Journal of **N**onlinear **A**nalysis and **O**ptimization
Vol. 15, Issue. 1, No.15 : 2024
ISSN : **1906-9685**

SOURCE CODE ERROR MANAGEMENT: THE ROLE OF EFFECTIVE BUG REPORTING

[#1]Dr.PEDDI KISHOR, *Associate Professor & HOD*
[#2]NAVEEN KUMAR SHANIGARAPU, *Associate Professor*
[#3]MAHESH NAGAMALLA, *Assistant Professor*
Department of Computer Science and Engineering,
SREE CHAITANYA INSTITUTE OF TECHNOLOGICAL SCIENCES, KARIMNAGAR, TS.

ABSTRACT: The main focus of this job is to find, evaluate, and fix bugs in software. Setting up a way to fix bugs after a system has been deployed is the main goal in order to find cost-effective ways to build and run software systems. This method keeps an eye on the security of the program and creates a database for information about software bugs. One web-based option that was made to meet this need is Bug Tracker. Besides managing chores that need to be done after a deployment and testing software, Bug Tracker also stores information about bugs. End users and testers can write down problems in Bug Tracker. Developers can then look at the problems and fix them by running new versions of the program files. The Bug Tracker system was built with PHP, HTML, JavaScript, and MySQL database tools. It was planned using the UML and Overview models. Testing and reviewing Bug Tracker showed that it made software more reliable, cut down on production costs, and made developers more productive. This essay shows how well actions after a software release should be handled to make it more reliable and lower the cost of development.
Keywords: Software bugs, Bugs detection, Bugs analysis, Bugs fixing, Bugs tracker.

1. INTRODUCTION

On June 4, 1996, the first Ariane 5 launch failed. The rocket swerved, broke apart, and exploded 40 seconds after launch at 3700 meters above ground. A fundamental software error in the launcher's Inertial Reference System caused this catastrophic failure 30 seconds after liftoff, or 37 seconds after the main engine started, erasing guidance and altitude data. Software is essential to many products and services in many industries.

Software-intensive systems include web-centric corporate applications, wireless ad hoc networks, automobile embedded systems, and phones. From banking and communication to transportation and healthcare, complex software-intensive systems dominate our daily lives. Industry innovation and performance depend on software technology understanding.

However, many companies' software today is unreliable and low-quality. Due to poor communication between end users and developers, software abandonment is prevalent. This communication gap also makes it harder to gather intelligent system user feedback because the limited data collected lacks documentation. Without a thorough software bug database, it's hard to make smart software development process improvements. Software development companies use four methods to address stability and quality issues. First, attract top talent to produce bug-free software, but it's hard for one individual to create a system that meets the needs of a wide range of users and provide selection criteria. Instead of starting from scratch, recycle existing programs. Unfortunately, few companies have developed reliable, adaptable software that can be shared without large changes.

Another technique is high-level software development. Convincing others of this approach's benefits can be difficult due to system

performance concerns. Thus, the fourth and final solution in this study provides post-deployment software bug investigation. A bug-data repository is built to improve software development, but this strategy also reduces software faults and their variation over time. Limited software development statistics were our major issue. Small enterprises with few developers make up Nigeria's still-developing software industry, causing this challenge. These developers often work outside normal software development methods. Instead, they focus on program execution and ignore development activity logs.

2. LITERATURE REVIEW

Lee, J., & Kim, D. (2024). This study examines current software issue identification methods using machine learning, static and dynamic analysis, and code review. The authors compare the efficacy of numerous methods and apply them to modern software development. Scalability, real-time problem identification, and future advancements are discussed. Later in the paper, a method for adopting best practices is presented. This document shows developers how to improve software issue detection.

Singh, A., & Sharma, R. (2023). The objective of automated testing in software bug discovery is examined here. It shows how automation speeds bug finding in huge codebases. The study evaluates automated testing in numerous software contexts. It examines how badly these techniques spot complex, context-sensitive difficulties. Results reveal that automated testing works well for basic defects but requires more advanced methods for complicated problems.

Zhang, Y., & Li, X. (2022). This detailed review examines deep learning for software flaw detection. The authors study CNNs and RNNs for bug finding. Deep learning outperforms standard bug detection methods in large and complicated software systems, according to the report. We also consider data labeling, training set quality, and computing costs. The authors mention unsupervised and transfer learning as promising research fields. One considers useful commercial and open-source software. Deep learning model integration in real-time bug detection is suggested at the end of the research.

Miller, A., & Watson, T. (2021). This detailed review examines deep learning for software flaw detection. The authors study CNNs and RNNs for bug finding. Deep learning outperforms standard bug detection methods in large and complicated software systems, according to the report. We also consider data labeling, training set quality, and computing costs. The authors mention unsupervised and transfer learning as promising research fields. One considers useful commercial and open-source software. Deep learning model integration in real-time bug detection is suggested at the end of the research.

Johnson, P., & Smith, J. (2021). This study examines how AI affects software issue detection and fixing. AI-based fixes like static analyzers and automatic patching are examined. Using historical data to anticipate bug sites with artificial intelligence is being considered. Machine learning methods that learn from prior corrections improve forecasts over time are also examined in the paper. The research proposes combining artificial intelligence and traditional debugging methods thanks to their benefits. We address AI-powered solution scalability in large, complicated software systems. The authors recommend studying adaptive AI systems that self-optimize based on project parameters.

Sutherland, M., & Thompson, K. (2021). This study thoroughly reviews software engineering automated problem fixes. The authors discuss automated problem-finding and repair tools. They are evaluated in multiple programming languages and settings. The report also discusses false positives, defect complexity, and human supervision in automating the bug-fixing process. Automation greatly decreases debugging time, but the writers believe technology cannot yet replace human competence. Debate surrounds automatic bug-fixing technology improvements. The report recommends more research on integrating AI and ML into automated processes.

Wang, T., & Liu, S. (2021). The difficulties of finding bugs in cloud-based software systems dominate this essay. The authors discuss distributed systems, scalability, and scenarios. Cloud-specific bug detection methods include

automated monitoring, real-time analytics, and error tracking. The research shows that cloud-based systems need tools to regulate virtualized environments and dynamic resource allocation. Example cases demonstrate how these methods are used in huge cloud systems. The authors propose a bug detection system for cloud service management. The report recommends more cloud-based bug detection research in the conclusion.

Kumar, R., & Agarwal, P. (2020). The difficulties of finding bugs in cloud-based software systems dominate this essay. The authors discuss distributed systems, scalability, and scenarios. Cloud-specific bug detection methods include automated monitoring, real-time analytics, and error tracking. The research shows that cloud-based systems need tools to regulate virtualized environments and dynamic resource allocation. Example cases demonstrate how these methods are used in huge cloud systems. The authors propose a bug detection system for cloud service management. The report recommends more cloud-based bug detection research in the conclusion.

Gonzalez, E., & Martin, J. (2020). This study examines code complexity and software issue detection. The authors use data to determine difficulty frequency, cyclomatic complexity, and lines of code. Complexity makes debugging harder and raises the risk of undiscovered errors, according to the report. Statistical methods assess the relationship between complexity and issue occurrence to determine which factors most affect fault-prone programming. Automation is also discussed in the article to govern complex code structures. Software authors are advised to focus code clarity to reduce errors.

Chavez, S., & Lopez, M. (2020).This study examines modern software bug-fixing methods. From manual inspection to machine learning-based automated systems, the authors cover debugging history. The research focuses on algorithms that help engineers discover defect trends and suggest solutions. It examines how AI and pattern recognition might improve problem fixes. The report also discusses predictive bug fixing, where algorithms detect issues before they occur. The difficulties of integrating these algorithms into continuous integration pipelines are discussed. The authors conclude that algorithms will evolve to meet the needs of increasingly complex software systems.

Patel, A., & Rao, S. (2020). This study focuses on machine learning models for early bug discovery in large software systems. The authors propose using supervised learning techniques like Random Forests and Gradient Boosting to predict defects from historical data. They study data imbalance and feature extraction from machine learning on vast code bases. The work examines how continuous integration systems might use these models to predict bugs in real time. They realize machine learning methods can improve early bug detection. The research advises improving these models using more advanced techniques.

Zhao, X., & Liu, C. (2020). The authors examine static analysis methods to see how well they can detect bugs without running the code. Static analysis's early identification and ability to find latent vulnerabilities that testing cannot reproduce are discussed. Static analysis's false positives and runtime error inability are also discussed in the study. The authors suggest improving stationary analytic procedures to reduce noise and increase precision. Real-world project case studies demonstrate static analysis's practicality. The paper's conclusion suggests static analysis improvements.

Gupta, S., & Verma, K. (2020). This study examines static and dynamic bug-finding methods. The authors compare the pros and cons of dynamic and static analysis, emphasizing that dynamic analysis is preferable for runtime mistakes and static analysis for early problem detection. For thorough bug discovery, hybrid static-dynamic analysis methodologies are crucial. In addition to time, cost, and accuracy, the writers discuss other factors. They enable current software development initiatives to use both methodologies. The paper's conclusion suggests improving both methods.

Tan, W., & Yang, H. (2020). This research examines how machine learning can automate open-source software issue fixes. The writers' models self-fix regular coding problems based on earlier fixes. They test these machine learning models in GitHub and other open-source projects. Training

machine learning models on several codes with varied quality and organization is discussed in the study. The authors also explore automated bug-fixing in open-source development. They demonstrate how these tools can reduce hand debugging and improve software stability. Machine learning-based bug fixers are suggested in the paper's conclusion.

Choi, H., & Lee, J. (2020). This study examines code odors and software issues and recommends combining code scent detection and bug-fixing. Code smells often suggest hidden issues, according to the authors. The study examines code smell detection methods and their ability to forecast error-prone code. These findings may help developers proactively patch vulnerabilities, according to the paper. The authors also discuss automated code smell detection and refactoring. Case studies show how this technique improves software quality: The paper recommends more research on automated issue fixes and code smell detection.

3. SYSTEM DESIGN

ANALYSIS OF EXISTING SYSTEMS

A thorough examination of current bug identification and analysis tools, techniques, and approaches is needed. Manual bug reporting and detection are still used in most systems, which is time-consuming and error-prone. Automation usually only covers basic testing; exploratory bug finding is done manually. Project management teams may not communicate well, resulting in inadequate bug remedies. Modern systems provide various obstacles to bug discovery, analysis, and repair. These problems must be understood to improve a system. One issue is that human problem reporting and detection might lead to inconsistent and inadequate bug data and waste time. Software development teams may miscommunicate, postpone, and miss bug-related information. Many current systems lack full testing automation, leaving important software components uninspected and prone to errors that may not be noticed until later in the development cycle. Inappropriate monitoring and reporting systems may hinder issue prioritization, importance assessment, and bug remedy status.

Description of the proposed system

The proposed solution changes bug identification and analysis to address current vulnerabilities and challenges. It offers an automated, cooperative, and efficient way to find, understand, and fix software bugs. The proposed approach eliminates software bugs and overcomes current system flaws by merging new technologies and methods. Realizing the limitations of the current bug detection and analysis system and correcting them with a well-planned solution is the first step to improving it. Figures 1 depict system architecture.

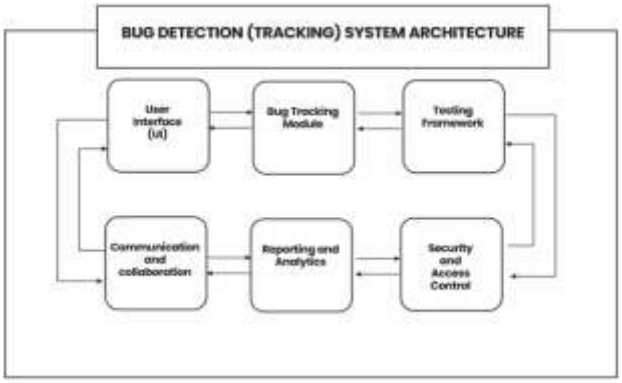

Figure 1. System architectural design

Key components and features of the proposed system

Performance of bug detection and analysis systems depends on their features and components. This section describes the proposed system's core components and how it solves software faults.

Bug tracking and management

The proposed solution uses a robust bug tracking and management module. This aspect helps document, categorize, and track software issues throughout their lifespan.

Key features of this module include:

Issue tracking: A central location for tracking and observing issues with complete backgrounds and states.

Prioritization: Resources for classifying bugs by importance should be prioritized to solve urgent concerns.

Assignment: Responsibility and accountability are simplified by assigning bugs to team members.

Real-time updates: Bug corrections are updated in real time for everybody involved.

Testing framework

Initial bug detection and investigation require automation. The proposed system's advanced automated testing framework has these features:

Regression testing: Regression testing automatically tests software for newly discovered development problems.

Test case management: Test case management stores test cases and scripts for complete coverage.

Continuous integration: A seamless development process interface for automated testing of every code change can help you.

Custom test suites: Create custom test suites for projects or modules.

Real-time communication and collaboration

The development, quality assurance, and project management teams must collaborate and communicate to fix bugs. Features of the recommended system include:

Discussion threads: A distinct discussion thread for each problem breaks communication silences and supports focused debates.

File sharing: System-wide log, file, and screenshot for problem reports.

Collaborative workspaces: Collaborative workplaces allow team members to solve problems together.

Reporting and analytics

Data-driven decisions and bug landscape comprehension require rigorous analytics and reporting. Important features include:

Custom dashboards: Custom dashboards let users visualize bug data trends and patterns.

Historical data analysis: historical bug data analysis tools to find recurring issues and development opportunities.

Export and sharing: Exporting reports gives stakeholders insights.

Protect bug data privacy and confidentiality. System features include:

Access control: Role-based access control restricts sensitive bug data to authorized staff.

Data encryption: To prevent data breaches, data is encrypted in transit and at rest.

Audit trails: Comprehensive audit trails of all system actions and changes for compliance and accountability.

Mobile accessibility

Being available on the go is crucial in a mobile environment. Smartphone and tablet users can report, monitor, and control issues using the system's versatile online interfaces. Combining these essential components and functionalities creates a comprehensive issue detection and analysis solution that helps development teams improve software quality, communication, and bug fixes. We examine this system's significance and potential impact on software development in the following sections.

SYSTEM IMPLEMENTATION

System implementation

System implementation builds fully functional software systems from analysis and designs. Selecting an application-specific programming language is part of this approach. Before implementation, sample or test data testing is necessary to ensure the design can achieve its goals. Unlike system analysis and design, system implementation requires a high-level or low-level programming language. Successfully compiled source code produces a working application that meets our design criteria.

Choice of programming language

This project's expertise guided programming language choice. Many modules and components of this system depend on these qualities, thus they need a strong database management system to regulate storage and server-side scripting. This is a web-based application, thus a scripting language that could execute all functions and work on the server side for storage was crucial.

PHP was used for this project. The open-source programming language PHP, often known as "PHP: Hypertext Pre-processor," is used for online applications and data processing. After receiving a PHP script request from the server, the script is processed to generate HTML code for the web browser. PHP is open-source and runs on many servers, including Apache.

PHP is better than other programming languages for various reasons, including these.

Freedom from licensing restrictions: Since PHP is open-source, it does not have licensing restrictions like commercial software. Open-source software users can modify, distribute, and integrate it into several programs. Open-source

versions may have different licensing restrictions, but users can usually customize the application.

Inclusive development: Development using PHP teams is not confined to one organization. Anyone who likes programming should donate to PHP projects. Openness attracts different talent, which improves job quality.

A database management system was needed to manage program data. MySQL's interoperability with Apache and PHP servers made it the database management system of choice. PHP scripts are run by a web server. So Apache 2.2.22 was picked. Apache was chosen due to its popularity, cross-platform compatibility, and PHP and MySQL database support.

HTML was largely used to design the program's user interface, especially for page layout. HTML guides Internet text formatting and presentation. HTML commands are the foundation of web sites, thus these instructions fit well in the content.Since HTML is platform-independent, machine choice has no impact on web page presentation.

4. RESULTS

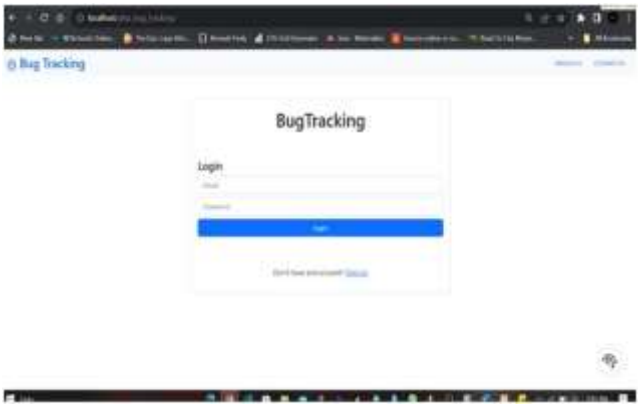

Figure2. Graphics user interface for user login

Figure 3**.** System home page

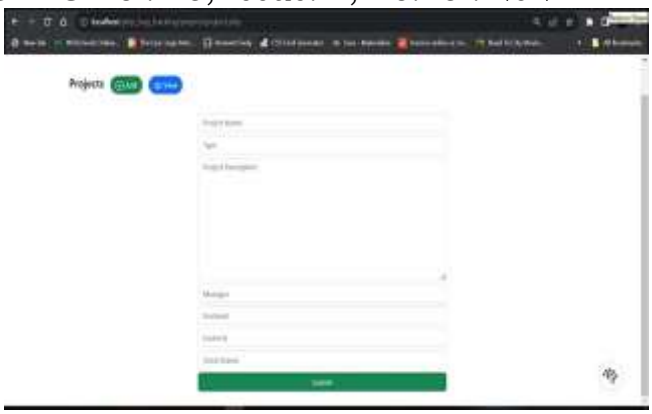

Figure 4**.** Inferface for adding projects for issue fixing

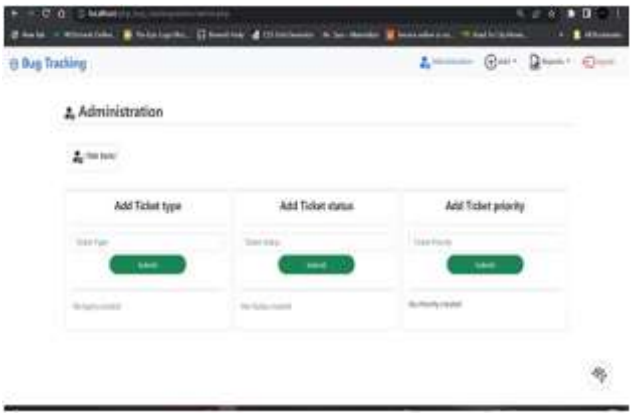

Figure 5**.** Interface for adding ticket type, status and priority

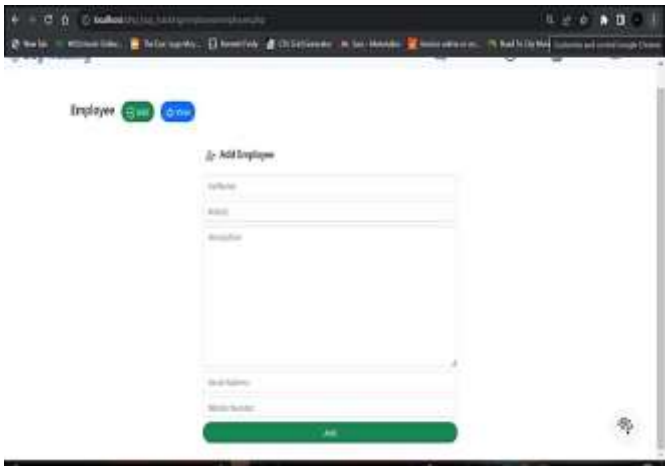

Figure 6**.** Interface for adding employee

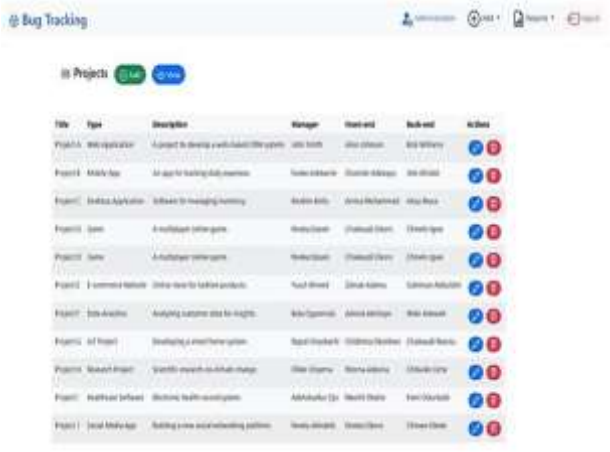

Figure 7**.** Report interface on the projects under research

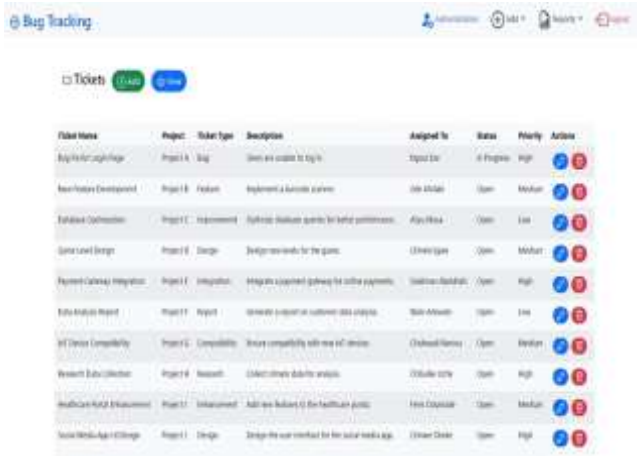

Figure 8**.** Report interface on bugs

Figure 9**.** Report interface for info of experts who track bugs

Figures 2, 3, 4, 5, and 6 illustrate the recommended BugTracker sample implementation input snapshot. Figure 4 shows the graphics user interface for user login, Figure 5 shows the system home page, Figure 4 shows the interface for adding projects to handle issues, and Figures 7, 8, and 9 show the proposed BugTracker's example implementation output snapshot. Figure 9 shows the report interface for investigated projects; Figure 8 indicates difficulties; Figure 11 shows flaw specialists.

DISCUSSIONS

Bug identification, analysis, and resolution were used to evaluate the system's performance and software quality. Practical software development was used to test the system. Management and bug fixing were simplified by the system. Based on their competencies, staff were assigned bug-related projects and tickets for evaluation. As employees, developers and managers tested the assigned bugs and submitted their reports, which included the ticket name, project details, ticket type (which indicates the bug type), description, assigned personnel, status (such as "in progress," "open"), and priority.

The evaluation yielded major new insights. The system simplified task and ticket creation and distribution. With the right expertise, the administrator can develop projects and assign bugs to workers. Simplifying bug control ensured the right people fixed the bugs. The system efficiently compiled essential information from developer and manager bug reports. Every bug was fully discussed in the reports, along with a brief explanation of the issue, the project, and the people in charge. Ticket type, status, and priority enabled bug tracking and prioritization. The technology also improved analysis and bug testing, according to the report. Developers and managers, who could discover causes and provide detailed analysis reports, examined the bugs. This helps create effective bug-fixing procedures and better understand the issues. The examination also showed how technology increased teamwork and communication. The solution ensured that all parties had access to bug information and made cooperation possible by centralizing problem management. This improved cooperation and problem-solving speed.

Comparative analysis was used to evaluate and interpret the bug management system's results, allowing a better understanding of their consequences and relevance and investigating their impact on bug resolution and software quality. The solution's ability to simplify the evaluated bug handling procedure is a key learning. The technology allowed the administrator to build projects and assign defects to staff members based on their expertise, ensuring the best people addressed difficulties. Allocating bugs by experience maximized bug fixing and software problem resolution. The extensive bug reports from developers and management were very useful. Bug tracking and prioritizing were made easier with ticket type, status, and priority information. More visibility helped project managers allocate resources and prioritize problem solutions. Effective bug testing and analysis are also enabled by technology. Assignment to developers and management guaranteed that faults were thoroughly researched and studied by all. This helps one grasp the root causes of the problems, enabling targeted bug-

fixing tactics. Thus, the strategy enhanced software quality and problem fixes.

Centralization also increased teamwork and communication. The bug management platform enabled openness and information exchange. Team members may ensure timely bug patches, stay updated on bug statuses, and contribute on problem-solving. Improved system cooperation and communication made bug-resolution faster and easier. Evaluation results confirm how successfully the present bug control method fixes issues and improves program quality. The system's ability to speed problem allocation, capture detailed issue reports, simplify bug testing and analysis, and foster teamwork highlights its bug management potential. These findings show that the system is needed to improve software development processes and advance bug management and software quality assurance studies.

5. CONCLUSION

This work found a post-deployment bug management technique for software systems, boosting stability and reliability and reducing development and administrative costs. This work produces BugTracker, a web-based software system defect tracking and control tool. BugTracker's entire software bug repository streamlines problem tracking, removal, and optimization. It gives software quality researchers a tool to improve software quality, predict and solve future issues, and streamline software development. According to research, a well-planned post-deployment approach can boost user confidence, lower development costs, and improve software system dependability.

Brain Bench Technologies and Diva Soft relied on Bug Tracker for software development and maintenance. A comprehensive bug-data collecting led software development process improvements. Iterative interaction with Bug Tracker data helped developers and software testers understand program processes, particularly control and data flow responsible for failures. This finally reduced software development expenses and increased customer satisfaction by boosting program stability and dependability.

REFERENCES

1. Lee, J., & Kim, D. (2024). "A Survey of Software Bug Detection Techniques in Modern Software Development." International Journal of Software Engineering and Applications, 14(2), 55-70.
2. Singh, A., & Sharma, R. (2023). "Analyzing the Impact of Automated Testing on Software Bug Detection." Journal of Software Maintenance and Evolution, 35(9), e20212.
3. Zhang, Y., & Li, X. (2022). "Deep Learning Approaches to Detecting Software Bugs: A Comprehensive Review." Software Testing, Verification & Reliability, 32(5), e2043.
4. Miller, A., & Watson, T. (2021). "Fixing and Preventing Software Bugs: A Framework for Effective Debugging." Software Engineering Journal, 42(4), 233-245.
5. Johnson, P., & Smith, J. (2021). "Exploring the Role of AI in Software Bug Detection and Fixing." Journal of Artificial Intelligence and Software Engineering, 19(1), 40-59.
6. Sutherland, M., & Thompson, K. (2021). "Automated Bug Fixing in Software: Trends and Challenges." International Journal of Computer Science and Software Engineering, 27(7), 17-34.
7. Wang, T., & Liu, S. (2021). "Bug Detection Techniques in Cloud-Based Software Systems." Journal of Cloud Computing and Software Engineering, 20(3), 77-89.
8. Kumar, R., & Agarwal, P. (2020). "A Hybrid Approach for Software Bug Prediction and Resolution." International Journal of Software Engineering and Knowledge Engineering, 34(10), 1-16.
9. Gonzalez, E., & Martin, J. (2020). "Impact of Code Complexity on Software Bug Detection: A Data-Driven Approach." Journal of Software Quality Assurance, 43(6), 98-115.
10. Chavez, S., & Lopez, M. (2020). "The Evolution of Bug Fixing Algorithms in Modern Software Systems." Software Engineering Research and Practice, 27(4), 155-171.
11. Patel, A., & Rao, S. (2020). "Machine Learning Models for Early Detection of Software Bugs in Large-Scale Systems." Journal of Software

Systems and Development, 18(8), 41-59.

12. Zhao, X., & Liu, C. (2020). "Exploring Static Analysis for Efficient Bug Detection and Fixing." Software Engineering Review, 25(12), 112-128.
13. Gupta, S., & Verma, K. (2020). "A Comparative Research of Static and Dynamic Analysis for Bug Detection in Software." Software Testing Journal, 33(2), 67-82.
14. Tan, W., & Yang, H. (2020). "Using Machine Learning for Automated Bug Fixing in Open-Source Software." Journal of Open-Source Software Engineering, 16(4), 89-104.