# Data-Driven Insights for Software Development Process Improvement: A Defect Analysis

Melike Takıl
The Scientific and Technological Research Council of Türkiye (TÜBİTAK) Informatics and Information Security Advanced Technologies Research Center (BİLGEM)
Ankara, Türkiye
email: melike.takil@tubitak.gov.tr

Zeliha Dindaş
The Scientific and Technological Research Council of Türkiye (TÜBİTAK) Informatics and Information Security Advanced Technologies Research Center (BİLGEM)
Ankara, Türkiye
email: zeliha.dindas@tubitak.gov.tr

***Abstract*— This paper presents an analysis of defects found in a software project at a Capability Maturity Model Integration (CMMI) Level 5 public institution, required to manage and improve their processes using statistical and other quantitative techniques, develops software for other public organizations. A dataset of software defects collected via a task and issue management platform was analyzed, focusing on defect severity, defect type, detected activity and affected components. Defects were classified and a root cause analysis was conducted to identify defect-prone areas and underlying causes. The motivation of this work is providing a practical perspective on how public-sector software teams operating under governmental regulatory constraints can use defect data to fix defects and to support long-term process improvement and quality assurance. The results of this research are intended to contribute future projects of the organization and provide referenceable value to other governmental software units aiming to enhance their defect management capabilities.**

***Keywords*— *software defect analysis; software quality; root cause analysis***

## I. INTRODUCTION

In software engineering, the identification, classification, and analysis of defects play a key role in providing product quality and maintaining process efficiency. Defects which are broadly defined as flaws, errors, or bugs in software have direct consequences on system reliability, maintainability and user satisfaction. Their early detection and resolution are crucial for reducing rework and cost while preserving the credibility of organizations, particularly in high-stakes public sectors. Defect analysis is an important component of software improvement process. It enables organizations to trace the origins of defects, understand the conditions under which they arise, and implement preventive measures to reduce their recurrence. Many studies have shown that systematic defect tracking and root cause analysis contribute significantly to achieving higher maturity in software processes, as seen in models such as the CMMI. Organizations at higher maturity levels (such as Level 5) are expected to leverage quantitative defect data for continuous process optimization and predictive quality management.

While defect analysis is a well-established practice in the private sector, its application within public-sector software development presents unique challenges and opportunities. Public institutions are often subject to greater regulatory oversight, extended stakeholder ecosystems, and longer procurement cycles. These factors underscore the importance of software quality and magnify the implications of defects. Moreover, since public-sector software is frequently reused, integrated, or interfaced with systems from other agencies, the downstream effects of unresolved or recurring defects can be profound.

The remainder of this paper is structured as follows. In Section 2, a review of the relevant literature and related works is presented in order to contextualize the study. In Section 3, the methodology is described, including the motivation for the study, its scope, the dataset and variables used, and the expected outcomes. In Section 4, the data is analyzed from multiple perspectives to uncover significant patterns and insights. Finally, in Section 5, the main conclusions are drawn and potential directions for future work are outlined.

## II. RELATED WORK

Defect tracking is a critical component to a successful software quality effort. In fact, Robert Grady of Hewlett-Packard stated in 1996 that “software defect data is the most important available management information source for software process improvement decisions,” and that “ignoring defect data can lead to serious consequences for an organization’s business” [1]. Defect and problem metrics are among the few direct and quantifiable indicators of software process and product quality. Although customer perceptions of software quality may vary, the frequency of defects is widely recognized as being inversely proportional to quality. Such measurements provide objective insights into reliability, correctness, efficiency, and usability of the software system [2]. Preventing defects early in the software development lifecycle is more effective and less costly than detecting them later. Key defect prevention strategies—such as formal methods, process improvements (e.g., CMMI), training, and automation—play a crucial role in enhancing software quality. This proactive approach complements the focus on post-deployment defect analysis by underscoring the importance of early quality assurance practices [3]. Defect Causal Analysis (DCA) is a structured approach used to identify systematic errors that repeatedly cause software defects and failures. This technique aims not only to prevent similar defects in the future but also to enable their earlier

detection through root cause analysis [4]. A common method within DCA is the use of defect classification data—such as Pareto charts—to identify the most frequent defect types, which often point to underlying process weaknesses [5]. Once these patterns are recognized, organizations can implement targeted process improvements to reduce recurrence of similar issues [6]. The present study follows a similar rationale by examining defect distributions and contributing factors to support software process improvement.

The outputs produced by a process can be characterized by some quality attributes, the values of which generally show some variation. The causes of variation can be classified as natural causes (also called common causes) or assignable causes (also called special causes). Natural causes are those that are inherent in the process and that are present all the time. Assignable causes are those that occur sometimes and that can be prevented. A process is said to be under statistical control if all the variation in the attributes is caused by natural causes [7][8]. Therefore, Statistical Process Control (SPC) control limits were used to detect defects throughout the software development process. Usage of control charts can lead to reduction in the control limits causing process improvements. It has been observed that rigorous monitoring of control charts plotted for process parameters like defect density and taking timely corrective and preventive actions would lead to process improvements [9].

## III. METHODOLOGY

This section outlines the methodological approach adopted to investigate defect trends and root causes within a public sector software project. It describes the rationale behind the study, the scope and structure of the dataset, the selected variables, and the expected outcomes, all of which contribute to a systematic and data-driven defect analysis process.

### A. Rationale and Scope

This study was carried out in response to a noticeable increase in software defects detected in the production environment of a public sector software project. Given the potential impact of such defects in public services, it became essential to investigate the nature, distribution, and timing of these issues. The primary objective was to identify critical defect patterns, root causes, and components most affected.

Software quality metrics are periodically monitored using dashboards visualized through a Business Intelligence (BI) tools. When defect counts began to increase, a more detailed investigation was required to identify trends, seasonal patterns, and component-level defect concentrations. Moreover, since data interpretation and context play a crucial role in defect analysis, the project's technical lead and project manager were actively involved in scoping the dataset. Their input ensured the inclusion of relevant variables and the exclusion of irrelevant entries, thereby improving the accuracy and relevance of the analysis. It is emphasized that the reactive aspect of defect management by analyzing already reported and resolved issues, aiming to support transition toward proactive quality assurance in future phases. It aligns with the principles of Total Quality Management (TQM) and CMMI, focusing on continuous improvement, data-driven decision-making, and stakeholder engagement [10].

### B. Dataset and Variables

The dataset used in this study was obtained from a task and issue management platform employed by the institution to coordinate and oversee software development activities. This platform is deeply integrated into the organization's software lifecycle and serves as a central hub for managing project workflows, including backlog planning, sprint execution, issue tracking and quality assurance processes.

Acting as the authoritative repository for work-related records, the platform enables the systematic logging, categorization, assignment, and resolution of software issues. It facilitates end-to-end traceability by capturing detailed metadata for each issue, including attributes such as issue type, impacted components, severity, detected activity, sprint association, assignee and current status. Additionally, the system records all updates, comments, workflow transitions, and timestamps, allowing for detailed temporal analysis and retrospective evaluations. The tool is actively used by cross-functional teams comprising developers, testers, analysts, project managers, and technical leads. It supports both agile and hybrid project methodologies through features such as sprint boards, user story hierarchies, version tagging, and customizable workflows. This makes it possible to track the lifecycle of a defect from discovery through resolution with a high degree of transparency and consistency.

For the purposes of this analysis, the scope of the issues was narrowed to defects. The selection criteria included:

- Only resolved and closed issues were considered, to ensure that the analysis reflects confirmed defects rather than pending or misclassified reports.
- Only defects reported in production environments were included, as these are considered more critical due to their direct impact on end users and operational services. Issues identified in test environments were excluded, since their occurrence is expected and does not necessarily indicate process deficiencies.
- The analysis covers the period from January 2024 to April 2025, selected in collaboration with project's technical lead to focus on periods when defect trends increased.

A total of 147 defect issues met the inclusion criteria. Rather than including the entire dataset, we present a representative snapshot of the dataset and its fields in Table 1.

To ensure the relevance and reliability of the dataset, a preliminary validation process was conducted with the project's technical lead and project manager. This included reviewing ambiguous entries, verifying proper classification of defect types and deciding on the most appropriate variables. During pre-processing, a limited number of missing values were addressed by directly consulting project

team, whose validated input was used to complete the dataset.

TABLE 1 DATASET SNAPSHOT FOR FIELDS

| Field | Value |
|---|---|
| Issue Key | 143 |
| Issue Type | Defect |
| Sprint Period | 01.04.24 |
| Severity | Medium |
| Defect Type | Coding |
| Component/s | A |
| Detected Activity | System Monitoring |
| Resolution | Done |
| Status | Closed |

The following key variables were extracted from the task and issue management platform and used in the analysis:

- Severity: This attribute indicates the relative criticality of the defect, typically ranked on a scale (e.g., minor, medium, major). It reflects the potential functional or user-facing impact of the issue.
- Detected Activity: This captures the specific development or operational phase in which the defect was discovered (e.g., Development, Integration Test, Code Review, System Monitoring). This variable supports root cause analysis by highlighting gaps in earlier detection efforts.
- Component(s): This denotes the subsystem(s) or modules affected by the defect. The platform allows for multiple components to be tagged per issue, enabling an analysis of module-level quality.
- Detected Sprint: This indicates the sprint during which the issue was logged. This supports time-based analysis, especially within agile projects where delivery and quality metrics are tracked in sprint cycles.
- Defect Type: This refers to the technical nature of the defect (e.g., coding, architectural design, data, integration, User Interface (UI), performance). Accurate classification in this field is crucial for identifying systemic weaknesses in development or architectural design practices.

Each of these structured variables was used to segment the dataset and support both descriptive and diagnostic analysis. By leveraging standardized fields available within the task and issue management platform, the study ensured traceable, reproducible, and context-aware outcomes. Nevertheless, several data quality considerations were taken into account:

- Timeliness and accuracy of data entry: As the platform relies on manual inputs from team members, discrepancies in timing or completeness of entries may affect the accuracy of the dataset.
- Subjectivity in classification: The interpretation of what constitutes a "defect" versus another issue type may vary across individuals or teams, potentially introducing inconsistency.
- Dataset size: Although the 147 production defects analyzed provide sufficient detail for meaningful pattern recognition, the moderate size limits the statistical generalizability of the results. Software engineering experiments often have small sample sizes [11]. One way to manage this challenge is through improving the dataset itself, as it has been noted that "the improvement of data sets through enhanced data collection, pre-processing and quality assessment should lead to more reliable prediction models, thus improving the practice of software engineering" [12].

Despite these limitations, the active use of a task and issue management platform significantly enhances the reliability and depth of the analysis. Its integration into daily workflows ensures that the defect data reflects the operational reality of software development in a complex institutional environment.

### C. Expexted Outcomes

Identifying the conditions under which defects most frequently arise, determining whether specific modules or time periods exhibit elevated defect counts, and tracing the root causes behind these occurrences form the basis of this study. By leveraging this knowledge, the institution can reduce defect density—an outcome strongly correlated with maintainability and user satisfaction [13]. Improvements in defect management ultimately lead to shorter release cycles, lower maintenance costs, and enhanced end-user trust.

## IV. ANALYSIS

This section presents a structured analysis of production defect data by leveraging variables extracted from the task and issue management platform. The aim is to identify defect trends, classify defect types, assess component-level impact, and examine detection patterns to support actionable quality improvement and data-driven decision-making.

### A. Monthly Defect Counts

Several analyses were performed by using the available fields within the task and issue management platform. Monthly total defect counts and especially major defect counts were visualized in Figures 1a and 1b to monitor trends over time. It was observed that defect counts increased notably in certain months, prompting further statistical investigation.

To determine whether these increases were statistically significant or merely due to natural variation, an Upper Control Limit (UCL) was defined using the formula mean + standard deviation ($1\sigma$).

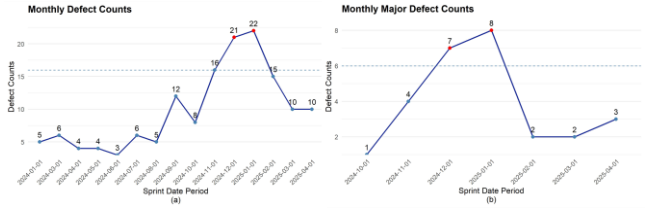


Figure 1. Monthly defect counts

This method provides a practical upper limit to identify months in which the defect count significantly exceeds the expected range, assuming a roughly normal distribution of the data. The rationale behind selecting the mean ± 1σ approach lies in its sensitivity to moderate but potentially meaningful anomalies. While traditional Shewhart control charts commonly employ mean ± 3σ, which encompasses 99.7% of all observations, such a strict threshold is more suitable for large datasets with high process stability, where false alarms must be minimized. A mean ± 2σ limit, capturing 95% of observations, offers a compromise but may still exclude relevant fluctuations in smaller or less stable datasets. In contrast, a mean ± 1σ threshold includes approximately 68% of data points under the normality assumption. This makes it particularly useful in exploratory analyses or early warning systems, where the primary aim is to flag unusual patterns for further review [14].

Using this method, the months of December 2024 and January 2025 were identified as exceeding the control limits, suggesting the presence of statistically unusual behavior. As a result, the possibility of seasonal effects influencing defect occurrences was explored. However, feedback obtained from the project team lead indicated that no seasonality was present. The variation was attributed to potential data entry adjustments or changes in reporting behavior. It was concluded that the increase in defects during these months likely stemmed from reporting-related factors rather than genuine increases in software issues. Then, it was agreed that team members should be provided with training or guidance on accurate and consistent data entry practices. Alternatively, the implementation of a control mechanism for validating input quality was proposed, aiming to improve the reliability of defect-related analytics in future reporting periods.

### B. *Defect Counts by Defect Type*

Defects were categorized into standard types such as coding, functionality, architectural design, data, UI, performance, integration and system-related issues.

According to the Pareto analysis shown in Figure 2a Coding defects dominated the dataset (77%), suggesting significant opportunities for improvement in development practices, code reviews and developer training. Functional, architectural design and data-related defects followed, indicating lesser but still notable concerns. Understanding the origin of major defects is essential for effectively issue prioritization, establishing risk management practices and enabling teams to focus on areas with the greatest potential impact on system reliability and user satisfaction. As shown in Figure 2b, the majority of major defects are Coding defects. This can be taken into account when planning actions.

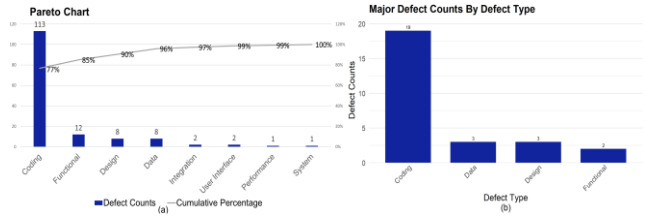


Figure 2. Pareto chart of defect counts by defect type

### C. *Defect Counts by Component*

A defect issue can affect multiple components simultaneously. As teams conduct a defect analysis to understand root causes, it becomes increasingly important to identify which specific components are associated with higher defect frequencies. This level of granularity enables teams to detect recurring patterns, assess component-level stability and prioritize quality improvement efforts where they are most needed.

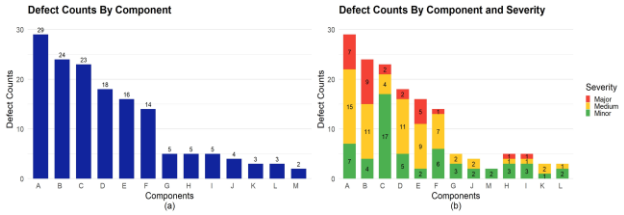


Figure 3. Defect counts by components

As illustrated in Figure 3a, a bar chart visualization was used to present the distribution of defects across different software components. The chart shows that three components exhibit a notably high concentration of defects compared to the others. Defect counts by severity level for each component displays in Figure 3b Identifying such major defect-prone components is critical, as it allows development teams to prioritize their efforts and conduct focused root cause analyses in the most problematic areas of the system. In the chart, although Component E exhibits a lower total number of defects compared to the others, it has a relatively high proportion of major defects. This aspect should be considered during task prioritization. Conversely, while Component C has a higher overall number of defects, the vast majority are classified as minor. Therefore, targeted interventions in this component may lead to a substantial reduction in the total defect count.

### D. *Defect Counts by Detected Activity*

The activity during which a defect is detected serves as a critical indicator for understanding the effectiveness of quality assurance practices throughout the software

development lifecycle. Conducting such analyses is essential for identifying defect patterns at a technical level and understanding in which activities defects are most frequently identified enables the implementation of targeted improvements and preventive measures.

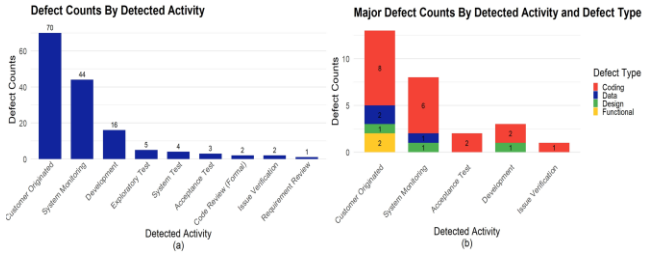


Figure 4. Defect counts by detected activity and defect types

To the Figure 4a, the analysis of defect detection activities revealed that the majority of defects were found after deployment, rather than during early development or testing stages. Specifically, the “Customer Originated” category accounted for the highest number of defects (70 cases), indicating that many issues were discovered directly by end users or stakeholders during actual system usage. The second most frequent detected activity was ”System Monitoring”(44 cases), reflecting the role of automated monitoring tools in identifying runtime anomalies and system-level issues. While this demonstrates that monitoring mechanisms are effectively capturing failures in the production environment, it also reinforces the need for earlier detection to reduce operational risk and customer impact.

Figure 4b presents the distribution of major defects categorized by detected activity and defect type. As illustrated, Coding defects represent the predominant defect type across all detected activities. This suggests that efforts aimed at minimizing coding-related defects could lead to a substantial reduction in the overall number of major defects. It may also mean that fewer of them will leak to the customer.

### *E. Analysis of Defect Issue Summarries*

As illustrated in Figure 5, detailed analysis of the defect issue summaries revealed that the most frequently reported defect issues were related to NULL (NPE) handling (26 instances), followed by business rule violations (23), and query-related defects (20). Additionally, a notable number of issues stemmed from incorrect data insertion operations (16), functional logic defects (13), and update operation failures (11).

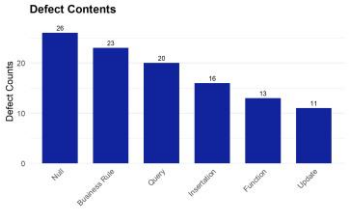


Figure 5. Defect issue summaries

This distribution indicates that a large proportion of the defects originate from NPEs, which typically occur when the code attempts to access or modify an object reference that has not been initialized. Previously, Figure 2 also highlighted that most defects were rooted in coding-related activities. The findings in this figure further corroborate that conclusion.

As for the null-related problems, static code analysis can detect some Null Pointer Exceptions (NPEs), particularly in cases where a method might return a null value, and the returned result is used directly—such as accessing a field or invoking a method—without checking for null. However, it cannot identify null values that originate from external sources such as databases or API inputs. In practice, many NPEs encountered during development tend to arise from such dynamic sources, which static analysis tools are generally unable to detect. Business rule violations rank as the second most common defect type after NPEs. Accordingly, allocating adequate resources and providing targeted training to address NPEs could significantly enhance code robustness. Furthermore, a detailed investigation into the origin of business rule violations specifically, whether they arise from customer miscommunication or analyst defects, would provide valuable insights to inform project decisions and process.

Ultimately, software defects are often the result of multiple, interrelated factors. Limited or superficial test coverage, insufficient domain knowledge, and time pressure can all contribute to quality issues. Late requirement changes and urgent requests disrupt planned workflows, reducing the time available for proper analysis and testing. Inadequate refactoring and poor adherence to clean code practices further degrade maintainability. Since many defects arise under complex conditions, identifying their root causes often requires detailed investigation.

## V. CONCLUSION AND FUTURE WORK

In this study, a comprehensive defect analysis was conducted for a public institution engaged in software development for public organizations. Given the critical nature of software applications and their direct impact on end users, identifying and understanding defects was of high importance. Monthly analyses were performed both for total defects and major defects, using ±1 sigma control limits to identify significant increases. Defects were categorized by defect type, component, detected activity. A Pareto analysis revealed that the majority of issues stemmed from Coding

defects. Components A and B were identified as the most defect-prone areas, both in terms of total and major defects.

When analyzed by detected activity, a significant portion of the defects, including major ones, originated from Customer Originated issues. The majority of these customer originated major defects were also related to coding. A deeper technical classification showed that most of the defects were associated with null-related problems, particularly Null Pointer Exceptions. Furthermore, a considerable number of defects resulted from misunderstandings of business rules, highlighting potential gaps in requirement analysis.

These findings highlight specific areas that require focused attention. The predominance of Customer Originated defects may indicate the need to intensify testing activities to identify issues prior to deployment. The high frequency of Coding defects related to null value handling suggests the necessity for targeted developer training and the establishment of best practices in coding standards. Additionally, relying solely on happy path testing is insufficient; comprehensive test coverage should include diverse input sets to ensure robustness against edge cases such as null values. The prevalence of business rule violations underscores the importance of conducting a thorough investigation into their root causes particularly to determine whether they stem from customer miscommunication or analyst errors. Such insights are expected to inform both project decisions and process improvements. The analysis presented here may serve as a foundation for future initiatives, such as increasing the involvement of analysts during early project phases.

While this study provides a detailed retrospective analysis of current defects, future efforts should shift towards predictive and preventive measures. To this end, a broader range of quality measures and metrics will be incorporated into the future work to demonstrate effectiveness and strengthen its scientific value. A potential direction for future research is the development of a predictive model capable of anticipating defect occurrences prior to deployment. Such a model would enable proactive mitigation strategies and contribute to enhanced overall software quality.

## ACKNOWLEDGMENT

The authors thank TUBITAK BILGEM for supporting this work. Special thanks go to the project team for their valuable contributions throughout the research.

## REFERENCES

[1] R. B. Grady, "Software failure analysis for high-return process improvement decisions," Hewlett-Packard Journal, vol. 47, no. 4, Aug. 1996.

[2] IEEE Standard for a Software Quality Metrics Methodology, IEEE Standard 1061-1990, Inst. Electr. Electron. Eng., New York, NY, USA, 1990.

[3] L. M. Laird and M. C. Brennan, "Software defect prevention," Proc. 14th Int. Symp. Softw. Rel. Eng. (ISSRE), Denver, CO, USA, 2003, pp. 2–13, doi: 10.1109/ISSRE.2003.1257423.

[4] F. Shull et al., "Investigating the role of defect causal analysis for software process improvement," Empirical Software Engineering, vol. 8, no. 4, pp. 357–382, Dec. 2003.

[5] G. D. Everett and R. McLeod, Software Testing: Testing Across the Entire Software Development Life Cycle. Hoboken, NJ, USA: Wiley-IEEE Computer Society Press, 2007.

[6] V. Basili and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments," IEEE Trans. Softw. Eng., vol. 14, no. 6, pp. 758–773, Jun. 1988.

[7] D. C. Montgomery, Introduction to Statistical Quality Control, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, 1996.

[8] D. J. Wheeler and D. S. Chambers, Understanding Statistical Process Control, 2nd ed. Knoxville, TN, USA: SPC Press, 1992.

[9] V. Vashisht, “Enhancing Software Process Management through Control Charts,” Journal of Software Engineering and Applications, vol. 7, no. 2, pp. 87–93, 2014.

[10] W. E. Deming, Out of the Crisis. Cambridge, MA, USA: MIT Press, 1986.

[11] B. Kitchenham and L. Madeyski, "Recommendations for analysing and meta analysing small sample size software engineering experiments," Empirical Software Engineering, vol. 29, no. 6, Article 137, 2024.

[12] Bosu, M.F., & MacDonell, S.G. (2013). Data quality in empirical software engineering: a targeted review. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE 2013), pp. 171–176.

[13] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 8th ed. New York, NY, USA: McGraw-Hill, 2014.

[14] D. C. Montgomery, Introduction to Statistical Quality Control, 8th ed. New York, NY: Wiley, 2019, pp. 18–19.