

A Framework for Defect Root-Cause Analysis in Cross-Functional Enterprise QA Teams

Samanuru Geetha Chakravarthy^[0009-0000-6614-0688]

Independent Researcher, USA

csamanuru@gmail.com

Published: Feb 23, 2023

[Vol. 5 No.5 \(2023\): AJAI](#)

Abstract

Defect root-cause analysis is an important software quality assurance activity used to identify the underlying reasons behind defects rather than focusing only on their immediate symptoms. In large enterprise software environments, defects may originate from multiple areas, including unclear requirements, design decisions, coding errors, integration problems, test-data issues, environment configuration, or gaps in testing processes. The involvement of cross-functional teams makes root-cause analysis more comprehensive but can also introduce challenges related to communication, ownership, technical dependencies, and differences in team perspectives. This study proposes a structured framework for performing defect root-cause analysis within cross-functional enterprise Quality Assurance (QA) teams. The framework focuses on systematic defect classification, evidence collection, impact assessment, identification of contributing factors, root-cause determination, corrective actions, and preventive measures. It also emphasizes collaboration among QA engineers, developers, business analysts, architects, operations teams, and other relevant stakeholders. The proposed approach aims to move organizations from reactive defect fixing toward proactive quality improvement by identifying recurring patterns and addressing weaknesses in development and testing processes. The study highlights the importance of standardized analysis techniques, clear responsibilities, reliable defect data, effective communication, and continuous monitoring. A structured root-cause analysis framework can help enterprise QA teams reduce recurring defects, improve development and testing processes, and strengthen overall software quality.

Keywords: Defect Root-Cause Analysis, Quality Assurance, Cross-Functional Teams, Enterprise Software, Defect Management, Software Testing

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

1. Introduction

1.1 Background of Defect Management in Enterprise Software

Defect management is an important part of software quality assurance that involves identifying, recording, analyzing, prioritizing, resolving, and verifying defects throughout the software development lifecycle. In enterprise software systems, defect management becomes particularly important because applications are often large, interconnected, and used by many different users or business functions. A defect in one component can sometimes affect several other components through shared services, databases, APIs, or business processes.

A software defect may occur at different stages of development and can have various sources. It may result from unclear or incomplete requirements, incorrect system design, programming errors, configuration problems, integration failures, incorrect test data, or gaps in testing. Effective defect management therefore requires more than simply reporting a failure. Teams need to understand the nature, severity, priority, and possible impact of each defect so that appropriate corrective action can be taken.

Enterprise QA teams often work with large volumes of defects across multiple applications and development teams. Without a structured defect-management process, similar defects may repeatedly occur, ownership may become unclear, and teams may focus primarily on fixing individual problems rather than improving the underlying process. For this reason, defect management increasingly includes activities such as trend analysis, defect classification, root-cause analysis, and preventive action.

A mature defect-management process also provides useful information about the overall quality of a software project. Defect patterns can reveal weaknesses in requirements, design, development, testing, or deployment practices. By analyzing this information systematically, organizations can identify areas where processes can be improved and reduce the likelihood of similar defects occurring in future releases.

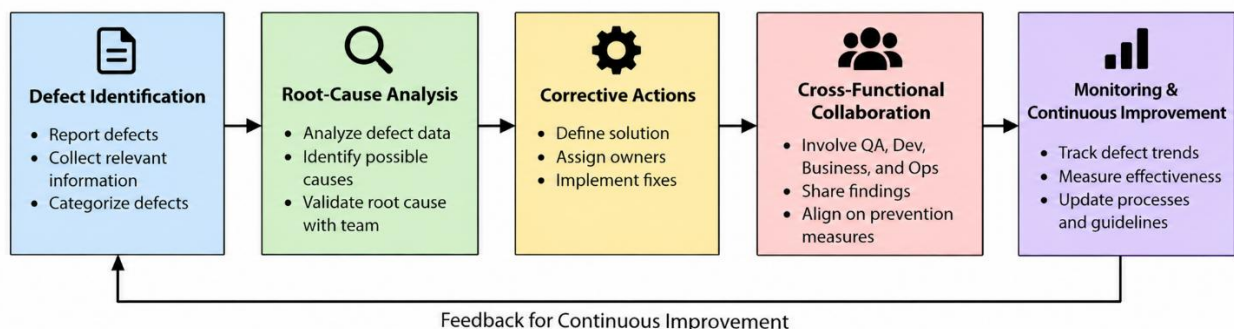


Figure 1. Framework for Defect Root-Cause Analysis in Cross-Functional Enterprise QA Teams

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

1.2 Importance of Defect Root-Cause Analysis

Defect root-cause analysis is the systematic process of identifying the underlying reason or combination of reasons that allowed a defect to occur. Traditional defect management often concentrates on reproducing the problem, fixing the affected code, and verifying the correction. While these activities are necessary, they may not prevent the same type of defect from appearing again. Root-cause analysis extends the investigation by asking why the defect occurred and what conditions allowed it to pass through earlier stages of the development and testing process.

For example, a defect discovered during system testing may appear to be a coding problem, but further investigation may show that the actual cause was an unclear requirement or an incorrect interpretation of a business rule. Similarly, a production issue may initially appear to be a software failure but may have been caused by incorrect configuration, incomplete test data, inadequate regression coverage, or a deployment process problem. Identifying the underlying cause allows organizations to address the source rather than only correcting the visible symptom.

Root-cause analysis is particularly valuable in enterprise environments because defects can involve several teams and interconnected systems. QA engineers, developers, business analysts, architects, database teams, operations teams, and other stakeholders may each have information relevant to the investigation. Collaboration between these groups can provide a more complete understanding of the defect and reduce the possibility of assigning an incorrect or incomplete root cause.

The results of root-cause analysis can also support defect prevention and continuous improvement. When recurring causes are identified, organizations can introduce improvements such as better requirements reviews, coding standards, additional test scenarios, improved automation, stronger integration testing, enhanced training, or changes to development and deployment processes.

1.3 Problem Statement

Enterprise software systems are becoming increasingly complex due to the use of distributed architectures, APIs, cloud services, databases, third-party integrations, and multiple application components. As system complexity increases, defects may have multiple contributing factors rather than a single easily identifiable cause. This makes defect investigation more difficult, particularly when different teams own different parts of the system.

A common challenge is that defect analysis may focus primarily on immediate correction instead of understanding the underlying cause. Teams may fix the reported problem without investigating why it was introduced, why it was not detected earlier, or what process weakness

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

allowed it to reach a later testing stage or production environment. As a result, similar defects can recur in subsequent releases.

Cross-functional teams also face challenges related to communication and ownership. Different teams may have different interpretations of the same defect and may focus on their own area of responsibility. In the absence of a common analysis framework, investigations can become inconsistent, time-consuming, or focused on assigning responsibility rather than understanding the actual cause.

Therefore, the central problem addressed by this study is the need for a structured and collaborative framework for defect root-cause analysis in cross-functional enterprise QA environments. The framework should help teams collect relevant evidence, classify defects, identify root and contributing causes, determine corrective and preventive actions, and use the resulting information for continuous quality improvement.

1.4 Objectives of the Study

The primary objective of this study is to develop a structured framework for performing defect root-cause analysis within cross-functional enterprise QA teams. The study aims to provide a systematic approach that can be followed when investigating defects, regardless of whether their source is related to requirements, design, development, testing, data, configuration, integration, or deployment.

Another objective is to identify and categorize common causes and contributing factors associated with enterprise software defects. The study considers technical causes as well as process and organizational factors that may influence defect introduction and detection. This classification can help QA teams identify recurring patterns and areas requiring improvement.

The study also aims to examine the role of cross-functional collaboration in defect investigation. By involving relevant stakeholders, the analysis can combine different technical and business perspectives and provide a more complete understanding of the defect. The framework further aims to establish a connection between root-cause findings and corrective or preventive actions.

Finally, the study seeks to demonstrate how defect-analysis information can support continuous improvement. By examining defect trends, recurrence, severity, detection stage, and root-cause categories, organizations can identify weaknesses in their development and testing processes and take measures to reduce the occurrence of similar defects in future releases.

1.5 Scope of the Study

The scope of this study covers defect root-cause analysis within enterprise software quality assurance environments. It focuses on defects identified during different stages of the software lifecycle, including development testing, integration testing, system testing, user acceptance

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

testing, and production. The study considers defects related to functional behavior, requirements, design, coding, data, configuration, integration, testing, and deployment.

The study also covers the participation of cross-functional stakeholders such as QA engineers, developers, business analysts, architects, database specialists, operations teams, and other relevant technical or business representatives. It examines how information from these groups can be combined during defect investigation and how responsibilities can be defined within the analysis process.

The framework focuses on defect identification, classification, evidence collection, root-cause determination, contributing-factor analysis, corrective actions, preventive actions, and continuous monitoring. It does not attempt to replace an organization's existing defect-tracking or project-management tools. Instead, it provides a structured approach that can be incorporated into existing quality assurance and defect-management processes.

The study is intended to provide general principles applicable to complex enterprise software environments. The specific techniques and processes may be adapted according to the organization's application architecture, development methodology, team structure, regulatory requirements, and quality objectives.

2. Defect Root-Cause Analysis Framework

2.1 Fundamentals of Root-Cause Analysis

Root-cause analysis (RCA) is a systematic approach used to identify the underlying reasons behind a problem rather than focusing only on its visible symptoms. In software quality assurance, RCA is used to understand why a defect was introduced, why it was not detected earlier, and what conditions allowed it to reach a later stage of testing or production. The main purpose is not simply to identify who made the mistake, but to understand the technical, process, and environmental factors that contributed to the problem.

A defect can have more than one contributing factor. For example, a software failure may be caused by an incorrect implementation, but the investigation may reveal that the implementation resulted from an unclear requirement and was not detected because the related test scenario was missing. Therefore, root-cause analysis should consider the complete sequence of events that led to the defect.

An effective RCA generally involves collecting evidence, reproducing the problem, understanding the affected workflow, identifying possible causes, evaluating those causes, determining the root cause or causes, and defining appropriate corrective and preventive actions. Evidence can include requirements, design documents, source-code changes, test cases, test data, application logs, configuration information, defect history, and deployment records.

Journal of AI & Innovation (AJAI)

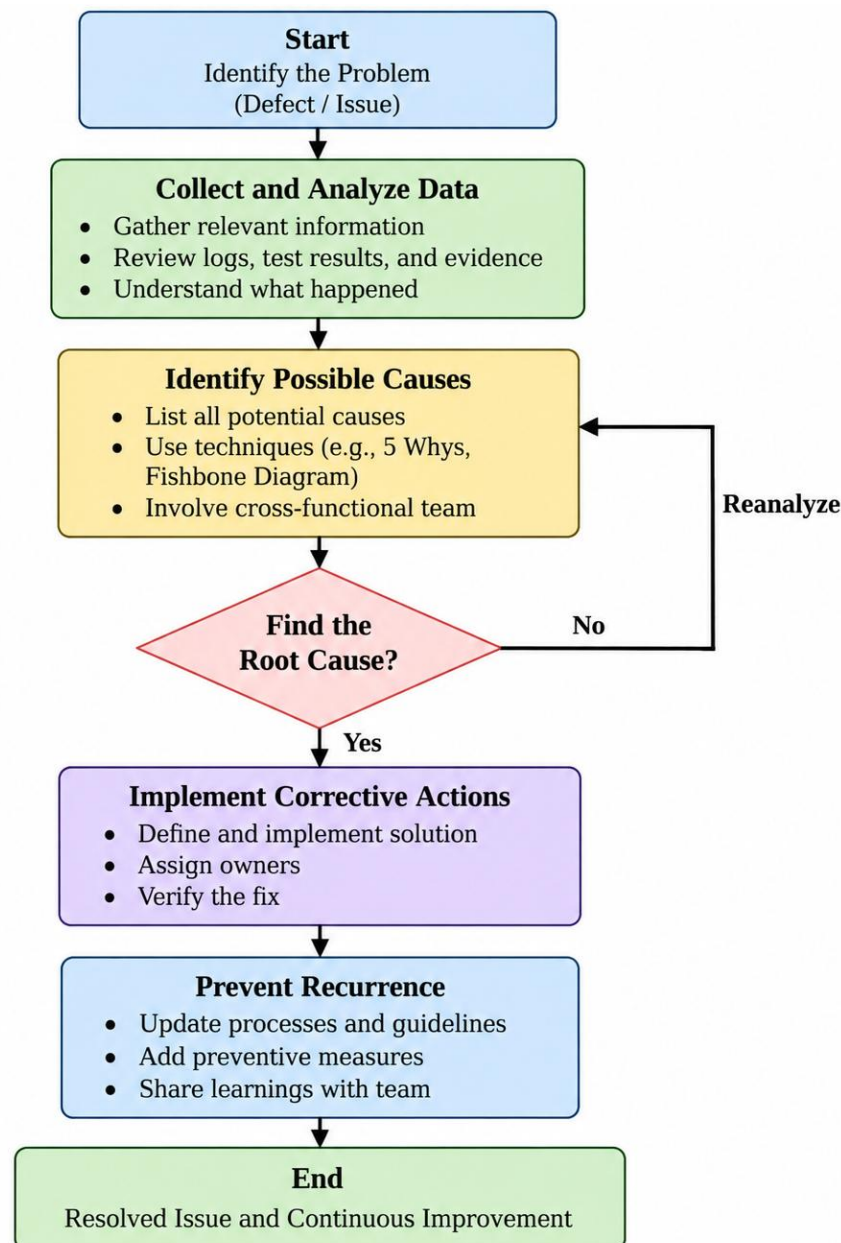
Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

RCA should also be conducted in a structured and objective manner. The purpose is to improve the software and development process rather than assign blame to an individual or team. A constructive approach encourages team members to share information openly and helps organizations identify process weaknesses that can be addressed systematically.



2.2 Defect Classification and Categorization

Defect classification is the process of organizing defects into meaningful categories based on their characteristics, source, severity, or stage of detection. Proper classification helps QA teams identify patterns and understand where defects are being introduced within the software

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

development lifecycle. Without consistent classification, historical defect information becomes difficult to analyze and may provide limited value for future quality improvement.

Defects can be categorized according to their technical or process-related causes. Common categories may include requirements defects, design defects, coding defects, integration defects, data defects, configuration defects, environment-related defects, test-case defects, and deployment-related defects. A single defect may also have contributing factors from more than one category, so the classification process should allow teams to record both the primary cause and additional contributing factors where appropriate.

Defect severity and priority are also important classification attributes. Severity describes the impact of a defect on the system or users, while priority indicates how urgently the defect should be addressed. For example, a defect that prevents an important business process from functioning may have greater severity than a minor user-interface issue. These classifications help teams focus their analysis and corrective efforts appropriately.

Consistent categorization also supports historical analysis. If a large number of defects are repeatedly classified as requirement-related, an organization may need to improve its requirements review process. Similarly, frequent integration defects may indicate a need for stronger interface testing or better communication between teams. Therefore, classification is not only useful for managing individual defects but also for identifying broader improvement opportunities.

2.3 Root-Cause Identification Techniques

Several techniques can be used to identify the root cause of software defects. One simple and commonly used approach is the **Five Whys technique**, in which the investigation repeatedly asks why a problem occurred until the underlying cause is identified. The number of questions does not have to be exactly five; the purpose is to move beyond the immediate symptom and understand the chain of contributing events.

Another useful technique is the **Fishbone or Ishikawa diagram**, which organizes possible causes into major categories. In a software environment, these categories may include people, processes, technology, requirements, data, tools, environment, and communication. The technique helps cross-functional teams consider different perspectives instead of immediately assuming that the defect was caused by programming.

Fault Tree Analysis (FTA) can also be useful for complex failures. It starts with an undesirable event and works backward to identify combinations of conditions that could have caused it. This approach can be particularly useful when multiple technical components or dependencies are involved.

Pareto analysis can be applied to historical defect data to identify the categories or causes responsible for a large proportion of defects. For example, if a small number of recurring causes

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

account for a significant portion of defects, addressing those causes may provide greater improvement than treating every defect independently.

The selected technique should depend on the complexity and nature of the defect. Simple problems may require a basic cause-and-effect analysis, while complex enterprise failures may require evidence from multiple systems and participation from several technical teams. Combining structured techniques with reliable evidence can improve the accuracy of root-cause identification.

2.4 Corrective and Preventive Actions

Corrective action refers to the steps taken to resolve an identified problem and restore the expected behavior of the software or process. For example, if a defect is caused by incorrect program logic, the corrective action may involve modifying the implementation and verifying the change through appropriate testing. Corrective action generally addresses the immediate problem identified during the investigation.

Preventive action focuses on reducing the possibility of similar problems occurring in the future. Instead of correcting only the individual defect, preventive action addresses the process or condition that allowed the defect to occur. Depending on the root cause, preventive measures may include improving requirements reviews, adding missing test scenarios, strengthening coding standards, introducing additional automated tests, improving test data, or modifying deployment procedures.

An effective RCA framework should connect each identified root cause with an appropriate corrective or preventive action. Actions should be specific, assigned to responsible stakeholders, and tracked until completion. Simply recording that an organization should "improve testing" may not be sufficient because it does not clearly identify what needs to change or how the improvement will be measured.

The effectiveness of corrective and preventive actions should also be reviewed after implementation. If similar defects continue to occur, the original root cause analysis may have been incomplete or the selected preventive action may not have addressed the actual cause. Continuous review therefore helps organizations determine whether RCA activities are producing meaningful improvements.

2.5 Role of Metrics in Defect Analysis

Defect metrics provide quantitative information that can support root-cause analysis and quality improvement. Metrics help organizations understand the number, type, severity, source, and detection stage of defects over time. When analyzed consistently, this information can reveal patterns that may not be visible when individual defects are considered separately.

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

Common metrics include defect density, defect severity distribution, defect leakage, defect recurrence, defect resolution time, and the percentage of defects detected at different testing stages. Root-cause distribution can also be measured to identify whether defects are primarily associated with requirements, design, coding, integration, testing, data, or other areas.

For example, a high number of defects detected late in the software lifecycle may indicate weaknesses in earlier review or testing activities. Similarly, repeated defects belonging to the same root-cause category may suggest that an existing corrective action has not addressed the underlying process problem. Metrics can therefore provide evidence for deciding where quality-improvement efforts should be concentrated.

However, metrics should be interpreted carefully and in context. A reduction in the number of reported defects does not automatically mean that software quality has improved because it could also result from reduced testing or changes in reporting practices. Therefore, defect metrics should be considered together with testing coverage, release activity, system complexity, and other relevant information.

In a cross-functional enterprise environment, a shared set of meaningful metrics can also improve communication between teams. QA, development, business, and management stakeholders can use common data to discuss recurring problems and improvement priorities. When combined with qualitative root-cause analysis, defect metrics provide a stronger foundation for continuous software quality improvement.

3. Cross-Functional Enterprise QA Case Study and Methodology

3.1 Enterprise QA Environment

Enterprise software environments are generally large and complex systems that support important business operations and serve a large number of users. Such environments may include web applications, mobile applications, databases, application programming interfaces (APIs), cloud services, third-party systems, reporting platforms, and internal enterprise applications. Because these components are interconnected, a defect in one component can sometimes affect the behavior of other components. Therefore, quality assurance in an enterprise environment requires continuous testing and coordination between different technical and business teams.

The QA environment normally includes several stages such as development, system testing, integration testing, user acceptance testing, staging, and production. Defects can be identified at any of these stages. QA engineers are responsible for designing and executing test cases, validating business requirements, performing regression testing, and reporting unexpected behavior. Developers investigate the technical implementation, while business analysts and product owners help determine whether the observed behavior matches the intended business requirement.

Table 1. Cross-Functional Enterprise QA Case Study and Methodology

Methodology Component	Description	Key Activities	Expected Outcome
Case Study Selection	Selection of an enterprise QA environment involving multiple functional teams.	Define system scope, QA processes, teams, and defect categories.	Clear case-study boundaries.
Defect Data Collection	Collection of historical and current defect information.	Gather defect reports, severity, priority, modules, timestamps, and resolution details.	Structured defect dataset.
Cross-Functional Analysis	Collaboration among QA, development, business, operations, and product teams.	Conduct defect-review meetings and share technical and business perspectives.	Broader understanding of defect causes.

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

Root-Cause Identification	Systematic investigation of why defects occurred.	Apply 5 Whys , Fishbone/Ishikawa , Pareto analysis , and defect trend analysis.	Verified root causes.
Cause Categorization	Classification of identified causes into meaningful categories.	Categorize causes as requirements, design, coding, testing, environment, data, or process related.	Consistent cause taxonomy.
Corrective Action	Development and implementation of solutions addressing identified causes.	Assign owners, define actions, implement fixes, and validate results.	Reduced defect recurrence.
Preventive Action	Improvements designed to prevent similar defects in future releases.	Update test cases, requirements, processes, automation, and QA guidelines.	Improved defect prevention.
Effectiveness Measurement	Evaluation of whether corrective and preventive actions worked.	Monitor defect recurrence, escape rate, resolution time, and root-cause trends.	Measurable QA improvement.
Continuous Improvement	Feedback of findings into the enterprise QA process.	Review lessons learned, update standards, and refine QA practices.	Sustainable improvement in software quality.

In a cross-functional enterprise environment, defect management cannot be treated as the responsibility of the QA team alone. A defect may involve multiple components or may be caused by a weakness in a process followed by another team. For example, a failure identified during system testing may appear to be a coding issue, but further investigation may reveal that the original requirement was incomplete or that a dependent service provided unexpected data. This makes collaboration between different functions essential for accurate root-cause analysis.

The enterprise QA environment also generates a significant amount of information that can be used during defect analysis. Test execution results, defect reports, application logs, database records, source-code changes, deployment information, configuration details, and monitoring data can provide useful evidence. Proper organization of this information helps teams

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

understand the complete history of a defect and determine where and why the failure was introduced.

3.2 Roles and Responsibilities of Cross-Functional Teams

Cross-functional root-cause analysis involves people with different responsibilities and areas of expertise. Each participant provides a different perspective that can help the team understand the defect more completely. The QA team generally identifies, reproduces, documents, and validates defects. QA engineers also provide information about the test conditions, expected behavior, actual behavior, test data, and environment in which the defect occurred.

Developers play an important role in investigating the technical cause of a defect. They examine source code, recent changes, application logic, database interactions, APIs, and error logs to determine how the unexpected behavior occurred. Developers may also identify technical dependencies or implementation decisions that contributed to the defect. Their involvement is particularly important when the defect is related to complex application logic or system integration.

Business analysts and product owners provide the business perspective during the analysis. They help verify the original requirement and clarify the expected behavior of the system. In some cases, a defect may occur because the requirement was ambiguous, incomplete, or interpreted differently by different teams. Business stakeholders can therefore help determine whether the issue originated from a misunderstanding of business rules rather than from the software implementation itself.

Architects, database specialists, DevOps or operations teams, and other subject-matter experts may also participate when the defect involves their respective areas. Architects can investigate design-level problems, database specialists can analyze data-related issues, and operations teams can examine infrastructure, deployment, or configuration problems. The involvement of these specialists prevents the analysis from being limited to a single technical viewpoint.

Effective responsibility distribution is important for preventing confusion during root-cause analysis. The team should clearly identify who owns the investigation, who provides technical or business information, who approves corrective actions, and who verifies the effectiveness of those actions. Such coordination helps ensure that the analysis remains focused on solving the underlying problem rather than assigning blame to an individual or department.

3.3 Defect Investigation and Analysis Process

The defect investigation process begins when a defect is identified and recorded in the organization's defect management system. The initial defect report should contain sufficient information to understand the problem, including the affected functionality, steps to reproduce the issue, expected result, actual result, environment, test data, severity, and relevant

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

screenshots or logs. A clear defect report reduces unnecessary communication delays and provides a reliable starting point for further investigation.

The next step is defect validation and reproduction. The responsible QA engineer or investigation team attempts to reproduce the reported behavior under the same or similar conditions. Reproduction helps confirm that the issue is genuine and allows the team to observe the failure directly. If the defect cannot be reproduced, the team may investigate differences in environment, configuration, test data, application version, or user conditions.

After reproduction, the team performs a detailed investigation to identify possible causes. The investigation may include reviewing requirements, design documents, source-code changes, database records, test cases, API responses, application logs, configuration files, and deployment records. The team should examine not only the immediate failure but also the sequence of events that led to it.

Root-cause identification is then performed using suitable analysis techniques. For simple defects, the Five Whys technique may be sufficient to trace the problem from its visible symptom to an underlying process or technical cause. For more complex defects, techniques such as fishbone analysis, fault tree analysis, or Pareto analysis may provide a broader view. Multiple contributing factors should be considered when a single cause cannot fully explain the defect.

Once the root cause has been identified, the team defines corrective and preventive actions. Corrective action addresses the current defect, while preventive action focuses on reducing the possibility of similar defects occurring in the future. For example, fixing incorrect code may resolve the immediate issue, while improving code review practices or adding automated regression tests may help prevent recurrence.

The final stage involves verification and closure. QA validates that the defect has been corrected and that the associated preventive actions have been implemented where applicable. The team may also monitor subsequent releases to determine whether similar defects appear again. If recurrence is observed, the original root-cause analysis may need to be reviewed and improved.

3.4 Evidence Collection and Data Analysis

Reliable evidence is one of the most important elements of effective root-cause analysis. Decisions based only on assumptions or individual opinions can result in incorrect conclusions. Therefore, cross-functional teams should collect objective information from different stages of the software development and testing lifecycle.

Important sources of evidence include defect management records, test execution results, requirements documents, design specifications, source-code repositories, code review records, application logs, database information, API responses, configuration files, deployment records,

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

and monitoring systems. The relevance of each evidence source depends on the nature of the defect. For example, a configuration-related defect may require detailed environment information, while a business-rule defect may require greater focus on requirements and acceptance criteria.

The collected evidence should be organized according to the defect timeline. Establishing a timeline can help the team determine when the defect was introduced, when it could have been detected, and when it was actually discovered. Comparing these stages can reveal gaps in requirements review, development practices, testing, integration, or deployment procedures.

Data analysis can also be performed across multiple defects rather than focusing on a single incident. Historical defect information can be grouped according to root-cause category, severity, application component, detection stage, recurrence, and resolution time. Such analysis can identify patterns that may not be visible when individual defects are examined separately.

For example, if a large number of defects are repeatedly associated with incomplete requirements, the organization may need to strengthen requirements review and stakeholder validation. Similarly, repeated integration defects may indicate a need for improved interface documentation, integration testing, or communication between dependent teams. Therefore, defect data can provide evidence for broader process improvements.

The quality of the analysis depends on the quality of the recorded data. Inconsistent defect categories, incomplete reports, missing logs, or inaccurate root-cause information can reduce the usefulness of historical analysis. Organizations should therefore establish common reporting standards and encourage teams to record sufficient and accurate information during the defect lifecycle.

3.5 Proposed Root-Cause Analysis Framework

Based on the requirements of cross-functional enterprise QA environments, a structured root-cause analysis framework can be proposed to provide a consistent method for investigating defects. The framework begins with **defect identification and classification**, where the reported issue is validated and categorized according to its type, severity, affected component, and possible source. This provides an initial understanding of the problem and helps determine which teams should participate in the investigation.

The second stage is **evidence collection and defect reproduction**. The investigation team gathers relevant logs, test results, requirements, code changes, configuration details, data, and deployment information. At the same time, the defect is reproduced whenever possible so that the team can directly observe the failure. This stage ensures that the analysis is based on actual evidence rather than assumptions.

The third stage is **cause analysis**. The team identifies immediate causes, contributing factors, and possible underlying causes using appropriate techniques such as Five Whys, fishbone

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

analysis, fault tree analysis, or Pareto analysis. Cross-functional participation is particularly important at this stage because different team members may identify causes that would otherwise be overlooked.

The fourth stage is **root-cause confirmation**. A proposed root cause should be supported by available evidence and should explain why the defect occurred and why it was not detected earlier. The team should distinguish between the direct technical cause and deeper process-related or organizational contributing factors. This prevents the analysis from stopping at a superficial explanation.

The fifth stage is **corrective and preventive action planning**. Corrective actions are defined to resolve the current problem, while preventive actions are designed to address the underlying weakness. Actions may include changes to code, requirements, design, test cases, automation, review processes, documentation, configuration, or team practices. Each action should have a clearly identified owner and an expected completion or verification point.

The final stage is **monitoring and continuous improvement**. After corrective and preventive actions are implemented, their effectiveness should be evaluated. Future defects can be compared with historical data to determine whether recurrence has decreased. Root-cause categories and defect trends can also be reviewed periodically to identify new areas requiring improvement.

The proposed framework therefore follows a continuous cycle of **identify, classify, investigate, collect evidence, analyze causes, confirm the root cause, implement corrective and preventive actions, and monitor results**. Its main purpose is to create a consistent and collaborative approach that can be applied across different enterprise QA environments. By combining structured analysis with cross-functional participation and data-based decision-making, the framework can support organizations in moving from reactive defect correction toward proactive defect prevention and continuous software quality improvement.

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

4. Findings, Challenges and Framework Analysis

4.1 Common Root Causes of Enterprise Defects

The analysis of enterprise software defects indicates that defects can originate from several stages of the software development and testing lifecycle. Unlike simple applications, enterprise systems usually involve multiple teams, technologies, integrations, databases, business rules, and deployment environments. As a result, identifying the actual root cause of a defect often requires examination of more than one component or process. The most common root causes can generally be grouped into requirements-related, design-related, coding-related, integration-related, data-related, environment-related, and testing-related categories.

Requirements-related issues are one of the major sources of defects in enterprise applications. Ambiguous requirements, incomplete business rules, frequently changing requirements, and differences in interpretation between business and technical teams can result in incorrect system behavior. In some situations, developers may implement a requirement correctly according to the available documentation, while the implemented behavior does not satisfy the actual business expectation. Such defects demonstrate that the problem may exist before coding begins.

Design-related defects can occur when the architecture or technical design does not adequately consider system dependencies, scalability, security, data flow, or integration requirements. A design decision that appears appropriate for one component may create problems when the component interacts with other enterprise systems. These defects may remain undetected until integration or production because their effects are not always visible during isolated component testing.

Coding defects are another important category. These may include incorrect logic, improper handling of exceptional conditions, incorrect calculations, boundary-condition errors, or failures to follow established coding standards. However, the coding error should not automatically be considered the complete root cause. Further investigation may reveal contributing factors such as insufficient code review, unclear requirements, inadequate unit testing, or time pressure during development.

Integration and interface problems are particularly common in enterprise environments because applications frequently communicate through APIs, messaging systems, databases, and third-party services. Differences in data formats, interface contracts, authentication mechanisms, service versions, or error-handling behavior can result in failures. Such defects often require collaboration between teams responsible for different systems.

Data-related defects may arise from incorrect, incomplete, outdated, or inconsistent data. Test environments may also contain data that does not adequately represent real business conditions. A system may therefore pass standard test cases but fail when it encounters unusual or

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

unexpected data combinations. Environment and configuration problems can similarly cause defects when development, testing, staging, and production environments are configured differently.

Testing-related root causes include insufficient test coverage, missing test scenarios, incorrect test data, inadequate regression testing, and failure to automate important repetitive scenarios. A defect that reaches production may therefore indicate not only a problem in the application but also a gap in the quality assurance process. Identifying these broader causes allows organizations to improve their development and testing practices instead of repeatedly fixing individual symptoms.

4.2 Cross-Functional Collaboration Challenges

Although cross-functional participation improves the breadth of root-cause analysis, it can also introduce several collaboration challenges. Different teams often have different responsibilities, priorities, technical knowledge, and definitions of success. QA teams may focus on system behavior and user expectations, developers may focus on implementation details, business analysts may focus on business requirements, and operations teams may focus on system stability and deployment. These different perspectives are useful, but they can also create communication difficulties during investigations.

One major challenge is unclear ownership of defects. A defect may involve multiple teams, making it difficult to determine which team should lead the investigation or implement the corrective action. For example, an application failure may involve a coding issue, an incorrect database value, and an integration problem at the same time. If ownership is not clearly established, teams may spend time discussing responsibility instead of identifying and resolving the underlying causes.

Communication gaps can also affect the quality of root-cause analysis. Technical information may not always be communicated in a form that is easily understood by business stakeholders, while business requirements may not always contain sufficient technical detail for developers and testers. Differences in terminology, documentation practices, and communication channels can result in misunderstandings or incomplete information.

Another challenge is the availability of evidence. Logs may be stored by operations teams, test results may be maintained by QA teams, and code changes may be available only through development repositories. If teams do not have appropriate access to relevant information, investigators may be forced to depend on incomplete evidence. This can increase the time required to identify the root cause and may lead to inaccurate conclusions.

Time and resource constraints are also significant challenges. Enterprise teams often work under release deadlines, and detailed root-cause analysis may initially appear to delay defect resolution. As a result, teams may focus on applying a quick fix rather than investigating the

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

underlying reason for the problem. While this approach may restore functionality quickly, it can increase the possibility of recurrence.

Differences in organizational priorities can further complicate collaboration. Teams may have separate performance measurements or delivery targets, which can unintentionally encourage local optimization rather than overall quality improvement. A successful cross-functional RCA process therefore requires clear responsibilities, shared quality objectives, open communication, and a culture that treats defect analysis as a learning and improvement activity rather than an exercise in assigning blame.

4.3 Defect Recurrence and Prevention

Defect recurrence occurs when the same or a closely related problem appears again after an earlier defect has supposedly been resolved. Recurring defects are particularly important because they can indicate that the original root cause was not properly identified or that the corrective action addressed only the visible symptom. For enterprise systems, repeated defects can increase maintenance costs, delay releases, affect users, and reduce confidence in the software.

One common reason for recurrence is the implementation of a narrow corrective fix. For example, if a particular input causes an application failure, developers may modify the code to handle that specific input without examining why the validation was missing in the first place. Although the immediate defect may disappear, similar failures may occur with other inputs. A broader root-cause analysis can identify the missing validation rule, incomplete requirement, or inadequate testing process responsible for the recurring problem.

Preventing recurrence requires organizations to distinguish between corrective and preventive actions. Corrective actions address the existing defect, while preventive actions address the conditions that allowed the defect to occur. Preventive measures may include improving requirements reviews, expanding test coverage, introducing automated regression tests, strengthening code reviews, improving test-data management, updating technical documentation, or establishing better integration testing practices.

Historical defect information can also support prevention. When defects are categorized consistently, organizations can identify recurring root-cause patterns across multiple releases and applications. If a particular category repeatedly contributes to defects, management and technical teams can focus improvement efforts on that area. This changes defect management from an incident-by-incident activity into a continuous quality improvement process.

The effectiveness of preventive actions should be monitored over time. Simply completing an action does not guarantee that the underlying problem has been eliminated. Teams should examine whether similar defects continue to occur and whether relevant quality metrics

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

improve after the action is implemented. This feedback allows organizations to revise ineffective preventive measures and strengthen successful practices.

4.4 Impact of Root-Cause Analysis on Software Quality

Root-cause analysis can have a significant impact on overall software quality because it encourages organizations to address the reasons behind defects rather than repeatedly correcting their symptoms. When RCA is performed systematically, teams gain a better understanding of weaknesses in requirements, design, coding, testing, integration, deployment, and operational processes.

One important impact is the reduction of recurring defects. When the underlying cause of a repeated problem is identified and addressed, the likelihood of similar failures can be reduced. This can improve the stability of applications and reduce the amount of time spent on repeated investigation and maintenance. The benefit becomes more significant in enterprise environments where the same application may support large numbers of users or business processes.

RCA can also improve testing effectiveness. Analysis of escaped defects can reveal which scenarios were not covered by existing test cases or which stages of testing failed to identify the problem. QA teams can use this information to improve regression suites, add boundary-value scenarios, introduce automation, or strengthen integration testing. In this way, production defects can become a source of learning for future testing activities.

Another important impact is improved collaboration between technical and business teams. A structured RCA process creates a common environment in which different stakeholders can examine evidence and discuss the defect systematically. Instead of focusing only on which team introduced the defect, the discussion can focus on what process or condition allowed it to occur. This can encourage a more constructive quality culture.

RCA can also support better decision-making through the use of defect metrics and historical information. Organizations can identify areas where defects occur frequently, determine which stages allow defects to escape, and prioritize improvement activities based on observed patterns. This provides a more evidence-based approach to quality management.

However, the impact of RCA depends on how effectively it is implemented. Poor-quality data, incomplete investigations, lack of stakeholder participation, or failure to implement preventive actions can reduce its value. Therefore, RCA should be treated as part of a broader quality management process rather than as an isolated activity performed only after serious defects occur.

4.5 Analysis of the Proposed Framework

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

The proposed root-cause analysis framework provides a structured approach for investigating defects in cross-functional enterprise QA environments. Its major strength is that it combines technical investigation with process-level analysis. Instead of stopping after identifying the immediate coding or configuration problem, the framework encourages teams to determine why the problem occurred and why existing controls did not prevent or detect it earlier.

The framework begins with defect identification and classification, which creates a common understanding of the issue. This stage is important because consistent classification allows organizations to compare defects over time and identify recurring patterns. The next stages, involving evidence collection and reproduction, ensure that the analysis is based on observable information rather than assumptions or personal opinions.

Another important feature is the involvement of cross-functional stakeholders. Enterprise defects frequently cross organizational and technical boundaries, so limiting an investigation to a single team may result in an incomplete explanation. By involving QA engineers, developers, business analysts, architects, database specialists, and operations teams when required, the framework allows the investigation to consider multiple possible causes.

The framework also emphasizes the separation of immediate causes, contributing factors, and root causes. This distinction improves the quality of corrective and preventive actions. For example, an incorrect line of code may be the immediate cause of a failure, but the deeper contributing factors could include an unclear requirement, missing review step, or insufficient test coverage. Addressing these factors can provide greater long-term value than correcting the code alone.

The framework supports continuous improvement through its final monitoring stage. After corrective and preventive actions are implemented, their effectiveness is evaluated using future defect information and relevant quality metrics. This creates a feedback loop in which lessons from previous defects are incorporated into future development and testing practices.

At the same time, successful implementation requires organizational support. Teams need sufficient time for analysis, access to reliable evidence, clearly defined responsibilities, standardized defect categories, and management support for preventive improvements. Without these conditions, RCA may become a documentation exercise rather than an effective quality improvement mechanism.

Overall, the proposed framework provides a systematic method for connecting defect management, root-cause investigation, cross-functional collaboration, and continuous improvement. Its practical value lies in helping enterprise QA teams move beyond immediate defect correction and develop a more preventive approach to software quality. The framework can also be adapted according to the size, architecture, development methodology, and operational requirements of different enterprise organizations.

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

Author Copy

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

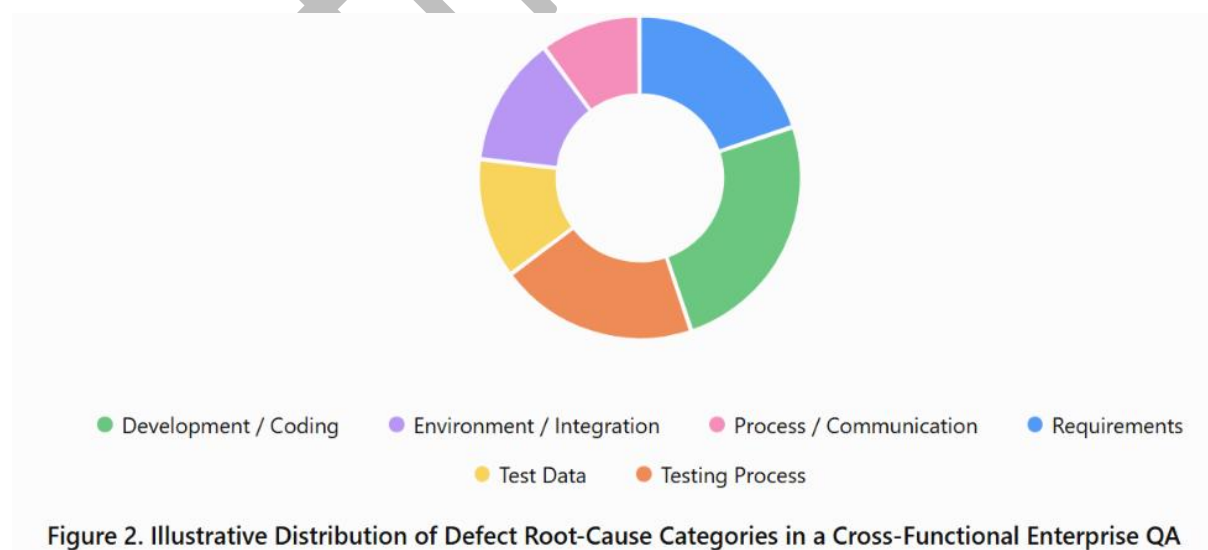
5. Conclusion and Future Scope

5.1 Conclusion

Defect root-cause analysis is an important activity for improving software quality in large and complex enterprise environments. Enterprise applications consist of multiple components, technologies, integrations, databases, business processes, and teams. Because of this complexity, a defect cannot always be understood by examining only the immediate technical failure. The actual cause may be related to requirements, design, implementation, test coverage, data, configuration, integration, deployment, or communication between different teams. Therefore, a systematic approach to root-cause analysis is required to understand defects in a comprehensive manner.

This study proposed a structured framework for performing defect root-cause analysis within cross-functional enterprise Quality Assurance teams. The framework focuses on defect identification and classification, evidence collection, defect reproduction, cause analysis, root-cause confirmation, corrective and preventive actions, and continuous monitoring. It recognizes that different defects may require different investigation techniques and that multiple contributing factors may exist behind a single failure.

The analysis also highlighted the importance of cross-functional collaboration. QA engineers, developers, business analysts, architects, database specialists, operations teams, and other stakeholders can provide different types of information during an investigation. Combining these perspectives can help teams identify causes that may not be visible from a single team's viewpoint. At the same time, clear ownership, effective communication, reliable evidence, and shared quality objectives are necessary to make collaboration successful.



A major finding of the study is that simply fixing a defect does not necessarily prevent it from occurring again. Organizations need to distinguish between immediate corrective actions and

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

preventive actions that address the conditions responsible for the defect. Historical defect data, root-cause categories, recurrence information, and quality metrics can be used to identify repeated weaknesses and guide process improvements.

The proposed framework therefore supports a shift from reactive defect management toward proactive quality improvement. Instead of treating every defect as an isolated incident, organizations can use defect investigations as opportunities to learn from failures and improve requirements, development, testing, integration, and deployment processes. When implemented consistently, root-cause analysis can contribute to more stable software, better testing practices, reduced defect recurrence, and stronger collaboration across enterprise teams.

5.2 Key Recommendations

Organizations implementing cross-functional defect root-cause analysis should establish a standardized RCA process that can be followed consistently across projects and teams. The process should clearly define the stages of investigation, required evidence, responsibilities of participants, documentation standards, and criteria for determining whether a root cause has been adequately established. Standardization can make investigations more consistent and improve the usefulness of historical defect information.

A second recommendation is to focus on evidence-based analysis rather than assumptions or individual opinions. Defect investigations should make appropriate use of requirements, test results, source-code changes, application logs, database information, configuration details, deployment records, and other relevant evidence. This approach can reduce incorrect conclusions and provide stronger justification for corrective and preventive actions.

Organizations should also establish clear roles and responsibilities for cross-functional investigations. A designated owner can coordinate the investigation, while relevant technical and business specialists contribute information according to the nature of the defect. Clear responsibility helps prevent delays caused by uncertainty regarding ownership and encourages teams to focus on resolving the underlying issue rather than assigning blame.

Another important recommendation is to maintain consistent defect classification and root-cause categories. Common categories such as requirements, design, coding, integration, data, environment, testing, and deployment can provide a useful basis for historical analysis. Organizations should periodically review these categories and update them when new patterns emerge.

Preventive actions should receive the same level of attention as immediate defect fixes. After identifying a root cause, teams should determine whether changes are required in requirements review, design validation, coding practices, test coverage, automation, test-data management, integration testing, documentation, or deployment procedures. Preventive actions should be assigned to responsible stakeholders and monitored until their effectiveness can be evaluated.

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

Finally, organizations should use defect metrics to support continuous improvement. Metrics such as defect recurrence, severity distribution, detection stage, resolution time, defect leakage, and root-cause distribution can provide useful information about software quality. These metrics should be interpreted together with project context rather than being used as isolated measures of team performance.

5.3 Future Enhancements

The proposed framework can be enhanced in the future through greater use of automation and intelligent data analysis. Enterprise organizations generate large volumes of defect information, including descriptions, logs, test results, code changes, and historical RCA records. Automated analysis techniques could help identify similar defects, group related issues, detect recurring root-cause patterns, and provide investigators with relevant historical information.

Artificial intelligence and machine learning can also be explored for supporting defect classification and root-cause prediction. For example, historical defect data could be analyzed to identify relationships between defect characteristics and previously identified causes. Such systems could assist QA teams by suggesting possible cause categories or related incidents. However, automated recommendations should be treated as decision-support information and should be validated by appropriate technical and business stakeholders.

Another potential enhancement is the integration of RCA activities with modern continuous integration and continuous delivery pipelines. Defect information could be connected with source-code commits, automated test results, deployment records, and monitoring information. This would allow organizations to establish a stronger relationship between a production failure and the development or deployment events that preceded it.

The framework can also be extended with real-time quality monitoring and predictive analytics. Instead of analyzing defects only after they occur, organizations could monitor indicators such as increasing failure rates, repeated test failures, frequently changed components, or growing defect concentrations. These signals could help teams identify areas requiring attention before they result in larger numbers of defects.

Future implementations may also include standardized organizational knowledge repositories. Lessons learned from previous root-cause investigations can be stored and made available to developers, testers, analysts, and other stakeholders. Such a knowledge base can reduce repeated investigation effort and help teams recognize previously observed failure patterns.

Finally, future research can evaluate the proposed framework across different industries and enterprise architectures. Comparative studies involving banking, healthcare, telecommunications, retail, education, government, and utility systems could provide information about how root-cause patterns differ across domains. Further empirical studies

Journal of AI & Innovation (AJAI)

Impact Factor: 16.4

Peer Reviewed Refereed Journal

ISSN-3245-459x (Online)

Acceptance Rate: below 5%

could also examine how structured RCA affects measurable indicators such as defect recurrence, production defect leakage, resolution effort, and software reliability.

5.4 References

1. Pressman, R. S., and Maxim, B. R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill Education.
2. Sommerville, I., *Software Engineering*, Pearson.
3. Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley.
4. Juran, J. M., and Godfrey, A. B., *Juran's Quality Handbook*, McGraw-Hill.
5. Ishikawa, K., *Guide to Quality Control*, Asian Productivity Organization.
6. Andersen, B., and Fagerhaug, T., *Root Cause Analysis: Simplified Tools and Techniques*, ASQ Quality Press.
7. IEEE Computer Society, *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, IEEE.
8. International Organization for Standardization, *ISO 9001: Quality Management Systems – Requirements*, ISO.
9. International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC/IEEE 29119 Software Testing*, ISO/IEC/IEEE.
10. International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC 25010 Systems and Software Quality Requirements and Evaluation (SQaRE)*, ISO/IEC.
11. Deming, W. E., *Out of the Crisis*, MIT Press.
12. Humble, J., and Farley, D., *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley.
13. Duvall, P. M., Matyas, S. M., and Glover, A., *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley.
14. Montgomery, D. C., *Introduction to Statistical Quality Control*, Wiley.
15. Pressman, R. S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, for concepts related to software quality, defect management, testing, and process improvement