Volume (1) No (8), 2025
Journal of Global Research in Multidisciplinary Studies (JGRMS)
Review Paper/Research Paper
Available online at https://saanvipublications.com/journals/index.php/jgrms/index

Emerging Trends and Tools in the Modern Software Development Life Cycle: A Survey of Strategies for Modern Engineering

Dr Chintal Kumar Patel
Associate Professor, CSE
Geetanjali Institute of Technical Studies
chintal.patel@gits.ac.in

***Abstract*—The Software Development Life Cycle (SDLC) has undergone dramatic change that has seen the end of rigid and linear approaches to software development and has replaced them with dynamic and iterative approaches to software development, heavily dependent on technology. The paper is a review of thought processes and tools that define new-fangled SDLC practice. It starts by looking at the conventional models, including the Waterfall and V-Model, that focus on sequential development and voluminous documentation. As the requirements of agility, speed/ agility, and customer focus have increased, approaches to meet these demands have become more focused on Cross-functional integration, iterative delivery, and continuous integration/continuous deployment (CI/CD), leading to the increase in the popularity of methodologies such as Agile, Scrum, and DevOps. Adoption of new technologies, such as machine learning (ML) and artificial intelligence (AI), cloud computing, adoption, and containerization, is also discussed in the paper and redefining the design, development, testing, and maintenance of software. There is also a focus on modern approaches of testing, such as shift-left and shift-right testing, which improve the quality of software and increase delivery. The paper also indicates the environmental and ethical factors that shape the sustainable practices in software engineering. This review can be of interest to researchers, developers, and industry professionals who have to look into the future of software engineering and are interested in receiving a complete picture of modern SDLC strategies and tools.**

Keywords*—Software Development Lifecycle (SDLC), Agile, DevOps, Continuous Delivery, Artificial Intelligence, Machine Learning, Shift-Left Testing, Software Engineering.

I. Introduction

The proliferation of programmable computers in the latter part of the 20th century led to an increase in the complexity and variety of software systems. The pressure on software has also increased, so now it has to be provided on a quicker note, with greater functionality, improvement in quality and at low costs. Conventional ways that were employed in the small-scale projects have not been effective in large-scale system development [1]. Consequently, different software development lifecycle (SDLC) models have been introduced to cover the increased complexity and changing demands of software engineering.

The software development life cycle (SDLC) plans proceed in sequential processes like the Waterfall model, then to incremental and iteration models, and finally transformative models like Agile and DevOps [2]. Iterative development has taken center stage in current software practice, which encourages continuous integration, delivery and feedback [3]. The newer concepts, like software architecture and component-based development, are more and more being incorporated into these models, to increase modularity and re-usability.

Another important aspect of contemporary SDLC is its ecological consideration. The computer software sector is also part of the global carbon footprint in the emissions of data centers and e-waste [4]. These difficulties are further complicated by the exponential increase in digital technologies. Not only is it a moral requirement, but also a strategic need, to seek answers to these environmental concerns, which is compelling organizations to adopt sustainable engineering practices and minimize their ecological footprint.

Artificial intelligence (AI) technologies are transforming the process of developing software by enhancing certain lifecycle characteristics. Predictive analytics and anomaly detection can be made possible through Machine Learning (ML), helping teams detect a problem early. Natural Language Processing (NLP) makes the process of requirement gathering efficient through translating user input into tasks [5]. While less widespread, Specialized area like automated UI design and image-based test coverage are also backed by Computer Vision. Next to AI, search-based software engineering (SBSE) has also come to prominence [6]. Genetic programming, simulated annealing, local search, and genetic algorithms are popular optimization methods of solving complex software engineering problems [7]. These approaches provide effective means of test case generation, estimate effort, refactoring, and resource distribution.

The scene of software development is changing due to the incorporation of higher lifecycle concepts, tools based on AI, and environmentally-friendly engineering [8]. These developments are essential to meeting the changing needs of modern systems, as well as the fact that software development is efficient, conscientious, and futuristic.

A. Structure of the Paper

The paper is organized as follows: Section I, the evolution of SDLC is introduced. Section II, the author discusses the traditional and contemporary SDLC models. Section III, new

trends and tools such as AI, ML, DevOps, and shift-left/ right testing are mentioned. Section IV talk is about modern engineering strategies. Section V, a literature review on the main developments in the domain is conducted. The last section VI which provides further study directions at the end of the paper.

II. Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) is a structured model that explains the steps necessary to produce a software product. The classical SDLC models need to follow a straight and systematic process with a great deal of planning, documentation and well-defined stages. These models have largely played a major role in software engineering in giving a regulated path to software development. They provide a graphical and formal representation of the whole cycle of software, all of the tasks required to finish a software product's lifecycle, to his ultimate end of life [9]. Essentially, SDLC frameworks coordinate all the processes inclusive of requirement capture, design, implementation, testing, deploying and maintaining the product to have uniformity and quality in the development lifecycle of the product [10]. As illustrated in Figure 1, the SDLC phases are multifarious.

Fig. 1. Software Development Life Cycle (SDLC) phases

A. *Traditional Software Development Life Cycle (SDLC) Models*

Traditional SDLC models are characterized by a systematic and sequential method in which every step must be completed before proceeding to the next. These models were among the earliest attempts to formalize the software engineering process and have laid the foundation for many of the methodologies used today. Some of the most widely recognized traditional SDLC models include:

1) *Waterfall Model*

A linear sequential SDLC technique, the waterfall model was initially introduced by Royce in 1970 [11]. The phases of design, coding, testing, and requirements analysis, and execution are followed in such a way that, once finished, a phase is not repeated and development does not proceed to the next step until the previous phase is finished. Figure 2 therefore illustrates the Waterfall SDLC paradigm in situations when the project requirements are changeable.

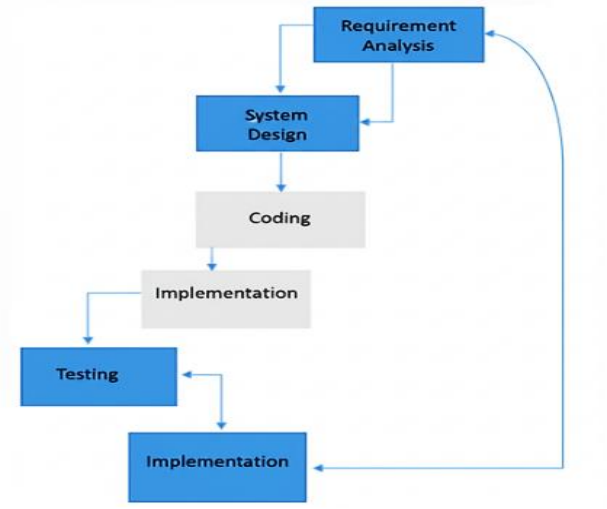

Fig. 2. Waterfall SDLC model

2) *V-Shape Model*

The Verification and Validation model is referred to as the V-model. Processes are carried out sequentially in the V-Shaped lifecycle, much like in the waterfall paradigm. Before moving on to the next phase, each must be finished [12]. The model with a V-shape is illustrate in Figure 3 below, and product testing is scheduled concurrently with a comparable phase of development:

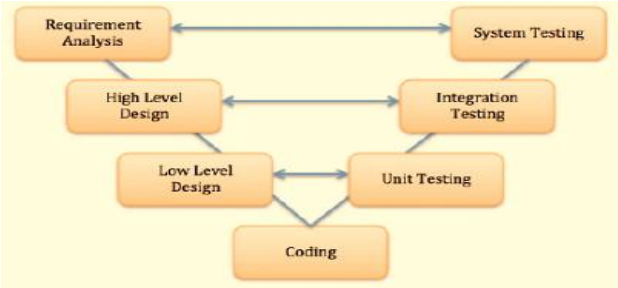

Fig. 3. V Shaped Model

3) *Iterative Model*

The iterative approach addresses the shortcomings of the waterfall paradigm. In contrast with the waterfall approach, which only requires requirements once, the iterative methodology collects requirements at each stage. The project is divided into smaller components so that the results may be used to the next phase [13]. After every increment, the client's input is gathered and utilized to build the next step. and make adjustments (Figure 4 illustrates the iterative model). At each stage, a new software version is created and repeated until the entire system is prepared.

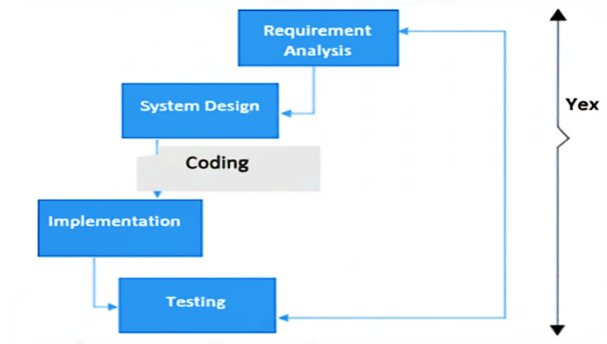

Fig. 4. Iterative Models

B. DevOps and Continuous Delivery

DevOps is the theoretical study of software development and delivery to infrastructure using an integrative and cooperative methodology between software operations (Ops) and developers (Dev). DevOps is an organizational strategy that aims to foster empathy, communication, and cooperation across departments and divisions [14]. The use of DevOps is one way to improve IT team cooperation, which is crucial for software development and maintenance [15].In general, continuous delivery consists of a few essential procedures. One involves developers segmenting their work into minor updates or modifications that are applied to the mainline or trunk in version control. Automated tests provide developers with quick feedback within minutes of committing a change, which is another crucial component [16]. Rapid feedback and learning are made possible by automated testing, which gives developers the chance to address issues as soon as they arise. Continuous integration is the term used to describe these methods used together. Subsequent procedures including thorough automated acceptance testing, manual exploratory testing, and performance testing are initiated if the build and tests are successful. These depend on the software's automated deployment to an environment generated from version control system-stored system and application configuration data.

C. Limitations and Challenges in SDLC

Software development may be approached methodically and systematically with the help of standard SDLC models, they exhibit several inherent limitations that make them less suitable for modern, dynamic development environments [17]. The key challenges include:

- Inflexibility in changing traditional models, particularly linear ones like the Waterfall model, lacks the flexibility to accommodate evolving requirements. It becomes challenging and expensive to go back and change earlier stages after a phase is finished. This rigidity can result in software products that do not fully meet stakeholder needs by the time of delivery.
- Delayed Testing and Feedback Testing in conventional models typically occurs after the implementation phase, which delays the identification and resolution of defects. This late validation raises the possibility of finding important problems at the end of the development cycle, increasing the cost and duration of remediation.
- Limited Stakeholder Involvement Traditional SDLC models often minimize stakeholder and end-user involvement during development. Feedback is usually collected only during the requirements and final testing phases. This limited engagement can lead to misinterpretation of user expectations and a final product that lacks user-centric functionality.
- Overhead in Documentation These models place significant emphasis on exhaustive documentation at each phase. While documentation is important, an overreliance on it can shift focus away from working software. Maintaining and updating extensive documentation also demands substantial time and resources.

III. Emerging Trends and Tools in Software Engineering

The dynamics of software engineering are changing fast, owing to breaking technology and the complexity of systems and the need to have faster and more dependable software production. Emerging trends such as DevOps, shift-left and shift-right testing, microservices architecture, and AI/ML integration are reshaping traditional software development practices [18]. Tools that facilitate containerization (e.g., Docker, Kubernetes), Continuous integration and deployment (CI/CD) and infrastructure as code (IaC), and observability (e.g., Prometheus, Grafana) have become essential in modern development workflows [19]. The trends are focused on automatizing, collaboration, early defect identification and real-time feedback, ultimately resulting in better software quality, a quicker time to market, and easier maintainability. With the software engineering sector perpetually adjusting to the ever-evolving digital environment, adopting the trends of the present and utilizing the modern tools is essential in the development of scalable, secure, and resilient systems.

A. Shift-Left and Shift-Right Testing

The Shift-Right and Shift-Left testing practices make a complete software testing strategy when incorporated together. The shift-left testing methodology, which includes UX review, unit testing, code reviews, test automation, and performance benchmarking, is used for remedial testing, which tries to find and stop flaws early in the product lifecycle. In this manner, teams may identify and address issues as they arise during the initial stages of growth, lowering the likelihood of later, more significant problems. Testing, Conversely, Shift-Right testing is carried out following deployment and entails evaluating the program in real-world circumstances [20]. In an effort to quickly identify and address any issues that may arise in production, this includes gathering user input, performance analysis, and logs. By combining the two approaches, the organizations are able to obtain a comprehensive Image of product quality. Figure 5 indicates Shift-Left testing (early defect prevention) and Shift-Right testing (post-release validation) to enhance software quality throughout the SDLC.

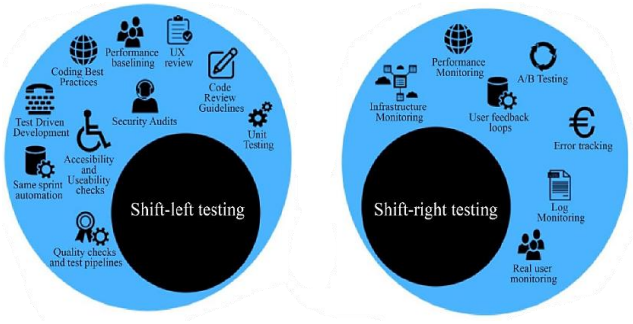

Fig. 5. Comparison of Shift-Left and Shift-Right Testing Activities

B. AI and ML in Software Development

The software development process has undergone significant change as a result of AI technologies, which have made several procedures more effective and efficient. AI is employed in many areas, one of which is automated coding. Code recommendations are given in context by tools like GitHub Copilot, which helps engineers create code more quickly and with fewer errors. These development tools facilitate this process by analyzing and suggesting suitable snippets to be deployed in existing codebases. Quality control

and testing are additional crucial areas [21]. It may be possible to increase the test process's precision and effectiveness by strategically developing and executing test cases using AI-based automated testing frameworks. Tools like Test might be mentioned as an example. Through early issue discovery in the development phase, machine learning enhances software dependability and assists AI in identifying and prioritizing testing scenarios so that developers may focus on high-priority features of their product.

Predictive analytics with machine learning (ML) algorithms allow a team to analyze data, identify patterns, and make prediction-based choices, they have become a crucial component of software projects. The three main categories of these algorithms are reinforcement learning, unsupervised learning, and supervised learning. learning while being watched over. Both the input data and the right response are known since the algorithms employed in supervised learning are trained using labelled data. The most often used ones are support vector machines, decision trees, and linear regression.

C. Cloud-Native and Serverless Architectures

Modern application development in Cloud environments is using serverless and cloud-native designs more and more to improve efficiency, scalability, and agility. Cloud-native approaches utilize technologies such as Using containers, microservices, and orchestration technologies (like Kubernetes) to create robust, modular applications that can rapidly adapt to change [22]. Serverless architecture, on the other hand, emphasizes event-driven computing, where cloud service providers scale apps dynamically in response to demand and manage infrastructure. Unlike traditional IaaS models that require upfront provisioning of resources, serverless applications run in lightweight containers that are triggered only when needed, reducing both operational overhead and costs. This model abstracts infrastructure concerns, freeing developers from tasks like server management, patching, scaling, logging, and monitoring. It supports seamless integration with other cloud services and accelerates deployment cycles [23]. Serverless applications can be built entirely using serverless functions or as hybrids incorporating traditional microservices, enabling flexible and efficient cloud solutions. Figure 6 shows a serverless architecture where events from the UI, API Gateway, and Cloud Event Sources are queued and dispatched to worker nodes for execution. Each worker runs isolated functions, enabling scalable and event-driven processing.

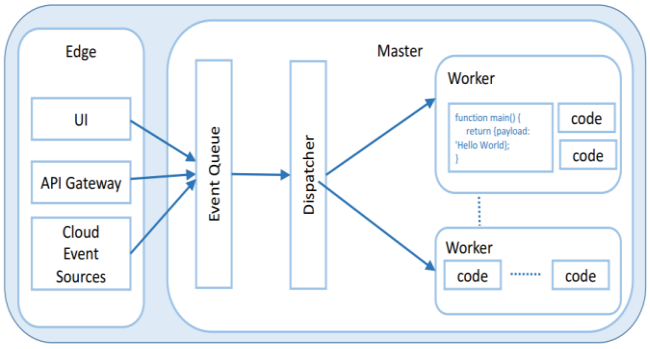

Fig. 6. Serverless Architecture

IV. Strategic Approaches in Modern Engineering

In order to support an organization's goals, strategic management is the art and science of creating, executing, and assessing cross-functional choices. Strategic management is the ongoing process of creating, implementing, and supervising broad plans that guide the organization in achieving its strategic goals in light of the internal and external environment [24]. Strategic planning is used by organizations to support strategic management. Strategic planning analyses the organization and its surroundings to develop strategies that produce more effective and efficient outcomes, such as enhancing current conditions, resources, and capacities to gain a competitive edge [25]. The strategic planning process's two main goals are generally developing strategies and choosing the best one from a variety of possibilities to accomplish the goals of the organization.

A. Agile and Scrum Practices

Agile methodologies have emerged as a dominant paradigm in modern software engineering, particularly for projects characterized by dynamic requirements and technological uncertainty [26]. The following key practices summarize the core principles and practical implementations of Agile and Scrum approaches, especially within distributed development contexts.

- Suitability of Agile for Uncertain and Evolving Projects: Agile methods are ideal for projects with uncertain scope, evolving technologies, and frequently changing requirements. Their flexibility and iterative nature make them especially suitable for Distributed Software Development (DSD), where adaptability is essential.
- Agile in Distributed Development: Tailored Agile practices enable effective coordination among remote teams, supporting asynchronous collaboration through digital tools. Frameworks like Scrum help overcome challenges related to time zones, cultural differences, and geographical dispersion.
- Scrum in Co-located Teams; Traditionally, Scrum has been used in small, co-located teams, where informal communication and rapid feedback enhance cohesion and iterative progress.
- Evidence of Scrum in DSD: Studies confirms successful Scrum adoption in distributed settings, highlighting benefits like improved transparency, team alignment, and stakeholder engagement.
- Inspect-and-Adapt Principle: Scrum promotes continuous improvement through regular inspection, adaptation, and feedback, ensuring responsiveness to change and iterative progress.

Agile, especially Scrum, suits uncertain and evolving projects like distributed software development. It supports flexibility, collaboration, and iterative delivery. While originally for small co-located teams, Scrum scales well for distributed teams using frameworks like SAFe. Studies show it improves transparency, team alignment, and adaptability through its core components and continuous improvement approach.

B. Site Reliability Engineering (SRE)

Software engineering and systems administration are combined in site reliability engineering (SRE), a cutting-edge field that guarantees the availability, performance, and dependability of large-scale systems [27]. Developed at Google, SRE focuses on using engineering principles to manage and automate operational tasks traditionally handled by operations teams. This method places a strong emphasis on using software to increase systems' dependability and

effectiveness, transforming the conventional role of system administrators into that of software engineers focused on operational excellence, as shown as Figure 7.

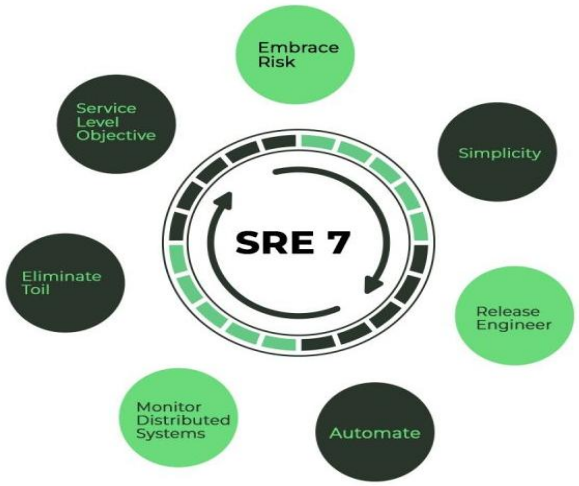

Fig. 7. Key Principles of Site Reliability Engineering (SRE)

The 7 pillars of Site Reliability Engineering (SRE) form the basis of SRE's fundamental philosophy. These include: Embrace Risk, Release Engineer, Automate, Monitor Distributed Systems, Eliminate Toil, Embrace Risk, and SLOs. Together, these principles guide the practices and mindset of SRE teams, emphasizing system reliability, scalability, efficiency, and continuous improvement in managing complex software systems.

C. Infrastructure as Code (IaC)

IaC is the process of writing textual descriptions of computer systems that are readable by machines, usually in the cloud, that can be run automatically. IaC scripts may be written in domain-specific languages on IaC systems as well as virtualization tools and plugins for cloud service provider integration [28]. Different deployment steps are covered by various IaC tools. Computing node software configuration is managed by tools like Ansible, Chef, and Puppet. They are able to deploy, configure, and manage containers and applications. Consequently, they frequently facilitate the embedding of shell scripts and other system configuration languages. The activities needed to achieve the desired configuration are clearly described by Ansible and Chef scripts, which may be regarded as imperative, whereas Puppet expresses them declaratively. Puppet actions are idempotent, meaning that doing them again, whatever many times, has the identical consequences as performing them only once.

D. Data-Driven Decision Making in SDLC

A deliberate change towards using historical and real-time data to inform each stage of contemporary software engineering is represented by data-driven decision making (DDDM) in the Software Development Life Cycle (SDLC). Evaluating user behavior metrics, feature usage metrics, bug trends, and project performance, teams can prioritize requirements, predict the effort that is needed to develop with accuracy, and predict risks early, and customize the testing strategies to focus on the riskiest components. Combining the predictive analytics and machine learning allows making better estimations and effective resource distribution. One-on-one monitoring with the assistance of such tools as Application Performance Monitoring (APM) can guarantee the swift feedback loops in terms of performance tuning and improvement of quality. Moreover, Data stored on CI/CD pipelines and user-centric design insights based on behavioral analytics are becoming the basis of making deployment decisions. DDDM, in general, helps to increase the level of decision accuracy, faster development, increased product quality, and flexibility to adjust to consumers' and businesses' evolving demands.

V. Literature Review

The Literature Review section is a summary of recent developments in SDLC particularized into security integration, dependability, environmental impact, cross-domaining, testing, and integrating the ML in various development phases.

Saeed et al. (2025) is to analyze the latest advancements in the SDLC's security integration field by examining publications written over the past 20 years and suggesting future directions. Planning, execution, and analysis are the three primary phases of the study. It is clear from these that a team effort is required to handle significant software security risks (CSSRs) using efficient risk management and estimating methods. By employing a numerical scale to quantify hazards, a thorough grasp of their seriousness may be obtained, which helps with targeted resource allocation and successful mitigation initiatives. By means of a thorough comprehension of possible weaknesses and Through protection poker-enabled proactive mitigation strategies, organizations may effectively deploy resources to ensure project and activity completion in a dynamic threat environment [29].

Yu and Yang (2024) analyzes the meaning of dependability life cycle, and on the basis of introducing the existing system, data and product life cycle models, constructs the system dependability life cycle model, including concept, development, realization, utilization and retirement/re-use of five life cycle stages. Considering the wide application of software system, the dependability life cycle model of software system is further studied and established. It is possible to foster the realization of system dependability and generate dependability value by implementing dependability activities at every stage of the system life cycle [30].

Simon et al. (2023) Consequently, provide a technique and related model to help stakeholders and the software development ecosystem estimate the environmental effect of their projects across a number of impact categories. A sample case study illustrates the significance of development influence on a software life cycle as well as the relative value of phases and resources used. This all-encompassing method provides useful insights to recognize possible transitions between stages, such as creation and consumption, and hotspots among the resources used to create and run software services [31].

Lee et al. (2023) An international standard for the functional safety of electrical and electronic systems installed in road vehicles, ISO26262/A-SPICE, and DO-178C, a software consideration in airborne systems and equipment certification, were compared in order to assess the applicability of software (SW) technologies accumulated in automobiles to aviation. A handbook was suggested to help software engineers in the automotive and aviation industries better understand one another. An analysis was carried out to compare the two viewpoints: the delivery perspective and the process perspective. Elements of consistency, similarity, and inconsistency were categorized after the goals and operations

of DO-178C were examined and contrasted with ISO26262 from a process standpoint [32].

Gupta and Gayathri (2022) Software testing is carried out to make software stronger and to stop issues. An integral and vital component of the testing stage of the software development life cycle. However, each association has a different way of finishing it altogether. Development now includes software testing, and it is better to begin testing early to prevent issues by addressing defects early. Furthermore, testing is done to enhance unwavering quality, execution, and other important elements that can be classified as SRS throughout the whole software development lifecycle. The SDLC is a comprehensive methodology that outlines the steps involved in designing, developing, and maintaining a particular software product. The organizational structure provides an explanation of each stage of the software development process [33].

Mashkoor et al. (2021) At many phases of the software development life cycle, ML research is utilised. In order to answer the question of whether ML prefers particular stages and techniques, the article's overall goal is to examine the relationship between the stages of the software development life cycle and the types, instruments, and techniques of machine learning. ML is being quickly adopted by the software engineering community to move contemporary software towards very intelligent and self-learning systems [34].

Table I provides the literature on Data Validation Techniques in Distributed Databases Used by Web-Based Systems, including important results, difficulties, and future prospects of the study.

TABLE I. LITERATURE REVIEW ON EMERGING TRENDS AND TOOLS IN THE MODERN SOFTWARE DEVELOPMENT LIFE CYCLE

Author	Study On	Approach	Key Findings	Challenges	Future Directions
Saeed et al. (2025)	Security integration in the Software Development Life Cycle (SDLC)	Systematic review of 100 articles (2005–2025); staged approach: planning, execution, analysis	Emphasizes the need for collaborative approaches to address Critical Software Security Risks (CSSRs); quantification of risks using numeric scales; highlights tools like Protection Poker for risk prioritization	Lack of unified methods for risk quantification; difficulty integrating security seamlessly into SDLC phases	Develop automated threat analysis and security testing tools; promote proactive risk estimation and mitigation practices
Yu and Yang (2024)	System dependability life cycle model	Construction of a dependability life cycle model	Proposed a comprehensive model including five stages: concept, development, realization, utilization, retirement/reuse, enhancing system dependability.	Integrating dependability activities consistently across life cycle phases.	Broaden application to diverse software systems to create continuous dependability value.
Simon et al. (2023)	The environmental impact of software development	Holistic methodology and impact model	Developed a methodology to estimate environmental footprint across software life cycle phases; revealed hidden costs and key consumption hotspots.	Lack of awareness and tools for environmental evaluation in SDLC.	Encourage sustainability-driven design decisions and integrate into DevOps workflows.
Lee et al. (2023)	Cross-domain applicability of SW safety standards	Comparative study between DO-178C and ISO26262/A-SPICE	Provided a bridge between automotive and aviation software practices by identifying similarities and inconsistencies.	High complexity in aligning domain-specific standards.	Develop unified frameworks for cross-industry collaboration and certification.
Gupta and Gayathri (2022)	Importance of early testing in SDLC	Review of software testing practices across organizations	Emphasized shift-left testing, improving quality and performance when testing starts early in the cycle.	Variability in testing methods across organizations.	Promote standardized shift-left testing practices across industries.
Mashkoor et al. (2021)	Machine learning in SDLC	Investigation of ML tools across SDLC phases	Demonstrated increasing use of ML for automation, predictive analytics, and decision support in development stages.	Identifying suitable ML models for each SDLC phase.	Advance AI/ML integration for autonomous and intelligent software development environments.

VI. CONCLUSION AND FUTURE WORK

The evolution of software development lifecycle (SDLC) models has progressed from traditional, linear approaches such as the Waterfall and V-Model to modern, adaptive methodologies like Agile, DevOps, and Continuous Delivery. It also explored the impact of emerging technologies, including AI, ML, cloud-native architectures, and complex testing techniques like shift-left and shift-right testing. These advancements are meant to increase software quality, expedite delivery, and simplify processes. Even with these developments, there are still issues. Among the primary limitations, one must state that no one particular SDLC framework can apply to every type of project and field. Also, the deliberate use of novel solutions, like the integration of AI and automation of DevOps processes, involves similarly high-competence employees, which becomes a problem when smaller organizations or teams with less technical background are to implement it.

The future work is to develop hybrid and customizable SDLC frameworks that are flexible to any project specifications and industrial requirements. It is necessary to create simple, automated solutions that allow non-expert users to use AI and ML technologies more frequently as well. Moreover, software development should also pay more attention to sustainability being concerned about energy-efficient algorithms, sustainable coding, and reducing carbon footprint of large systems. Finally, additional empirical, cross-industry research is required to assess the on-the-job efficacy and investment payback (ROI) of the current tools and techniques across the different organizational contexts.

REFERENCES

[1] K. Kyeremeh, “Overview of System Development Life Cycle Models,” *SSRN Electron. J.*, 2019, doi: 10.2139/ssrn.3448536.

[2] A. Atadoga, U. J. Umoga, O. A. Lottu, and E. O. Sodiya, “Tools, techniques, and trends in sustainable software engineering: A critical review of current practices and future directions,” *World J. Adv. Eng. Technol. Sci.*, vol. 11, no. 1, pp. 231–239, Feb. 2024, doi: 10.30574/wjaets.2024.11.1.0051.

[3] V. Prajapati, “Advances in Software Development Life Cycle Models : Trends and Innovations for Modern Applications,” *J. Glob. Res. Electron. Commun.*, vol. 1, no. 4, pp. 1–6, 2025.

[4] R. Patel and P. B. Patel, "The Role of Simulation & Engineering Software in Optimizing Mechanical System Performance," *TIJER – Int. Res. J.*, vol. 11, no. 6, pp. 991–996, 2024.

[5] S. Rongala, S. A. Pahune, H. Velu, and S. Mathur, "Leveraging Natural Language Processing and Machine Learning for Consumer Insights from Amazon Product Reviews," in *2025 3rd International Conference on Smart Systems for Applications in Electrical Sciences (ICSSES)*, 2025, pp. 1–6. doi: 10.1109/ICSSES64899.2025.11009528.

[6] N. Upadhyaya, "The Role of Artificial Intelligence in Software Development: A Literature Review." pp. 1–5, 2022. doi: 10.13140/RG.2.2.12291.92965.

[7] M. Harman, S. A. Mansouri, and Y. Zhang, "Search Based Software Engineering : A Comprehensive Analysis and Review of Trends Techniques and Applications," 2009.

[8] Z. Wang, B. Li, and Y. Ma, "An Analysis of Research in Software Engineering : Assessment and Trends," pp. 1–25, 2014.

[9] D. Patel, "Zero Trust and DevSecOps in Cloud-Native Environments with Security Frameworks and Best Practices," *Int. J. Adv. Res. Sci. Commun. Technol.*, vol. 3, no. 3, 2023.

[10] B. Acharya and K. Sahu, "Software Development Life Cycle Models: A Review Paper," *Int. J. Adv. Res. Eng. Technol.*, vol. 11, no. 12, pp. 169–176, 2020, doi: 10.34218/IJARET.11.12.2020.019.

[11] A. Mishra and D. Dubey, "A Comparative Study of Different Software Development Life Cycle Models in Different Scenarios," *Int. J. Adv. Res. Comput. Sci. Manag. Stud.*, vol. 1, no. 5, pp. 2321–7782, 2013.

[12] S. S. Kute and S. D. Thorat, "A Review on Various Software Development Life Cycle (SDLC) Models," *Int. J. Res. Comput. Commun. Technol.*, vol. 3, no. 7, pp. 776–781, 2014.

[13] G. Gurung, R. Shah, and D. P. Jaiswal, "Software Development Life Cycle Models-A Comparative Study," *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol.*, 2020, doi: 10.32628/cseit206410.

[14] M. H. R. Istifarulah and R. Tiaharyadini, "DevOps, Continuous Integration and Continuous Deployment Methods for Software Deployment Automation," *JISA(Jurnal Inform. dan Sains)*, vol. 6, no. 2, pp. 116–123, 2023, doi: 10.31326/jisa.v6i2.1751.

[15] G. Modalavalasa, "The Role of DevOps in Streamlining Software Delivery: Key Practices for Seamless CI/CD," *Int. J. Adv. Res. Sci. Commun. Technol.*, vol. 1, no. 12, pp. 258–267, Jan. 2021, doi: 10.48175/IJARSCT-8978C.

[16] Y. SKA and J. P, "A Study and Analysis of Continuous Delivery, Continuous Integration in Software Development Environment," *J. Emerg. Technol. Innov. Res.*, vol. 6, no. 9, pp. 96–107, 2019.

[17] A. Goyal, "Optimising Cloud-Based CI/CD Pipelines: Techniques for Rapid Software Deployment," *Tech. Int. J. Eng. Res.*, vol. 11, no. 11, pp. 896–904, 2024.

[18] S. Laato, M. Mäntymäki, A. K. M. N. Islam, S. Hyrynsalmi, and T. Birkstedt, "Trends and Trajectories in the Software Industry: implications for the future of work," *Inf. Syst. Front.*, vol. 25, no. 3, pp. 929–944, Apr. 2022, doi: 10.1007/s10796-022-10267-4.

[19] A. Goyal, "Optimising Software Lifecycle Management through Predictive Maintenance : Insights and Best Practices," *Int. J. Sci. Res. Arch.*, vol. 07, no. 02, pp. 693–702, 2022.

[20] P. Purushothaman, "Beyond Deployment: Unveiling the Dynamics of Shift-Right Testing," *Int. J. Comput. Trends Technol.*, vol. 72, no. 2, pp. 22–26, 2024, doi: 10.14445/22312803/ijctt-v72i2p104.

[21] H. Hourani, A. Hammad, and M. Lafi, "The impact of artificial intelligence on software testing," *2019 IEEE Jordan Int. Jt. Conf. Electr. Eng. Inf. Technol. JEEIT 2019 - Proc.*, pp. 565–570, 2019, doi: 10.1109/JEEIT.2019.8717439.

[22] V. Prajapati, "Cloud-Based Database Management: Architecture, Security, challenges and solutions," *J. Glob. Res. Electron. Commun.*, vol. 01, no. 1, pp. 07–13, 2025.

[23] S. V. Srivastava and S. Bhosale, "DevSecOps in Cloud-Native and Serverless Architectures," 2023.

[24] H. de S. Andrade and G. Loureiro, "A Comparative Analysis of Strategic Planning Based on a Systems Engineering Approach," *Bus. Ethics Leadersh.*, vol. 4, no. 2, pp. 86–95, 2020, doi: 10.21272/bel.4(2).86-95.2020.

[25] S. P. Kalava, "Enhancing Software Development with AI-Driven Code Reviews," *North Am. J. Eng. Res.*, vol. 5, no. 2, pp. 1–7, 2024.

[26] E. Hossain, M. A. Babar, and H. Y. Paik, "Using scrum in global software development: A systematic literature review," *Proc. - 2009 4th IEEE Int. Conf. Glob. Softw. Eng. ICGSE 2009*, no. May 2014, pp. 175–184, 2009, doi: 10.1109/ICGSE.2009.25.

[27] C. Mokkapati, S. Jain, and S. Jain, "Enhancing Site Reliability Engineering (SRE) Practices in Large-Scale Retail Enterprises," *Int. J. Creat. Res. Thoughts (IJCRT*, vol. 9, no. 11, pp. 870–886, 2021.

[28] M. Chiari, M. De Pascalis, and M. Pradella, "Static Analysis of Infrastructure as Code: A Survey," *2022 IEEE 19th Int. Conf. Softw. Archit. Companion, ICSA-C 2022*, pp. 218–225, 2022, doi: 10.1109/ICSA-C54293.2022.00049.

[29] H. Saeed, I. Shafi, J. Ahmad, A. A. Khan, T. Khurshaid, and I. Ashraf, "Review of Techniques for Integrating Security in Software Development Lifecycle," *Comput. Mater. Contin.*, vol. 82, no. 1, pp. 139–172, 2025, doi: 10.32604/cmc.2024.057587.

[30] M. Yu and C. Yang, "Research on dependability life cycle model for software system," in *2024 5th International Conference on Computer Engineering and Application (ICCEA)*, 2024, pp. 489–493. doi: 10.1109/ICCEA62105.2024.10604201.

[31] T. Simon, P. Rust, R. Rouvoy, and J. Penhoat, "Uncovering the Environmental Impact of Software Life Cycle," in *2023 International Conference on ICT for Sustainability (ICT4S)*, 2023, pp. 176–187. doi: 10.1109/ICT4S58814.2023.00026.

[32] D. Lee, J. Baek, D. Kim, H. Kim, K. Park, and J. Lee, "Adopting automotive software technology to aviation through a comparative analysis of software development standards," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023, pp. 1–4. doi: 10.1109/DASC58513.2023.10311335.

[33] S. Gupta and N. Gayathri, "Study of the Software Development Life Cycle and the Function of Testing," in *International Interdisciplinary Humanitarian Conference for Sustainability, IIHC 2022 - Proceedings*, 2022. doi: 10.1109/IIHC55949.2022.10060231.

[34] S. Shafiq, A. Mashkoor, C. Mayr-Dorn, and A. Egyed, "A Literature Review of Using Machine Learning in Software Development Life Cycle Stages," *IEEE Access*, vol. 9, pp. 140896–140920, 2021, doi: 10.1109/ACCESS.2021.3119746.

