International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING

ISSN:2147-6799 www.ijisae.org Original Research Paper

Enhanced Bug Localization through Version Tag Embedding: A Comprehensive Approach to Efficient Software Development

[1*] N Rama Rao, K. Suresh[2]

Submitted: 12/02/2023 **Revised:** 13/04/2023 **Accepted**: 10/05/2023

Abstract: In order to localize bugs, this study suggests putting version markers in software delivery files. The article focuses on the importance and method of the planned build process, in which developers upgrade internal version numbers prior to registration to aid with precise problem localization. It is suggested that version tagging automation be used as a workable solution to problems like identifying file sources and exploiting vulnerabilities. The suggested solution has advantages in better managing project complexity, protecting against software infiltration, lowering costs, improving quality, and boosting productivity and dependability. The use of date tagging and external release numbers for software identification and compatibility testing is also covered in the article. Operating systems, version control, build tools, web servers, application servers, scripting languages, bug tracking tools, databases, and programming languages are all used in the research's specialised testing and development environment. The branching, merging, check-out, check-in, parameters, build numbering approach, and tagging operations are all thoroughly detailed. The suggested approach shows how to embed and update internal version tags in deliverable files, improving traceability and facilitating problem fixes throughout the build process. Discussions of the approach's importance in terms of bug detection and eradication, release management, complexity handling, intrusion defence, cost savings, and dependability are included. The page includes graphs that show the number of builds and the link between the number of builds and the state of problems that have been discovered and repaired before and after releases. Overall, the research offers workable solutions for effective software development and problem management by presenting a unique bug localization technique employing version tagging.

Keywords: *Build and Release Management, Software Configuration Management, Embedding Version Tag, Integration, Software Release Management, Version Control System.*

1. Introduction

A key component of software development and maintenance is bug localization, which is locating and fixing flaws or problems in software systems. Traditional bug localization techniques can be time-consuming and error-prone since they frequently rely on manual labour. This research report suggests a unique bug localization strategy by inserting version tags in software deliverable files to solve these issues.

The suggested method makes use of the software's internal version numbers in an effort to increase the precision and effectiveness of bug localization. Before registration, developers update the internal version numbers to make sure that the precise file sources may be quickly identified [1]. This article goes into great depth on the expected build process, which includes this upgrading procedure as a crucial step.

The automation of the suggested process is one of its main benefits. Developers may lower the risk of human mistakes by automating the version tagging process and ensuring that the fields are updated accurately during check-in. This automation not only saves time but also improves the overall traceability of defects and lessens the likelihood that unauthorized parties would exploit a vulnerability [2].

The suggested approach gives software development projects various advantages in addition to resolving the difficulties associated with bug localization. It makes it possible to manage more complicated projects, offers protection against software infiltration, lowers costs, boosts software performance and quality, and increases productivity and dependability [3]. By implementing the provided strategy, businesses may manage software development and bug fixes efficiently while streamlining their release management procedures.

The suggested approach also emphasizes the value of date marking and external release numbers for software identification and compatibility testing. Users may quickly determine if the programmer they are running has been updated or not by adding dates to release tags. This makes it easier to choose software versions and compatibility, which improves user experience and minimizes compatibility problems.

* *Associate Professor, School of Engineering, Department of CSE (AIML), Mallareddy University, Hyderabad, Email ID: ram.narvaneni@gmail.com.*
*2 *Associate Professor in School of Information Technology, JNTUH, India. Email Id:[2]kare_suresh@jntuh.ac.in*

The research makes use of a particular setting for testing and development in order to assess the efficacy of the suggested approach. Operating systems, version control software (CVS or SVN), build tools (Ant and Make), web servers (Apache), application servers (Tomcat), scripting languages (JavaScript, shell scripting), bug tracking software (Bugzilla), databases (MySQL), and programming languages (C, C++, Java) are some of the components that make up this environment.

The suggested solution heavily relies on the build numbering scheme. Project ID, major, minor, revision, build, and timestamp are some of the factors taken into account throughout the build numbering process. With the help of these attributes, every build is guaranteed to have a distinctive identity that enables effective tracking and management throughout the development lifecycle.

The study also discusses branching, merging, tagging, check-in, check-out, and other crucial build process elements. To provide readers a thorough grasp of how the suggested strategy combines with current software development practises, these procedures are described in depth.

The paper also provides a case-based examination of the bug localization process, highlighting the difficulties encountered and the suggested remedies. Discussions of the approach's importance in terms of bug detection and eradication, release management, complexity handling, intrusion defence, cost savings, and dependability are included.

Overall, this study proposes a unique method for localising bugs using software deliverable files that incorporate version identifiers. The suggested approach provides workable solutions to increase build process efficiency, software quality and stability, and bug localization accuracy and efficiency. The article has graphs that show how the suggested strategy affects bug fixes and release management. By using this strategy, businesses may enhance their bug management procedures and streamline their software development procedures.

The following describes the structure of this document. The second segment investigates the review of the literature. The suggested construction procedure is discussed in section 3. The implementation plan and outcomes are presented in section 4 of the document. The fifth and final section examines conclusions and future contributions.

2. Related Literature

The studies in [1] and [4] emphasise how important it is to apply the appropriate "software configuration management" (SCM) during project development since it enables effective change management. To increase efficiency and enhance the quality of software development with SCM, consistent and organised change management is required. Improvements must be duplicated and tested in order to increase system reliability and quality. Additionally, [5] describes change management as a process for handling alterations to specified objects while preserving quality and uniformity.

In order to find, track, and fix problems, version control systems (VCS), which are often used in project development, must be utilised with a "change management procedure" (CMP) that is clearly defined [6]. Software creation, release management, and file version recognition are some of these duties. Furthermore, the usual practises of branching, merging, and tagging in VCS [7] provide appropriate programme enhancements and problem solutions. The build system is similarly important since stakeholders communicate throughout the development phase. Engineering releases include information about the stability and quality of application releases, both qualitatively and quantitatively [8].

The costs and missed market opportunities resulting from product delivery delays highlight the need of risk management through revised timelines [9] [10]. Successful risk management techniques may increase revenue, lessen project flaws, reduce costs, and create opportunities for new projects. Additionally, it ensures on-time completion of projects [10, 11].

Utilising consistent naming conventions and tagging builds in VCS helps speed up project delivery [12]. To efficiently manage a high volume of mistakes, it is also required for large software systems to stay up to date on which files have been changed in subsequent releases [13]. Automating build processes has several benefits, such as boosting productivity and enabling the compilation of Java and C modules using build tools like Ant and Maven [14].

There is always room for development in the realm of software maintenance, despite the fact that there have been several contributions to the literature on bug localization. Recently, the promise of a "convolutional neural network" (CNN)-based method that combines natural language processing with programming language processing was shown [15]. However, because they primarily focus on search-based bug localization, these techniques' performance is still limited, frequently falling between 65% and 80%. This constraint resulted in performance improvements over other contemporary methods when taking the input of the programming structure into account.

This study offers a novel approach that uses version tags to get over the drawbacks mentioned in the literature. This method seeks to promptly inform developers of the intended sources of issue reporting. By automating the build process and making internal adjustments to each delivery, the recommended method raises product quality, makes module compilation simpler, minimises repetition, prevents the creation of erroneous builds, and lessens dependence on key people. Release management is a crucial part of the IT industry, and the timing of software releases is essential for reducing errors during runtime. "Continuous build integration" (CBI) is advised as a tool to track changes during integration [8] as opposed to build automation systems like Maven and Ant.

The literature review closes by providing a complete summary of relevant studies and highlighting the significance of SCM, managing changes, controlling versions, automating build processes, localising bugs, and continuous integration in the development of software. While addressing present issues, the recommended build method seeks to improve software development practises.

3. Proposed Build Process

One of the most important parts of this study contribution is the suggested construction method. It focuses on the internal and external version numbers connected with software builds and discusses the importance and method of the build process. The build process' goal is to make sure that programmers update their internal version numbers before registering in order to provide precise file source recognition.

The suggested approach places a strong emphasis on automating the version tagging process in order to get around problems with manual updates and potential vulnerability abuse. During check-in, developers may accomplish the desired results and lower the risk of unauthorised access by automating the modifications of version data. A crucial component of the suggested approach is this automation solution, which provides workable methods to improve the build process.

The recommended approach assists software development initiatives in a number of ways. It makes it possible to create complex projects, protects against software infiltration, lowers costs, boosts performance and quality, and increases productivity and dependability. External release numbers are assigned by marketing staff and begin with "Release 1.0," with following iterations designated as "Release 1.1," "Release 1.2," and so on. This numbering scheme makes bug localization procedures' communication easier.

Users may also find updates more quickly by connecting release tags and dates. Informed decisions regarding which software versions to employ and which are compatible with one another are made by end users as a result. A particular collection of hardware and software, such as an OS, VCS, build tools, application servers, web servers, scripting languages, bug tracking tools, programming languages, and databases are used to develop and test the proposed build system. The proposed build process as follows:

Operating System: Red-hot Linux and Windows are just two of the operating systems that are supported by the suggested build method. These operating systems offer a solid and dependable platform for the creation and use of software.

Version control: Tools for managing revisions such as CVS and SVN may be used in conjunction with the build method. Version control solutions help engineers collaborate effectively by enabling precise versioning and tracking of code changes.

Build Tools: Versatile build tools like Ant and Make are used during the build process. The compilation, packaging, and deployment of software components are all automated by these technologies. While Make is frequently used for C and C++ projects, Ant is particularly helpful for creating Java programme.

Web servers: Apache, a popular web server, is supported by the build process. Web applications may be hosted on a stable and scalable platform using Apache, making the deployment and maintenance of web-based projects simple.

Application Servers: The build process incorporates Tomcat, a well-known Java application server. Java-based web applications may be deployed and operated using Tomcat, which also offers a dependable runtime environment for server-side processing.

Web technologies are supported by the build process, including XML, HTML, DHTML, and style sheets. The development of dynamic and visually attractive web pages requires these technologies.

Scripting: The build process supports shell scripting using programmes like sed and awk, as well as scripting languages like JavaScript. These scripting languages enable the execution of personalised scripts for certain jobs and improve the construction process's automation capabilities.

The build procedure interfaces with Bugzilla, a well-known bug tracking programme. Software faults may be efficiently tracked, managed, and resolved with the help of Bugzilla, ensuring that problems are found and fixed in a methodical way.

Database: MySQL, a popular open-source database management system, is utilised during the construction process. MySQL is appropriate for a variety of applications since it offers dependable data storage and retrieval capabilities.

Programming Languages: C, C++, and Java are just a few of the many languages that may be used throughout the construction process. The presence of these languages, which are extensively used for software development, guarantees flexibility and compatibility with various project needs. Additionally, the build procedure notably connects with the J2EE framework, including technologies like the Struts framework for developing Java web applications.

Server technologies such as Active Directory or LDAP for authentication, file servers for file management, and DHCP (Dynamic Host setup Protocol) and Domain Naming System (DNS) for network setup are all smoothly integrated into the development process. By offering necessary services for efficient operation, these server components improve the software system's overall functionality and performance.

Software releases will now use a versioning system thanks to the suggested build procedure. The external version number is defined as "PROJID_MAJNO_MINNO_REVNO_BUILD_NO_TIMESTAMP". The project ID (PROJID), major number (MAJNO), minor number (MINNO), revision number (REVNO), build number (BUILD_NO), and timestamp (TIMESTAMP) are all included in this external version number. It acts as a distinctive identification for each software release.

The internal version number, on the other hand, is denoted by the notation "PROJID_MAJNO_MINNO_REVNO_BUILD_NO". Along with the project ID, it also contains the major number, minor number, revision number, and build number (BUILD_NO). The internal version number is mostly utilised throughout the software development and testing process to keep track of various programme iterations and updates.

The build process provides accurate identification and administration of software releases by differentiating between the external and internal version numbers. Users and stakeholders may easily identify and follow software upgrades thanks to the external version number's distinct and useful identification. On the other hand, the internal version number enables developers to organise and keep track of the many phases of software development, testing, and issue repair.

The major, minor, revision, build, and date, as well as the pID ("project ID"), are taken into consideration by the build numbering mechanism. The timestamp and internal version number are combined to create the external version number, which is represented by the string PID_MaN_MiN_RN_BN_TIMESTAMP. When there are major functional or technological changes, new software versions are generated, but new releases might happen when bugs are resolved. PID_MaN_MiN_RN_BN_TIMESTAMP.jar is one of the six fields in the external release tag, while PROJID_MAJNO_MINNO_REVNO_BUILD_NO is one of the five fields in the internal release tag. The code repository then receives a separate file with the release tag of the updated build.

On a file server, all publicly accessible *.jar files are saved in one location. The BN ("build number"), MiN ("minor number"), pID ("product ID"), and MaN ("major number") are taken into account while creating the release tag. The major number indicates more elements of the solution, whereas the minor number indicates slight alterations. The BN and RN both display the build version information.

Modern build processes stand out for being automated and flexible, enabling them to run immediately when new code is checked in. The project moves onto the tagging phase when all of the code has been produced, tested, and adjusted as necessary. There are specific guidelines for checking in, labelling cases, and checking out.

In order to improve software development, the suggested build process combines automation, flexibility, and labelling. In order to facilitate effective and dependable software development, it solves issues with version control, file source identification, and vulnerability abuse.

3.1 Check out

It is standard procedure to complete the check-out procedure before starting work at the neighbourhood workplace. The repository's server configuration should be ready. At check-out, fresh folders are created and filled with the pertinent source files. The repository's original source files can then be updated or modified by programmers. The following step is to confirm the repository's updated status.

3.2 Check in

Developers check their code in to ensure that it is consistent with everyone else's. However, it is often desirable to assess, test, and complete the programme before doing a check-in. The development procedure proceeds without a hitch since all contemporary developers are kept abreast of the most recent modifications.

A workaround is to construct a bug-fixed version utilising version 1.0 if the product version 1.3 is not trustworthy enough for distribution because of a serious problem introduced in version 1.0. The fixed problems must to be applied to crucial portions of the code to guarantee the security of next releases. Upgrading or merging is the process through which two separate development streams combine to create version 1.4.

3.3 Branching, Merging, and Tagging

The "version control systems" (VCS) aspect of branching enables developers to designate different development paths for their projects. A fork's modifications do not spread to the others. Branching enables the administration of distinct product versions, facilitating the distribution of updated and supplementary features. The method by which these branches are entirely assimilated into the main stem is known as "merging."

The VCS repository employs tags to build a new branch after each software release that will serve as a guide for future bug fixes and assessments of the effectiveness of the operations. This branch is created when there is a bug fix or a minor release. While the main trunk is used for upkeep and improvements, this branch is used to produce patches. After a branch has been made public, all modifications must be merged back into the mainline. This merging process might cause additional issues for the VCS, thus branch tags like "branch_name" and "merged" "branch_name" are added to stop merging the same branch twice.

Tags are the norm when distributing new software versions. Before it can be made public, all of the source code for the next release must be checked out with the proper tag. Unlike check-ins, tags can be modified and used as a benchmark for comparison. The releases for the build are saved on a central server and often take the form of build.jars.

3.4 Case-Based Reasoning of Bug Localization by Embedding Version Tags

The case study in this section demonstrates how to use version tags. The research shows the issues that came up when attempting to incorporate version metadata in the deliverable files prior to the release of the constructed programme. Learning how to obtain version data from the repository server's code at first proved to be challenging. Other challenges may be encountered when analysing data from output files like obj or class, finding dependencies for certain deliverables, and identifying the updated file list. Uncertainty around the proper version tag format and how it should be introduced hindered the process of coming up with a solution.

Developers frequently write code and then check it into the VCS. After the code is complete, the files must be tagged to prevent confusion between the old and new versions. The code is given the LIVE tag when it has been tested and any required adjustments have been made. The next region of growth is the trunk. The suggested construction method, which include embedding internal version tags, is described in the phases that follow.

Step 1: Clean out the local workspace of any unnecessary files or folders.

Step 2 is to retrieve the code from the version control system's repository.

Step 3: Extract the source files and the version details from the repository to create the lvf ("latest_versions_file").

The VCS repository's components each handle VCS files under corresponding folders that contain the file locations, names, and most recent versions of the files.

In order to include version information in each of the deliverable files, mapping between the source and the deliverable files is required. Get the repository's whole collection of file revisions.

Write the names of all the VCS files connected to the development module to the vef ("vcs_entries_file").

Step 4: Identify the files that have changed since the application's last build. For adding or updating version information in updated files, this is crucial.

To find out which files have changed since the last build, compare the ovf ("old_ver_file") and lvf. The difficulty in identifying updated files is involved here.

To keep track of file changes, use the dlf ("dependent_list_file"), which provides dependence information. If a file is changed, it should be added to the "modified_files_list" (mfl).

New versions of files should be added to the mfl with a version number of 0.0.

The source and output files for each Java file are identical to one another. However, when compiling collections of C++ or C files using many-to-one mapping, dependency information is utilised. The "internal version tag" (ivt) should be updated to reflect the change.

Step 6: Find the version of the code and check it in.

Create a new build by committing the modifications to the server.

Step 7: Run the automatic build that compiles all of the modules.

Include rt ("release_tag") checks in the automated build process. The rt is made up of the MiN, BN, and. A timestamp will only be added to.tar or.jar files that have been made accessible for analysis, quality assurance, or production.

Utilise programmes like Ant, which may provide results in C++, Java, or even C. These software control log files, check-ins, branches, VCS merges, check-outs, tags, and data used for troubleshooting.

Make sure the build includes all recently updated files.

Step 8: Add the MiN, BN, RN, MaN, as well as timestamp to the final deliverable.

Step 9: Log the troubleshooting details in the lfbn ("log_file_build_no") if the build fails at any step.

To improve the product in terms of qualitative and quantitative characteristics, more investigations were made. An better change control method was produced by combining the suggested build process with the updating of the "internal version tag" for each deliverable file [5]. Figure 1 provides an illustration of this integration.

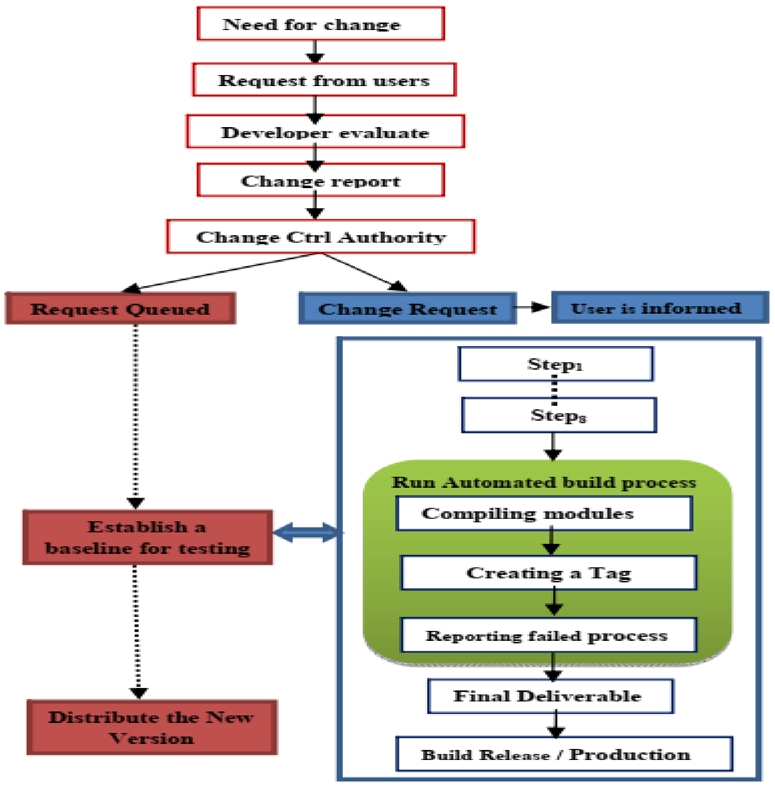

Fig. 1: Include or update the IVT to Deliverable file in the build procedure that has been recommended.

We investigated the association between the prevalence of defects during the build process and their subsequent correction using the suggested method, increased Bug Localization via Version Tag Embedding (EBL-VTE) [16], [17] employing the increased traceability feature. Figure 2 displays the number of reported issues both before and after the release. The suggested method offers a number of benefits, including a lower defect failure rate, improved release management and build processes, the ability to build more complex products, protection from software invasion, improved quality [18], [19], [20] improved performance, lower costs, higher reliability, and global scalability. To ensure effective build releases and software product development, it makes use of an incremental approach connected to the spiral technique. In order to produce a product that fully satisfies the market's expectations, incremental delivery is the best course of action. As the system gains knowledge from customer input, its fault tolerance rises.

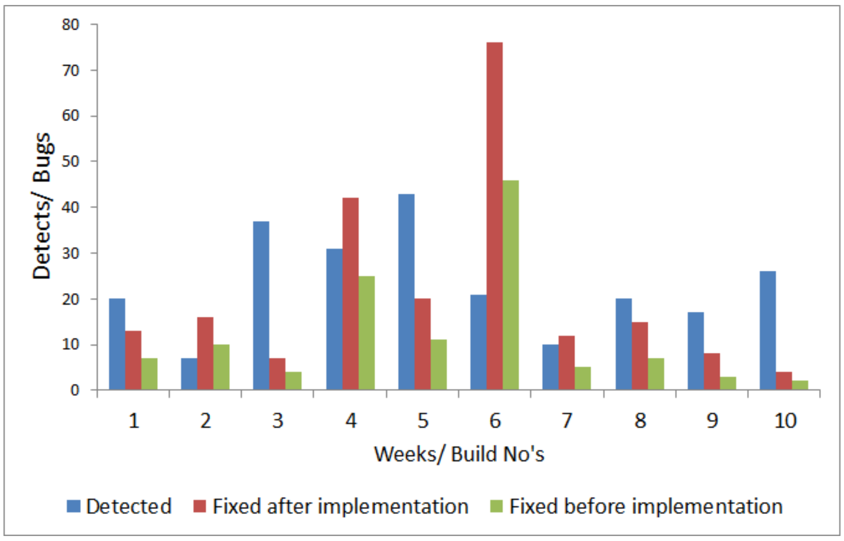

Fig. 2: Resolved bugs versus version numbers

The quantity of reported flaws that were fixed before and after a certain version's release is also shown in Figure 3.

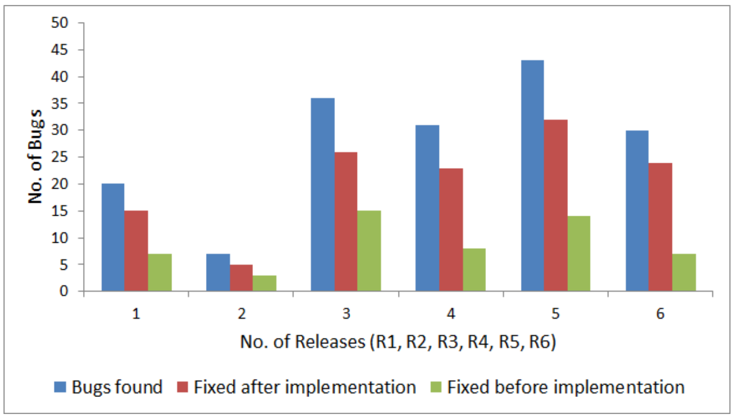

Fig. 3: Bugs discovered and fixed status in relation to fresh releases released

Additionally, Figure 4 highlights the relationship between the number of builds and the state of issues that have been found and addressed across various versions.

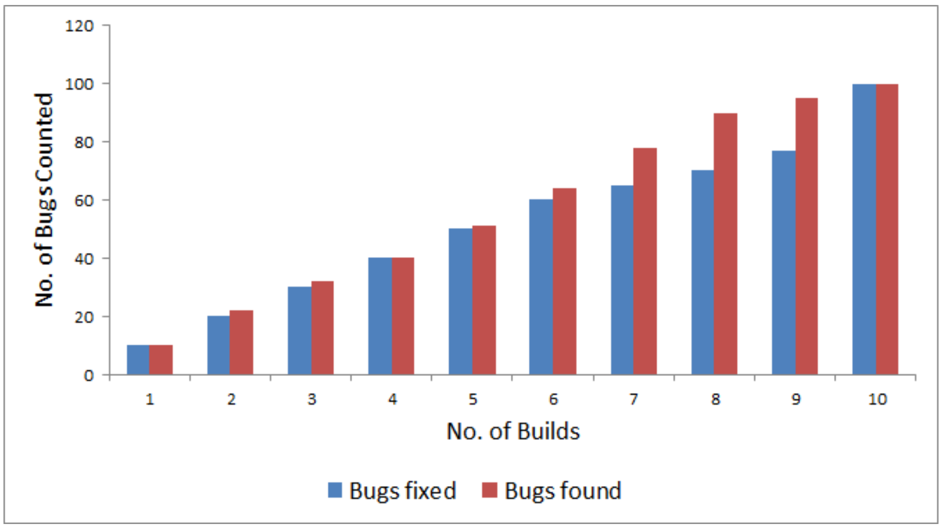

Fig. 4: Status of identified and repaired bugs in relation to the number of builds

4. Experimentation and Assessment of Results

In this section, an experimental study is given to assess the effectiveness and longevity of the Enhanced Bug Localization by Version Tag Embedding (EBL-VTE) approach. The study's statistics, which are displayed in Table 1, were compiled from a number of sources, including [21], [22], [23]. A performance evaluation of the suggested EBL-VTE technique was conducted utilising Natural as well as Programming Language Processing using Convolutional Neural Networks (NP-CNN) [15], a cutting-edge bug localization methodology. For this investigation, performance measures were hit rate (HT), information retrieval accuracy (IRA) using Mean Reciprocal Index (MRI), and mean best fit (MBF).

4.1 Input Data

The input data consisted of a set of source files tagged with version tags specified by EBL-VTE. To decide which source files should be marked with which version numbers, bug reports were used. A compilation of the defects from the PDE, Platform, and JDT bug tracking systems resulted in a total of 6267, 3900, and 3954 complaints. There were, respectively, 7153, 2319, and 3696 files of source code for the PDE, Platform, and JDT projects (Table 1). According to the issue reports, the vocabulary sizes for the JDT, PDE, and Platform projects were 4304, 2964, and 3677, respectively. The source files connected to each bug report were grouped together and sorted according to how closely they related to the report.

Table 1: Statistics for the sets of data

Data Set	bug reports	Correlated sources
JDT	6267	7153
PDE	3900	2319
Platform	3954	3696

4.2 Performance Measures

Indicators of performance such as hit rate, mean reciprocal index (MRI), and mean best fit (MBF) were employed to assess the effectiveness of EBL-VTE. The hit rate is the proportion of actual correct sources to the projected number of accurate sources for a certain target strategy (Eq. 1). The average location of the first proposed source inside the ordered collection of genuine sources is evaluated by mean best fit (Eq. 3), whereas the average reciprocal rank of the suggested sources is measured by mean reciprocal index (Eq. 2).

Eq. 1: For rs_i s_i, hr = (1/|D|) * _(i=1)(|D|) (|rs_i|/|s_i|) // In this equation, the hit rate is denoted by hr, the total number of bug reports is denoted by |D|, the set of sources proposed by the bug localization technique for the i-th bug report is denoted by rs_i, and the set of sources actually linked with the i-th bug report is denoted by s_i. The formula determines the average difference between the size of the real sources set and the size of the consequent sources set for each bug report, divides that value by the total number of problem reports, and returns the result.

Eq. 2 states that MRI = (1/|D|) * _(i=1) _(j=1) _(|rs_i|) 1/(index_of(r_j, s_i)) //In this equation, MRI stands for the mean reciprocal index, |D| for the total number of bug reports, index_of(r_j, s_i) for the index of source j in the ordered set s_i of actual sources, and rs_i for the set of sources that the bug localization method recommends for report i. For each bug report included in set D, the equation determines the average reciprocal value of the monitored sources' index positions among the resulting sources.

Eq. 3: Where index_of(r_1, s_i) exists for r_1 in rs_i, MBF = (1/|D|) * _(i=1)(|D|) index_of(r_1, s_i);In this equation, MBF stands for the mean best fit, |D| for the total number of bug reports, index_of(r_1, s_i) for the index of the first source in the ordered set of actual sources s_i, and rs_i for the set of sources that the bug localization method recommends as a result for the i-th bug report. For each bug report included in set D, the equation determines the average index position of the first monitored source among the resulting sources.

4.3 Statistical Analysis

Table 2 displays the experimental findings that were attained by the bagging procedure and 10-fold cross-validation. The statistics show the EBL-VTE model's promising performance in bug localization when compared to the modern NP-CNN approach.

Table 2: EBL-VTE as well as NP-CNN Performance Statistics Comparison

DATASET	JDT		PDE		PLATFORM	
METHOD	NP-CNN	EBL-VTE	NP-CNN	EBL-VTE	NP-CNN	EBL-VTE
HIT RATE	0.796 ± 0.003	0.882 ± 0.01	0.727 ± 0.006	0.863 ± 0.012	0.75 ± 0.008	0.839 ± 0.009
MRI	0.074 ± 0.003	0.096 ± 0.009	0.067 ± 0.004`	0.081 ± 0.01	0.015 ± 0.005	0.014 ± 0.005
MBF	7.043 ± 0.129	4.471 ± 0.199	6.481 ± 0.188	4.396 ± 0.151	16.94 ± 2.921	9.747 ± 0.401

On the JDT, PDE, and PLATFORM datasets, the EBL-VTE (bug localization by embedding version tag) strategy outperformed the NP-CNN (natural and programming language processing and convolutional neural networks) model in terms of hit rate. For the JDT, PDE, and PLATFORM datasets, respectively, the EBL-VTE produced hit rates that were 8.5%, 6.9%, and 8.4% higher than NP-CNN (see Figure 5). This suggests that, in bug reports, the EBL-VTE approach had a better likelihood of correctly choosing the right sources for the target method.

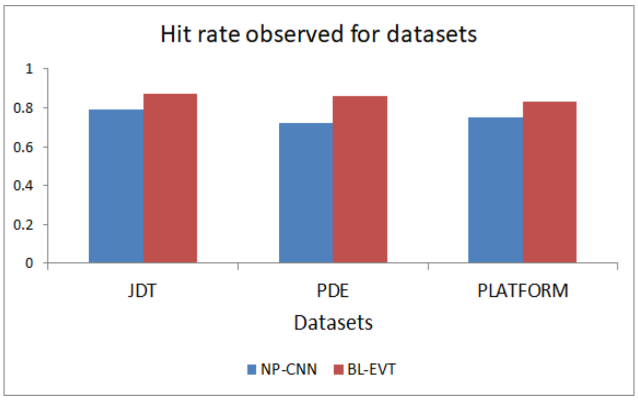

Fig. 5: Hit rate for the JDT, PDE, and LATFORM datasets was observed

Similarly, the EBL-VTE approach showed superior performance over NP-CNN when measuring the Mean Reciprocal Index (MRI). According to Figure 6, the MRI obtained by the EBL-VTE was 2% greater than that of the NP-CNN. The bug localization technique's correct sources' average reciprocal rank, measured by the MRI metric, shows that the EBL-VTE approach offered more accurate and pertinent suggestions.

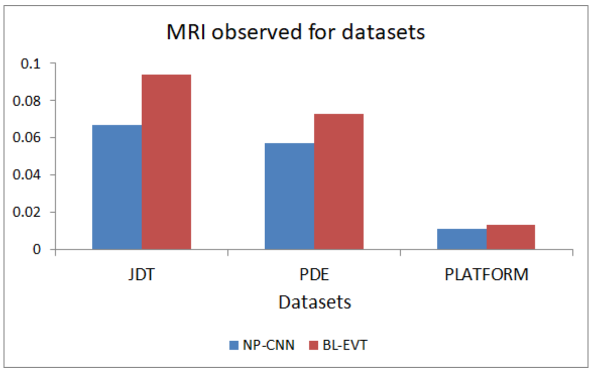

Fig. 6: MRI findings for the datasets JDT, PDE, and PLATFORM

The data collected for the mean best fit (MBF) measure likewise demonstrated the EBL-VTE method's significant performance advantage. Lower values in the mean best fit measure indicate a better fit between the suggested sources and the actual sources. The average mean best fit for the EBL-VTE technique was 4.5, which was around 3 indices lower than the average mean best fit for the NP-CNN approach of 7.5 (see Figure 7). This shows that in terms of precisely matching the proposed sources with the actual sources, the EBL-VTE technique fared better than NP-CNN.

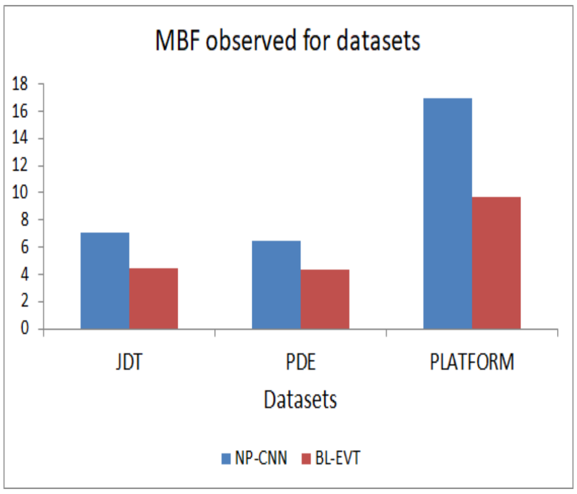

Fig. 7: MBF detected for the datasets from JDT, PDE, and PLATFORM

Overall, the experimental findings show that, when compared to the NP-CNN model, the EBL-VTE technique performed better in terms of hit rate, mean reciprocal index, and mean best fit. The usefulness and benefits of the EBL-VTE technique in precisely localising issues and offering more pertinent source code recommendations are supported by these findings.

5. Conclusions and Future Work

In conclusion, the suggested EBL-VTE (bug localization by embedding version tag) technique has significantly outperformed the current NP-CNN (natural and programming language processing and convolutional neural networks) model in terms of performance. The experimental investigation on the JDT, PDE, and PLATFORM datasets has revealed important details about the efficacy and stability of EBL-VTE.

The first finding from the hit rate measure was that EBL-VTE had greater odds of correctly choosing the proper sources for the target technique in bug reports. With an increase in hit rate of 8.5%, 6.9%, and 8.4% for the JDT, PDE, and PLATFORM datasets, respectively, the technique surpassed NP-CNN. This demonstrates how effective EBL-VTE is in recommending pertinent sources.

Second, compared to NP-CNN, EBL-VTE offered more accurate and pertinent suggestions, according to the Mean Reciprocal Index (MRI) measure. The MRI obtained by EBL-VTE was around 2% higher than that of NP-CNN, suggesting that the accurate sources indicated by the bug localization approach had a higher average reciprocal rank.

The mean best fit (MBF) statistic also demonstrated EBL-VTE's greater performance in precisely matching suggested sources with real sources. In comparison to the average mean best fit of 7.5 for NP-CNN, the method's average mean best fit was 4.5, which is a difference of almost 3 indices. This highlights how much better the EBL-VTE's source code suggestions fit and are relevant.

Overall, the experimental findings show that, in terms of bug localization relevance and accuracy, EBL-VTE outperforms NP-CNN. The suggested strategy significantly improved the software development and problem fixing processes by precisely identifying and proposing the appropriate sources for issue solutions.The results of this study have significant ramifications for researchers and software developers working in the bug localization sector. The EBL-VTE approach may be used to increase the speed and efficacy of bug correction, resulting in software products of greater quality. Developers may more accurately locate and fix issues by using the suggested localization technique and incorporating version tags in software deliverable files. As a consequence, defect failure rates are decreased and software dependability is increased.

In conclusion, the experimental investigation and performance assessment have shown that EBL-VTE outperforms NP-CNN in the localisation of bugs. The suggested approach offers a useful approach for precisely locating and repairing issues, enhancing the calibre of software, and eventually enhancing software development procedures. The EBL-VTE approach may be further investigated and improved in future studies, taking into account its potential uses in a variety of software development settings.

References

[1] Rao, N. R., & Sekharaiah, K. C. (2015). Embedding version tag in software file deliverables before build release. 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 1–6. https://doi.org/10.1109/ICRITO.2015.7359255

[2] Hamdy, A., & Arabi, A. E. (2022). Locating faulty source code files to fix bug reports: International Journal of Open Source Software and Processes, 13(1), 1–15. https://doi.org/10.4018/IJOSSP.308791

[3] Liu, G., Lu, Y., Shi, K., Chang, J., & Wei, X. (2019). Mapping bug reports to relevant source code files based on the vector space model and word embedding. IEEE Access, 7, 78870–78881. https://doi.org/10.1109/ACCESS.2019.2922686

[4] Bendix, L., Kojo, T., & Magnusson, J. (2011, August). Software configuration management issues with industrial opensourcing. In 2011 IEEE Sixth International Conference on Global Software Engineering Workshop (pp. 85-89). IEEE. https://doi.org/10.1109/ICGSE-W.2011.21

[5] Neville-Neil, G. V. (2009). Kode viciousSystem changes and side effects. Communications of the ACM, 52(4), 25–26. https://doi.org/10.1145/1498765.1498777

[6] Lekha Tummala,Hruthik Gavva,.Maanvitha Gona, Lakshmi Tulasi.P (2021).Virtual Controller: managing a remote computer using network communication. International Journal of Computer Engineering In Research Trends. 8(12), 216-219,

[7] Walrad, C., & Strom, D. (2002). The importance of branching models in SCM. Computer, 35(9), 31–38. https://doi.org/10.1109/MC.2002.1033025.

[8] P. Siva (2022). Prediction of Knee Osteoarthritis Using Deep Learning. International Journal of Computer Engineering in Research Trends. 8(12), 228-235.

[9] Lai, R., Garg, M., Kapur, P. K., & Liu, S. (2011). A study of when to release a software product from the perspective of software reliability models. Journal of Software, 6(4), 651–661. https://doi.org/10.4304/jsw.6.4.651-661

[10] Rao, N. R., & Sekharaiah, K. C. (2013). An incremental risk management framework for realizing project efficiency using version control. In CCSN and IJCA India 2013 (pp. 1-6), https://research.ijcaonline.org/ccsn2013/number3/ccsn1301.pdf.

[11] Neely, S., & Stolt, S. (2013, August). Continuous delivery? easy! just change everything (well, maybe it is not that easy). In 2013 Agile Conference (pp. 121-128). IEEE, DOI: 10.1109/AGILE.2013.17.

[12] Elbaz, M. (2011, August). To deliver faster, build it in reverse. In 2011 Agile Conference (pp. 230-233). IEEE, DOI: 10.1109/AGILE.2011.32.

[13] Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering, 31(4), 340–355. https://doi.org/10.1109/TSE.2005.49

[14] McIntosh, S., Adams, B., & Hassan, A. E. (2010, May). The evolution of ANT build systems. In 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (pp. 42-51). IEEE, DOI: 10.1109/MSR.2010.5463341.

[15] Huo, X., Li, M., & Zhou, Z. H. (2016, July). Learning unified features from natural and programming languages for locating buggy source code. In IJCAI (Vol. 16, pp. 1606-1612), http://129.211.169.156/publication/ijcai16npCNN.pdf.

[16] Kim, D. Y., & Youn, C. (2010, June). Traceability Enhancement Technique through the integration of software configuration management and individual working environment. In 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement (pp. 163-172). IEEE, DOI: 10.1109/SSIRI.2010.27.

[17] Rudra Kumar, M., Pathak, R., Gunjan, V.K. (2022). Diagnosis and Medicine Prediction for COVID-19 Using Machine Learning Approach. In: Kumar, A., Zurada, J.M., Gunjan, V.K., Balasubramanian, R. (eds) Computational Intelligence in Machine Learning. Lecture Notes in Electrical Engineering, vol 834. Springer, Singapore. https://doi.org/10.1007/978-981-16-8484-5_10

[18] Rudra Kumar, M., Pathak, R., Gunjan, V.K. (2022). Machine Learning-Based Project Resource Allocation Fitment Analysis System (ML-PRAFS). In: Kumar, A., Zurada, J.M., Gunjan, V.K., Balasubramanian, R. (eds) Computational Intelligence in Machine Learning. Lecture Notes in Electrical Engineering, vol 834. Springer, Singapore. https://doi.org/10.1007/978-981-16-8484-5_1

[19] Pingili, Madhavi & Sreenivasulu, K. & Maloth, Bhav Singh & Saheb, Shaik & Saleh, Alaa. (2022). Bug2 algorithm-based data fusion using mobile element for IoT-enabled wireless sensor networks. Measurement: Sensors. 24. 100548. 10.1016/j.measen.2022.100548.

[20] M. M. Venkata Chalapathi, M. Rudra Kumar, Neeraj Sharma, S. Shitharth, "Ensemble Learning by High-Dimensional Acoustic Features for Emotion Recognition from Speech Audio Signal", Security and Communication Networks, vol. 2022, Article ID 8777026, 10 pages, 2022. https://doi.org/10.1155/2022/8777026

[21] Ramana, Kadiyala, et al. "Leaf disease classification in smart agriculture using deep neural network architecture and IoT." *Journal of Circuits, Systems and Computers* 31.15 (2022): 2240004. https://doi.org/10.1142/S0218126622400047

[22] Bugzilla main page. (n.d.). Retrieved 13 May 2023, from https://bugs.eclipse.org/bugs/

[23] http://git.eclipse.org/, https://github.com/eclipse/.