibe coding, agentic coding, agentic software engineering, developer role transformation, and software development automation. Both foundational studies and recent publications were considered to ensure comprehensive coverage of the evolution of application development.

2.3. Literature Analysis and Conceptual Synthesis

Rather than analysing the selected studies independently, the literature was synthesized conceptually to identify recurring themes that characterize the evolution of application development. Three analytical dimensions consistently emerged across the literature:

- Development abstraction
- Developer role transformation
- Degree of automation

These dimensions were adopted as the analytical framework for interpreting how application development has evolved over time. Figure 1 illustrates the conceptual framework employed in this study.

2.4. Conceptual Framework Development

The proposed conceptual framework was developed through a conceptual synthesis of the reviewed literature. Rather than examining individual technologies independently, this study integrates findings from multiple research streams to establish a unified perspective on the evolution of application development. The overall framework development process is illustrated in Figure 2.

The framework development began by synthesizing literature related to software engineering, framework-based development, low-code/no-code platforms, AI-assisted software development, and agentic software engineering. Through this synthesis, three recurring analytical dimensions were identified: development abstraction, developer role transformation, and degree of automation. These dimensions provide a common conceptual basis for interpreting how application development has evolved over time.

Using these three analytical dimensions, the study examines five major paradigms of application development: Manual Coding, Framework-based Development, Low-Code Development, AI-assisted Development (Vibe Coding), and Agentic Development. Rather than representing isolated technological innovations, these paradigms are interpreted as successive stages reflecting increasing levels of abstraction, changing developer responsibilities, and greater automation throughout the application development process.

It is important to emphasize that the proposed framework does not suggest fundamental changes to the Software Development Life Cycle (SDLC). Core software engineering activities, including requirements analysis, system design, implementation, testing, deployment, and maintenance, remain essential across all development paradigms. Instead, the framework explains how the methods used to perform these activities have evolved over time. As development

paradigms progress, developers increasingly operate at higher levels of abstraction, their roles shift from manual coding toward orchestration and supervision, and software development activities become progressively more automated.

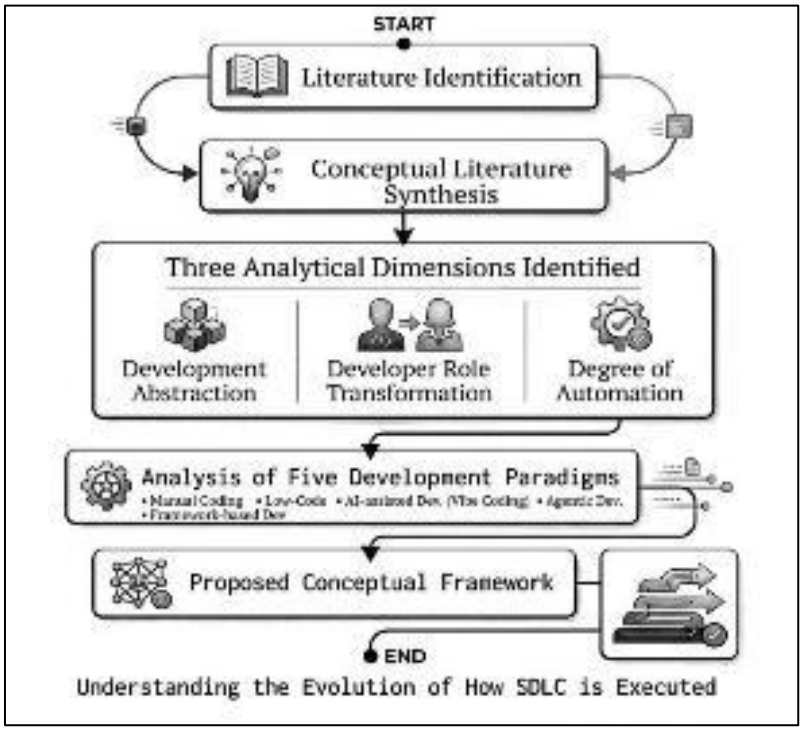

Figure 2 Research process for developing the proposed conceptual framework

Consequently, the proposed framework provides a conceptual explanation of the evolution of application development by integrating technological advances into a coherent analytical model. Rather than focusing on individual tools or programming technologies, the framework highlights the broader transformation in how software is developed in the era of artificial intelligence.

3. Results and Discussions

3.1. Evolution of Application Development Paradigms

The conceptual synthesis of the reviewed literature indicates that application development has evolved through a series of major paradigms, each introducing higher levels of abstraction [7], changing the role of developers [8], and increasing the degree of automation [2]. Rather than representing isolated technological innovations, these paradigms reflect successive approaches to developing software in response to increasing application complexity and advances in software engineering technologies.

As illustrated in Figure 3, this study identifies five major paradigms in the evolution of application development: Manual Coding, Framework-based Development, Low-Code Development, AI-assisted Development (Vibe Coding), and Agentic Development. Each paradigm represents a distinct approach to software development while preserving the fundamental activities of the Software Development Life Cycle (SDLC)[12], including requirements analysis, system design, implementation, testing, deployment, and maintenance. What has evolved is not the SDLC itself, but the methods, tools, and levels of abstraction through which these activities are performed.

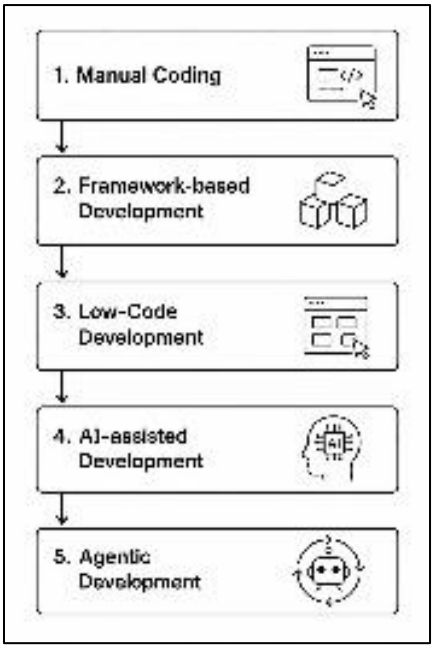

Figure 3 The five paradigms of application development evolution

The earliest paradigm, Manual Coding, required developers to implement software by writing source code directly, providing maximum flexibility but demanding extensive programming expertise [13]. As software systems became increasingly complex, Framework-based Development introduced reusable software components and standardized architectural patterns to improve productivity, maintainability, and code quality.

The emergence of Low-Code Development [10] further increased the level of abstraction by allowing applications to be created through visual models and configuration rather than extensive manual coding. More recently, advances in generative AI have enabled AI-assisted Development, in which developers interact with large language models using natural language prompts to generate and refine source code. The latest paradigm, Agentic Development, extends AI assistance by enabling autonomous software agents to plan, coordinate, execute, and refine development tasks with limited human intervention.

Although these paradigms differ significantly in their technologies and development approaches, they share the common objective of reducing development complexity while improving productivity and software quality. This study argues that these paradigms should not be viewed as independent technological milestones, but as successive stages in the continuing evolution of application development. The following sections examine this evolution through three analytical dimensions: development abstraction [7] , developer role transformation [14], and degree of automation [9].

3.2. Development Abstraction Across the Five Paradigms

One of the most significant findings of this study is that application development has continuously evolved toward higher levels of development abstraction [7]. Rather than requiring developers to manage every implementation detail, each new development paradigm progressively abstracts lower-level technical complexities, allowing developers to concentrate on increasingly higher-level aspects of software development.

In the manual coding paradigm, developers were responsible for implementing every component of an application directly in source code. All business logic, user interface, data access, and system integration had to be explicitly programmed [12], requiring extensive knowledge of programming languages, algorithms, and system architecture. Consequently, software development was highly dependent on coding expertise and required substantial development time. Visual abstractions, reusable development components, and automated code generation were generally unavailable, requiring developers to manually implement most application functionalities [10].

Framework-based development introduced the first major increase in development abstraction [4]. Instead of building every software component from scratch, developers increasingly assembled applications using reusable libraries, standardized architectures, and predefined software components, reflecting the component-based software

engineering principle of "reuse rather than rebuild" [15]. Frameworks reduced repetitive coding activities while improving maintainability, consistency, and software quality[16].

The emergence of low-code development further elevated the level of development abstraction by enabling application development through visual programming, reusable components, automatic code generation, and minimal hand-coding [10]. According to Rokis and Kirikova [10], low-code development provides a rapid and flexible approach that translates business requirements into software applications through graphical interfaces and visual abstraction rather than extensive manual programming. This evolution is consistent with the broader trend toward Low-Code/No-Code (LCNC) platforms described by Sufi [17], reflecting a progressive shift from traditional programming to increasingly abstract forms of application development. Rather than focusing primarily on syntax implementation, developers increasingly concentrated on application workflows, business processes, and user requirements. In addition to reducing coding effort, low-code platforms have been reported to shorten development cycles, improve collaboration between business and IT stakeholders, increase responsiveness to changing business requirements, and enable wider participation of citizen developers in software development [10]. Although programming knowledge remained valuable, significantly fewer implementation details required manual coding.

AI-assisted development represents another major shift in abstraction. Instead of constructing software through code or visual components alone, developers increasingly express application requirements using natural language prompts [18] [19]. Large Language Models translate these high-level descriptions into executable source code, allowing developers to focus more on problem definition, solution refinement, and code verification than on writing every implementation detail themselves [9][20].

The highest level of abstraction is represented by agentic development [2]. In this paradigm, AI systems extend beyond code generation to autonomously perform multiple software engineering activities, including planning development tasks, generating and revising code, testing software, debugging defects, interacting with external tools, and coordinating development workflows [2]. Consequently, developers increasingly define objectives, constraints, and expected outcomes, while AI agents execute much of the implementation process autonomously under human supervision[20].

Figure 4 summarizes this continuous increase in development abstraction. Across the five paradigms, the primary responsibility of developers gradually shifts from writing low-level implementation code toward defining business objectives, validating AI-generated solutions, and orchestrating increasingly autonomous software development processes. This finding suggests that the evolution of application development is characterized not merely by technological advancement, but by a fundamental transition in the level of abstraction through which software systems are created.

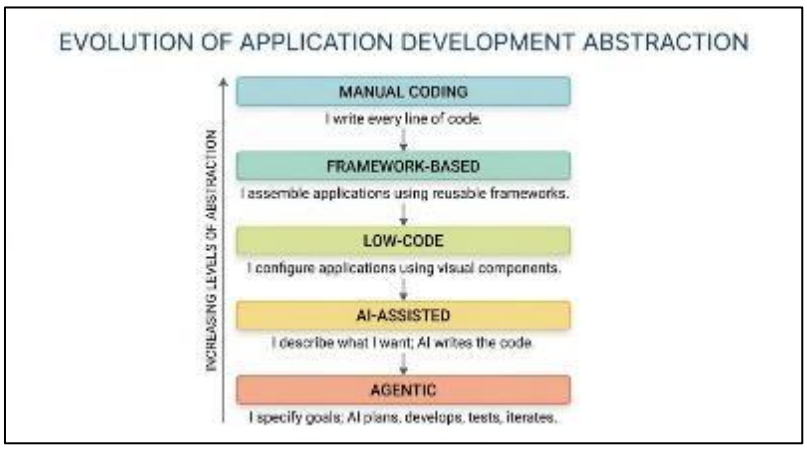

Figure 4 Increasing levels of development abstraction across the five application development paradigms

3.3. Developer Role Transformation

Another important finding of this study is the continuous transformation of the developer's role throughout the evolution of application development. As development abstraction and automation increased, developers gradually shifted their attention from low-level programming activities toward higher-level software design, business problem solving, and system orchestration.

In the manual coding paradigm, developers were responsible for virtually every aspect of software implementation [13] . They designed algorithms, implemented business logic, managed data access, developed user interfaces, performed testing, and resolved implementation issues manually. Consequently, software development relied heavily on individual programming expertise and detailed knowledge of programming languages and system architecture.

The introduction of software frameworks changed the developer's responsibilities by providing reusable architectures, standardized programming patterns, and pre-built software components. Rather than repeatedly implementing common functionalities, developers increasingly focused on integrating business requirements with framework capabilities while still maintaining responsibility for application logic and software quality.

The emergence of low-code platforms further transformed the developer's role from software programmer to application builder. Visual development environments enabled developers and even citizen developers to create applications through graphical modelling, reusable components, and business process configuration instead of extensive manual coding [10]. Consequently, developers devoted more attention to business processes, user experience, and application functionality than to programming syntax.

The adoption of generative AI introduced another significant shift in developer responsibilities. Rather than manually implementing every software component, developers increasingly collaborate with AI systems that generate code, documentation, test cases, and implementation alternatives from natural language instructions [21]. As a result, developers spend more time reviewing, validating, refining, and integrating AI-generated outputs than writing every line of source code themselves.

This transformation becomes even more evident in the emerging paradigm of vibe coding and agentic development. Meske et al. argue that software development is shifting from deterministic programming toward intent-mediated collaboration, in which developers express high-level objectives through natural language dialogue while AI assumes increasing responsibility for implementation. Consequently, software development redistributes cognitive and technical work between humans and AI, shifting developer expertise from technical implementation toward collaborative orchestration, architectural reasoning, and strategic decision making [22].

Figure 5 summarizes the transformation of the developer's role across the five development paradigms. Overall, the findings indicate that developers are progressively moving away from low-level programming activities toward higher-level responsibilities involving solution design, business understanding, AI supervision, and software orchestration.

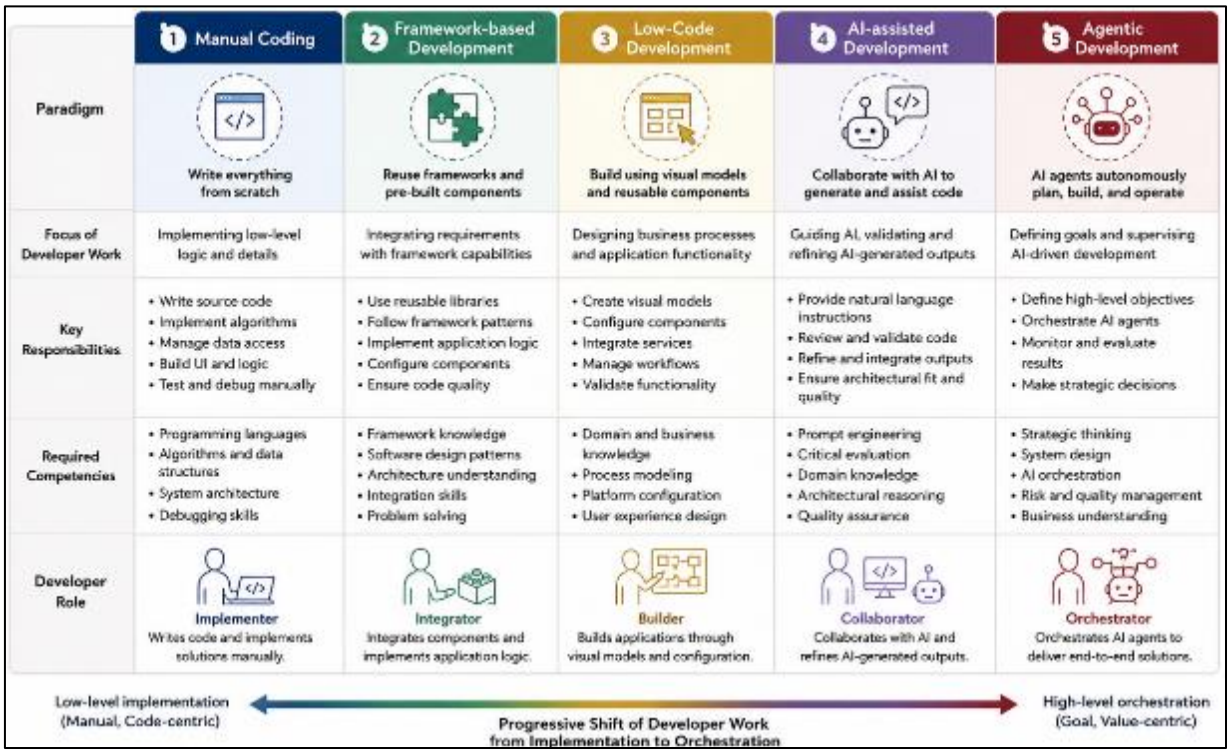

Figure 5 Transformation of developer roles across the five application development paradigms

It should be noted that the five paradigms proposed in this study are not mutually exclusive. Contemporary application development frequently combines multiple paradigms within a single project. For example, developers may implement business logic manually, utilize framework services, configure low-code components, collaborate with AI coding assistants, and supervise agentic AI systems simultaneously. Therefore, the proposed framework should be interpreted as representing the dominant evolution of software development practices rather than sequential replacement of earlier paradigms.

3.4. Increasing Degree of Automation

The third dimension identified in this study is the increasing degree of automation throughout the application development process. Similar to the evolution of development abstraction and developer roles, automation has progressively expanded across software development paradigms. Rather than merely accelerating programming activities, each paradigm has introduced new mechanisms for reducing manual effort and automating increasingly complex software engineering tasks.

In the manual coding paradigm, software development was almost entirely dependent on human effort. Developers manually implemented application logic, created user interfaces, tested software, identified defects, and maintained source code [13]. Automation was generally limited to basic compiler functions and development tools, leaving most software engineering activities under direct human control.

Framework-based development introduced partial automation by providing reusable software components, standardized development patterns, built-in libraries, and development utilities [16]. Common functionalities such as authentication, routing, database access, session management, and input validation were increasingly handled by frameworks rather than implemented repeatedly by developers. As a result, repetitive programming tasks were significantly reduced while maintaining flexibility for application-specific business logic.

Low-code development further increased the degree of automation by automatically generating substantial portions of application code from visual models, predefined components, and business process configurations [10]. Development platforms increasingly automated user interface generation, database integration, workflow implementation, and deployment activities, allowing developers to focus on application functionality rather than implementation details. Consequently, software development became faster and more accessible to users with limited programming experience [10].

AI-assisted development represents a significant advancement in software engineering automation. Large Language Models assist developers by generating source code, explaining programming concepts, identifying programming errors, creating software documentation, generating test cases, and recommending implementation alternatives based on natural language instructions [19] [21]. Rather than replacing developers, recent studies emphasize that GenAI establishes a collaborative human–AI workflow in which AI augments developers' capabilities while developers remain responsible for validation, architectural decisions, and software quality [21].

Recent empirical studies also indicate that software development practices are shifting from traditional coding toward increasingly automated and AI-supported workflows. Yu argues that this progression reflects a broader paradigm shift in coding productivity, where GenAI extends automation beyond conventional development pipelines by enabling AI-assisted code co-creation rather than merely supporting implementation tasks [23].

The highest degree of automation is observed in the emerging paradigm of agentic development. Rather than assisting individual programming tasks, AI systems increasingly participate throughout the software development lifecycle by planning tasks, generating code, executing tests, reviewing implementations, and interacting with external development tools under human supervision [22]. As argued by Meske et al., this emerging paradigm fundamentally reconfigures software development from deterministic programming toward intent mediation, where developers express high-level objectives through conversational interaction while AI assumes increasing responsibility for implementation [22]. Consequently, developer expertise shifts from syntax implementation toward collaborative orchestration, architectural reasoning, and strategic decision making.

Figure 6 summarizes the increasing degree of automation across the five application development paradigms. The findings suggest that automation has evolved from supporting isolated programming activities to coordinating integrated software engineering processes. Nevertheless, this evolution does not eliminate the role of human developers. Instead, automation increasingly complements human expertise by reducing repetitive implementation

tasks while enabling developers to focus on higher-level decision making, business problem solving, and AI orchestration.

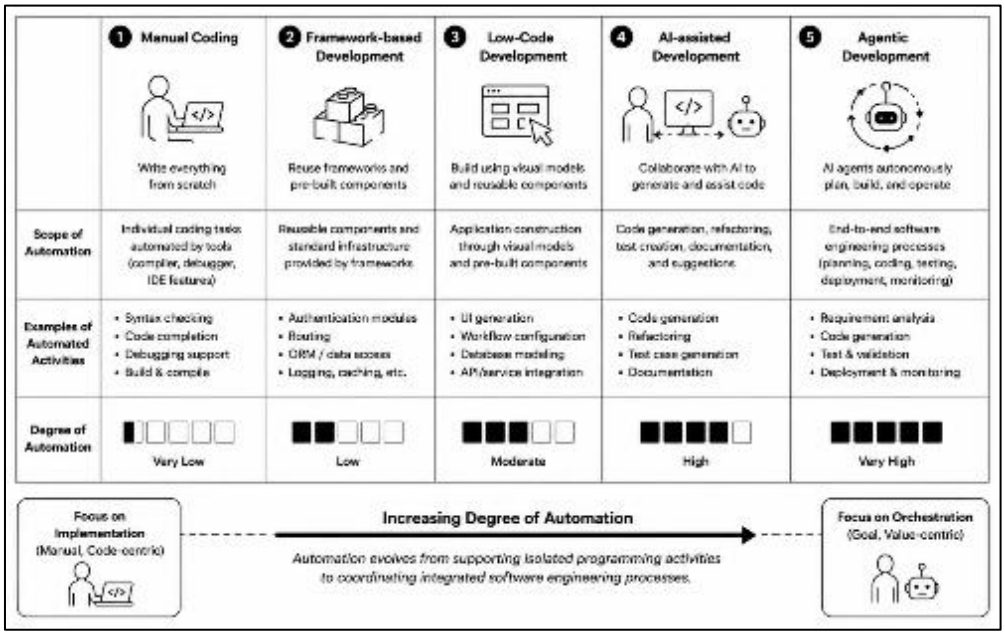

Figure 6 Increasing degree of automation across the five application development paradigms

4. Conclusion

This study proposed a conceptual framework to explain the evolution of application development through five major paradigms: Manual Coding, Framework-based Development, Low-Code Development, AI-assisted Development, and Agentic Development. Rather than treating these paradigms as isolated technological innovations, this study conceptualizes them as successive stages in the continuous evolution of software development.

The proposed framework demonstrates that this evolution can be understood through three interrelated dimensions: increasing development abstraction, transformation of developer roles, and increasing degrees of automation. Across these paradigms, software development has progressively shifted from manually implementing every software component toward defining higher-level business objectives while increasingly collaborating with intelligent development tools and autonomous AI agents.

The findings also suggest that the evolution of application development should not be interpreted as the replacement of earlier paradigms. Instead, contemporary software development frequently combines multiple paradigms within the same project. Developers may simultaneously write application logic manually, utilize software frameworks, configure low-code components, collaborate with AI coding assistants, and supervise agentic AI systems. Therefore, the proposed framework represents the evolution of dominant development paradigms rather than mutually exclusive development approaches.

From a theoretical perspective, this study contributes a unified conceptual framework that integrates previously fragmented discussions on software development paradigms into a coherent evolutionary perspective. Existing studies have typically examined manual programming, low-code development, AI-assisted coding, or agentic software engineering independently. In contrast, this study positions these approaches within a single evolutionary model that explains how software development has progressively evolved over time.

From a practical perspective, the proposed framework provides useful insights for software engineering researchers, educators, and practitioners. For educators, the framework highlights the need to rethink application development curricula by complementing traditional programming skills with higher-level competencies such as system architecture, business analysis, AI collaboration, and AI orchestration. For practitioners, the framework provides a conceptual understanding of how emerging AI technologies reshape software development practices while emphasizing that human expertise remains essential throughout the development lifecycle.

This study has several limitations. The proposed framework is conceptual in nature and has not yet been empirically validated. Future research may evaluate the framework through expert interviews, Delphi studies, surveys of software practitioners, or longitudinal analyses of software development practices across different industries. In addition, future studies may investigate the educational implications of agentic development and examine how software engineering curricula should evolve to prepare future developers for increasingly AI-driven development environments.

References

[1] F. W. Todorovic, "Enhancing Software Development with AI ," KTH School: School of Electrical Engineering and Computer Science, 2024. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1948945/FULLTEXT01.pdf

[2] U. Karwal, V.P., Kalucha, C., Sharma, C., Jain, A., Komal, Hariharan, "Multi-agent AI Framework for Developer Assistance: A New Paradigm in Software Engineering Automation," 2024, pp. 306–312. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-032-03769-5_18

[3] N. R. Rahwani and G. N. Nugraha, "Inventory Application with Average Perpetual System By using Visual Basic 2015," *J. INTEKNA Inf. Tek. dan Niaga*, vol. 16, no. 1, pp. 15–21, 2016, [Online]. Available: https://www.researchgate.net/publication/377534333_Inventory_Application_with_Average_Perpetual_System_By_using_Visual_Basic_2015

[4] S. Khan and A. T. Khanam, "Study on MVC Framework for Web Development in PHP," *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol.*, pp. 414–419, 2023, doi: 10.32628/cseit2390450.

[5] N. R. Rahwani, Hikmahwati, and M. A. Budiman, "Meninjau Kembali Dilema Buy-vs-Build Menuju Opsi Pengembangan Hybrid : Studi Kasus Aplikasi Proyek Pengadaan yang Didanai Investor," *BERNAS J. Pengabdi. Kpd. Masy.*, vol. 6, no. 4, pp. 3188–3197, 2025, doi: 10.31949/jb.v6i4.15742.

[6] N. R. Rahwani, "Evaluating the buy vs build dilemma: A case study approach to corporate accounting information systems," *Int. J. Sci. Res. Arch.*, vol. 14, no. 1, pp. 1901–1904, 2025, doi: 10.30574/ijsra.2025.14.1.0321.

[7] M. Jackson, "Aspects of abstraction in software development," in *Softw Syst Model*, no. July, 2012, pp. 1–20. [Online]. Available: https://doi.org/10.1007/s10270-012-0259-7

[8] E. Meade *et al.*, "The Changing Role of the Software Engineer," in *Communications in Computer and Information Science*, 2019, pp. 682–694. doi: 10.1007/978-3-030-28005-5_53.

[9] A. E. Hassan *et al.*, "Agentic Software Engineering: Foundational Pillars and a Research Roadmap," *arXiv Comput. Sci.*, vol. 1, no. 1, 2025, doi: 10.48550/arXiv.2509.06216.

[10] K. Rokis and M. Kirikova, "Exploring Low-Code Development : A Comprehensive Literature Review," *Complex Syst. Informatics Model. Q.*, no. 36, pp. 68–86, 2023, doi: 10.7250/csimq.2023-36.04.

[11] F. Geng *et al.*, "Exploring Student-AI Interactions in Vibe Coding," in *28th Australasian Computing Education Conference (ACE 2026), February 09â•fi13, 2026, Melbourne, VIC, Australia*, Association for Computing Machinery, 2025, pp. 45–54. doi: 10.1145/3786228.3786236.

[12] N. R. Rahwani, M. M. Sadewa, N. Nikmah, N. Mukhlisah, and S. Iriawan, "Boosting Efficiency: Integrating Inventory Apps in Accounting Information Systems," *Indones. J. Appl. Account. Financ.*, vol. 3, no. 2, pp. 153–166, 2023, doi: 10.31961/ijaaf.v3i2.2269.

[13] C. Patil, Y. Nikhade, M. Patel, and A. Pande, "Manual Vs Automated Coding A Comparative Analysis Of Productivity And Code Quality," *SSRN Electron. J.*, 2025, doi: 10.2139/ssrn.5220466.

[14] K. Qiu, N. Puccinelli, M. Ciniselli, and L. Di Grazia, "From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, 2025, doi: 10.1145/3709353.

[15] R. Qureshi and M. Sandhu, "Survey-Based Analysis of the Proposed Component-Based Development Process," *arXiv:1202.2498*, 2012, [Online]. Available: http://arxiv.org/abs/1202.2498

[16] U. Hasan, A. Sholihin, and U. L. Yuhana, "Reusable Components as Base Project for Standardizing Software Quality," *Proceeding - 2024 Int. Conf. Inf. Technol. Res. Innov. ICITRI 2024*, pp. 323–328, 2024, doi: 10.1109/ICITRI62858.2024.10699015.

[17] F. Sufi, "Algorithms in Low-Code-No-Code for Research Applications: A Practical Review," *MDPI - Algorithms*, vol. 16, no. 2, 2023, doi: 10.3390/a16020108.

[18] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI," *arXiv Comput. Sci.*, 2025, doi: 10.48550/arXiv.2505.19443.

[19] S. C. Necula, F. Dumitriu, and V. Greavu-Şerban, "A Systematic Literature Review on Using Natural Language Processing in Software Requirements Engineering," *MDPI-Electronics (Switzerland)*, vol. 13, no. 11, 2024, doi: 10.3390/electronics13112055.

[20] P. Mandl and P. Mandl, "AI-driven Software Development: A Pragmatic Path to Agentic Development Processes," *arXiv Prepr. arXiv2606.15283*, pp. 1–34, 2026, doi: 10.48550/arXiv.2606.15283.

[21] L. Banh, F. Holldack, and G. Strobel, "Copiloting the future: How generative AI transforms Software Engineering," *Inf. Softw. Technol.*, vol. 183, no. September 2024, p. 107751, 2025, doi: 10.1016/j.infsof.2025.107751.

[22] C. Meske, T. Hermanns, E. Von der Weiden, K. U. Loser, and T. Berger, "Vibe Coding as a Reconfiguration of Intent Mediation in Software Development: Definition, Implications, and Research Agenda," *IEEE Access*, vol. 13, no. October, pp. 213242–213259, 2025, doi: 10.1109/ACCESS.2025.3645466.

[23] L. Yu, "Paradigm shift on Coding Productivity Using GenAI," in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering , EASE, 2025 edition, EASE 2025*, Association for Computing Machinery, 2025, pp. 708–713. doi: 10.1145/3756681.3757081.