Journal of Computers, Mechanical and Management

Journal homepage: www.jcmm.co.in

Volume 2 Issue 1

Reviewing Software Testing Models and Optimization Techniques: An Analysis of Efficiency and Advancement Needs

Sarvesh Kumar*

Department of Computer Science Engineering, Babu Banarasi Das University, Lucknow, Uttar Pradesh, India 226028

Abstract

Software testing is a crucial component of software engineering that aims to confirm the intended functionality of software modules and minimize the likelihood of future failures. This paper provides a comprehensive review of various software testing models and optimization techniques available in the literature, emphasizing their performance analysis and related research papers. The paper analyzes and discusses the most commonly used software testing models, including waterfall, incremental, V-model, agile, and spiral models, and identifies several areas for improvement to increase their effectiveness. These areas include using machine learning techniques to automate and optimize testing processes, reducing the number of test cases required, and introducing new metrics to gauge the success of testing. Moreover, the paper suggests developing entirely novel methods to deal with the challenges of contemporary software programs, such as the Internet of Things and artificial intelligence. This paper aims to analyze various software testing models and optimization techniques thoroughly, highlight their advantages and disadvantages, and suggest improvements to increase their efficiency and effectiveness. By continuously improving and optimizing software testing processes, software modules can function as intended, minimizing the likelihood of future failures.

Keywords: Software Testing; Optimization Techniques; Test Cases; Performance Analysis

1 Introduction

Testing is a validation and verification process determining whether a specific system meets its originally specified requirements. It aims to identify bugs, errors, or missing requirements in a developed system or software. The process provides stakeholders with precise information about the product's quality [1–4]. Software testing is essential for various reasons, such as saving money by identifying and fixing bugs early in the development process, ensuring the security of users' personal information, improving the quality of the product, enhancing customer satisfaction, and facilitating the development process [5]. Software testing must be thoroughly executed throughout the software development lifecycle to ensure that software modules meet the desired quality standards. Software testing can be divided into two types: static and dynamic. Static testing involves examining the code passively, including code reviews, syntax checks, and walkthroughs. On the other hand, dynamic testing examines the code as it runs, allowing security checks to be performed while the code or application executes. Both approaches are suitable and complement each other [6–8]. Despite decades of research and advancements in software testing, it remains a challenging aspect of continuous software development. The complexity of software systems and their environments contributes to this challenge, as software systems run on various platforms and environments. Although there have been efforts to automate the testing process, achieving 100% automation is still not feasible [2]. Additionally, long-standing issues remain when qualifying and evaluating testing criteria and reducing retesting after software changes.

*Corresponding author: kr.sarvi91@gmail.com

Received: 16 January 2023; **Accepted:** 23 February 2023; **Published:** 28 February 2023

© 2023 Journal of Computers, Mechanical and Management.

This is an open access article and is licensed under a Creative Commons Attribution-Non Commercial 4.0 International License.

DOI: 10.57159/gadl.jcmm.2.1.23041.

Researchers' interest in this topic has grown with numerous specialized events and workshops and an increasing percentage of testing papers in software engineering conferences and journal periodicals [9]. This review paper aims to provide a comprehensive analysis of commonly used software testing models and optimization techniques. The paper examines the effectiveness of selected testing models and optimization techniques and discusses their strengths and limitations. Additionally, the paper identifies areas where improvements and advancements are needed to increase the efficiency and effectiveness of software testing. The review paper is structured in the following way: the first section provides an overview of software testing and its importance in software engineering, the second section reviews various software testing models and evaluates their effectiveness in identifying flaws and reducing testing time and effort, the third section discusses optimization techniques such as test case reduction, fault localization, mutation testing, and combinatorial testing, the fourth section analyzes the efficiency and need for advancement of these models and techniques, and finally, the paper concludes by summarizing the findings and suggesting future research areas.

2 Software Testing Models

Software testing models serve as the foundation for systematically testing software applications. Testing is an essential aspect of the software development life cycle. Various models or approaches can be utilized throughout the development process, each with advantages and disadvantages [10]. This section discusses commonly used software testing models, providing an overview of their strengths and limitations.

2.1 Waterfall model

The waterfall model is an early and straightforward software development process involving a linear phase sequence. The model is named after the flow of one phase into the next, resembling a cascade. In the waterfall approach, user research is conducted at the beginning and end of the project to inform the requirements and evaluate a working prototype or finished product. The methodology requires completing phases in a specific order, each with formal exit criteria. The approach involves a detailed list of tasks, supporting documentation with exit criteria, and larger companies often mandate the use of SDLC methodology products [11–13]. Advantages of the waterfall methodology include early completion of requirements, better resource utilization, superior application design, and easier measurement of project status. However, there are also drawbacks to the approach, including difficulties in obtaining comprehensive business requirements upfront, the need for a highly detailed breakdown of tasks and deliverables, and projects that frequently span months or quarters, resulting in being behind schedule, exceeding budget, and not meeting expectations [14]. Despite its limitations, researchers and practitioners still widely use the waterfall model to solve problems that require a structured and sequential approach to software development. For instance, Swara et al. [15] presented a study on developing an Android-based information system for business travel to improve marketing, ordering, payment, and departure processes. The authors utilized the waterfall model as a development methodology and conducted observations to evaluate the system's success. The study concluded that the system achieved 100% functionality after black box testing, and users were highly satisfied with the system, as indicated by the average satisfaction score of 4.44 on a Likert scale.

This article provides insights into the practical application of the waterfall model in developing an information system and demonstrates its effectiveness in achieving project goals. Researchers have applied the waterfall model in various contexts, such as developing a coffee shop website in Malang by Ardhiansyah et al. [16], developing a Job Training Management Information System at Trunojoyo University Madura by Herawati et al. [17], developing an application for processing report evaluation of Islamic junior high schools based on boarding Pesantren by Rahayu et al. [18], developing a prototype application for finding and ordering boarding houses in Telang by Negara et al. [19], building a web-based conference registration system by Badri et al. [20], and creating a monitoring system for pregnant women and newborns by Purba and Sondang [21].Figure 1 depicts the waterfall model approach in its general form.

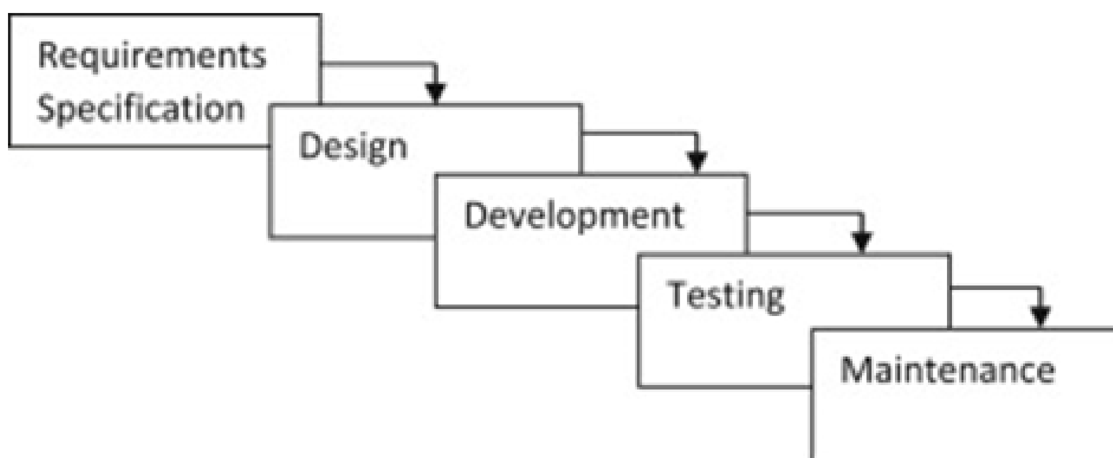

Figure 1: Waterfall model approach of software testing [22].

2.2 Incremental model

The incremental model is a software development methodology that involves multiple iterations of smaller cycles, including requirements, design, development, and testing. Each iteration produces a prototype of the software that builds on the previous prototype, making testing and managing easier with early error detection. This approach allows the user to plan the system's use and determine its requirements, and it also supports changing user requirements. However, frequent user feedback can lead to scope creep, and a never-ending development loop may occur. Since the requirements for the entire system are not gathered at the start of the project, the system architecture may be affected in later iterations. Additionally, a build-and-patch approach can lead to poor code design. Despite these drawbacks, the incremental model remains an effective approach, particularly for novel systems with unclear constraints or requirements [22–26]. Figure 2 represents the incremental model approach.

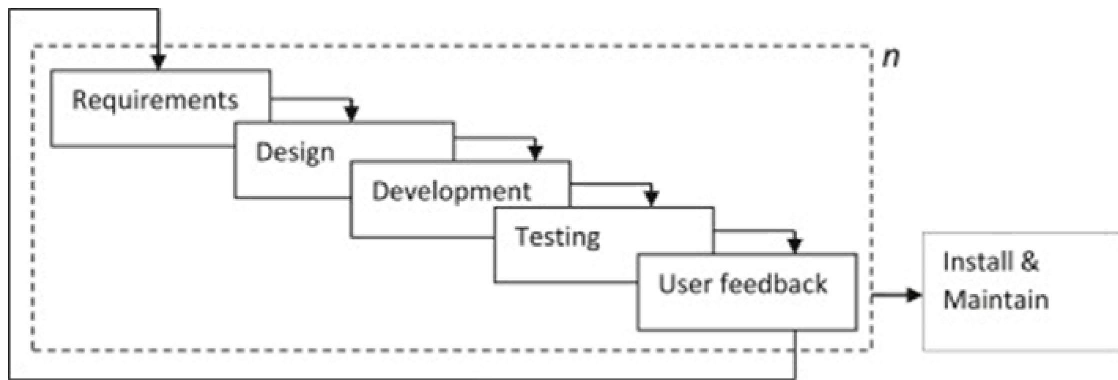

Figure 2: Incremental model approach [22].

Several recent studies have demonstrated the importance and effectiveness of incremental models in software development and testing, and the proposed strategies and models provide useful insights into the application of incremental models in various domains. For example, Shahzad et al. [27] proposed a strategy for identifying and mitigating software risks in the incremental software development model. The authors identified risk factors that may exist in other software development processes but are particularly relevant to the incremental model. Qiu and Riesbeck [28] described an incremental model for developing educational critiquing systems that integrated manual critiquing with critique authoring to facilitate the development of educational critiquing systems with the less upfront development effort.

Andreansyah et al. [29] described the implementation of incremental models in the development of web-based loan cooperative applications. The authors developed a web-based application to overcome the problems of a particular cooperative and tested it with 31 respondents from the company. The application received high ratings in terms of usability and usefulness. Lity et al. [30] proposed an automated change impact analysis based on incremental model slicing for incremental software product line (SPL) testing. The authors applied incremental slicing to determine the impact of applied model changes and to explain their potential retest. The effectiveness and applicability of the proposed approach were evaluated using four SPLs. Lochau et al. [31] also proposed an automated change impact analysis based on incremental model slicing for incremental SPL testing. The authors applied incremental slicing to determine the impact of applied model changes and to explain their potential retest. The proposed approach was evaluated using four SPLs, and the authors demonstrated its applicability and effectiveness.

2.3 V-model

The V-Model is an approach to system development projects that the State of Germany commissioned. It considers the entire lifecycle of a system and is nicely fitting for the line of thinking in systems engineering [32]. The V-model is regarded as an extension of the waterfall model, and in this methodology, software development processes occur in a sequential approach with a V shape that involves a sequence of processes. It is also referred to as the verification and validation model. The V-model is considered a high-level design of Test Driven Development (TDD), where each software development phase is directly associated with a corresponding testing phase. Each corresponding testing phase is planned in parallel with the development phase. Test cases are developed in the development phase to be implemented in the corresponding testing phase, but typically, testing is conducted once the software is completed [33]. The V-model is best suited for projects with clear and well-defined requirements. The model is unsuited for projects with frequently changing requirements because it does not allow much iteration or adaptation [34]. Figure 3 represents the simple V-model. The V-model integrates testing into every stage of the development process, making it highly effective for projects with a clear set of requirements. Defects can be detected and resolved early on in the process, reducing the overall cost and time required for testing. Furthermore, the high level of traceability between requirements, design, and testing ensures that every aspect of the product is thoroughly tested and meets the specified requirements. The V-model also encourages better collaboration between development and testing teams, as each stage has a corresponding testing stage executed in parallel. This ensures regular communication and coordination between team members, essential for producing high-quality software. Despite its many advantages, the V-model does have some limitations. One of the main drawbacks of this model is its limited scope for iteration and adaptation. Once a stage is completed, it can be challenging to make changes or adjustments without disrupting the entire process. Additionally, the cost and time required for testing can be higher than other models, as testing is integrated into each stage of the development process.

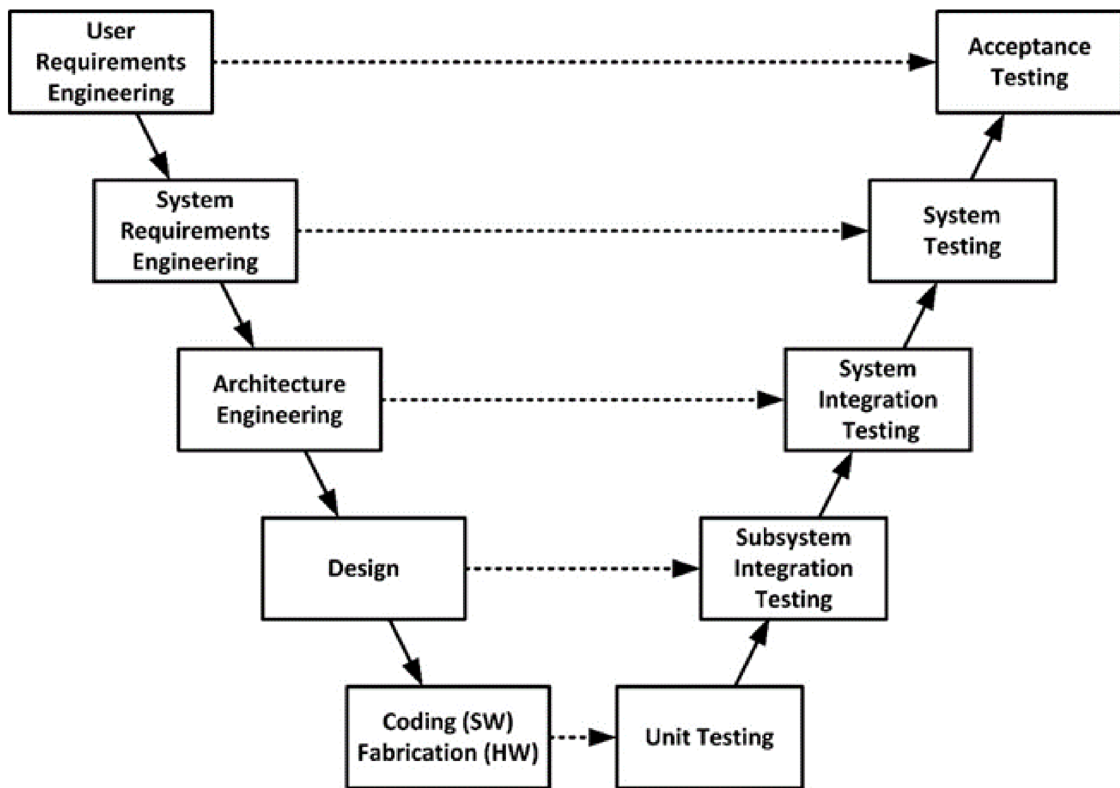

Figure 3: Representation of a simple V-model [35].

This can make the V-model less suitable for projects with limited resources. Finally, the V-model is unsuited for projects with frequently changing requirements, as it is designed to work best with a clear and well-defined set of requirements. If requirements are constantly changing, it cannot be easy to maintain the level of traceability needed for effective testing [35–37]. Despite its limitations, the V-model remains a popular and effective software development model for projects with clear and well-defined requirements. At present times, several researchers are using improved versions of the V-model. For example, Liu et al. [38] proposed an improved V-model process for automotive development. This process introduced early and continuous integrated verification enabled by simulation-based development to address the increasing complexity of modern vehicle systems. In another study, Lim and Chin [39] presented a V-model mobile app development technique incorporating two fuzzy quality function deployment (FQFD) phases. The study demonstrated the development of an online statistical process control app.

FQFD was used to structurally relate user requirements, system requirements, and design strategies in the V-model's verification phases. Integrating FQFD and the V-model in developing mobile apps is a novel contribution of this study. Hynninen et al. [40] addressed the gap between software engineering process terminology in formal education and the practical skills relevant to testing-related work. The authors proposed an approach to map the V-model development phases and testing levels with corresponding actual testing techniques. The approach was evaluated by designing the weekly topics, learning goals, and testing activities for a 7-week introductory course on the basics of software testing and quality assurance. Based on the course outcomes and recent literature, the strengths and weaknesses of the proposed curriculum were discussed. The study aimed to solve the problem of students having little knowledge about what is done during the V-model's development phases and testing levels, as the V-model is mainly conceptual and tied to the steps in the Waterfall model.

Khan et al. [41] proposed an enhanced V-model for developing complex medical devices in their study. They noted that with technological advancement, medical devices have evolved from simple hardware machines to integrated hardware and software systems. Regulatory bodies have imposed rules that medical devices must adhere to ensure the safety of both the hardware and software. However, the traditional Waterfall or Agile models may not be suitable for developing these devices. Therefore, the authors proposed an enhanced V-model and applied its recommendations to developing their wave therapeutic device. Febriyani et al. [42] proposed a verification and validation model of business processes to achieve business goals in implementing Enterprise Architecture designs based on the V-model. They emphasized the importance of testing and validating the design results of the Enterprise Architecture, specifically Business Architecture, to ensure their accuracy and suitability for the company's needs. The study suggests that using Enterprise Architecture designs based on the V-model in business process modeling is a good approach for error checking and achieving the company's business goals.

2.4 Agile model

The term ”Agile Software Development” was coined by the Agile Manifesto. It is an iterative approach that emphasizes incremental specification, design, and implementation, while also requiring full integration of testing and development. The Agile development process was influenced by the Rapid Application Development (RAD) methodology [2].

Agile Software Development is well-established in the software industry and has been widely adopted by hundreds of large and small companies to reduce costs and increase their ability to handle changes in dynamic market conditions [43, 44]. The Agile approach prioritizes communication, continuous integration, rapid delivery of software modules, and an iterative and incremental approach. However, it also has limitations, such as a lack of upfront planning, insufficient documentation, and a lack of predictability [45]. Figures 4 (a) and 4 (b) illustrate the Agile Software Development cycle and the generalized process flow for the Agile model, respectively [46].

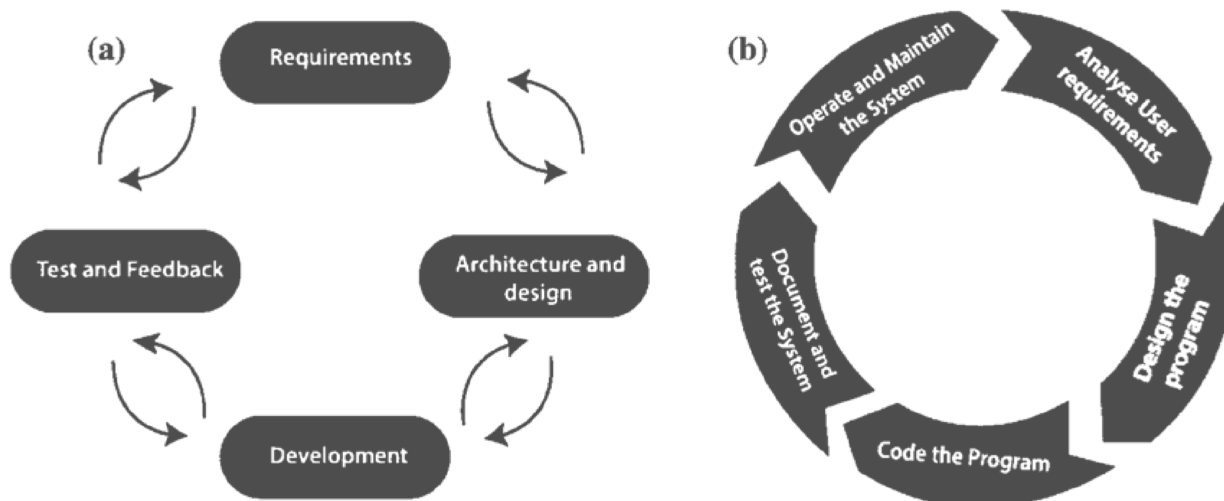

Figure 4: Agile model **(a)**software development cycle; **(b)**process flow [46]

Several researchers have been working on Agile testing models since its development. For example, Kahles et al. [47] presented a methodology for automating root cause analysis in Agile software testing environments using machine learning techniques. They extracted relevant features from raw log data, clustered them, and labeled them with ground-truth categories defined by testing engineers. Artificial neural networks were then trained on the labeled data to classify or pre-process it for clustering. In their study, Dhir and Kumar [48] discussed their approach to automating root cause analysis using machine learning techniques in Agile software testing environments. They proposed an Agile testing model that worked with the productization team in a planned and organized manner to deliver the products in the sprint. They conducted experimental work on a web application to evaluate the outcomes of their approach using the Agile testing model and compared it with traditional automated testing models. Elgrably and Oliveira [49] investigated the construction of a software testing syllabus using Agile practices to support its adherence to academic teaching skills. The research provided a set of skills considered favorable for teaching software testing, which academic program managers and teachers can use to facilitate the construction of syllabuses or subjects related to tests, learning objects, and academic syllabuses. Moreover, Kaur et al. [51] stressed the importance of the development team's continuous integration and deployment in Agile software development. They argued that automated testing could help improve software quality by allowing developers to identify and fix issues early in the development cycle. They also highlighted the importance of testing in real-world conditions and recommended combining manual and automated testing methods.

2.5 Spiral model

The Spiral model offers several advantages over other development models. It provides high flexibility and adaptability to the development process, allowing for continuous refinement and improvement throughout the project's lifecycle. It enables early identification and mitigation of potential risks and issues, as risk analysis is crucial to each spiral iteration. The model also allows for better stakeholder involvement and communication, as each spiral iteration allows stakeholders to review and provide feedback on the project's progress [52, 53]. Fig. 5 provides a general representation of the spiral model. The Spiral Model is a software development framework that uses the Waterfall Model for each version, with subsequent spirals adding functionality to the baseline spiral. It assumes hierarchical requirements and explicitly specifies risk analysis and management. While suitable for well-defined problems, it is not recommended for independent functions. The model provides a framework for designing processes based on project risk levels and can accommodate any development process model. It focuses on identifying and eliminating high-risk problems through careful process design guided by risk management principles [22], [54]. Nevertheless, researchers globally have been working on various applications of the Spiral model. For example, Sharma and Saha [55] proposed an improved Moth-Flame Optimization (MFO) algorithm for object-oriented testing. The paper explained that software testing had become a challenging task with the introduction of object-oriented technology due to its key concepts, such as polymorphism, encapsulation, and inheritance, which introduced various threats to testing. The paper stated that model-based testing was a cheaper and faster methodology, but optimal test path generation was still an open research area. The paper presented the use of the Fermat spiral instead of the logarithmic spiral to capture the spiral motion of moths and optimize the MFO algorithm. The improved algorithm was applied to State Transition Diagrams (STDs) of seven object-oriented software applications to produce test paths.

Aimicheva et al. [56] proposed a Spiral model for teaching mobile application development to bridge the programming knowledge gap between high school and higher education. The model covered all levels of programming education and aimed to effectively train highly qualified mobile developers in Kazakhstan's education system. Supiyandi et al. [57] developed a web-based Village Information System (SID) for Tomuan Holbung Village in North Sumatra, using the Spiral method for development. The system includes analysis, design, coding, testing, and entity relationship diagrams for the database. It facilitates effective and efficient processing of village information data and provides information about village government and activities. The web-based system can be adjusted by users, serving as a means of information in web development for the village. Khadapi [58] implemented the Spiral method to design and analyze a financial information system for the Cashier Financial Management Section (Cash Information Replacement) of PT Telekomunikasi Indonesia.Tbk cooperative. This research aimed to address the problems faced by employees, such as mixed financial data and poor data management. The Spiral method was used to develop a web-based application system that could enumerate financial arrangements, making it easier for employees to manage their finances.

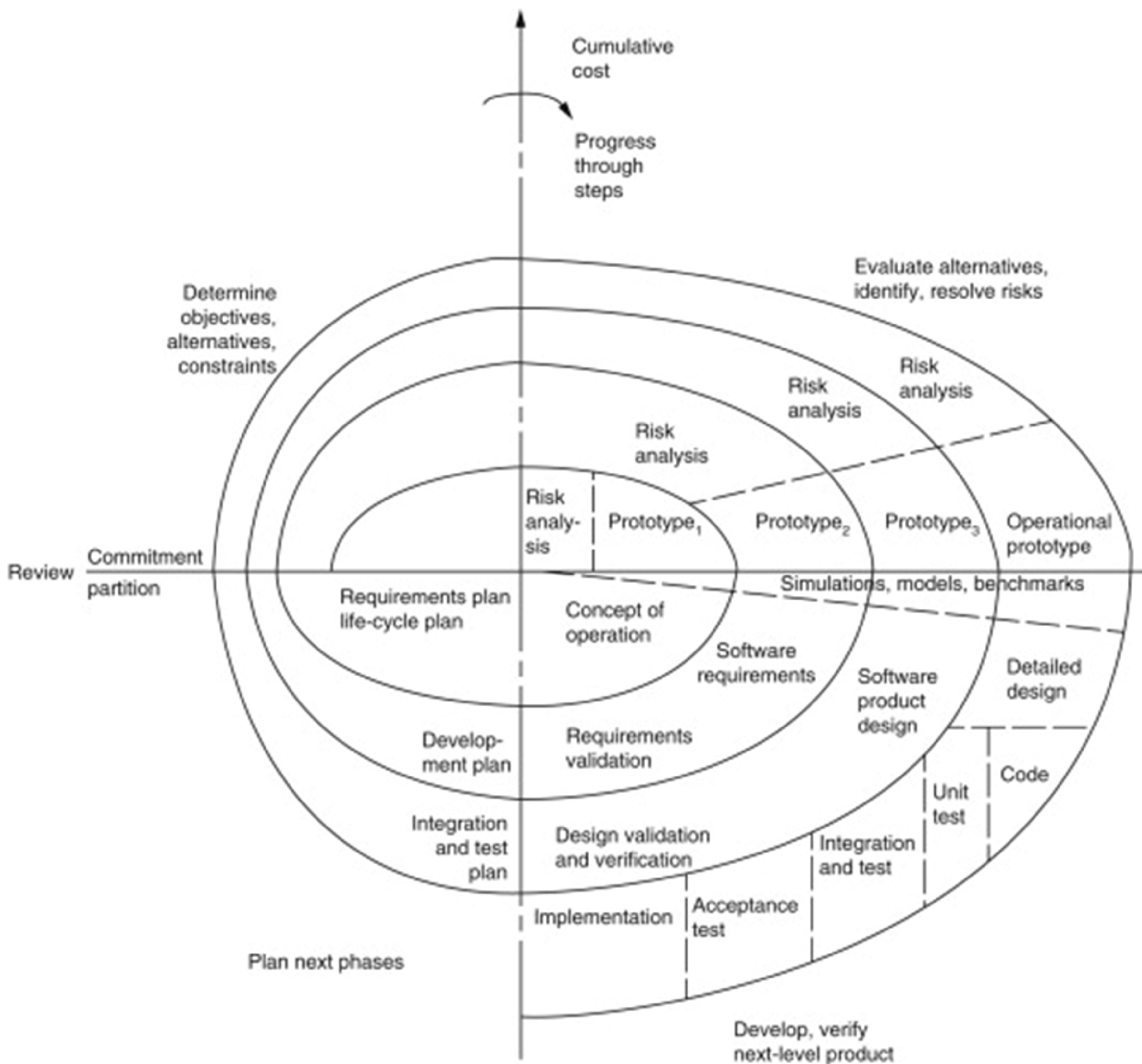

Figure 5: Spiral model of software testing [53].

3 Optimization Techniques

Software testing is a crucial aspect of software engineering, and it seeks to confirm that software modules function as intended and lower the likelihood of future failures [59]. Several optimization techniques have been developed to increase the efficiency and effectiveness of software testing [60]. In this section, we will review some of the most commonly used optimization techniques: test case reduction techniques, fault localization techniques, mutation testing, and combinatorial testing. We will also assess their advantages, disadvantages, and ability to improve software testing efficiency and effectiveness.

3.1 Test case reduction techniques

Test case reduction techniques minimize the number of test cases required to test software modules while maintaining adequate test coverage. These techniques reduce the testing effort and time required while ensuring the testing process effectively detects errors and defects [61, 62]. The first test case reduction technique is random testing, which involves randomly selecting a subset of test cases from the entire set of test cases based on a probability distribution function. This approach is useful in identifying errors or defects in the software module but cannot guarantee that all possible scenarios are tested, and it may be ineffective in detecting complex errors [63–66]. The second technique is prioritization, which involves ranking test cases based on their level of importance and executing them in order of priority. This approach ensures that high-priority test cases are executed first, which can help identify critical errors in the software module. However, lower-priority test cases may not be executed, which may miss identifying less critical errors [67–71]. The third technique is clustering, which involves grouping test cases with similar features and executing one representative test case from each group. This method can reduce the required test cases and may be more efficient than executing all test cases. However, this approach may miss identifying some errors that may only appear in specific test cases [66, 72–76]. While these test case reduction techniques can enhance software testing models by reducing the required test cases, they may not effectively identify all faults in the software module. Therefore, selecting an appropriate technique should be based on the specific characteristics of the software module under test.

3.2 Fault localization techniques

Fault localization is a crucial aspect of software testing that involves identifying the specific locations of faults or defects in the software module. Fault localization techniques are designed to reduce the time and effort required to identify and fix faults in the software, thereby improving the overall efficiency of the software testing process [77–79]. One of the commonly used fault localization techniques is spectrum-based or static techniques. This technique uses information such as statement or branch coverage to identify faulty statements in the software module. Spectrum-based techniques evaluate the coverage data collected during testing and calculate the suspiciousness score of each statement or code segment. The suspiciousness score indicates the likelihood of a faulty statement or code segment. The technique then prioritizes the statements or code segments based on their suspiciousness scores, and developers can investigate the highest-ranked statements or code segments to identify the fault [80–83]. However, while fault localization techniques have proven useful in enhancing software testing models, they may not be effective in identifying complex faults. Complex faults can occur when multiple defects or errors interact, making it challenging to identify the root cause of the fault. In such cases, fault localization techniques may not be sufficient to pinpoint the exact location of the fault.

3.3 Mutation test technique

Mutation testing is a technique used to evaluate the quality and adequacy of test suites. It involves creating artificial defects or mutations in the program under test by making small syntactic modifications, creating a modified version of the program called a mutant. The purpose of creating these mutants is to simulate common mistakes that programmers make while coding a program [84]. The goal of mutation testing is to ensure that the test cases effectively detect and identify defects in the program. By generating mutants and executing them with the same test suite, the adequacy of the test suite can be evaluated. Mutants that behave differently from the original program are considered dead, and equivalent mutants are identified when they exhibit the same behavior as the original program for all test cases. New test cases are then created to kill alive mutants, meaning mutants still exhibit different behavior than the original program [85]. Mutation testing has been applied at various levels of software development, including unit testing, integration testing, and specification testing. It has been used as a white-box unit testing technique for many programming languages, as well as for integration testing. Additionally, mutation testing has been applied at the design level to test program specifications or models [86]. Despite the benefits of mutation testing in terms of effectiveness in evaluating test cases, it also has some drawbacks. One of the major issues is the high number of mutants generated, which can be computationally expensive to execute. Additionally, identifying equivalent mutants can be a time-consuming and challenging task. However, despite these challenges, mutation testing has reached a maturity phase and is gradually gaining popularity in both academia and industry as a powerful tool for evaluating the quality of test suites [84–88].

3.4 Combinatorial test technique

Combinatorial testing (CT) is a dynamic testing technique that aims to detect interaction faults in software systems caused by the combined effect of multiple configurable parameters. It involves specifying test factors and their respective test settings based on requirements, system implementation, and other available information. Then, test cases are generated as combinations of one test set for each test factor. This technique detects failures triggered by parameter interactions in complex software systems with distributed environments [89]. CT is widely used to handle large test combination spaces in practical testing scenarios. However, the performance of existing constraint-handling methods can rapidly degrade when constraints are involved between parameters. Thus, more intelligent test data sampling mechanisms are needed to detect interaction faults in complex software systems [90]. The main goal of CT test generation is to generate covering arrays that cover all t-way parameter combinations. A covering array is a compact test suite that ensures that every t-tuple of parameter values occurs at least once in the test suite. The covering strength, t, determines the level of interaction between the parameters in the software system. CT is highly effective in detecting failures caused by interactions between parameters that are otherwise hard to detect using traditional testing methods [91].

In summary, combinatorial testing is a powerful dynamic technique that can detect complex interaction faults in software systems caused by multiple configurable parameters. It offers an effective testing mechanism to handle large test combination spaces and has been widely used for 20 years. However, more intelligent test data sampling mechanisms are needed to detect interaction faults in practical testing scenarios, especially when constraints exist between parameters [89–91].

4 Analysis of Efficiency and Advancement Needs

4.1 Comparison of software testing models and optimization techniques

This section will compare the software testing models and optimization techniques reviewed in the previous sections. The comparison of the four commonly used optimization techniques: test case reduction, fault localization, mutation testing, and combinatorial testing, is discussed based on their purpose, advantages, limitations, and applicability. This will help readers understand the differences between these techniques and choose the appropriate one for their specific testing needs. The information is summarized in Table 1. The comparison of the five commonly used software testing models: Waterfall, Incremental, V-Model, Agile, and Spiral, is discussed based on ten different important aspects of each technique to help readers understand their differences and choose the appropriate technique for their specific testing needs. The information is summarized in Table 2.

4.2 Identification of areas for improvement and advancement

Software testing is a crucial component of software engineering. There are various software testing models and optimization techniques available in the literature. Each model and technique have advantages and disadvantages, and their ability to identify flaws and reduce testing time and effort varies. Therefore, it is essential to assess and compare these models and techniques to identify areas for improvement and advancement. Based on the comparison of different testing models and optimization techniques, it is clear that there is no one-size-fits-all approach to software testing. The choice of model and optimization technique depends on the project requirements and context. However, several areas for improvement and advancement can increase the effectiveness and efficiency of software testing. One area for improvement is the integration of different software testing models and techniques. For example, it is possible to use the V-model for verification and validation and the Agile model for collaboration and feedback. Similarly, combining different optimization techniques to achieve better results is possible.

Table 1: Comparative analysis of optimizing techniques.

Aspects	**Test Case Reduction**	**Fault Localization**	**Mutation Testing**	**Combinatorial Testing**
Purpose	Minimize test cases while maintaining coverage	Identify specific fault locations	Evaluate the quality and adequacy of test suites	Detect interaction faults caused by parameter combinations
Advantages	Reduce overall testing time and effort	Improve the efficiency of the software testing process	Effective in evaluating the quality and adequacy of test suites	Highly effective in detecting failures triggered by parameter interactions
Limitations	May not identify all faults in the software module	It may not be effective in identifying complex faults	Computationally expensive to execute, challenging task of identifying equivalent mutants	Performance can rapidly degrade when constraints are involved between parameters
Applicability	When reducing the number of test cases is necessary	When identifying specific fault locations is critical	When evaluating the quality and adequacy of test suites	In complex software systems with distributed environments to detect interaction faults

Another area for improvement is the automation of software testing. Automation can reduce the time and effort required for testing and improve the accuracy and consistency of the results. Automation can also enable continuous testing, which is essential for Agile development. Machine learning techniques can be used to improve software testing in several ways. For example, machine learning can generate test cases automatically based on the software code analysis and its behavior. Machine learning can also prioritize test cases based on their likelihood of detecting faults. In addition, machine learning can be used for fault prediction and localization. Traditional metrics, such as code coverage and defect density, are useful but limited in their ability to gauge the success of software testing. New metrics are needed to capture the quality and effectiveness of software testing. In addition, it is important to measure the impact of software testing on the overall project, such as its effect on customer satisfaction and business value. Contemporary software programs, such as those used in the Internet of Things (IoT) and artificial intelligence (AI), pose unique challenges for software testing.

These programs are often distributed, heterogeneous, and complex, which makes testing more difficult. Thus, novel approaches are needed that consider the specific characteristics of these programs. One approach is to use model-based testing techniques that leverage formal models of the system under test to generate test cases automatically. This approach has shown promising results in testing IoT systems where many devices are connected, and the system's behavior is non-deterministic. Another approach is to use AI and machine learning techniques to improve the effectiveness and efficiency of software testing. For example, machine learning algorithms can be trained to automatically generate and prioritize test cases based on their likelihood of uncovering defects. This approach has shown promising results in reducing the number of test cases needed to achieve high test coverage.

Table 2: Comparison of software testing models.

Aspect	**Waterfall Model**	**Incremental Model**	**V-Model**	**Agile Model**	**Spiral Model**
Testing Approach	A sequential and linear approach	Iterative approach	A sequential and linear approach	An iterative and incremental approach	Iterative approach
Testing Phases	Testing is done after development is complete	Testing is done after each iteration or module is complete	Testing is done in parallel with development phases	Testing is done continuously throughout the project	Testing is done after each iteration
Documentation	Emphasis on documentation	Documentation is important but not emphasized as much as in the Waterfall model	Emphasis on documentation	Documentation is not the main focus, but agile methodologies have their own set of documentation practices	Emphasis on documentation
Communication	Communication is limited to predefined stages	Communication is ongoing and frequent	Communication is limited to predefined stages	Communication is ongoing and frequent	Communication is ongoing and frequent
Flexibility	Changes are difficult to implement	Changes are easier to implement	Changes are difficult to implement	Changes are easy to implement	Changes are easier to implement
Team Structure	Hierarchical structure with separate teams for each phase	Cross-functional teams with shared responsibilities	Hierarchical structure with separate teams for each phase	Self-organizing teams with shared responsibilities	Hierarchical structure with separate teams for each phase
Customer Involvement	Limited customer involvement	Moderate customer involvement	High customer involvement	Moderate customer involvement	Moderate customer involvement
Risk Management	Risk analysis is done before development starts	Risk analysis is done before each iteration	Risk analysis is done before development starts	Risk analysis is done continuously throughout the project	Risk analysis is done before each iteration
Testing Speed	Slower testing due to the sequential approach	Faster testing due to the iterative approach	Slower testing due to the sequential approach	Faster testing due to the continuous testing approach	Faster testing due to the iterative approach
Adaptability	Less adaptable to changes	Highly adaptable to changes	Less adaptable to changes	Highly adaptable to changes	Highly adaptable to changes

5 Conclusion

In conclusion, software testing is a critical aspect of software engineering that ensures software modules function as intended and minimizes the likelihood of future failures. The presented review paper comprehensively reviews various software testing models and optimization techniques, highlighting their advantages and disadvantages. The analysis showed no one-size-fits-all approach to software testing, and the choice of model and technique depends on the project's requirements, resources, and constraints. Additionally, the paper suggested areas for improvement and advancement to increase the effectiveness and efficiency of software testing, such as the use of machine learning techniques to automate and optimize testing processes, the introduction of new metrics to gauge the success of testing, and the development of novel approaches to deal with contemporary software programs. Ultimately, the software testing process should be continuously improved and optimized to ensure software modules function as intended and minimize the likelihood of future failures.

Declaration of Competing Interests

The author declares that he has no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding Declaration

This research did not receive any grants from governmental, private, or nonprofit funding bodies.

Author Contribution

Sarvesh Kumar: Conceptualization, Methodology, Data curation, Writing - Original draft preparation, Writing – Reviewing

References

[1] M. Jamil, M. Arif, N. Abubakar, and A. Ahmad, Software testing techniques: a literature review, in 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Nov. 2017, pp. 177–182.

[2] F. Lonetti and E. Marchetti, Emerging software testing technologies, in Advances in Computers, 108, 2018, pp. 91–143.

[3] M. Mayeda and A. Andrews, Evaluating software testing techniques: A systematic mapping study, in Advances in Computers, 123, 2021, pp. 41–114.

[4] M. Wolf, Program design and analysis, in Computers as Components, Elsevier, 2023, pp. 219–319.

[5] C. Moseley, 7 Reasons why collaboration is important, Jostle, 2021.

[6] M. Poženel and B. Slivnik, Using clickstream data to enhance reverse engineering of Web applications, in Advances in Computers, 116, (1), 2020, pp. 305–349.

[7] E. Conrad, S. Misenar, and J. Feldman, Security assessment and testing, in Eleventh Hour CISSP®, Elsevier, 2017, pp. 135–144.

[8] D. Hartley, Reviewing code for SQL injection, in SQL Injection Attacks and Defense: Second Edition, Elsevier, 2012, pp. 89–138.

[9] A. Bertolino, Software testing research: achievements, challenges, dreams, in FoSE 2007: Future of Software Engineering, May 2007, pp. 85–103.

[10] Sayantini, Software Testing Models, Edureka, 2019.

[11] C. Tupper, Data organization practices, in Data Architecture, Elsevier, 2011, pp. 175–190.

[12] E. Goodman, M. Kuniavsky, and A. Moed, Balancing needs through iterative development, in Observing the User Experience, Elsevier, 2012, pp. 21–44.

[13] R. Hartson and P. Pyla, Agile lifecycle processes and the funnel model of agile UX, in The UX Book, Elsevier, 2019, pp. 63–80.

[14] R. Sherman, Project management, in Business Intelligence Guidebook, 33, (3), Elsevier, 2015, pp. 449–492.

[15] G. Swara, I. Warman, and D. Putra, Implementation of the waterfall model on android-based travel ticket booking applications, "Journal of Information System, Informatics and Computing," 6, (1), pp. 235–245, 2022.

[16] A. Ardhiansyah, D. Putra, J. Kristanto, N. Budhianto, and F. Maulana, Waterfall Model for Design and Development Coffee Shop Website at Malang, in 2022 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), Nov. 2023, pp. 230–234.

[17] S. Herawati, Y. Negara, H. Febriansyah, and D. Fatah, Application of the waterfall method on a web-based job training management information system at Trunojoyo university Madura, "E3S Web of Conferences," 328, p. 04026, 2021.

[18] T. Rahayu, Susanto, and Suwarjono, Application Report Process of Islamic School Based on Pesantren boarding using waterfall model, "Journal of Physics: Conference Series," 1569, (2), p. 022025, 2020.

[19] Y. Dwi Putra Negara, D. Rizal Setiawan, E. Rochman, and F. Ayu Mufarroha, Development of a boarding house search information system using the waterfall model, "E3S Web of Conferences," 328, p. 04030, 2021.

[20] F. Badri, R. Maulana, K. Khotimah, R. Budiarti, and A. Andhyka, Design and build a web app-based conference registration system using the waterfall model, “Applied Technology and Computing Science Journal,” 4, (2), pp. 119–127, 2022.

[21] R. Purba and S. Sondang, Design and build monitoring system for pregnant mothers and newborns using the waterfall model, “INTENSIF: Jurnal Ilmiah Penelitian dan Penerapan Teknologi Sistem Informasi,” 6, (1), pp. 29–42, 2022.

[22] P. Ganney, S. Pisharody, and E. Claridge, Software engineering, in Clinical Engineering, 11, (2), Elsevier, 2020, pp. 131–168.

[23] K. Genter, N. Agmon, and P. Stone, Role-based ad hoc teamwork, in Plan, Activity, and Intent Recognition: Theory and Practice, Elsevier, 2014, pp. 251–272.

[24] N. Bhatt and S. Visvanathan, Incremental kinetic identification based on experimental data from steady-state plug Flow Reactors, in Computer Aided Chemical Engineering, 37, 2015, pp. 593–598.

[25] D. Rodrigues, J. Billeter, and D. Bonvin, Global identification of kinetic parameters via the extent-based incremental approach, in Computer Aided Chemical Engineering, 40, 2017, pp. 2119–2124.

[26] S. Hanks and D. Madigan, Probabilistic temporal reasoning, in Foundations of Artificial Intelligence, 1, (C), 2005, pp. 315–342.

[27] B. Shahzad, I. Ullah, and N. Khan, Software risk identification and mitigation in incremental model, in 2009 International Conference on Information and Multimedia Technology, ICIMT 2009, 2009, pp. 366–370.

[28] L. Qiu and C. Riesbeck, An incremental model for developing educational critiquing systems: Experiences with the Java Critiquer, ”Journal of Interactive Learning Research,” 19, (1), pp. 119–145, 2008.

[29] A. Rachman, Andreansyah, and Rahmi, Implementation of incremental models on development of web-based loan cooperative applications, ”International Journal of Education, Science, Technology, and Engineering,” 3, (1), pp. 26–34, 2020.

[30] S. Lity, T. Morbach, T. Thüm, and I. Schaefer, Applying incremental model slicing to product-line regression testing, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9679, 2016, pp. 3–19.

[31] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity, Incremental model-based testing of delta-oriented software product lines, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7305 LNCS, 2012, pp. 67–82.

[32] T. Weilkiens, Introduction, in Systems Engineering with SysML/UML, Elsevier, 2007, pp. 1–22.

[33] M. Awotar and R. K. Sungkur, Optimization of software testing, ”Procedia Computer Science,” 132, pp. 1804–1814, 2018.

[34] S. Shylesh, A study of software development life cycle process models, ”SSRN Electronic Journal,” pp. 1–7, 2017.

[35] D. Firesmith, No using v models for testing title, SEI Blog, 2013.

[36] G. Regulwar, P. Deshmukh, R. Tugnayat, P. Jawandhiya, and V. Gulhane, Variations in V model for software development, ”International Journal of Advanced Research in Computer Science,” 1, (2), pp. 135–140, 2010.

[37] M. Durmuş, İ. Üstoğlu, R. Tsarev, and J. Börcsök, Enhanced V-model, ”Informatica (Slovenia),” 42, (4), pp. 577–585, 2018.

[38] B. Liu, H. Zhang, and S. Zhu, An incremental V-model process for automotive development, in Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 2016, 0, pp. 225–232.

[39] C. Lim and J. Chin, V-model with fuzzy quality function deployments for mobile application development, ”Journal of Software: Evolution and Process,” 35, (1), 2023.

[40] T. Hynninen, A. Knutas, and J. Kasurinen, Designing early testing course curricula with activities matching the V-model phases, in 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019 - Proceedings, May 2019, pp. 1593–1598.

[41] A. Khan, M. Akram, A. Salam, and W. Butt, An enhanced agile-V model for system engineering in complex medical device development, in 2022 2nd International Conference on Digital Futures and Transformative Technologies, ICoDT 2022, May 2022, pp. 1–6.

[42] W. Febriyani, F. Kistianti, and M. Lubis, Validation and verification of business architecture process based on the V. model, in 2022 Seventh International Conference on Informatics and Computing (ICIC), Dec. 2023, pp. 01–06.

[43] M. Ali Babar, Making software architecture and agile approaches work together: foundations and approaches, in Agile Software Architecture: Aligning Agile Processes and Software Architectures, Elsevier, 2013, pp. 1–22.

[44] P. Rodríguez, M. Mäntylä, M. Oivo, L. Lwakatare, P. Seppänen, and P. Kuvaja, Advances in using agile and lean processes for software development, in Advances in Computers, 113, 2019, pp. 135–224.

[45] A. Agrawal, M. Atiq, and L. Maurya, A Current study on the limitations of agile methods in industry using secure google forms, ”Physics Procedia,” 78, pp. 291–297, 2016.

[46] A. Hoffman, Agile software development life cycle explained - vintank, JavaTpoint, 2020.

[47] J. Kahles, J. Torronen, T. Huuhtanen, and A. Jung, Automating root cause analysis via machine learning in agile software testing environments, in Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019, Apr. 2019, pp. 379–390.

[48] S. Dhir and D. Kumar, Automation software testing on web-based application, ”Advances in Intelligent Systems and Computing,” 731, pp. 691–698, 2019.

[49] I. Elgrably and S. Oliveira, Construction of a syllabus adhering to the teaching of software testing using agile practices, ”Proceedings - Frontiers in Education Conference, FIE,” 2020-Octob, 2020.

[50] S. Gochhait, S. A. Butt, T. Jamal, and A. Ali, Cloud enhances agile software development, ”Research Anthology on Agile Software, Software Development, and Testing,” 1, pp. 491–507, 2021.

[51] S. Kaur, S. Hooda, and H. Deo, Software quality management by agile testing, “Agile Software Development,” pp. 221–233, 2023.

[52] E. Conrad, S. Misenar, and J. Feldman, Software development security, in Eleventh Hour CISSP, Elsevier, 2014, pp. 63–76.

[53] E. Conrad, S. Misenar, and J. Feldman, Software development security (understanding, applying, and enforcing software security), in CISSP Study Guide, Elsevier, 2016, pp. 429–477.

[54] M. Jazayeri, Software engineering, in Encyclopedia of Physical Science and Technology, Elsevier, 2003, pp. 1–14.

[55] R. Sharma and A. Saha, Fermat spiral-based moth-flame optimization algorithm for object-oriented testing, Springer, Singapore, 2020, pp. 19–34.

[56] G. Aimicheva, Z. Kopeyev, Z. Ordabayeva, N. Tokzhigitova, and S. Akimova, A spiral model teaching mobile application development in terms of the continuity principle in school and university education, “Education and Information Technologies,” 25, (3), pp. 1875–1889, 2020.

[57] Supiyandi, C. Rizal, B. Fachri, M. Eka, and Y. Nasution, Development of a village information system using the spiral method, “International Conference on Sciences Development and Technology,” 2, (1), pp. 112–117, 2022.

[58] M. Khadapi, Implementation of the spiral method for analyzing and designing financial information systems and financial archives for cashier financial management section (cash information replacement), “Journal of Artificial Intelligence and Engineering Applications,” 2, (2), pp. 53–58, 2023.

[59] J. Musa, Software reliability engineering, in Reliability and Maintenance of Complex Systems, S. Özekici, Ed. Springer Berlin Heidelberg, 2013, pp. 319–332.

[60] L. Lazic, Software testing optimization by advanced quantitative defect management, “Computer Science and Information Systems,” 7, (3), pp. 459–487, 2010.

[61] F. Elberzhager, A. Rosbach, J. Münch, and R. Eschbach, Reducing test effort: A systematic mapping study on existing approaches, “Information and Software Technology,” 54, (10), pp. 1092–1106, 2012.

[62] R. Singh and M. Santosh, Test case minimization techniques: A Review, “International Journal of Engineering Research & Technology,” 2, (12), pp. 1048–1056, 2013.

[63] E. Narciso, M. Delamaro, and F. Nunes, Test case selection: a systematic literature review, “International Journal of Software Engineering and Knowledge Engineering,” 24, (04), pp. 653–676, 2014.

[64] R. Huang, H. Chen, W. Sun, and D. Towey, Candidate test set reduction for adaptive random testing: An overheads reduction technique, “Science of Computer Programming,” 214, p. 102730, 2022.

[65] T. Chen, F. Kuo, R. Merkel, and T. Tse, Adaptive random testing: The ART of test case diversity, “Journal of Systems and Software,” 83, (1), pp. 60–66, 2010.

[66] H. Pei, B. Yin, M. Xie, and K. Cai, Dynamic random testing with test case clustering and distance-based parameter adjustment, “Information and Software Technology,” 131, p. 106470, 2021.

[67] S. Yoo and M. Harman, Regression testing minimization, selection and prioritization: a survey, “Software Testing, Verification and Reliability,” p. n/a-n/a, 2010.

[68] G. Barbosa, É. de Souza, L. dos Santos, M. da Silva, J. Balera, and N. Vijaykumar, a systematic literature review on prioritizing software test cases using Markov chains, “Information and Software Technology,” 147, p. 106902, 2022.

[69] R. Mukherjee and K. Patnaik, A survey on different approaches for software test case prioritization, “Journal of King Saud University - Computer and Information Sciences,” 33, (9), pp. 1041–1054, 2021.

[70] M. Khatibsyarbini, M. Isa, D. Jawawi, H. Hamed, and M. Suffian, Test case prioritization using firefly algorithm for software testing, “IEEE Access,” 7, pp. 132360–132373, 2019.

[71] S. Mirarab and L. Tahvildari, A prioritization approach for software test cases based on bayesian networks, in Fundamental Approaches to Software Engineering, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 276–290.

[72] G. Kumar and P. K. Bhatia, Software testing optimization through test suite reduction using fuzzy clustering, “CSI Transactions on ICT,” 1, (3), pp. 253–260, 2013.

[73] A. Upadhyay and A. Misra, Prioritizing Test Suites Using Clustering pproach in Software Testing, “Ijsce.Org,” (4), pp. 222–226, 2012.

[74] S. Yoo, M. Harman, P. Tonella, and A. Susi, Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge, in Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA 2009, Jul. 2009, pp. 201–211.

[75] S. Shekhar, Y. Li, R. Y. Ali, E. Eftelioglu, X. Tang, and Z. Jiang, Spatial and spatiotemporal data mining, in Comprehensive Geographic Information Systems, Elsevier, 2018, pp. 264–286.

[76] I. Witten, E. Frank, M. Hall, and C. Pal, Probabilistic methods, in Data Mining, Elsevier, 2017, pp. 335–416.

[77] D. Slane, Fault localization in in vivo software testing by, Bard College at Simon’s Rock Great Barrington, Massachusetts, 2009.

[78] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, A survey on software fault localization, “IEEE Transactions on Software Engineering,” 42, (8), pp. 707–740, 2016.

[79] T. Ostrand, E. Weyuker, and R. Bell, Predicting the location and number of faults in large software systems, “IEEE Transactions on Software Engineering,” 31, (4), pp. 340–355, 2005.

[80] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. Van Hoorn, and D. Lo, A critical evaluation of spectrum-based fault localization techniques on a large-scale software system, “Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017,” pp. 114–125, 2017.

[81] H. He, J. Ren, G. Zhao, and H. He, Enhancing spectrum-based fault localization using fault influence propagation, “IEEE Access,” 8, pp. 18497–18513, 2020.

[82] R. Abreu, P. Zoeteweij, and A. van Gemund, Spectrum-based multiple fault localization, in 2009 IEEE/ACM International Conference on Automated Software Engineering, Nov. 2009, pp. 88–99.

[83] H. de Souza, D. Mutti, M. Chaim, and F. Kon, Contextualizing spectrum-based fault localization, “Information and Software Technology,” 94, pp. 245–261, 2018.

[84] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, Mutation testing advances: an analysis and survey, in Advances in Computers, 112, 2019, pp. 275–378.

[85] R. Silva, S. de Souza, and P. Lopes de Souza, A systematic review on search based mutation testing, “Information and Software Technology,” 81, pp. 19–35, 2017.

[86] Y. Jia and M. Harman, An analysis and survey of the development of mutation testing, “IEEE Transactions on Software Engineering,” 37, (5), pp. 649–678, 2011.

[87] N. Shomali and B. Arasteh, Mutation reduction in software mutation testing using firefly optimization algorithm, “Data Technologies and Applications,” 54, (4), pp. 461–480, 2020.

[88] R. Just, The major mutation framework: Efficient and scalable mutation analysis for Java, “2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings,” pp. 433–436, 2014.

[89] R. Kacker, D. Kuhn, Y. Lei, and J. Lawrence, Combinatorial testing for software: An adaptation of design of experiments, “Measurement: Journal of the International Measurement Confederation,” 46, (9), pp. 3745–3752, 2013.

[90] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, Generating combinatorial test suite using combinatorial optimization, “Journal of Systems and Software,” 98, pp. 191–207, 2014.

[91] C. Nie and H. Leung, A survey of combinatorial testing, “ACM Computing Surveys,” 43, (2), 2011.