

Bug Tracking and Resolution System: A Rule-Based and Python-Driven Approach

Aditi Gupta^{1,*}, Rao Ranaditya², Devendra Kumar³

Abstract

The way things operate and how they will age from a performance and user interaction standpoint, what types of loads they can handle, and how maintenance will be needed in the future are all connected to how working out glitches affects the future and how it feels to the users. Many older systems record problems, track the steps taken to resolve them, and provide very little in the way of organizing and automating solutions. Some recent studies have started focusing on using machine learning to identify reporting issues by pattern recognition to sort and rank operational errors. For many entry-level and self-taught programmers and budget-constrained organizations, these systems are expensive and often economically unfeasible. Managing bugs from scratch is complex. This time we tried to manage them using basic Python. No black box models, just simple rules, and structured data. Reports in the standard format trigger automatic sorting based on presets. The combination of severities and the frequency of occurrence determine the urgency. Solutions are suggested based on the old fixes stored in the system's memory. Everything is done through a clear process; nothing is opaque. It is easy to comprehend because everything is connected to history. A new approach to error handling is being tested, and it is simple to implement. No high-level tech skills are necessary. The system is designed to process large volumes of data and to remove the sorting backlog. The tests show that it deals adequately with common bugs, which shows that there are smart solutions, even without sophisticated algorithms.

Keywords: Automated bug triaging, bug tracking and resolution, rule-based systems, Python-based systems, issue management, software engineering

INTRODUCTION

Today's digital technologies are deeply intertwined with many fundamental services, such as hospitals, banks, schools, and messaging systems. They are continuously updated and improved, and as a consequence, services become interwoven, leading to design flaws, improper configurations, errors in daily user interactions, system crashes, or security leaks. Tools such as Bugzilla, JIRA, and GitHub Issues capture user feedback, but identifying what needs to be addressed most urgently remains a task for humans, resulting in delays as feedback submissions accumulate [1].

*Author for Correspondence

Aditi Gupta
E-mail: aditisapna2002@gmail.com

¹Student, School of Compute Science and Engineering, Galgotias University, Uttar Pradesh, India

²Student, School of Compute Science and Engineering, Galgotias University, Uttar Pradesh, India

³Assistant Professor, School of Compute Science and Engineering, Galgotias University, Uttar Pradesh, India

Received Date: June 24, 2026

Accepted Date: August 25, 2026

Published Date: September 28, 2026

Citation: Aditi Gupta, Rao Ranaditya, Devendra Kumar. Bug Tracking and Resolution System: A Rule-Based and Python Driven Approach. Recent Trends in Programming Languages. 2026; 13(2): 17–27p.

To eliminate manual bugs, software developers are tasked with recreating software errors and then subjectively determining the severity of such issues. This is particularly true for large software systems. Despite the existence of predictive and sortative automation technologies, in most cases, the required algorithms must be complemented with extensive data, considerable preparation time, and significant computing power. Gentle, rule-based systems are better in most of these cases. Balanced with the right system objectives and a system in place to prioritize what needs to be addressed urgently, these systems remain available for their users, absorb system usage, and are more efficient.

Here, we introduce a new method of sorting software bugs that is based on a simple design, logical clarity, and low computing requirements. Next, we will look back at other similar attempts and then examine how this approach is constructed. The findings, as they will, appear shortly afterward, unfiltered.

PROBLEM STATEMENT

Software teams face endless bug reports during building, checking, and later updates. Tracking systems help record problems yet sorting them still relies heavily on people reading each case. One developer might check details, judge impact, weigh urgency, and propose solutions using past knowledge—slow work. Mistakes happen. Attention fades. Fatigue builds. Decisions waver under load [2].

As codebases expand, quicker fixes may go unnoticed. Overworked engineers have to deal with torrents of error comments that cause backlogs. Important bugs remain pending for a long time because manually sorting them is too time-consuming. Issue trackers and logging tools do very little to provide any kind of automatic resolution. Without smart tools, order in chaos is practically impossible to discern. Researchers are currently experimenting with smart tools that use language processing and pattern recognition to auto-sort software bugs. This type of system is becoming increasingly sophisticated but still requires massive amounts of pre-sorted data, powerful computers, and lengthy periods of time to configure. Small tech companies and educational institutions do not have the funds, technology, or expertise, causing them to fall behind.

While consulting, bug history could help in understanding ongoing problems that seem to be rather ignored. Automation, when introduced, seems to be more of a burden, creating inefficient systems that increase the workload on developers with little improvement in the end result. There is a clear demand for simple systems that are resource-efficient and fully transparent in their operations, quickly deployable and capable of handling repetitive tasks.

When coding meets clear instructions, a new paradigm can be established. Instead of making an educated guess, historical data are used to inform definable actions. Patterns make suggestions for improvements.

LITERATURE REVIEW

More researchers are interested in software development, lifecycle software and how it manages and classifies bugs. Most of this type of research attempts to automate processes using machine learning. For automation, the most noticeable example is Dinc and Tuzun [3] for the classification of case reports and automated generation of case report summaries. There is no doubt that automation is good at reducing the volume of work that must be done by people, but it is extremely poor because it is done using only historical and a priori rules. The manual component of error sorting is extremely tedious, but so is the automation of the process, which is highly computational and documentation-intensive.

There is a new angle to the large language model problem prioritization, but the outcomes are not clear. The usefulness of large language models is evident in the positive outcomes and performance on many tasks, but the operational cost is extremely high. The complexity of the large language model's advisory services is also evident, and the rationale behind their conclusions is often unclear. Smaller teams seem to be the most affected by the challenge of integrating these models into their workflow.

Software challenges can be addressed using historical data related to similar software challenges. Software ideas can be generated based on extrapolated data to be used as solutions for future software bugs. Emeto programs review historical data with descriptions of software bugs and how they were resolved to create suggestions. This helps teams maximize productivity by spending less time resolving bugs. This is especially true with large sets of historical data, as bugs tend to create data gaps and provide problems that are more difficult to gain insights from.

Regardless of what is going on, deep learning continues to be acknowledged by many practitioners, citing, among other factors, the construction of word embeddings as the source of progress. Rather than following the same path, the team led by Wu chose to experiment with report structuring networks. Yes, it demanded a lot more computational resource. In any case, the evidence showed that it produced better results.

Examining tools such as Bugzilla, JIRA, and GitHub Issues, it becomes evident how the workflow functionality works. However, studies are lacking when it comes to prioritization of tasks. These tools help in battles, but they do almost nothing in the way of effort- and time-saving automation. Most of the literature focuses on simplified situations and more ambiguous control systems. Most fruitful insights are generated by systems with a minimalistic approach. This is what the current study aims to achieve.

PROPOSED SYSTEM

I have yet to come across another tool that tackles software bugs this way. I have also not seen any other tool that prioritizes simple and clear instructions over long processes.

The tool uses Python. Instead of using artificial intelligence (AI) such as language models, deep learning, or other forms of AI, this tool runs only on the rules and data it has. It is designed to work as simply as possible, with clear and understandable actions. The workload during the sorting, ranking, and issue-fixing steps of the process is reduced because the routine tasks are performed by the process rather than the people. The code may be designed to make decisions, but those decisions are kept open to review, rather than hidden behind code like the rest of the functions [4].

Each time a new bug is reported, a new sprint is created. There is a process that must be followed when documenting a bug. This report minimizes misunderstandings and misinterpretations of the issues. Attention to detail is crucial. Make sure to document the bug's title, description, location, type, impact, the scenario that triggered the bug, and its reproducibility. The goal is to provide as much data as possible so that the validators can be optimized.

With every new report, automated sorting begins. For the most part, this is directed by the same set of rules that are frequently applied to identify what is wrong in the code. Automation can take care of issues that are structural, functional, data/storage, display, or connectivity. This is how text-to-system methodologies function. They take faulty data and build a system that recognizes patterns to organize the process.

The first step in handling bugs is assigning score-based priorities. This is not only about the severity of a bug. The frequency of occurrence is also important. If something non-core breaks functionality, that drives urgency. If users can easily reproduce the problem, it significantly raises the score. With the criteria, a total score is calculated to determine its level of priority, that is, high, medium, or low. Most would come to diverse conclusions, but this method ensures a uniform outcome. The reference numbers point to supporting sources at the end.

Another feature of this system is the ability to suggest potential solutions based on previous bug records. It comes preloaded with a history of resolved bugs, including one for each that contains a description of the problem and a record of the solution. When a new issue is entered, the system searches for previous issues based on possibly relevant criteria, including area, segment, and type of problem.

When an issue is detected that meets these criteria, the programmer is presented with the previous solution. When no appropriate previous solutions are available, the problem is considered significant enough to require manual inspection and is stored to help the system improve in the future [5].

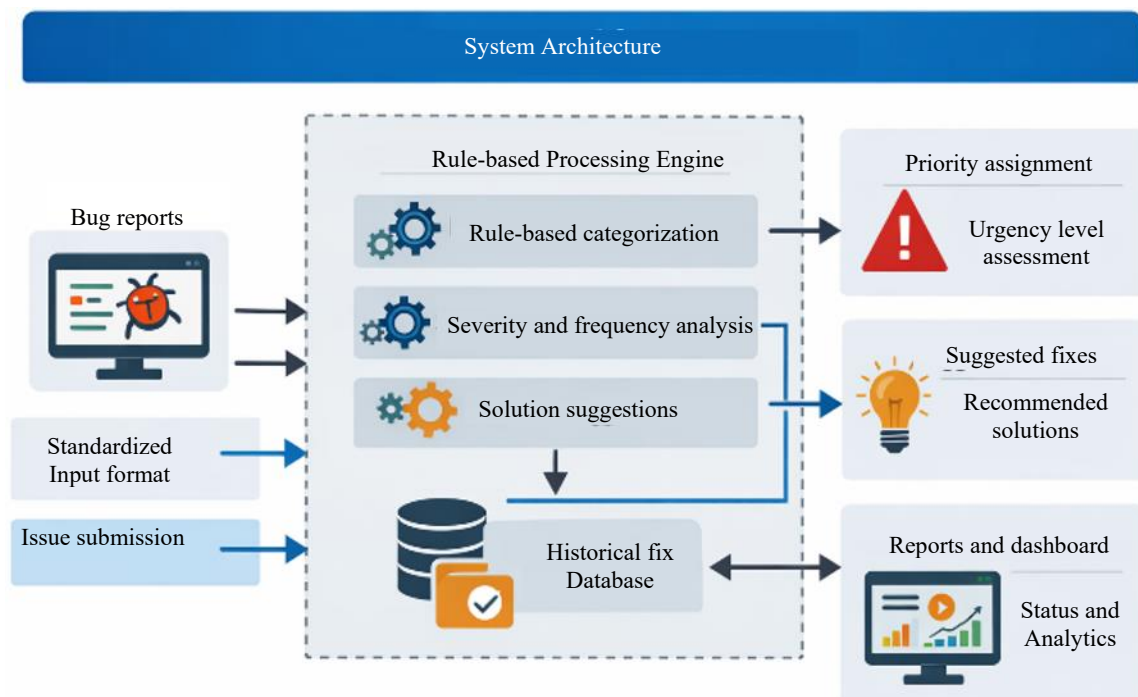


Figure 1. Proposed three-layer system architecture for bug processing and management.

One part discusses the methods used by users to submit bugs and how they track the status of their bugs. A section of the tool written in Python deals with the rules and decisions the system must make. The bugs and fixes are stored in the section below.

SYSTEM ARCHITECTURE

The project is divided into three layers, each with specific tasks to streamline the end-to-end bug processing lifecycle. The project does not leverage AI, Large Language Models (LLMs), or any other machine learning (ML) techniques. Instead, this project uses a system of rules coupled with methods based on a set of predetermined rules and structures to guide the processing of information [6]. The overall system architecture is shown in Figure 1.

Architectural Overview

The system architecture layers are:

1. Presentation layer
2. Application layer (Python core)
3. Data layer (Database)

Presentation Layer

The presentation layer of the system offers a web application and/or a command-line interface that users can use to enter bug reports and check the status of these reports. The system interface collects information in a specific structure to ensure that all required fields (such as title, description, module, error type, and level of severity, along with the environment in which the bug occurred) are completed. The system's goal is to ensure uniformity and consistency in the type of information/data being input so that the information is structured and organized before being processed.

Application Layer (Python Core)

The processing layer is the core of the system and carries out all other functions based on a system of predetermined rules and structures. This processing layer categorizes issues and assigns priority based on their level of impact, refers to previously documented solutions to the same bugs, and controls the status of issues based on their level of processing [7].

This processing layer is implemented to ensure that system/digital decisions are made in a consistent, transparent, and orderly manner, and to make it easier to manage and streamline workflows. In addition, it also provides a basis for incorporating future system enhancements.

Data Layer

The data layer obtains and stores the system's data using a relational database system, such as MySQL or SQLite. It keeps data on bug reports systematically and records the attributes and classifications of priority and resolution of the bugs. Organizing the data in a single location allows for bug resolution data retrieval, and when a recommendation system is data-rich, it increases the frequency and improves the accuracy of the algorithm.

System Workflow

The system takes in and processes bug reports in a predetermined manner. The submitted report first goes to the presentation layer, where the inputs are validated before being sent to the processing layer. In this layer, the system utilizes classification algorithms to determine the release classification of the bug and assign the priority before matching it to a previously recorded similar bug report. The system then updates the bug status in the data layer and records the resolution suggestion [8].

The system performs efficiently with a simple layered approach and is designed for small or educational teams with limited budgets. The result is displayed to the user via the system interface.

METHODOLOGY

This whole thing began with a clean slate. Step by step, someone puts together the tracker with care. Issues landed in place, sorted, some pushed ahead without help. Fixes appear where they are needed most. Behind each assignment is quiet reasoning at work. Something else drives it instead of smart algorithms. Logic steps in where complexity might otherwise take hold. Each output comes from matching inputs to predictable outcomes. Following paths happen only after decisions are made one by one. Without preset directions, motion cannot emerge at all. Patterns shape results here through order, not chaos. Other models pull answers from noise, but this one lines up each step like bricks in a wall. Starting again from saved steps makes things run smoother once all actions lock into place. Even if small mistakes happen, steady routines keep going because they were made to last. One kind runs basic upkeep tasks, whereas other shifts on its own as conditions change, getting better over time. The system moves ahead in one clear direction each cycle. Out of several possible routes shown at first, real tests shaped both this line and the entire setup behind it, as shown in Figure 2

Initial Stage: Collecting Reports on Bugs

The first step is to begin collecting bug reports. Reports must be filed uniformly. Each report should consistently include a bug title, bug summary, affected module, fault type, fault severity, location, fault occurrence, and bug report number. Using the same structure increases clarity, and the report will be easier to process with minimal ambiguity [9].

Segmentation of Processed and Reported Bugs

Once a bug report is submitted, the system that automates bug report processing begins to classify the report based on a number of definable features using a coding defect system: construction bug, execution crash bug, data flow bug, display bug, connectivity bug, etc.

The system does not “advocate” learning from the documents. However, it uses a set of predetermined criteria and processes the report in a typical rationale-based approach. Because this process relies entirely on a predetermined pathway, it is considered fully adaptive, with a predictable outcome [10].

Prioritization of Bug

First, we need to organize bugs, and from this, we will be able to rank them based on prioritization, frequency, impact, and consistency.

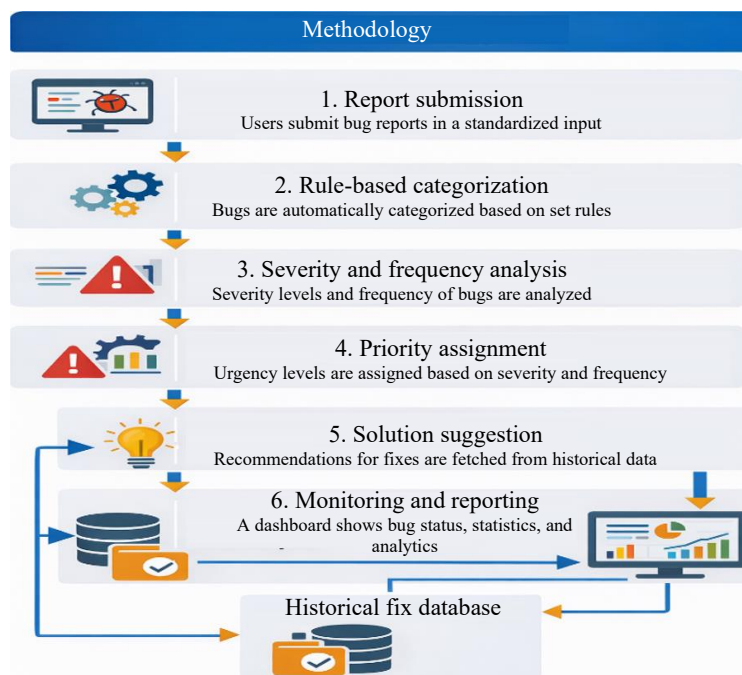


Figure 2. Proposed rule-based methodology for bug report processing and workflow management.

All these factors go into the ranking, with some features being more influential on the outcome.

Using Past Data to Improve Suggestions

What must be clarified is the influence of solutions already documented on the automation of new problems. The system uses stored information to resolve new problems by looking at past solutions. The category, software section, and type of error retrieve the memories of previous documents. The system, if available, can auto-suggest a stored solution. If there is no match, the user must manually review the case, and the new case will be added to the system for future use.

Workflow Management

The new case has a status of ‘New’ and does not change until a user interacts with it. There are no arbitrary status changes; there is a reason, be it system internal logic or user external input. When a user takes ownership of the case, its status changes to “Assigned,” and it remains there until the user is ready to start working on it. As soon as the user acts, the status changes to “In Progress,” and some changes occur. When the user completes the action, the case is not closed.

Instead, it will need to go through a review process before it can have a status change to “Resolved.”

TECHNOLOGY STACK

What is fascinating is how bug trackers integrate with slim setups built to flex without losing form. Most slot into system boundaries as if they have been there from the beginning. A class experimenting with some scripts might find them as useful as a small team coding collaboratively. Built light purposefully, the design chooses usefulness over show, sliding into place without weighing anyone down.

Python as a Programming Language

Python is the language that runs the system. It is clear and straightforward and outlines how things should work. It integrates with almost any system and makes coding less tedious. It provides a complete organizational structure and makes it easier to plan future changes. Aligned guides that are built into the system help to control the flow of information and data, and everything is ready to run [11].

User Interface

Initially, there is nothing, as there is always room to add bugs. You might be using a browser or a terminal version. One is web-based, and the other is command-window-based. Both run without making any sound. There is nothing to fill in, and then everything is set, ready for inputs to be processed.

DBMS

MySQL and SQLite are examples of data management systems. They contain data on bugs (what types, their priorities, what solutions, what fixes, etc.). Data is structured in a tabular form to maintain data integrity. Data integrity is maintained after alterations. Due to referential integrity, quick access to a single record is possible. Data structure often provides a means of easy manipulation with Python. Because everything is aligned early, less work is required in the future.

Utilized Libraries and Frameworks

A new browser window opens for the Flask Framework. The connection is instant, and complex systems are not necessary because the integration is done with the existing scripts. Do not use more complicated frameworks; the main functions are in small files now. Simple scripts are used to organize the rules, persistent data, relationships, and functions. When everything is held together, it is more flexible for development, and when numerous modifications are made, a high order is maintained when components are intact.

Tools for Development and Deployment

It is common practice to use software such as Visual Studio Code or PyCharm to set up the environment. Although there are many alternatives to Git, most users choose Git as their versioning tool for its simplicity and ease of use [12]. To get things running, you do not need an expensive laptop. Because of this, the setup is easy to showcase.

IMPLEMENTATION DETAILS

An example of a bug tracker that works in Python. It is a scalable bug tracker and keeps an unbundled design. It has modular components. This design offers small-scale addition flexibility. Due to these components, rules are set, and actions are automated. As more components are added, the design behavior remains the same.

Module Design

A module can be unified as one script that performs one function in Python. Then, there is one that identifies bugs, followed by another that tries to structure issues in a problem-solving order based on some preset rules. Then, there is one that prioritizes order, followed by one that offers recommendations. The last one governs how issues are processed. You can change any of these without affecting others. Having the modules separated in this way makes adaptation easier.

Bug Reporting Module

If something is broken, one tool initiates a bug report. The reporter explains this in a simple form that describes what broke, where it was located, the severity, and the actions to reproduce it. It is processed by someone after the report is submitted. A manual check is performed for everything. Nothing goes uninspected, and it can only be stored once all parts are done [13, 14].

Rule-Based Classification Module

Bugs can be attributed to many factors, from the code structure to their flexibility, glitches, illogical constructs, poor data flows, rendering issues, fractured networks, and so on. When a structure allows flexibility, patterns can change. Updates can be incorporated without a break in the structure.

Priority Assignment Module

Bugs cause a score to be assigned for every occurrence. Each bug's attributes severity, frequency, impact, and consistency are assigned a score, and each score can be adjusted with a multiplier to

determine a value. Consequently, the bug receives a rank: high, medium, or low. This happens every time, irrespective of the evaluator.

Fix Suggestion Module

Past solutions should not only be viewed fondly out of nostalgia. Along with every answer, the cause and reason for the answer are also documented. When a system bug occurs, the system attempts to find an answer based on type, category, and failing pattern. Often, this also means retrieving a legacy answer for the developers. In cases where the system fails to find an answer that meets the defined criteria, a person must step in to assess the situation, answer it, and insert the answer into the system afterwards.

Workflow and State Handling

Bugs have not yet developed the ability to hop. They are sequential crawlers. It then goes to assignment, followed by in progress, and finally, it lands on resolved. Crawling between these phases can be done manually (by a developer) or automatically (by the system). Once a bug has been created, it cannot be erased. Bugs can show many action statuses; therefore, a developer can always track the actions and attribute them to a specific user [15, 16].

Database Integration

Using connection APIs, a thread can link Python with its database. Here, it connects to a database where items (bug reports, types of bugs, priorities of bugs, resolutions of bugs, and fix history) are stored. Based on instructions, actions are performed for retrieval, updating, and streamlining. Each of these is a single, uncomplicated, and singularly focused task. They may also be arranged in a sequence.

Error Handling and Logging

Mistakes are first identified by simple trackers in the code. When things do not go as planned, these logs help avoid the issue in the future. Documenting bugs from the past allows for safer changes in the future.

RESULTS AND DISCUSSION

This section details our experience building and testing the Python bug tracking tool. Following this, we will discuss our observations. We focus on the effectiveness of the rules in sorting bugs automatically. We also discuss the issues of assigning levels of urgency and the suggestion of possible added features.

Experimental Results

Pages filled with notes or printed by machines carried familiar mistakes here and there—code built the wrong way, programs failing once started. Every problem reached the team in the same way real users sent them. Every new problem that came in with software demanded strict handling. Some messages pointed to patches in stored data that were broken, while others showed what glitches were on the screen. Nothing jumped ahead; everything moved forward step by step, as shown in Figure 3.

The majority of bug reports were sent where they needed to be, and this was made possible because of the clearly defined sorting rules. Every time a specific variety of bugs appeared, there was a specific and unchanged response. People describe problems in different ways, and this is where the uniqueness of the bugs comes from: a human's way of capturing data in a tedious, unpredictable, and often erroneous way. Compared to the irregular results from manual handling, there was a lot of stability in this.

Bugs were categorized using an early point-based system that allowed them to be classified as high, medium, or low, as shown in Figure 4. On the top tier, there were problems that occurred frequently, broke critical features, and had serious consequences. Given that this method relied on unambiguous, objective parameters, teams were able to identify urgent issues early in the process.

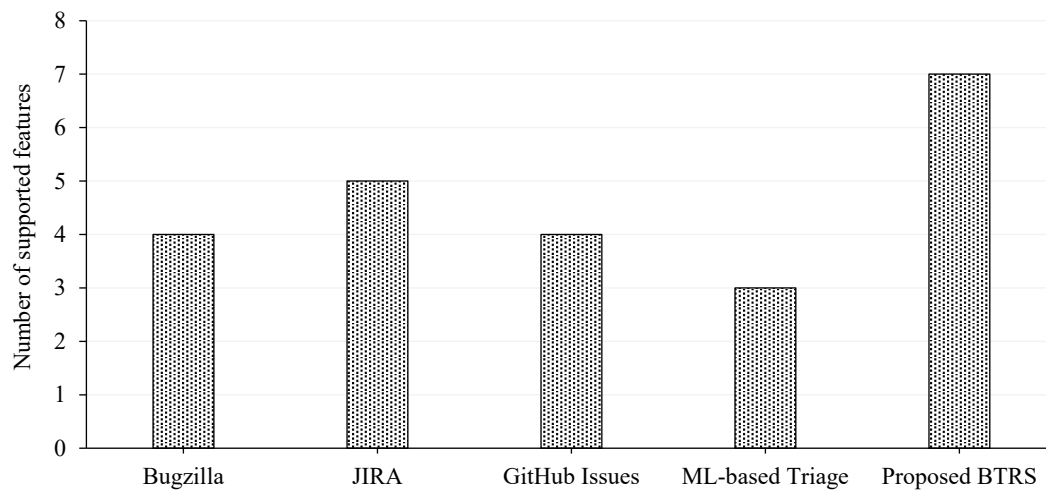


Figure 3. Comparison of bug management capabilities.

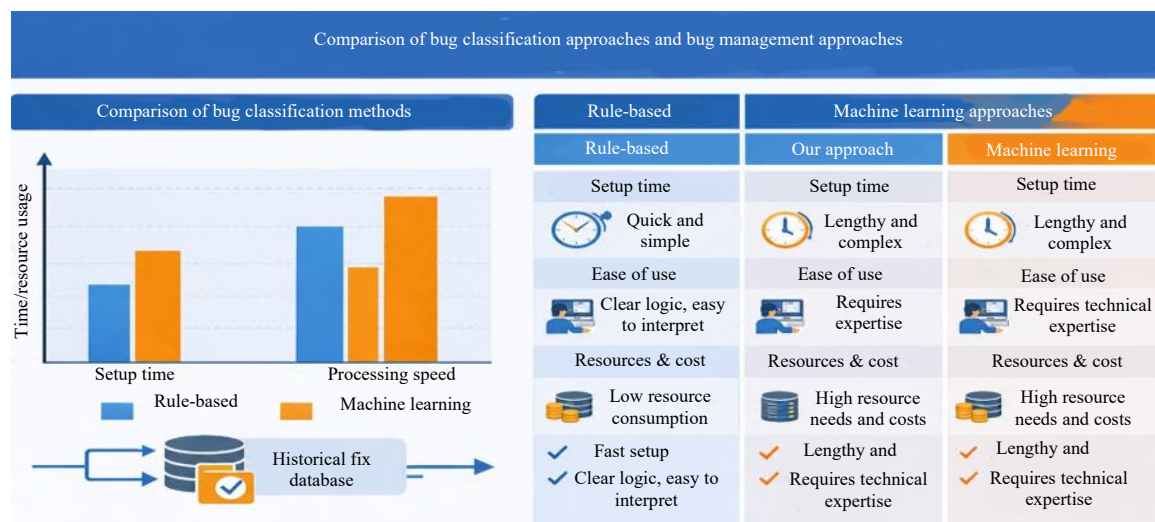


Figure 4. Comparison of bug classification and management approaches.

Most recurring bugs were of great help to the system in proposing ideas for fixes. Given the previous data, the system usually suggested previous solutions that were more effective, thus eliminating the need for a lengthy review. In the absence of historical data, issues were left for humans to resolve. Meanwhile, these new issues were added to the system's database, helping the system improve in small increments. Gradually, the system's knowledge reduced the unfilled gaps.

DISCUSSION

Current tests indicate that the system minimizes the time spent organizing tickets or fixing bugs, and because the system operates autonomously, the tagging, prioritization, and decision-making processes become automated. Additionally, the system will make daily repetitive tasks take less effort, allowing the user to shift their focus to more complex coding challenges. The removal of mundane tasks will enhance overall efficiency and productivity.

Prioritizing bugs was a guessing exercise, and that is a thing of the past. There are rules that govern each step taken, and as a result, the decisions made are documented. There are no black boxes; the opacity is gone as the user gains visibility into the process as the output is produced. University colleagues recognize this immediately. Once made, the code operates silently and is available to small collections. Gaining results is the belief–effort recognized, time respected, no extravagant promises, no overselling.

Seamless operation? Simple arrangements are better at managing complexity than convoluted ones. Small numbers become irrelevant; when focus is on clarity, the data can be prioritized. The right people can bring joy to surprises. Clean, unprocessed data wins every time. Simple rules make data collection easier. Hidden system inefficiencies are compensated for by other technologies to complete tasks at a higher speed. Where deep programming is needed, attention shifts to the gaps, and the gaps are filled with a lack of accuracy.

The events that occurred show that implementing a few simple rules can help make error correction easier in real-life situations.

ADVANTAGES

The system saves time and allows employees to focus their efforts elsewhere by automating the sorting, ranking, and offering possible solutions to the problems. The design ensures consistency, solid transparency, and unquestionable ease of processing. Furthermore, the system does not require modernization because it does not involve ML. The recommendations for possible solutions become more effective over time because of the integrated historical dataset for recurring problems. This system is the most optimal solution for educational systems or smaller or mid-sized development teams.

LIMITATIONS

The system is designed to follow certain processes. This design causes problems when trying to process incomplete descriptions or descriptions that are more free form. The system is incapable of processing novels and complex situations because of the design. Furthermore, the system does not process new datasets, meaning that the system will quickly become incapable of managing the environment if it ever becomes sufficiently large.

FUTURE SCOPE

Current rule-based systems are structured and streamlined, but the ability and precision of future systems can be increased. Lightweight natural language processing could be a good solution for copyright tools and/or influence some sort of classification for bugs. Adding visual tools would give analytics the ability to recognize and/or determine some sort of patterns that are repetitively stressing out the system. Simple analytics may be able to pinpoint systems and stress and may be able to recognize patterns that aid in detecting or alleviating stress. Integrating with GitHub or JIRA would allow for an automated sync of assigned tasks and/or updates to the workflow. Overall control of the data, the management, the system's flexibility, efficiency, and control, and the future growth of the system should be focused on to keep the system expandable.

CONCLUSION

The goal of this project is to demonstrate that with a little bit of rule-based automation, it is possible to convert Python workflows to manipulate bugs without any sophisticated tools. Automating bug detection, bug classification, and bug resolution explain the processes and the results of each task in a more simplified manner. The success of the project lies more in the simplicity of the explanation of the processes and the results, rather than in the sophistication. This is something very quiet academically that can be done in small team settings. Even with small systems and simple frameworks, basic structured processes are very efficient.

REFERENCES

1. Kumari N, Ahmed J, Raghuwanshi KK, Agarwal P. Automating bug triage: unleashing the power of machine learning and natural language processing [Preprint]. 2024. doi:10.21203/rs.3.rs-4294244/v1.
2. Chhabra D, Chadha R. Automatic bug triaging process: an enhanced machine learning approach through large language models. *Eng Technol Appl Sci Res*. 2024;14(6):18557-18562. doi:10.48084/etasr.8829.

3. Emeto IC, Anthony UW, Galadima AA, Ajayi CO, Fahm-Yesufa A, Amad AMB, et al. A web-based automated bug detection and fix suggestion system. *Int J Comput Sci Mob Comput*. 2025;14:28-38. doi:10.47760/ijcsmc.2025.v14i03.004.
4. Wang R, Ji X, Xu S, Tian Y, Jiang S, Huang R. An empirical assessment of different word embedding and deep learning models for bug assignment. *J Syst Softw*. 2024;210:111961. doi:10.1016/j.jss.2024.111961.
5. Morovati MM, Nikanjam A, Tambon F, Khomh F, Jiang ZM. Bug characterization in machine learning-based systems. *Empir Softw Eng*. 2024;29(1):14. doi:10.1007/s10664-023-10400-0.
6. Picus O, Serban C. Bugsby: a tool support for bug triage automation. In: *Proceedings of the 2nd ACM International Workshop on AI and Software Testing*. 2022. p.17-20. doi:10.1145/3536168.3543301.
7. Gartziandia A, Arrieta A, Ayerdi J, Illarramendi M, Agirre A, Sagardui G, et al. Machine learning-based test oracles for performance testing of cyber-physical systems: an industrial case study on elevators dispatching algorithms. *J Softw Evol Process*. 2022;34(11):e2465. doi:10.1002/smr.2465.
8. Sepahvand R, Akbari R, Jamasb B, Hashemi S, Boushehrian O. Using word embedding and convolution neural network for bug triaging by considering design flaws. *Sci Comput Program*. 2023;228:102945. doi:10.1016/j.scico.2023.102945.
9. Wu H, Ma Y, Xiang Z, Yang C, He K. A spatial-temporal graph neural network framework for automated software bug triaging. *Knowl Based Syst*. 2022;241:108308. doi:10.1016/j.knosys.2022.108308.
10. Liu Y, Qi X, Zhang J, Li H, Ge X, Ai J. Automatic bug triaging via deep reinforcement learning. *Appl Sci (Basel)*. 2022;12(7):3565. doi:10.3390/app12073565.
11. Zhang T, Chen J, Yang G, Lee B, Luo X. Towards more accurate severity prediction and fixer recommendation of software bugs. *J Syst Softw*. 2016;117:166-184. doi:10.1016/j.jss.2016.02.034.
12. Mani S, Sankaran A, Aralikkatte R. DeepTriage: exploring the effectiveness of deep learning for bug triaging. *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*. 2019. p.171-9. doi:10.1145/3297001.3297023.
13. Dinç E, Tüzün E. Judge the votes: a system to classify bug reports and give suggestions. *2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. 2025. p.1-10. doi:10.1109/AIware69974.2025.00025.
14. Montgomery L. Issue tracking ecosystems: context and best practices [Preprint]. 2025. arXiv:2507.06704.
15. Třeštlík P. Automatic evaluation of the frequency of errors occurring in real-world projects [master's thesis]. Plzeň: University of West Bohemia, Faculty of Applied Sciences; 2025. [Czech].
16. Long G, Gong J, Fang H, Chen T. Learning software bug reports: a systematic literature review. *ACM Trans Softw Eng Methodol*. 2026;35(7):1-47. doi:10.1145/3750040.