

A Survey of Intelligent Programming and Code Optimization Techniques in Software Development

Dr. Neha Upadhyay

Assistant Professor

MANIT, Bhopal

nupadhyay@manit.ac.in

Abstract: *Software development is continuously evolving with the integration of Artificial Intelligence (AI) techniques that enhance productivity, software quality, and development efficiency. This survey reviews intelligent programming and code optimization techniques applied across the Software Development Life Cycle (SDLC), including requirements analysis, coding, testing, deployment, and maintenance. It examines traditional software development approaches and modern architectures, followed by intelligent programming techniques based on Machine Learning (ML), Natural Language Processing (NLP), Deep Learning, Transformers, and Generative AI. The survey is limited to intelligent source code completion, AI techniques for code generation, classical optimization methods such as code cloning, invariant code motion, dead code elimination, local optimization, global optimization, and inter-procedural optimization. Recent studies demonstrate that AI-assisted approaches can improve code quality, development efficiency, vulnerability detection, energy efficiency, cost estimation, and software testing. However, challenges related to data dependency, interpretability, developer trust, technical integration, and the reliability of AI-generated code remain significant. The survey highlights current research trends and identifies future opportunities for developing reliable, explainable, scalable, and efficient AI-driven programming and code optimization solutions for modern software development.*

Keywords: Intelligent Programming, Code Optimization, Artificial Intelligence, Software Development, Machine Learning

I. INTRODUCTION

Efficient software systems are essential for achieving better performance, reliability, scalability, and resource utilization across modern computing applications, including web and mobile platforms, cloud computing [1], Kubernetes-based environments, distributed systems, embedded systems, Internet of Things (IoT), sensor-based systems, edge computing, enterprise environments, and networked applications [2][3]. Software engineering is a thoughtful practice of designing, developing, and maintaining software systems that leads to efficient, scalable, and reliable solutions [4]. To begin with the basics, software is a set of instructions that can be read by a machine and direct the computer's processor to perform operations as per the instructions provided. The software development process, which is really defined as the software development lifecycle (SDLC), basically consists of various phases like planning, requirements analysis, design, implementation, testing, deployment and maintenance. SDLC is a set of predefined tasks performed at every step of software development [5]. There are many methodologies used in software development. Some new methodologies used for software development are as following: Reuse-Based Development. Component-Based Development. Agile Software Development (ASD). These models differ and cannot meet the diverse requirements of the software industry and its users [6]. There must be a change to a new method that has



lightning-fast development and delivery if wants customers to be happy. So, for this purpose, a method called ASD is used. But ASD not only shrinks the timeline in software development but also cuts the cost. ASD is able to deliver a product which is according to the needs of the customer and market needs [7].

AI technologies, including ML, NLP, and computer vision, have introduced new paradigms in how developers approach various phases of the software development lifecycle. ML algorithms enable predictive analytics and anomaly detection, helping developers identify potential issues before they escalate. NLP assists in translating user requirements into actionable development tasks, making requirement gathering more efficient and accurate. Computer vision, though less directly applicable, supports niche areas in software development, such as automated design generation and image-based testing [8]. The integration of AI-driven tools has enabled automation of complex and repetitive tasks, such as code generation, bug detection, cybersecurity analysis, network monitoring, sensor-data processing, and optimization of deployment strategies across cloud, edge, and distributed computing environments [9]. These tools not only enhance productivity but also facilitate error reduction, contributing to higher quality software and more efficient development cycles [10].

Automation through the novel element of Artificial Intelligence has become a key driver of this change in the conventional approaches to code optimization and refactoring [11]. Code Optimization is the process of transforming source code to produce more efficient target code. Efficiency is measured in both time and space. Optimization is generally implemented using a set of optimization transformations, i.e., algorithms that take a piece of source code and transform it to produce semantically equivalent output code that uses fewer resources. The majority of optimization methods focus on making the target code quicker by either replacing slower sequences of instructions with faster ones or by removing unneeded instructions from the object code [12].

The paper is organized as follows: Section II introduces Software Development, Section III explores Intelligent Programming and Code Optimization, Section IV reviews recent literature, and Section V concludes the paper while outlining potential future research directions.

II. AN OVERVIEW OF SOFTWARE DEVELOPMENT

The management of software development is achieved through the introduction of software processes, which can be defined as “the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy and maintain a software product” [13]. Several models have been introduced to drive software development activities in startups, however, without achieving significant benefits.

A. Software Development Life Cycle (SDLC)

The Software Development Life Cycle is a process that outlines the stages and activities involved in developing software from conception to deployment and maintenance. It provides a structured framework for managing and controlling the software development process. Here is a simplified description of the typical phases in the SDLC are shown in Fig. 1 below:



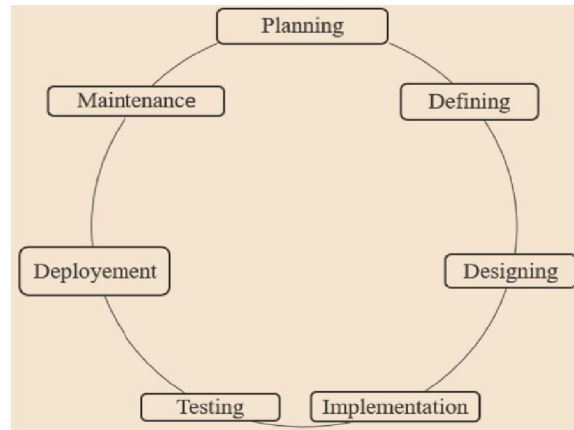


Fig. 1. Software Development Life Cycle Phases

- **Planning:** Define project scope, objectives, and requirements. Identify stakeholders and their roles. Create a project plan, including timelines and resources.
- **Analysis:** Gather and document detailed requirements from users and stakeholders. Analyze the gathered information to understand the system's functional and non-functional requirements [14]. Develop use cases, user stories, or functional specifications.
- **Design:** Create architectural design, defining how the software components interact. Develop detailed technical specifications. Design the user interface (UI) and user experience (UX).
- **Implementation (Coding):** Write and test the actual code based on the design and specifications. Conduct unit testing to ensure individual components work as intended. Integrate code modules as needed.
- **Testing:** Perform various levels of testing, including integration testing, system testing, and user acceptance testing (UAT). Identify and fix defects and issues. Verify that the software meets the defined requirements.
- **Deployment:** Deploy the software to a staging environment for final testing and validation. Prepare for the production environment by configuring servers and databases. Roll out the software to end-users or customers.
- **Maintenance and Support:** Monitor and maintain the software in the production environment. Address and fix any reported issues or defects. Implement updates, enhancements, and patches as needed.

B. Traditional Software Development Model

The traditional model is one of the oldest approaches to the software development process [15]. The model's sequential approach, in which each step is performed in turn, is one of its most remarkable characteristics. The traditional approach is mainly based on planning, analysis, and detailed discussion of the project.

- **Waterfall Model:** The waterfall model is the first model within the SDLC approach [16]. This model is easy to understand and can be used successfully to develop new projects. In addition, this model is known as the sequential linear life cycle model. In this model, the software development process is applied in separate stages [17].
- **Iterative Model:** The iterative model is one of the SDLC models that relies on implementing the requirements of software development through small units repeatedly to reach a completed system. The iterative or incremental model is divided into several architecture units to meet the requirements of the software development process.
- **Spiral Model:** Essentially, the model progressively improves the product through an iterative process around the spiral. The model starts with the stage of collecting business requirements in the baseline spiral [18]. In addition, during the product maturity stage in subsequent spirals, the remaining main system, subsystem and unit requirements are identified.



C. Intelligent Applications in Software Development

The advent of AI has precipitated transformative shifts across various facets of software development [19]. The integration of AI-driven tools and methodologies has enhanced the efficiency, accuracy, and adaptability of software engineering processes.

- **Relevance to Industry Applications:** Tools and datasets that are widely used in real-world software engineering contexts, such as GitHub Copilot and IBM's defect prediction tools, were prioritized [20].
- **Recency:** The research aims to represent the current status of the field by giving preference to tools and datasets that have been published or utilized actively during the previous five years [21].
- **Accessibility:** Open-source datasets and tools with publicly available documentation were selected to facilitate reproducibility [22].
- **Coverage of Development Phases:** Examples were chosen to cover diverse phases of the software development lifecycle, including coding, testing, and maintenance.
- **Impact and Adoption:** The selection emphasized tools and datasets with demonstrated effectiveness, as reported in industry and academic studies.

D. Code Optimization Across the Software Development

AI automation tools can identify architectural problems, such as incorrect module interfaces or even reusable software entities that are too tightly coupled. These systems analyze the code structure to provide guidance on enhancing modularity and scalability.

- **Automated Code Reviewing Systems:** Automated code review systems popularly employ artificial intelligence to implement an important process of engineering, the code review.
- **Refactoring Suggestion Engines:** The second AI innovation in software engineering is called refactoring suggestion engines [11]. Automated Code Review tools scan the implemented code and offer suggestions with regard to the ways that the code can be refactored and made more readable and sustainable.
- **GNNs for Code Structure Processing:** GNNs are a relatively new AI method that has been applied in classic computational models for the purpose of analysis as well as reorganizing code that contains dependencies on each other [23].

III. INTELLIGENT PROGRAMMING AND CODE OPTIMIZATION

Intelligent programming integrates AI techniques into software development to automate tasks such as code generation, completion, debugging, and refactoring. Code optimization focuses on improving code performance, efficiency, maintainability, and resource utilization. Traditional techniques eliminate redundant or unnecessary code, while AI-based methods identify optimization opportunities automatically. Together, these approaches enhance developer productivity, software quality, and overall development efficiency.

A. Intelligent Programming Approaches

Neural networks, knowledge-based systems, supervised and unsupervised learning, and reinforcement learning are examples of AI technologies utilized in contemporary software engineering. These technologies allow for automation of development tasks, provide better decision-making capabilities and enable intelligent coding, testing and deployment operations in Fig. 2.

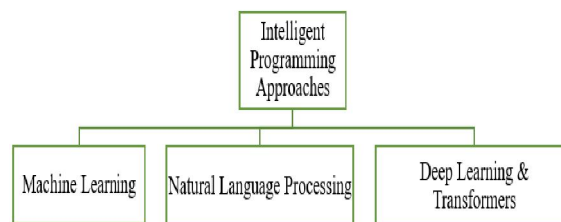


Fig. 2. Intelligent Programming Approaches



- **Machine Learning:** Contemporary software engineering outside of research labs is increasingly incorporating ML approaches to not only automate but also facilitate, to a wide degree, various tasks that are part of the development lifecycle[24]. A variety of conventional supervised and unsupervised ML techniques have been used for defect prediction, code recommendation, effort estimation, and maintenance prioritization.
- **Natural Language Processing:** Since numerous software artifacts, e.g., requirements documents, bug reports, code comments, and documentation, are expressed in natural language, NLP techniques have been incorporated into software engineering processes. NLP is instrumental in parsing, classifying, and extracting useful informational parts from textual artifacts.
- **Deep Learning & Transformers:** The use of DL techniques, especially neural networks that use structured or sequential code representations, allows AI to enter the software development space independently of human developers[25]. By representing code mathematically (i.e., an "embedding") for the code token, syntax tree, and control/data flow graph, DL can recognize higher-level concepts in big software systems from the patterns it discovers in the software structure [26].

B. Conventional Approaches to Intelligent Code Completion

These environments can be broadly classified based on their AI integration approach, with some utilizing rule-based systems and others employing ML algorithms for more sophisticated, context-aware code suggestions.

- **Intelligent Source Code Completion:** Intelligent assistants consider the context from both the current program and various other projects to recognize common patterns [27]. By discerning these patterns, they can gauge the developer's intent. This determination often hinges on variable names or method sequences, leading to contextually relevant suggestions. Instead of offering all possible suggestions like traditional assistants, intelligent ones provide those aligned with recognized patterns, streamlining the developer's task.
- **Traditional Source Code Completion:** Traditional source code completion assistants usually list all the attributes or methods that are available at a certain point of the source code, usually after "." is pressed. The developer can then select an appropriate method from an alphabetically ordered list. The process is often slower than writing the method's name manually [28].

C. Benefits and Improvements Achieved Through AI-based Code Generation

There are some benefits and improvements achieved through AI-based Code Generation some of them are listed below:

- **Increased Productivity and Efficiency:** AI- based code generation automates repetitive and time- consuming tasks in software development, allowing developers to focus on higher-level tasks [29].
- **Enhanced Code Quality:** AI models can analyze large codebases, identify patterns, and learn best coding practices from existing high-quality code [30].
- **Enabling Code Generation from Alternative Representations:** AI models can generate code from different representations, such as images, diagrams, or sketches. This allows developers to express their ideas visually or graphically and automatically convert them into executable code [31].
- **Code Completion and Autocompletion:** AI models trained on vast code repositories can provide intelligent code completion suggestions based on the context [32].

D. Classical Code Optimization Methods

The code optimization techniques mostly include three types of methods that focus on optimizing code and help improve performance and reduce code size. Some compiler optimization techniques operate on the code, but they are completely dependent on the compiler.

- **Code Cloning:** Code cloning is a code piece like each other in terms of semantics and punctuation. It is a comparable code structure present in various structures in programming frameworks [33]. Analysts have indicated that numerous projects contain amazing amount of copied code.
- **Invariant Code Motion:** Invariant code motion decreases the amount of code in the loop. It takes a statement or expression that can be moved outside the loop body. It can reduce in the loop and improve code space and runtime performance.



- **Dead code elimination:** Dead code elimination is an improvement technique used to eliminate dead code from the program so if the code is killed it won't be an effect on code results. It abbreviates the program so execution time less similarly as space taken by the program declines.

E. Code Optimization Techniques

Code optimization involves applying systematic techniques to improve the efficiency, performance, and resource utilization of software programs. These techniques can be applied at different levels, ranging from individual statements or code segments to the entire program [34]. In general, locally scoped techniques are easier to implement but typically provide smaller performance improvements, whereas global optimization techniques can achieve greater gains by considering the overall program structure. The optimization techniques considered in this study are illustrated in Fig. 3.



Fig. 3. Code Optimization Techniques

- **Local Optimization:** The code optimization phase in a compiler begins with partitioning the sequences of three-address instructions into basic blocks. These basic blocks become the nodes of a flow graph. Local optimization is performed within each basic block.
- **Global Optimization:** Global optimization techniques act on whole functions. In global optimization, improvement takes into account what happens across basic blocks. Most global optimization techniques are based on data-flow analysis.
- **Inter-Procedural Optimization:** Inter-procedural optimization (IPO) techniques are applied to programs that contain many frequently used functions. The technique is called inter-procedural because it analyses the entire program, whereas other optimizations such as local optimization or global optimization look at only a single function, or even a single block of code.

F. Challenges in AI-Based Programming and Code Optimization

There are some challenges in Code Optimization which are AI-Based and a few are listed below:

- **Data Dependency:** AI models require vast amounts of high-quality data to function effectively. Training data scarcity, especially for specific domains, can limit the performance of AI tools.
- **Interpretability:** One of the significant limitations of AI models is their interpretability. Complex models, particularly DL algorithms, often function as “black boxes,” making it difficult to understand the reasoning behind their suggestions.
- **Trust:** Developers may hesitate to trust AI-driven decisions if they cannot interpret how these models arrived at a specific conclusion.
- **Technical Challenges:** Integrating AI-driven tools into development workflows poses technical challenges. Setting up continuous integration/continuous deployment (CI/CD) pipelines with AI-driven testing tools requires expertise and infrastructure.



IV. LITERATURE REVIEW

This section highlights related studies on the integration of intelligent programming, code and software optimization, emphasizing AI-driven code generation, completion, optimization, testing, vulnerability detection, and improved software quality and development efficiency.

Pattlola Srinivas et. al. (2026) presented a comprehensive Generative AI-assisted framework that integrates large language models and ML techniques to optimize and secure the SDLC. The proposed framework leverages generative AI capabilities across multiple phases, including requirements analysis, code generation, vulnerability detection, automated testing, and deployment optimization. By incorporating AI-driven threat modeling and continuous security assessment, the framework addresses critical challenges in identifying vulnerabilities early in development while maintaining code quality and reducing time-to-market. The system employs advanced prompt engineering techniques and context-aware code analysis to generate secure code patterns, detect security anomalies, and suggest remediation strategies in real-time [35].

Yousef Younes & Yousef Elsheikh (2026) investigate these issues in AI-generated Java code, with a focus on common object-oriented problems such as switch statements, temporary fields, and refused bequest. A structured approach is proposed that combines static analysis tools with explainable AI techniques to better understand why these problems appear. Based on the realized insights, prompts are optimized to guide the AI model towards generating cleaner and more structured code. The results showed clear improvement after the prompt optimization, where the number of detected code smells was reduced by 66%, and completely removed (100% reduction) in some cases [36].

Tanush Sharanarathi (2025) proposed system leverages transformer-based code embeddings for semantic analysis, a FAISS-powered contextual memory for historical learning, and a collaborative multi-agent system consisting of specialized AI agents for energy profiling, compliance monitoring, resource allocation, and maintainability assessment. Adaptive reinforcement learning dynamically refines energy recommendations based on developer interactions, ensuring continuous improvement over time. Empirical evaluations demonstrate that the framework effectively reduces redundant feedback, improves system-wide energy efficiency, and enhances developer trust through explainable AI techniques [37].

Mabroukah Amarif & Salha Owaydat (2024) presented an optimal optimization of software development cost estimation using a genetic algorithm. It provides a fine solution to set the uncertain and ambiguous properties of software factors. It adopts the COCOMO II model formulas and fine-tunes parameters to enable accurate effort and scheduled-time software cost estimation. An experiment has been conducted using a NASA dataset to improve the proposed algorithm. The experimental results show a significant optimization for both software effort up to 97.27% accuracy and scheduled time up to 98.88% [38].

QiaoLi Chen, LiWen Guo & YunChen Feng (2023) propose to improve software development efficiency, reduce software testing costs, improve software quality and user experience, and finally advocate the use of Scrum mode and TDD in agile development, combined with the CDIO concept to strengthen team cooperation, so as to more efficiently complete the implementation and delivery tasks of software projects [39].

Jinsheng Zhang (2022) first introduced the background and significance of software development process management, analyzed the status quo of software development process management under the background of big data, and finally provide the software development process management under the background of big data, maximizing the supervision of the process management information system based on JAVA. The software design, development, maintenance and other processes [40].

Table I summarizes recent studies on AI-assisted software development, code optimization, agile practices, cost estimation, and intelligent software process management.



TABLE I. COMPARATIVE ANALYSIS OF INTELLIGENT PROGRAMMING AND CODE OPTIMIZATION APPROACHES

Pattlola Srinivas et al. (2026)	Generative AI-assisted Software Development Life Cycle (SDLC)	Integrates LLMs, machine learning, prompt engineering, context-aware code analysis, AI-driven threat modeling, and continuous security assessment across SDLC phases.	Improves vulnerability detection, code quality, development efficiency, and secure code generation while reducing time-to-market.	Dependence on quality contextual data; reliability and security issues in AI-generated code; challenges in integrating AI across SDLC phases.	Develop reliable and secure GenAI frameworks with improved context awareness, automated vulnerability remediation, and broader SDLC integration.
Yousef Younes & Yousef Elsheikh (2026)	Code Smell Detection and Optimization in AI-Generated Java Code	Combines static analysis and explainable AI to identify code smells and uses optimized prompts for cleaner code generation.	Prompt optimization reduced detected code smells by 66%, with up to 100% reduction in some cases.	Limited focus on selected Java code smells; effectiveness depends on prompt quality and AI model behavior.	Extend the approach to more programming languages, code smells, and automated refactoring.
Tanush Sharanarathi (2025)	Intelligent Code Optimization and Energy-Efficient Software Development	Uses transformer-based code embeddings, FAISS contextual memory, multi-agent AI, energy profiling, compliance monitoring, resource allocation, and adaptive reinforcement learning.	Reduces redundant feedback, improves software energy efficiency, and increases developer trust through explainable AI.	Multi-agent coordination complexity; dependence on historical data; challenges in measuring software energy consumption.	Develop scalable multi-agent systems with real-time energy optimization and improved adaptive learning.
Mabroukah Amarif & Salha Owaydat (2024)	Software Development Cost and Effort Optimization	Applies a genetic algorithm to optimize COCOMO II parameters using NASA software project data for effort and schedule estimation.	Achieved 97.27% accuracy for effort estimation and 98.88% for scheduled-time estimation.	Performance depends on dataset quality and parameter tuning; validation may be limited by available project datasets.	Integrate hybrid AI optimization algorithms and larger real-world software project datasets.
QiaoLi Chen, LiWen Guo & YunChen Feng (2023)	Agile Software Development and Testing Optimization	Combines Scrum, TDD, and CDIO to improve collaboration, testing, software quality, and project delivery.	Improves development efficiency, reduces testing costs, enhances software quality, and strengthens	Requires effective team coordination and agile adoption; implementation varies by project and organization.	Integrate AI-assisted agile development, automated testing, and intelligent project management.



			team collaboration.		
Jinsheng Zhang (2022)	Big Data-Based Software Development Process Management	Examines software process management in big-data environments and proposes a Java-based system for supervision of design, development, and maintenance.	Supports improved supervision and management of software development processes using big-data technologies.	Dependence on effective data management and system integration; scalability and adaptability challenges.	Incorporate AI-based analytics, predictive process monitoring, and intelligent decision-making.
Pattlola Srinivas et al. (2026)	Generative AI-assisted Software Development Life Cycle (SDLC)	Integrates LLMs, machine learning, prompt engineering, context-aware code analysis, AI-driven threat modeling, and continuous security assessment across SDLC phases.	Improves vulnerability detection, code quality, development efficiency, and secure code generation while reducing time-to-market.	Dependence on quality contextual data; reliability and security issues in AI-generated code; challenges in integrating AI across SDLC phases.	Develop reliable and secure GenAI frameworks with improved context awareness, automated vulnerability remediation, and broader SDLC integration.

V. CONCLUSION AND FUTURE WORK

The programming and code optimization techniques in modern software development, emphasize the growing role of Artificial Intelligence across the Software Development Life Cycle. Traditional development approaches and modern software architectures were examined to establish the foundation for AI-driven development. The study discussed ML, NLP, DL, Transformers, and Generative AI for intelligent programming tasks such as source code completion, code generation, automated review, debugging, refactoring, and optimization. Classical optimization techniques, including code cloning, invariant code motion, dead code elimination, local optimization, global optimization, and inter-procedural optimization, were also considered. Recent studies indicate that intelligent approaches can improve development productivity, code quality, security, energy efficiency, cost estimation, and testing. However, challenges involving data dependency, interpretability, developer trust, technical integration, and reliability remain. Overall, intelligent programming and optimization provide significant opportunities for developing efficient, maintainable, secure, and high-quality software systems.

Future research should focus on developing reliable, explainable, and scalable AI-based programming systems that integrate effectively across the complete SDLC. Greater attention is required toward automated vulnerability remediation, intelligent refactoring, energy-efficient optimization, multilingual code generation, real-time learning, and improved developer trust through transparent and interpretable AI.

REFERENCES

1. R. Lingam, "AI-powered framework for evaluating child-friendly mobile applications," Int. J. Hum. Comput. Intell., vol. 4, no. 4, 2025, doi: 10.5281/zenodo.15624307.



2. N. Karra, N. Karra, L. Echefu, and G. Bandapati, "DevOps-Enabled Cloud-Native Architectures for Scalable and Resilient IoT Applications: A Kubernetes-Based Approach," in 2025 9th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Kirtipur, Nepal: IEEE, 2025, pp. 48–53, October. doi: 10.1109/I-SMAC65734.2025.11597587.
3. S. K. Malaraju, G. C. Kakaraparthi, and R. C. Bondalapati, "Efficient Deployment of Scalable AI Models via Kubernetes Orchestration in Hybrid Cloud Setups," in 2025 International Conference on Intelligent and Secure Engineering Solutions (CISES), Greater Noida Gautam Budh Nagar, India: IEEE, 2025, pp. 1450–1454, August. doi: 10.1109/CISES66934.2025.11265182.
4. M. Z. Haider Ali, A. Arif, S. Z. Haider, M. Azam, M. Malik, and A. Hussain, "A Comprehensive Review Of Software Development Methodologies: Models, Mindset, And Misunderstandings," vol. 3, pp. 996–1029, 2025, doi: 10.5281/zenodo.16414601.
5. P. Sankhe and M. Dixit, "A Review and Survey of Software Development Methodologies," vol. 11, no. 5, pp. 726–733, 2023.
6. C. S. Pareek, "Accelerating Agile Quality Assurance with AI-Powered Testing Strategies," Int. J. Sci. Res. Eng. Manag., vol. 6, no. 01, pp. 1–11, JAN, 2022, doi: 10.55041/IJSREM15369.
7. H. Singh and Y. Singh, "Review paper on software development," vol. 3, no. 2, pp. 268–271, 2015.
8. P. Patel, "Uncertainty-Aware Hybrid Deep Learning for AI-Generated vs Human Image Classification with Adaptive Human-AI Routing," in 2026 2nd International Conference on Artificial Intelligence Systems (AIS), San Antonio, TX, USA: IEEE, May 2026, pp. 537–543. doi: 10.1109/AIS69919.2026.11621535.
9. P. Jinka, "Database Evolution: The Transformation of Data Partitioning and Indexing in the Cloud Era," J. Comput. Sci. Technol. Stud., vol. 7, no. 5, pp. 16–22, May 2025, doi: 10.32996/jcsts.2025.7.5.3.
10. N. Upadhyaya, "The Role of Artificial Intelligence in Software Development: A Literature Review," 2022. doi: 10.13140/RG.2.2.12291.92965.
11. S. Konakanchi, "Artificial Intelligence in Code Optimization and Refactoring," Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol., vol. 11, no. 2, pp. 1197–1211, Mar. 2025, doi: 10.32628/CSEIT25112463.
12. A. Kumar, "New trends and Challenges in Source Code Optimization," Int. J. Comput. Appl., vol. 131, no. 16, pp. 27–32, Dec. 2015, doi: 10.5120/ijca2015907609.
13. N. Paternoster, C. Giardino, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software development in startup companies: A systematic mapping study," Inf. Softw. Technol., vol. 56, no. 10, pp. 1200–1218, Oct. 2014, doi: 10.1016/j.infsof.2014.04.014.
14. M. I. Hossain, "Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management," Int. J. Multidiscip. Res., vol. 5, no. 5, pp. 1–36, Sep. 2023, doi: 10.36948/ijfmr.2023.v05i05.6223.
15. A. Alazzawi, Q. Yas, and B. Rahmatullah, "A Comprehensive Review of Software Development Life Cycle methodologies: Pros, Cons, and Future Directions," Iraqi J. Comput. Sci. Math., vol. 4, no. 4, pp. 173–190, Nov. 2023, doi: 10.52866/ijcsm.2023.04.04.014.
16. M. F. Nasir, M. J. U. Qureshi, P. Mitra, and T. Islam, "Introducing a New SDLC Trigon Model for Software Development," in Proceedings of the International Conference on Sustainable Development in Technology for 4th Industrial Revolution 2021 (ICSDTIR-2021), Chattogram, 2021, pp. 1–7. doi: 10.13140/RG.2.2.26057.01122.
17. P. Agarwal, A. Singhal, and A. Garg, "SDLC Model Selection Tool and Risk Incorporation," Int. J. Comput. Appl., vol. 172, no. 10, pp. 6–10, Aug. 2017, doi: 10.5120/ijca2017915143.
18. S. Shaikh and S. Abro, "Comparison of Traditional & Agile Software Development Methodology: A Short Survey," Int. J. Softw. Eng. Comput. Syst., vol. 5, no. 2, pp. 1–14, Jan. 2020, doi: 10.15282/ijsecs.5.2.2019.1.0057.



19. M. Alenezi and M. Akour, "AI-Driven Innovations in Software Engineering: A Review of Current Practices and Future Directions," *Appl. Sci.*, vol. 15, no. 3, p. 1344, Jan. 2025, doi: 10.3390/app15031344.
20. M. Barenkamp, J. Rebstadt, and O. Thomas, "Applications of AI in classical software engineering," *AI Perspect.*, vol. 2, no. 1, p. 1, Dec. 2020, doi: 10.1186/s42467-020-00005-4.
21. H. Sofian, N. A. M. Yunus, and R. Ahmad, "Systematic Mapping: Artificial Intelligence Techniques in Software Engineering," *IEEE Access*, vol. 10, pp. 51021–51040, 2022, doi: 10.1109/ACCESS.2022.3174115.
22. H. W. Marar, "Advancements in software engineering using AI," *Comput. Softw. Media Appl.*, vol. 6, no. 1, p. 3906, 2024, doi: 10.24294/csma.v6i1.3906.
23. M. Jaiswal, "Software Architecture and Software Design," *SSRN Electron. J.*, vol. 06, no. 11, 2019, doi: 10.2139/ssrn.3772387.
24. S. Panchali, U. Kavirayani, K. Mylavarapu, J. Pilli, P. K. Jannu, and J. Mohammad, "Smart Software Development A Review on AI-Driven Productivity in Coding, Automated Testing, and Deployment Practices," *Int. J. Emerg. Trends Comput. Sci. Inf. Technol.*, vol. 5, pp. 172–181, 2024, doi: 10.63282/3050-9246.IJETCSIT-V5I1P117.
25. V. Methuku, S. Kamatala, P. Naayini, and P. R. Vontela, "From Ethical Principles to Technical Safeguards: A Unified Framework for Safe and Human-Centered Artificial Intelligence," *Am. Int. J. Comput. Sci. Technol.*, vol. 4, no. 5, pp. 26–34, 2022, doi: 10.63282/3117-5481/AIJCSIT-V4I5P103.
26. S. Arveti, M. Kalal, P. Bose, L. A. Yeruva, S. Inamdar, and R. K. Kanneganti, "Model Driven and Modular Software Architecture for Scalable Development," in *2026 International Conference on Computing Theory and Wireless Communications (ICCTWC)*, Ichalkaranji, India: IEEE, Apr. 2026, pp. 1–10. doi: 10.1109/ICCTWC68241.2026.11583718.
27. T. Hliš, L. Četina, T. Beranič, and L. Pavlič, "Evaluating the Usability and Functionality of Intelligent Source Code Completion Assistants: A Comprehensive Review," *Appl. Sci.*, vol. 13, no. 24, p. 13061, Dec. 2023, doi: 10.3390/app132413061.
28. A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and Memory-Efficient Neural Code Completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, May 2021, pp. 329–340. doi: 10.1109/MSR52588.2021.00045.
29. A. Odeh, N. Odeh, and A. S. Mohammed, "A Comparative Review of AI Techniques for Automated Code Generation in Software Development : Advancements , Challenges , and Future Directions," vol. 13, no. 1, pp. 726–739, 2024, doi: 10.18421/TEM131.
30. E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," *IEEE Access*, vol. 10, pp. 82434–82455, 2022, doi: 10.1109/ACCESS.2022.3196347.
31. D. Baulé, C. G. Von Wangenheim, and A. Von Wangenheim, "Automatic code generation from sketches of mobile applications in end-user development using Deep Learning," no. Robinson 2019, 2021, doi: 10.48550/arXiv.2103.05704.
32. J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro, "Automated Source Code Generation and Auto-Completion Using Deep Learning : Comparing and Discussing Current Language Model-Related Approaches," vol. 2, no. 1, pp. 1–16, 2021, doi: 10.3390/AI2010001.
33. T. Patil, A. Jadhav, H. Patil, S. Joshi, and T. Patil, "Comparative Analysis of Code Optimization Techniques with Eminent Applications (Tools)," *Solid State Technol.*, vol. 63, no. 6, 2022.
34. S. Sharma, P. Mangla, and P. Kumari, "Research Paper on Significance of Code Optimization," vol. 7, no. 9, pp. 58–64, 2018.
35. P. Srinivas, A. Rahmath Mohammad, T. Neelima, M. S. Kundiladi, S. Ramamoorthy, and S. Katta, "Generative AI-Assisted Framework for Secure and Autonomous Software Development Lifecycle Optimization," in *2026 6th International Conference on Image Processing and Capsule Networks (ICIPCN)*, 2026, pp. 1700–1705. doi: 10.1109/ICIPCN67432.2026.11438791.



36. Y. Younes and Y. Elsheikh, "Understanding and Reducing Code Smells in AI-Generated Java Code Using Explainable AI and Prompt Optimization," in 2026 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AECT), 2026, pp. 347–352. doi: 10.1109/AECT69724.2026.11657848.
37. T. Sharanarathi, "Adaptive Multi-Agent AI Framework for Real-Time Energy Optimization and Context-Aware Code Review in Software Development," in 2025 5th International Symposium on Computer Technology and Information Science (ISCTIS), 2025, pp. 353–358. doi: 10.1109/ISCTIS65944.2025.11066037.
38. M. Amarif and S. Owaydat, "An Optimal Optimization of Software Development Cost Estimation Using Genetic Algorithm," in 2024 IEEE 4th International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA), IEEE, May 2024, pp. 654–659. doi: 10.1109/MI-STA61267.2024.10599709.
39. Q. Chen, L. Guo, and Y. Feng, "Agile development research based on software engineering to software testing," in 2023 9th International Conference on Systems and Informatics (ICSAI), IEEE, Dec. 2023, pp. 1–3. doi: 10.1109/ICSAI61474.2023.10423308.
40. J. Zhang, "Research on Management Information System of Logistics Industry Software Development Process under the Background of Big Data," in 2022 3rd International Conference on Smart Electronics and Communication (ICOSEC), IEEE, Oct. 2022, pp. 832–835. doi: 10.1109/ICOSEC54921.2022.9951981.

