Volume 9, Issue 11, November – 2024 International Journal of Innovative Science and Research Technology
ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV177

Early Bug Detection through Shift Left Testing

Ashwin Venkitaraman
Independent Researcher
Dept. of Electrical Engineering
Fremont, CA – 94536, USA

Abstract:- Shift-Left Testing is a preventive approach in the SW development process of identifying and handling defects where testing is performed before the flow proceeds to the subsequent phases of SDLC. In most situations, testing is done after development, and this means that any defects get discovered later contributing to high costs and more time to complete the project. Essentially, Shift-Left Testing implies that testing should be conducted during the design or the coding stage and is beneficial due to the fact that in those stages of development, it is considerably less expensive to rectify problems that are detected. It uses integrated strategies including continuous integration, static code analysis and automated testing, in which the development and the test team work together from the start. Consequently, the approach results in enhanced quality of the software, their development time, and minimization of the post-release faults. Although Shift-Left Testing is changing many ways in software development for the better, it has some problems, for instance, changing organizational culture and has high demands to test automation frameworks.

Keywords:- *Shift-Level Testing, Early Bug Detection, Software Quality Assurance, Test Automation, Software Development Lifecycle (SDLC).*

I. INTRODUCTION

Shift-left testing is a preventive approach in which bug detection is done at the initial stage of software development as a way of enhancing the quality of the software. The software development model that specified the testing phase after the development phase is not efficient because the bugs are detected late, therefore, the correction of the bugs is costly and time-consuming. Shift-left testing, as the name implies, means moving testing activities up or to the 'left' on the V-model of software development life cycle. This approach focuses on regular testing, with the idea everyone who is involved in the development of an application, from developers to testers, is involved from the beginning, with the aim of avoiding bugs rather than dealing with them when they occur.

Shift left testing is basically aimed at identifying a problem at a level where it is still simple and cheap to rectify the same. Such problems associated with the logic, functionality and performance of a code can be identified much early should the code be tested during the development process. Some of the common practices that are used to ensure that the software is fine from the early stages include; unit testing, integration testing and the code analysis. Automation is also involved in this approach where developers are able to run tests consistently on the build process.

This is because the testing starts early in the development phase and is done continuously; this helps in minimizing the number of bugs that get to the ultimate stages of the project such as user acceptance testing or production phase. It also allows for quicker feedback loops to be provided to the developers so that they may solve problems as they arise without having to escalate. Testing performed during the development phase improves the reliability of the software and therefore improves on user satisfaction and reduces on the maintenance costs in the future. Therefore, shift-left testing as an early bug detection technique is a beneficial approach that helps to produce excellent-quality software quickly while reducing the dangers and delivery time of products.

A. Benefits of Shift-Left Testing for Early Bug Detection

The benefits of shift-left testing for early bug detection are substantial and play a crucial role in improving the overall software development lifecycle. Some of the key advantages include:

- *Reduced Cost of Bug Fixes*

It is also important to note that it will cost a developer a significantly higher amount of resources to fix bugs that are located deeper into the code pipeline. Shift left testing prevents bugs getting detected later at a later stage in the development hence saves effort that otherwise would have been spent on rectifying slips made during development, thus saves costs.

- *Improved Software Quality*

For complicated programs, early bug detection makes sure that the code base is subjected to error check as early as possible. This leads to the creation of higher quality of the software that has fewer vital problems signifying themselves in a later stage of the testing or even when the software is released to be used.

- *Faster Time-to-Market*

Correcting defects as early as possible decreases the number of working hours to be spent for restoring bugs in the future. Therefore, errors are detected early and the development process is improved so as to facilitate faster delivery of the product without being held up by major problems at the eleventh hour.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV177

- *Increased Developer Productivity*

When testing is incorporated during the initial development stages, it gives the developers quicker feedback, because they correct them at that stage. This makes the development process continuous and does not allow for the tendency of having to look for bugs at a later stage in the development cycle.

- *Enhanced Collaboration between Teams*

Shift left testing implies that testing must be carried out while developers are in the initial stage of the development process. Another test automation approach is shift-left testing, it means testing should start at the early stages of the development process involving the cooperation of the development and testing teams.

- *Better Risk Management*

Shift-left testing enables one to address possible risks created by bugs that might degenerate upon realization at a later stage. It also has the advantage of averting some of the disruptions in cycle of development and enhances the general stability of the projects.

- *Continuous Improvement*

The concept of early testing leads to a test feedback loop that allows code to be enhanced as the development process continues. This helps to check that successive versions of the product are better consolidated, more reliable and closer to expectations of the user and the business.

Shift-left testing is a proactive strategy that integrates testing into the earlier phases of development, leading to better software quality, faster releases, and reduced costs.

B. Need of the Study

This work is necessary because the contemporary software systems are becoming more sophisticated, and the requirements for creating stable and efficient applications are rising. As the software projects have grown, the cost and effort of rectifying the bugs at the later stages of the development or even post-release has become more expensive and cumbersome as compared to the bugs which are found at the initial stages. This has prompted the shift-left testing approach that focuses on the early identification of bugs in the development cycle. The frequency at which new software releases are made possible by agility and automated pipelines has increased the pressure for effective and preventive testing. It is, therefore, important to note that quality assurance methodologies that are practiced in the traditional approach where quality assurance is conducted after development are inadequate to cope with these cycles. Thus, it is essential to discuss such approaches as shift-left testing, which represents the idea of testing from the beginning and continuously. This research is not only going to benefit software quality, but it is also going to offer an efficient way of cutting down the costs of development and shortening the time it takes to get the software to the market while at the same time increasing the satisfaction level of the end users through the delivery of more effective software products.

II. LITERATURE REVIEW

Li, Z., Tan, L., Wang, et al (2006). Over the years, bugs in the contemporary OSS have acquired certain features mainly as a result of the new trends in software development, complexity of the code and the open-source communities. Nowadays, almost all the open-source projects use continuous integration, testing, and issue tracking systems that allow them to manage bugs more efficiently. Still, some of the problems remain the same which include security issues, memory issues, and performance issues. One of the primary challenges of OSS development is that it is decentralized which can result in incongruity of bug report and time taken for their resolution. Research done on this shows that nature and type of bugs are affected by the size of the code and the activity level of the coders. Besides, bug identification and reporting in OSS is done by a larger group of users with different levels of experience making bugs in OSS to have different characteristics compared to bugs in proprietary software.

Regehr, J., Chen, Y., et al (2012). Test-case reduction for C compiler bugs is an important in the debugging process that is used for making the test case as small as possible but still containing some behavior that causes the failure. If a bug is found in C compiler, the first test case may be large and complicated and the cause of the bug may be hard to find. By systematically reducing the test case, developers can easily identify the real conditions that could lead to the bugs, which in turn eases identification and resolution of the issue at hand. Delta debugging or other similar methods or tools like “C-Reduce” do this automatically by constantly eliminating unrelated sections of the code while ensuring that the code contains only the parts necessary to reproduce the bug. Test-case reduction is crucial for compiler bugs as compilers are exposed to various inputs and optimization techniques making bug reproduction in such a setting rather difficult. A smaller test case can also help in developers’ interaction and provide a faster bug fix. Regression test suites are created by adding reduced test cases to the original test suites so that the same bug does not revisit the subsequent versions of the compiler. It should be realized that, test-case reduction improves the effectiveness of the debugging process, which in turn improves the stability and efficiency of the compiler.

Pan, K., Kim, S., & Whitehead, E. J. (2009). Recognizing bug fix patterns is crucial for optimization of software maintenance and for amplification of development activities. Bug fix patterns can be described as the patterns of how developers address the problem of defects in the code. These patterns may differ depending on the type of bug to be fixed, the programming language used and the project under development. Research indicates that the majority of bugs fixed fall under a few major types, including logical problems, memory related problems, and syntax problems. Some of the changes that can be seen in bug fix patterns include changes in the structure of the code as well as changes in the variables that are assigned to different values and even changes that may lead to the introduction of new algorithms. This analysis helps the developers to predict which kind of

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV177

bugs is likely to occur in similar situations so that it can be prevented.

Williams, C. C., & Hollingsworth, J. K. (2005). Source code repositories mining is a useful approach that augments bug searching techniques by employing historical data that could be available in repositories such as GitHub, GitLab or Bitbucket. These sources consist of valuable data such as commit histories, bug reports, code changes and developers' discussions that can be helpful to understand bug patterns and code evolution. Through proper use of machine learning and data mining, one is able to discover patterns of where bugs frequently occur, what types of bugs are usually found and how bug correction develops over time. Exploring these repositories makes it possible to identify code smells, security flaws, and places with high bug density that explain high testing or refactoring priority. This approach can help build models that will predict the future bugs based on patterns on the past defects and it will be possible to prevent things that may hinder the proper running of the software. Automatic mining also contributes to the development of both static and dynamic analysis tools as they are given actual bug data to enhance the former's capacity to detect even more intricate and sophisticated bugs. Through continual learning from vast amounts of code and bug fix data, these techniques can improve the effectiveness of today's advanced bug finding tools and the resulting software systems.

Yang, X., Chen, Y., Eide, E., et al (2011). Debugging bugs in C compilers is intricate since the inputs that compilers come across are vast and are rather complex. The process of translation of high level human-readable C code into the machine-readable code is done with the help of C compilers that face multiple challenges in terms of the language features, optimization levels, and hardware architectures. Compiler bugs may result in a variety of scenarios such as the wrong code being generated, a program halting or slowing down. These bugs, in general, are detected with the help of various testing methods, including differential testing, which implies the comparison of the results of the same code compilation with different compilers or with different options.

Bader, J., Scott, A., et al (2019). Automated bug fixing is a new promising field that aims at using machine learning and artificial intelligence to correct faults in the software. It is a technique which aims at decreasing the time that developers spend on debugging by delivering to them the most helpful information for bug identification, analysis and resolution. Automated bug fixing is usually performed based on historical data of software repositories, bug reports, code modifications, and commit histories to learn common patterns and developers' solutions. With these data, models can recommend or even execute fixes with similarities to the bugs seen earlier. One way can be for example to apply the neural networks or genetic algorithms to create patches for the faulty code.

Nama, Prathyusha (2023). presents a comprehensive analysis of how artificial intelligence (AI) is transforming user interaction in mobile applications through intelligent features and context-aware services. The research highlights how AI-powered mobile apps leverage machine learning algorithms to understand user behavior, provide personalized recommendations, and deliver intelligent, context-aware experiences. The study emphasizes the integration of various AI-driven functionalities, including voice recognition, natural language processing, and predictive analytics, which collectively enhance user engagement and satisfaction while improving accessibility for diverse user groups.

The research also addresses critical implementation challenges, particularly regarding privacy and data security in AI deployment. Through analysis of successful case studies and emerging trends, Nama demonstrates how context-aware services enable applications to respond dynamically to users' environments and situations, leveraging data about location, time, and activity to deliver tailored services. The findings suggest that while AI is revolutionizing mobile user interaction through features like content generation, digital assistance, and predictive analytics, developers must carefully balance personalization with ethical considerations regarding user privacy. The study concludes that AI-powered mobile applications represent a significant advancement in user interaction, making experiences more intuitive and personalized while emphasizing the need for continued research in areas such as Edge AI and conversational interfaces.

III. METHODOLOGY

The study shall use survey research with a mix of both quantitative and qualitative research in establishing customer satisfaction with internet banking services across the different demographic segments. This combination enables the researcher to understand the research problem in depth as it combines numerical data with the participants' perception of the problem. The quantitative aspect will assist in determining the level of satisfaction of customers while the qualitative will offer more information in relation to the perception and experiences of customers. The target population comprises the people aged between 18-60 years who engage in online banking. A purposive sampling technique of stratified random sampling will be used so as to capture participants from different generations, occupation, and income status. It will be easier to ensure that the findings come from different point of views hence increasing the generality of the results. The survey sample will be 300 respondents, whereas 20 in-depth interviews will be used to gain more insights into the respondents' experiences in their own words.

Keeping this in mind the primary data will be collected from customers via online survey which will have pre-constructed questions for measuring the satisfaction level of the customers about internet banking and factors that affect their usage of internet banking services. Besides the aforesaid surveys, exploratory will be conducted to get more details on certain aspects of customer experience that may not

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV177

necessarily be reflected in quantitative statistics. Secondary data shall be gathered from journals, previous researches, and reports from the Indian banking sector, which will give wider perspective about the study. Informed consent will be sought from all participants in the study, so that the participants are fully aware of the nature of research to be conducted. To ensure the respondents' confidentiality and privacy, their responses will also not be disclosed. In order to conduct the research in an ethical manner, ethical clearances will be sought from the ethical committee of the institution.

IV. RESEARCH PROBLEM

Main research problem in this research is centered on the continuous difficulty of detecting and eradicating software bugs at the early stage of development. Even with agile development methodologies in place and the focus on delivering software more frequently, it remains a challenge for organizations to identify defects early in the development life cycle, thereby suffering from extra development cycles, longer time to release and poor quality software products. Most of the conventional testing techniques that are practiced after the development phase of a software also report bugs at a later stage which increases problems such as time overrun, increased costs, and unsatisfied customers. Specifically, as the size of software systems increases there is a higher demand for early detection of bugs in order to avoid issues of growth. As much as shift-left testing has provided a solution to this challenge, little is still known regarding how to apply it in a variety of development settings and how it generates a positive impact on the quality of software and its delivery time. This research problem therefore aims to identify the techniques, methods, and approaches to adopt for the incorporation of early testing into the development life cycle processes with the view of optimising the shift-left testing initiative. This is an important factor that can enable movement from a model of continuously patching flawed software to one where software developers design quality solutions from the onset.

V. RESULTS

Table 1: Comparison of Bug Detection Phases in Traditional Testing vs. Shift-Left Testing

Testing Phases	Traditional Testing (Bugs Detected)	Shift-Left Testing (Bugs Detected)
Requirements Gathering	2%	15%
Design	5%	20%
Development	10%	25%
Integration Testing	20%	25%
System Testing	30%	10%
User Acceptance Testing	20%	3%
Post-Deployment	13%	2%

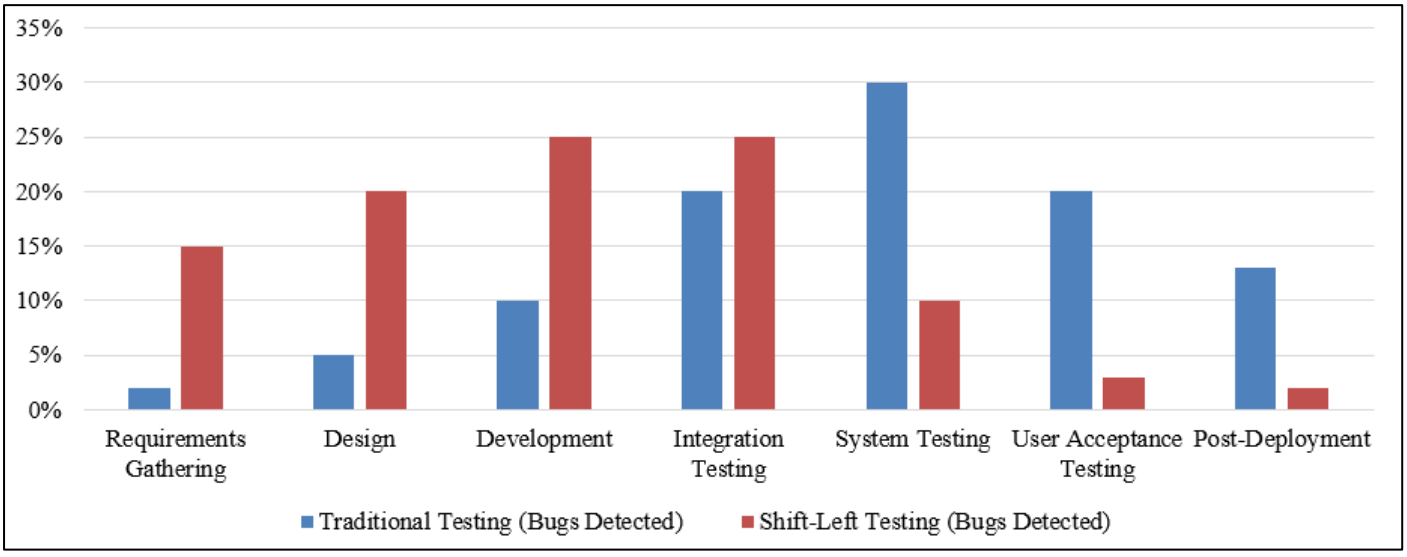

Fig 1:

The table shows the effectiveness of traditional testing and shift-left testing of bugs in various stages of SDLC. In the traditional software testing approach, only 2% of the defects are identified during the requirements gathering phase, while in shift-left testing, 15% of the total defects are identified which greatly emphasizes on the early phase of testing. In design phase, the traditional testing identifies only 5% of bug and shift-left testing identifies only 20% of bugs which shows that shift-left testing is more effective in comparison to a traditional way of testing. Likewise, during development, shift left testing offers 25% of bugs and 10% of traditional testing, which emphasis the testing during the coding.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV177

While in the integration testing phase, both detect about 25% of the bugs, the difference is evident in other phases. It is found that 30 % of bugs are detected in system testing while shift left testing identifies them at 10 % as many are eliminated. Traditional user acceptance testing contributes to bug detection at 20% while shift-left contributes at 3% thus making the user experience more seamless. The percentage of bugs detected after the shift-left testing is also considerably smaller, 2% in shift-left testing and 13% in the conventional testing, thus minimizing the need to correct issues that have made it into the production versions. In all, shift-left testing is better placed to detect bugs early hence minimizing the risks and costs that are incurred in bug fixes at later stages.

Table 2: Cost of Bug Fixing Across Development Stages (in USD)

Development Stage	Cost in Traditional Testing	Cost in Shift-Left Testing
Requirements Gathering	$500	$100
Design	$1,000	$250
Development	$3,000	$1,500
Integration Testing	$5,000	$2,000
System Testing	$7,000	$3,000
User Acceptance Testing	$10,000	$5,000
Post-Deployment	$20,000	$8,000

The table shows how much it would cost to adopt shift left testing against traditional testing at each phase of the development process. During the requirements gathering phase, the cost of handling problems in traditional testing costs $500 while shift-left testing costs only $100; proving that it is cheaper to test earlier. Likewise, in the design phase, conventional testing costs $1, 000 but shift-left testing costs $250 as it detects defects right from the onset.

And as the development increases, the difference in the cost between the two becomes more significant. During the development phase, traditional testing takes $3,000 while shift left testing takes $1,500 in the same phase, proving the advantage of testing as it progresses with coding. During integration testing it costs $5,000 in traditional testing while it costs $2,000 in shift-left testing because issues are detected early hence not complicated to solve. In the traditional method of testing, system testing cost $7,000, while the user acceptance test cost $10,000 but using the shift-left testing, the cost is reduced to $3,000 cost for system testing and $5,000 for user acceptance testing. The largest gap is observed in post-deployment where the traditional approach costs $20,000 while shift-left testing costs $8,000. This table shows reduced cost impact of shift-left testing as most of the root causes of costly rework and post-deployment fixes are caught and fixed early.

Table 3: Comparative Results Table for Shift-Left vs. Traditional Testing

Aspect	Traditional Testing	Shift-Left Testing
Bug Detection Rate	Lower in initial phases, higher post-development	Higher during initial phases, lower post-development
Testing Involvement	Testing begins after development completion	Testing begins during the design and coding phases
Developer Productivity	Lower (due to frequent context switching and late rework)	Higher (focused bug fixing during coding stages)
Test Coverage	Limited coverage, primarily functional testing	Broader coverage, including unit, integration, and static analysis
Time Spent on Debugging	More time spent on debugging after development	Less time, as bugs are caught during development
Impact on Release Deadlines	Frequently causes delays due to late-stage defects	Fewer delays as most bugs are detected early
Code Quality	Lower (due to deferred testing and bug accumulation)	Higher (continuous testing improves code quality)
Maintenance Costs	Higher, with frequent post-release patches	Lower, as fewer defects remain post-release
Error Propagation	Higher, as bugs go unnoticed until later stages	Lower, as early detection prevents error propagation
Team Communication	Less cross-team interaction, isolated roles	High collaboration between development and QA teams
Test Data Creation	Test data created at later stages, often delayed	Test data prepared early in the development process
Risk of Regression Bugs	Higher due to late testing	Lower, as early testing helps in maintaining system stability
Security Issues Identification	Identified post-development	Identified during code development through static analysis and code review

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV177

VI. CONCLUSION

Shift-left testing can be said to be an important strategy in solving the problems that arise in the current software development processes due to early identification of bugs. It is much more effective to incorporate testing activities into the early stages of the development life cycle as this will reduce the number of efforts, time and money needed to check and correct the defects. This approach in software development not only enhances the quality of the end product, but also reduces the time taken to complete the software since most of the bugs are detected early enough before they progress to other difficult to handle phases. Thus, the work focuses on the preventive approach to testing activities with an emphasis on integrating developers and testers as often as possible: unit testing, static code analysis, and automated testing frameworks. These strategies must therefore be aligned to the agile and CI/CD way of working to be able to support the needs of development cycles that are fast. When testing is performed right from the start of the development cycle, it produces more effective software systems for the customers and decreases maintenance expenses.

REFERENCES

[1]. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., & Zhai, C. (2006, October). Have things changed now? An empirical study of bug characteristics in modern open source software. In Proceedings of the 1st workshop on Architectural and system support for improving software dependability (pp. 25-33).

[2]. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., & Yang, X. (2012, June). Test-case reduction for C compiler bugs. In Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (pp. 335-346).

[3]. Pan, K., Kim, S., & Whitehead, E. J. (2009). Toward an understanding of bug fix patterns. Empirical Software Engineering, 14, 286-315.

[4]. Williams, C. C., & Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. IEEE Transactions on Software Engineering, 31(6), 466-480.

[5]. Yang, X., Chen, Y., Eide, E., & Regehr, J. (2011, June). Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (pp. 283-294).

[6]. Liblit, B., Aiken, A., Zheng, A. X., & Jordan, M. I. (2003). Bug isolation via remote program sampling. ACM Sigplan Notices, 38(5), 141-154.

[7]. Bader, J., Scott, A., Pradel, M., & Chandra, S. (2019). Getafix: Learning to fix bugs automatically. Proceedings of the ACM on Programming Languages, 3(OOPSLA), 1-27.

[8]. Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012). Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices, 47(6), 77-88.

[9]. Nama, Prathyusha. "AI-Powered Mobile Applications: Revolutionizing User Interaction Through Intelligent Features and Context-Aware Services." (2023).

[10]. Jang, J., Agrawal, A., & Brumley, D. (2012, May). ReDeBug: finding unpatched code clones in entire os distributions. In 2012 IEEE Symposium on Security and Privacy (pp. 48-62). IEEE.

[11]. Meszaros, G. (2007). xUnit test patterns: Refactoring test code. Pearson Education.

[12]. Herzig, K., Just, S., & Zeller, A. (2013, May). It's not a bug, it's a feature: how misclassification impacts bug prediction. In 2013 35th international conference on software engineering (ICSE) (pp. 392-401). IEEE.

[13]. Liblit, B. R. (2004). Cooperative bug isolation. University of California, Berkeley.

[14]. Pewny, J., Schuster, F., Bernhard, L., Holz, T., & Rossow, C. (2014, December). Leveraging semantic signatures for bug search in binary programs. In Proceedings of the 30th Annual Computer Security Applications Conference (pp. 406-415).

[15]. Sun, C., Le, V., & Su, Z. (2016, October). Finding compiler bugs via live code mutation. In Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications (pp. 849-863).

[16]. Lu, S., Park, S., Seo, E., & Zhou, Y. (2008, March). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (pp. 329-339).

[17]. Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M., & Poshyvanyk, D. (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(4), 1-29.

[18]. Hooimeijer, P., & Weimer, W. (2007, November). Modeling bug report quality. In Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering (pp. 34-43).

[19]. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., & Holz, T. (2015, May). Cross-architecture bug search in binary executables. In 2015 IEEE Symposium on Security and Privacy (pp. 709-724). IEEE.

[20]. Park, S. B., Hong, T., & Mitra, S. (2009). Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28(10), 1545-1558.