Bug Tracking in Cloud Computing

**Pavan Yendluri1\***

**1**Depeartment of computer science, Lovely Professional University, Phagwara-144411, India.

\*[pavanyendluri588@gmail.com](mailto:pavanyendluri588@gmail.com)

***Abstract*— Bug tracking in cloud computing is critical to ensuring the stability and security of cloud services and applications. Cloud computing presents unique challenges for bug tracking, including complexity, scalability, lack of visibility, and security. To overcome these challenges, it is essential to adopt best practices such as continuous monitoring, a collaborative approach, root cause analysis, and automated testing. With these best practices in place, cloud computing environments can be more stable and secure, ensuring that applications and services run smoothly and without interruption. As cloud computing continues to evolve and becomes more prevalent, the importance of bug tracking will only increase. It is essential for organizations to prioritize bug tracking and invest in the tools and processes necessary to identify and resolve bugs in a timely manner. By doing so, they can ensure that their cloud services and applications remain stable and secure, providing value to their customers and driving business success.**

**.**

***Keywords-component; ageing-related bugs; cloud computing; Key-value store; MapReduce; Open source software.***

1. INTRODUCTION

Cloud computing has become an integral part of modern-day software development. The ability to store and access data and applications through the internet has made it easier for businesses to manage their software applications and services. However, like any software system, cloud computing is susceptible to bugs and errors. Therefore, there is a need for a robust bug-tracking system for cloud computing. The bug tracking system is a software application that helps in managing and tracking bugs or defects in a software system. It allows developers to keep track of the reported bugs, assign them to the relevant team members, and monitor their progress until they are resolved. A bug-tracking system for cloud computing is specifically designed to track bugs in cloud-based software applications.

Cloud-based bug tracking systems have several advantages over traditional on-premises bug tracking systems. First and foremost, cloud-based bug-tracking systems provide easy accessibility from any location. With a cloud-based system, developers can access the system from any device with an internet connection. This makes it easier for developers to work remotely and collaborate on resolving bugs in real time. Secondly, cloud-based bug-tracking systems are more scalable than on-premises systems. Cloud-based systems can easily handle large volumes of data and users. This makes them ideal for businesses that have a large number of software applications or services to manage.

Thirdly, cloud-based bug-tracking systems offer better data security. With cloud-based systems, data is stored on secure servers that are monitored and managed by the cloud provider. This ensures that the data is protected from unauthorized access, theft, or loss.

The cloud provider takes care of the hardware and software infrastructure, and businesses only pay for the services they use. However, despite the many advantages of cloud-based bug-tracking systems, there are also some challenges that need to be addressed. One of the main challenges is data privacy. With cloud-based systems, data is stored on servers that are owned and managed by the cloud provider. This raises concerns about data privacy and security.

Diagram

Description automatically generated

Another challenge is integration with other cloud-based software applications. Cloud-based bug-tracking systems need to be integrated with other software applications, such as project management tools, to ensure effective bug tracking and resolution. This can be challenging, especially if the software applications are from different vendors. To overcome these challenges, cloud-based bug-tracking systems need to have certain features. Firstly, they need to have strong data encryption and security features to ensure that data is protected from unauthorized access. Secondly, they need to be integrated with other cloud-based software applications to ensure effective bug tracking and resolution. Finally, they need to have robust reporting and analytics capabilities to enable developers to monitor and track the progress of bugs in real time.

A bug tracking system for cloud computing is essential for businesses that use cloud-based software applications. Cloud-based bug-tracking systems offer several advantages, including easy accessibility, scalability, data security, and cost savings. However, they also face certain challenges, such as data privacy and integration with other software applications. To overcomplications and cloud-based bug-tracking systems need to have strong data encryption and security features, be integrated with other software applications, and have robust reporting and analytics capabilities.

[1] Software testing plays a critical role in identifying and detecting bugs, which often arise from errors and confusion in a program's source code or technical design. Detecting and managing bugs in software applications can be a challenge due to their various forms and complexities. To address this issue, bug-tracking systems have been developed to monitor and keep track of reported bugs in an application. Today, there are several proprietary and open-source bug-tracking systems available, and new systems are continuously being developed to cater to the changing software requirements that give rise to bugs. To address the need for a tool that can effectively fix and track the progress of bug fixes in the rapidly evolving software landscape, this paper presents a comparative study of five defect-tracking systems, including both open-source and proprietary options. Additionally, we propose a new defect tracking system called "Bugtrac," designed to address all types of software applications and the potential defects that may arise from them. We anticipate that this system will be a more promising defect tracking solution compared to existing options.

[2]deploying multitenant applications on the cloud, achieving adequate isolation between tenants presents a significant challenge. To address this challenge, this paper explores the use of COMITRE, or Component-based approach to Multitenancy Isolation Through Request RE-routing, to evaluate the effectiveness of three multitenancy patterns: shared component, tenant-isolated component, and dedicated component. Specifically, we apply this approach to a cloud-hosted Bug tracking system that utilizes Bugzilla and empirically assesses the degree of isolation between tenants offered by each pattern.

Our study reveals that the use of dedicated components offers the highest degree of isolation, particularly in cases of database transactions where support for locking is enabled. On the other hand, for tenant isolation based on performance metrics like response time, the use of shared components appears to be more favourable compared to resource consumption metrics like CPU and memory, where the use of dedicated components is preferred. Finally, we discuss key challenges and recommendations for implementing multitenancy for application components in cloud-hosted bug-tracking systems, with an emphasis on ensuring isolation between multiple tenants.

[3] In software development, the main problem is recognizing the security-oriented issues within the reported bugs due to their unacceptable failure rate to provide satisfactory reliability on customer and software datasets. The misclassification of bug reports has a direct impact on the effectiveness of the bug prediction model.

Diagram

Description automatically generated

Fig: Multitenancy Data Isolation Architecture

The misclassification issue surely compromises the accuracy of the system. Manually reviewing bug reports is necessary to solve this problem, but doing so takes a lot of time and is tiresome for developers and testers. This paper proposes a novel hybrid approach based on natural language processing (NLP) and machine learning. To address these issues, the intended outcomes are multi-class supervised classification and bug prioritization using supervised classifiers. After being collected, the dataset was prepared for vectorization, subjected to exploratory data analysis, and preprocessed. The feature extraction and selection methods used for a bag of words are TF-IDF and word2vec. Machine learning models are created after the dataset has undergone a full transformation. This study proposes, develops, and assesses four classifiers: multinomial Naive Bayes, decision trees, logistic regression, and random forest. The hyper-parameters of the models are tuned, and it is concluded that random forest outperformed with a 91.73% test and 100% training accuracy. The SMOTE technique was used to balance the highly imbalanced dataset, which was initially created for the justified classification. The comparison between balanced and imbalanced dataset models clearly showed the importance of the balanced dataset in classification as it outperformed all experiments.

Diagram

Description automatically generated

Fig: COMITRE Architecture

[4] In the context of open-source development projects, it is common to have an open bug repository where both developers and users can report bugs. However, triaging these reports can be a time-consuming task, particularly for large open-source projects that receive a high volume of bug reports. To address this issue, we propose a semi-automated approach that streamlines the assignment of bug reports to developers. Our approach involves using a machine learning algorithm to analyze the open bug repository and learn which types of reports each developer is most skilled at resolving. When a new bug report is submitted, the algorithm recommends a shortlist of suitable developers to assign the report to. Our approach has achieved precision levels of 57% and 64% on the Eclipse and Firefox development projects, respectively. We have also applied our approach to the gcc open-source development project, although with less favorable outcomes. We discuss the conditions under which this approach is most effective and share the lessons we have learned about applying machine learning to repositories used in open-source development.

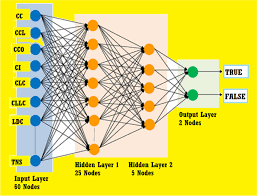


Fig: Neural Network used to bug tracking

[5] Bug repositories like Bugzilla and Jira are abundant with data from numerous projects, including various issue types such as bug reports, feature improvements, new product features, and tasks. Each issue type has multiple attributes, and manually collecting such vast amounts of data can be time-consuming, tedious, and error-prone. This paper focuses on reducing errors caused by human mistakes and improving accuracy by automating the collection of bug reports from Jira. The bug report collection system described in this paper is implemented in C# and utilizes REST APIs to extract data from the Jira repository. By making an HTTP request and parsing the response into objects, this tool automatically extracts information from over 100 Apache projects maintained by Jira and generates reports containing several bug attributes such as bug ID, one-line bug description, bug priority, bug components, bug long description, affected versions, assignee, and other attributes. Researchers can utilize these reports for further analysis, such as bug classification by type, bug prioritization, and prediction of bug severity using machine learning. Therefore, these generated reports are useful for researchers to analyze and prioritize bugs based on their assigned priorities and classify the frequency of different types of bugs in various projects, saving time and manual effort.

[6] During the era of big data and cloud computing, ensuring the reliability of distributed systems is of utmost importance. Regrettably, distributed concurrency bugs, commonly known as DCbugs, are prevalent and can be difficult to detect. These bugs reside within the vast state space of distributed cloud systems and manifest in a non-deterministic manner, depending on the timing of distributed computation and communication. To address this issue, this research paper proposes a novel solution, DCatch, for the detection of DCbugs. DCatch utilizes an analysis of the correct execution of distributed systems to predict and identify DCbugs. The development of DCatch involved the creation of a set of happens-before rules that model a wide range of communication and concurrency mechanisms present in real-world distributed cloud systems. Additionally, runtime tracing and trace analysis tools were designed to identify concurrent conflicting memory accesses in these systems, while tools to help eliminate false positives and trigger DCbugs were also developed. The effectiveness of DCatch was evaluated on four open-source distributed cloud systems, including Cassandra, Hadoop MapReduce, HBase, and ZooKeeper, by monitoring the correct execution of seven workloads on these systems. DCatch successfully identified 32 DCbugs, with 20 of them being genuinely harmful.

[7] When implementing multi-tenancy in a cloud-based software service, maintaining optimal performance and resource consumption for each tenant can be challenging. Designers and architects must ensure that the actions of one tenant do not negatively impact the experience of others. This research aims to identify the trade-offs, commonalities, and differences to be considered when implementing the necessary level of tenant isolation. By conducting a cross-case analysis of several open-source cloud-hosted software engineering tools, we empirically evaluated various degrees of isolation between tenants. Our research uncovered five common factors: disk space reduction, use of locking, low cloud resource consumption, customization and use of plug-in architecture, and choice of multi-tenancy pattern. Two of these factors, customization and plug-in architecture can compromise tenant isolation. In contrast, careful consideration of workload handling, data and process locking, and appropriate multi-tenancy pattern selection can enhance isolation.

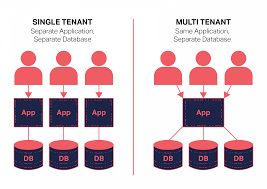


Fig: Single vs Multi-Tenant Cloud System

We also found five case study differences: the size of generated data, cloud resource consumption, sensitivity to workload changes, software process effect, and client latency and bandwidth. Our results indicate that large data size, high resource consumption by specific processes, high or fluctuating workload, low client latency, and bandwidth when transferring multiple files can impair isolation. Moreover, we developed an explanatory framework that maps tenant isolation to different software development processes, cloud resources, and layers of the cloud stack. This framework also explains the trade-offs involved in achieving tenant isolation, such as resource sharing, the number of users/requests, customizability, the size of generated data, the scope of control of the cloud application stack, and business constraints. Our research suggests that software architects must consider the trade-offs, commonalities, and differences identified in this study to meet their tenant isolation requirements.

[8] When implementing multi-tenancy in a cloud-based software service, maintaining optimal performance and resource consumption for each tenant can be challenging. Designers and architects must ensure that the actions of one tenant do not negatively impact the experience of others. This research aims to identify the trade-offs, commonalities, and differences to be considered when implementing the necessary level of tenant isolation. By conducting a cross-case analysis of several open-source cloud-hosted software engineering tools, we empirically evaluated various degrees of isolation between tenants. Our research uncovered five common factors: disk space reduction, use of locking, low cloud resource consumption, customization and use of plug-in architecture, and choice of multi-tenancy pattern. Two of these factors, customization and plug-in architecture can compromise tenant isolation.

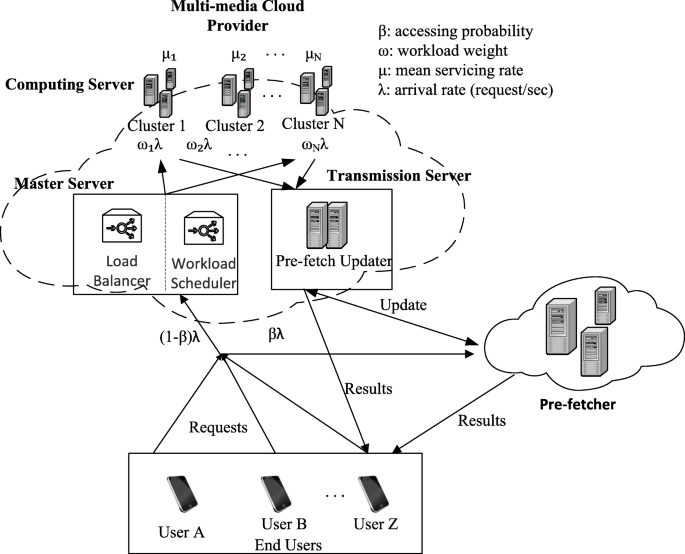


Fig: Data transmission between cloud and end users in the presence of pre-fetcher

In contrast, careful consideration of workload handling, data and processes locking, and appropriate multi-tenancy pattern selection can enhance isolation. We also found five case study differences: the size of generated data, cloud resource consumption, sensitivity to workload changes, software process effect, and client latency and bandwidth. Our results indicate that large data size, high resource consumption by specific processes, high or fluctuating workload, low client latency, and bandwidth when transferring multiple files can impair isolation. Moreover, we developed an explanatory framework that maps tenant isolation to different software development processes, cloud resources, and layers of the cloud stack. This framework also explains the trade-offs involved in achieving tenant isolation, such as resource sharing, the number of users/requests, customizability, the size of generated data, the scope of control of the cloud application stack, and business constraints. Our research suggests that software architects must consider the trade-offs, commonalities, and differences identified in this study to meet their tenant isolation requirements.

[9] The demand for a comprehensive understanding of fault analysis and recovery mechanisms in service-oriented systems (SOSs) is growing due to their increasing scale and complexity. To meet this need, a systematic literature review (SLR) was conducted to provide a summary of the current state-of-the-art in SOSs fault analysis. The SLR followed a predefined method and included a manual search of 11 highly reputable international journals and three conference proceedings. The papers were categorized and organized based on research questions pertaining to fault analysis approaches, types of faults, current issues and challenges, and tools. The resulting review includes 123 papers published between January 2008 and April 2017, with 65 addressing fault handling approaches, 35 reporting SOS-specific faults, 51 addressing issues, challenges, and testing of SOSs, and 17 related to tooling. The review discusses the main approaches, techniques, concepts, contributions, and research methods used for fault analysis in SOSs. Overall, this comprehensive review provides empirical evidence that can inform future research agendas in this field.

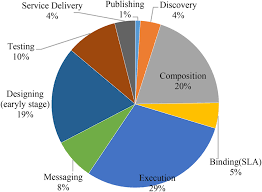


Fig: Fault analysis of service-oriented systems

[10] In a cloud-hosted application, tenants often share components and resources to reduce costs and resource consumption. However, this can lead to performance issues and access problems if one tenant experiences a sudden increase in workload and the application does not scale up accordingly. This problem can be exacerbated when certain components have varying degrees of isolation. To address this, this paper proposes a decision-based model that consists of an optimization model, an open multiclass queuing network model, and a metaheuristic algorithm to determine the optimal deployment of components for a cloud-hosted application that ensures the required degree of multitenancy isolation. The model was tested through experiments, which showed that the optimal solutions obtained from the model had low variability and deviation when compared to actual optimal solutions. The paper also discusses the potential applications of the optimization model and highlights the challenges and recommendations for deploying components with varying degrees of isolation.

[11] Cloud-hosted application tenants seek to decrease costs and reduce resource consumption by sharing components and resources. However, resource sharing can impact access and performance for tenants if one experiences a sudden increase in workload, especially if the application does not handle the workload. This issue can be severe when there is a higher or varying degree of isolation between components. The aim of this paper is to introduce new solutions for deploying components of a cloud-hosted application to ensure the necessary level of multitenancy isolation through a mathematical optimization model and metaheuristic algorithm. The research conducted in this paper demonstrates that the optimal solutions achieved through the model have low variability levels and a small percentage deviation. Additionally, this paper explores the possible areas of application of our optimization model, as well as the challenges and recommendations for deploying components with varying degrees of isolation.

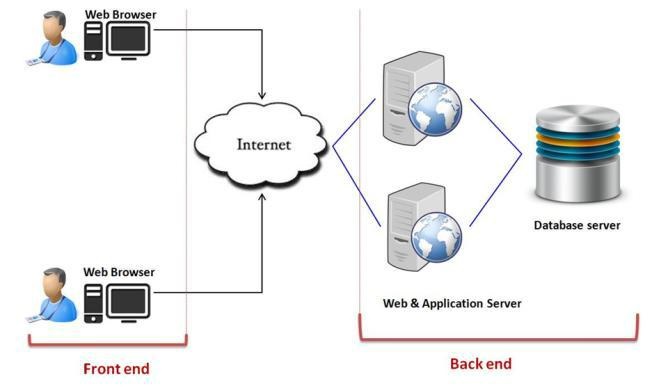


Fig: optimal resource management in the cloud environment

[12] Multitenancy is the capability that allows multiple tenants to access a single instance of a cloud offering. However, there has been a lack of focus on implementing the necessary degree of isolation, which can differ among tenants, despite the variety of approaches available for implementing multitenancy. In response, this paper presents a framework for attaining the required level of isolation between tenants who are accessing a cloud offering, which ensures that the performance, resource utilization, and access privileges of one tenant do not impact other tenants during workload fluctuations. The framework consists of two key components: 1) the component-based approach to multitenancy isolation via request re-routing (COMITRE); and 2) an optimization model that generates optimal solutions for deploying components of a cloud-hosted service. We showcase the effectiveness of the framework by employing it in a case study of a cloud-hosted bug-tracking system and a synthetic dataset, attaining the necessary level of isolation and optimal deployment solutions.

[13] Cloud-hosted applications often share components and resources among tenants to reduce costs and resource consumption. However, this sharing can lead to issues where the behaviour of one component impacts the performance, resource usage, and access privileges of others, especially if the application fails to scale up during sudden workload spikes. These issues are exacerbated when components have different or varying degrees of isolation. This paper proposes a mathematical optimization model and a simulated annealing-based metaheuristic solution to deploy components of a cloud-hosted application while ensuring the necessary degree of isolation between them. Our experiments demonstrate that the near-optimal solutions obtained from our model had low variability and percent deviation compared to the optimal solution. Additionally, we provide recommendations for deploying components with varying degrees of isolation.

[14] Software engineering research areas like mutation testing, automatic repair, fault localization, and fault injection rely on empirical knowledge of frequently occurring bug-fixing code changes. Previous studies in this field have focused on what changes have been made due to bug-fixes in terms of code edit actions but have overlooked the context of the change. Knowing about the context of bug-fix changes can potentially narrow down the search space for many software engineering techniques by focusing on specific parts of the software. Moreover, most previous research has focused on C and Java projects, leaving a gap in empirical evidence about bug-fixing changes in Python software. This paper addresses these gaps by conducting an extensive empirical analysis of bug-fixing changes in three OpenStack projects, considering both the what and the where of the changes. The findings indicate that recurring change patterns are not oblivious to the surrounding code but tend to occur in specific code contexts.

[15] This paper addresses the challenge of detecting quality bugs, which are often difficult to identify due to the scattered nature of quality-related features across the codebase. To overcome this limitation, the authors propose a solution that leverages a Hierarchical Dirichlet Process (HDP) topic modeling technique in conjunction with structural and textual analyses to capture the hierarchical topical relationships among quality features. Their solution, called SOFTQUALTOPICDETECTOR, clusters scattered quality concerns into a meaningful hierarchy to infer a set of candidate classes that are relevant for recommending the repair of quality bugs. The authors also incorporate three visual features to monitor, prioritize, and trace suspicious classes for improving maintainability, functional suitability, and traceability. The empirical evaluation of the SOFTQUALTOPICDETECTOR shows a significant improvement over the baseline and state-of-the-art approaches in terms of average precision and recall.

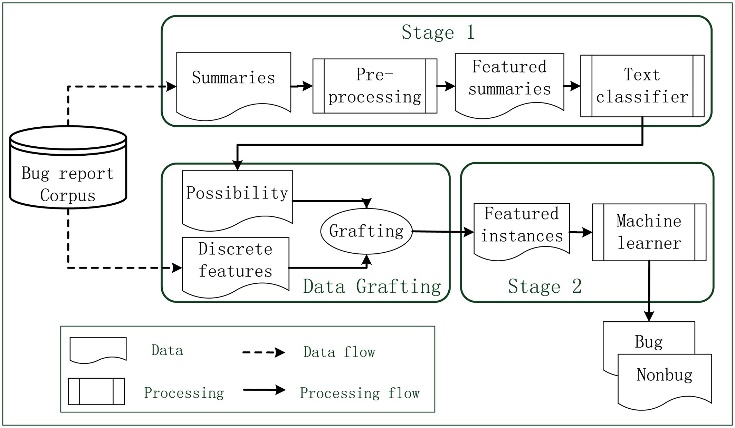


Fig: Combining text mining and data mining for bug report classification

[16] This paper proposes a mining approach to identify fix patterns of Python programs by extracting fine-grained bug-fixing code changes. The approach employs the repetitiveness of code changes to generate bug fix patterns, which can be used as a critical component of Automatic Program Repair (APR) to automatically detect and fix bugs in software products. The paper collects bug reports from GitHub and clusters similar bug-fixing code changes using abstract syntax tree edit distance to generate fix patterns. The effectiveness of these fix patterns is evaluated by applying them to single-hunk bugs in two benchmarks, BugsInPy and QuixBugs. The results show that 13 out of 101 real bugs can be fixed without human intervention, and for each complex bug, 15% of the bug code could be fixed, and 37% of the bug code could be matched by fix patterns. The paper also highlights the limitations of using fix patterns proposed for other programming languages in Python programs due to syntactic incompatibilities and lack of analysis of dynamic features. The proposed approach addresses these limitations and demonstrates the potential of using fix patterns to improve the quality of Python programs.

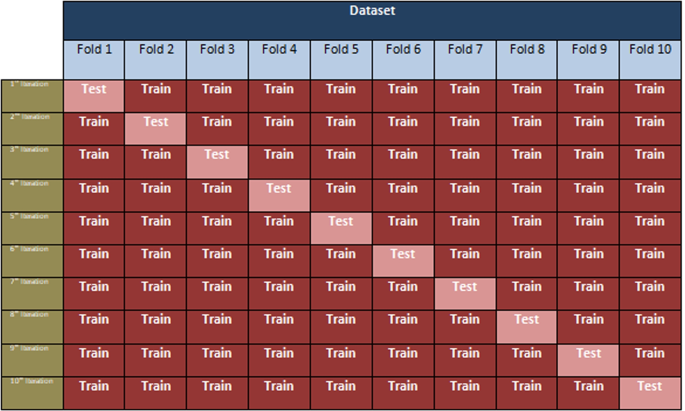


Fig: Bug tracking software defects analysis using deep learning.

[17] The paper describes the development of a tool called Python Change Miner that can discover code change patterns in the histories of Python projects. The tool was validated and used to identify patterns in a dataset of 120 projects from four different domains of software engineering. The authors manually categorized the patterns that occur in more than one project and compared different domains and patterns in terms of their structure and content. They conducted a survey of the authors of the discovered changes and found that most of them could give the change a name and expressed their desire to have the changes automated. The authors also interviewed members of a popular integrated development environment (IDE) development team to estimate the feasibility of automating the discovered changes. The study found that independence from context and high precision made a pattern a better candidate for automation, and several patterns were ranked as very likely for automation. Overall, the study suggests that discovering code change patterns can provide insights into the nature of software development and lead to practical solutions for developers.

[18] ProFIPy is a new fault injection tool for Python software that enables users to specify their own software fault model using a domain-specific language (DSL) for fault injection. The tool is designed to be programmable, allowing users to customize the fault injection process to their specific needs. Additionally, ProFIPy is provided as software-as-a-service, making it more user-friendly and easier to access. The tool supports users through the configuration of the fault load and workload, analysis of failure data, and full automation of experiments using container-based virtualization and parallelization. With these features, ProFIPy can help developers and software engineers test the resilience and robustness of their software, and identify and fix potential faults before they cause significant problems or downtime.

[19] Deep Learning (DL) frameworks have become essential tools for developers to create DNN models without having to learn the underlying algorithms and models. These frameworks have been deployed in safety-critical areas like self-driving cars and medical diagnostics, making it crucial to identify and characterize bugs that could potentially impact their reliability. The objective of our research is to classify common DL framework bugs at the source code level, analyze their symptoms, root causes, and solutions, and provide insights to researchers to develop quality assurance techniques such as automatic repair and fault location techniques.

To achieve this objective, we began by summarizing the reference architecture of DL frameworks and proposed a DL framework bug taxonomy. We then analyzed 1,127 bug reports from eight popular DL frameworks and categorized the bug types, root causes, and symptoms. Our research findings show that DNN model building bugs and general type bugs account for one-third of the total defects. DNN model building bugs are more prone to algorithm logic constraints, internal API errors, and data/numerical errors. Additionally, we identified fifteen bug-fixing patterns that can be used as a reference for DL framework bug repair and future research on automatic DL framework bug detection tools.

Our analysis of bug-fixing changes has provided valuable insights into the occurrence, root causes, symptoms, and fixing of these bugs. This study can help researchers ensure DL framework quality and provide actionable recommendations for DL framework developers to enhance the quality of their code.

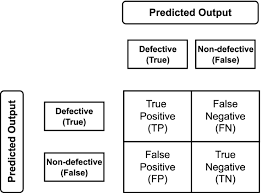


Fig: Prediction matrix of the deep learning mode.

[20] The rapid pace of evolution in cloud computing software is creating a dilemma for our increasing reliance on cloud computing. While cloud software must be agile to remain competitive, critical services are becoming increasingly dependent on the cloud and require high availability through strict Service Level Agreements (SLAs) for cloud infrastructures. This constant push to increase the frequency of cloud upgrades while maintaining service availability is unsustainable. In this paper, we discuss the challenges and opportunities for cloud upgrades. We examine the release histories of several cloud applications to analyze their pace of evolution and discuss the limitations of current cloud upgrade mechanisms. We propose several solutions for sustaining this evolution while improving availability, by focusing on the unique characteristics of cloud computing. By outlining several promising directions for realizing this vision, we suggest a research agenda for the future of software upgrades in the cloud.

SUMMURY

Effective bug tracking is a crucial aspect of software development, and it becomes even more vital in cloud computing due to its distributed nature. Although cloud computing offers various benefits, including scalability, elasticity, and cost-effectiveness, the rapid evolution pace of cloud software is in conflict with the growing reliance on cloud computing, making it imperative to improve bug tracking in cloud computing.

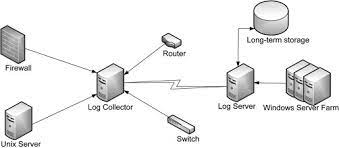


Fig: Bug Tracking System

The objective of this research is to investigate the challenges and opportunities for bug tracking in cloud computing. The study underlines the importance of efficient bug tracking in cloud computing as the number of critical services dependents on the cloud continues to increase. Additionally, it is essential to ensure high availability through firm Service Level Agreements (SLAs) for cloud infrastructures.

This research explores the shortcomings of current bug tracking mechanisms in cloud computing and proposes solutions to overcome these challenges. The study analyzes the release histories of several cloud applications to determine their evolution pace, and it identifies the challenges of cloud upgrade mechanisms. The proposed solutions to improve bug tracking efficiency in cloud computing include automated bug tracking, better collaboration between developers and users, and improved integration of bug tracking systems with cloud infrastructure. The research suggests that these solutions can help sustain the evolution pace of cloud software while enhancing availability.

Finally, the study suggests a research agenda for the future of bug tracking in cloud computing. It emphasizes the importance of more research in areas such as automated bug detection, root cause analysis, and machine learning-based approaches to bug tracking. The research also underscores the need for collaboration between researchers, developers, and users to ensure effective bug tracking in cloud computing.

REFERENCES

1. Gopal, M. K., Govindaraj, M., Chandra, P., Shetty, P., & Raj, S. 2022, February). Bugtrac–a new improved bug tracking system. In *2022 IEEE Delhi Section Conference (DELCON)* (pp. 1-7). IEEE.
2. Ochei, L. C., Petrovski, A., & Bass, J. M. (2016, June). Implementing the required degree of multitenancy isolation: A case study of cloud-hosted bug tracking system. In 2016 IEEE International Conference on Services Computing (SCC) (pp. 379-386). IEEE.
3. Tabassum, N., Namoun, A., Alyas, T., Tufail, A., Taqi, M., & Kim, K. H. (2023). Classification of Bugs in Cloud Computing Applications Using Machine Learning Techniques. Applied Sciences, 13(5), 2880.
4. Anvik, J., Hiew, L., & Murphy, G. C. (2006, May). Who should fix this bug?. In Proceedings of the 28th international conference on Software engineering (pp. 361-370).
5. Kaur, A., & Jindal, S. G. (2017, January). Bug report collection system (BRCS). In 2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence (pp. 697-701). IEEE.
6. Liu, H., Li, G., Lukman, J. F., Li, J., Lu, S., Gunawi, H. S., & Tian, C. (2017). Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. ACM SIGARCH Computer Architecture News, 45(1), 677-691.
7. M. Grottke, R. Matias Jr., and K. S. Trivedi, The fundamentals of software aging, In Proc. of the 1st Int'l Workshop on Software Aging and Rejuvenation (WoSAR2008), pp.1-6, 2008.
8. Ochei, L. C., Bass, J. M., & Petrovski, A. (2018). Degrees of tenant isolation for cloud-hosted software services: a cross-case analysis. Journal of Cloud Computing, 7(1), 1-39.
9. Bhandari, G. P., & Gupta, R. (2018). Fault analysis of service‐oriented systems: a systematic literature review. IET Software, 12(6), 446-460.
10. Ochei, L., & Petrovski, A. (2018). A Decision-based Model for Optimizing the Deployment of Cloud-hosted Application Components to Guarantee Multitenancy Isolation.
11. Ochei, L. C., & Ejiofor, C. I. (2018). A model for optimising the deployment of cloud-hosted application components for guaranteeing multitenancy isolation. Advances in Science, Technology and Engineering Systems Journal, 3(2).
12. Ochei, L. C., Petrovski, A., & Bass, J. M. (2018). A framework for achieving the required degree of multitenancy isolation for deploying components of a cloud-hosted service. International Journal of Cloud Computing, 7(3-4), 248-281.
13. Ochei, L. C., Petrovski, A., & Bass, J. M. (2016, October). Optimizing the deployment of cloud-hosted application components for guaranteeing multitenancy isolation. In 2016 International Conference on Information Society (i-Society) (pp. 77-83). IEEE.
14. Cotroneo, D., De Simone, L., Iannillo, A. K., Natella, R., Rosiello, S., & Bidokhti, N. (2019, October). Analyzing the context of bug-fixing changes in the openstack cloud computing platform. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE) (pp. 334-345). IEEE.
15. Krasniqi, R., & Do, H. A Hierarchical Topical Modeling Approach for Recommending Repair of Quality Bugs.
16. Yang, Y., He, T., Feng, Y., Liu, S., & Xu, B. (2022). Mining Python fix patterns via analyzing fine-grained source code changes. Empirical Software Engineering, 27(2), 48.
17. Golubev, Y., Li, J., Bushev, V., Bryksin, T., & Ahmed, I. (2021). Changes from the trenches: Should we automate them?. arXiv preprint arXiv:2105.10157.
18. Cotroneo, D., De Simone, L., Liguori, P., & Natella, R. (2020, June). Profipy: Programmable software fault injection as-a-service. In 2020 50th annual IEEE/IFIP international conference on dependable systems and networks (DSN) (pp. 364-372). IEEE.
19. Yang, Y., He, T., Xia, Z., & Feng, Y. (2022). A comprehensive empirical study on bug characteristics of deep learning frameworks. Information and Software Technology, 151, 107004.
20. Neamtiu, I., & Dumitraş, T. (2011, September). Cloud software upgrades: Challenges and opportunities. In 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (pp. 1-10). IEEE.

families in Google BigTable [[2].](#_bookmark8) It is designed to handle large amounts of data spread out across clustered node while providing a highly available service with no single point of failure. Cassandra is used in Facebook, Rackspace, Twitter, Netflix, Cisco, etc. The code is written in Java and the current latest release is version 1.1.4, which was released in August 2012. Software bugs in Cassandra are tracked by JILA in Apache software foundation. The bug reports have been posted from March 2009.

1. *Eucalyptus*

Eucalyptus is a middleware for constructing a private Infrastructure as a Service (IaaS) cloud that is accessible via APIs compatible with Amazon EC2 6 and Amazon S3 7 . Eucalyptus was originally developed by MAYHEM labs in UC Santa Barbara [[3].](#_bookmark9) Currently Eucalyptus fully supports KVM 8 and Xen. In addition, the Enterprise Edition Eucalyptus supports the proprietary VMware hypervisor. Several programming languages are used for the implementation including Java, C, python and perl. The current stable release of Eucalyptus is version 3.1, which was released in June 2012. Software bugs in Eucalyptus were originally tracked on Launchpad9, but later they are migrated to JILA-based system. The bugs are reported from the version 1.5.x series in 2009.

1. *memcached*

Memcahed is a widely adopted general-purpose distributed memory caching system. It maintains a key-value associative array in memory. Clients populate this array and query it through simple API. Many organizations use memcached to speed up web application systems using data bases by caching query results. The software is used in Wikipedia, YouTube, Twitter, Mixi, etc. It is implemented by C and the latest version is 1.4.14 released in July 2012. The project is managed in Google code and bug reports are posted on the issued list. On the issue list, bugs are tracked from the version 1.2.6 in October, 2008.

1. *Xen*

Xen is an open source implementation of virtual machine monitor (VMM) that allows multiple operating systems execute on a single physical server concurrently. Xen was originally developed in University of Cambridge Computer Lab. and the first public release of Xen is occurred in 2003. Currently, non-commercial XenServer is fully open-source and available to the public. Xen is implemented by C except some scripts which are written by python. The latest release of Xen hypervisor is version 4.1.3 which was released in August 2012. Software bugs in Xen projects are managed in Bugzilla hosted by xen.org. The bugs are tracked from an unstable version in April 2005.

6 Amazon EC2, <http://aws.amazon.com/ec2/>

7 Amazon S3, <http://aws.amazon.com/s3/>

8 Kernel based Virtual Machine, <http://www.linux-kvm.org/>

9 Launchpad, https://launchpad.net/

1. BUG REPORTS

For the five OSS projects, we investigate the bug reports on the bug tracking systems from the initial reports to the last reports until January 31st, 2012. In the reported bugs, there are duplicated reports originated from the same software bug, invalid reports which are not exactly associated with the faults in the software, and unsolved issues whose causes are not identified so far. First, we remove those duplicated reports and invalid reports based on the resolution status assigned by the developers. Unsolved issues are counted in the results because we analyze the amount of unresolved issues as well. Next, we filter the reports by keyword search using the key words: “leak”, “increas”, “descreas”, “deplet”, “exhaust”, “exceed”, “aging”, “Out of memory” and “OOM”. The verb keywords are written in a special form (e.g., “increas” instead of “increase”) to match both present and progressive tenses. By reading the descriptions of filtered bug reports carefully, we identify the unique set of reports caused by aging-related bugs following the definition in [[1].](#_bookmark7) The statistics of aging-related bugs are summarized and discussed in the following sections.

1. *Amount of aging-related bugs*

First, we count the total number of the valid bug reports and the number of the reports caused by aging-related bugs as in [TABLE I.](#_bookmark0) Aging-related bugs are confirmed in all of the five projects. The results reveal that the recent OSS products for cloud computing contain aging-related bugs regardless of the functionalities provided by the software. The related statistics are also summarized in [TABLE I.](#_bookmark0) Bug rates are computed by dividing the total number of valid bug reports by the bug tracking period in months. The ratio of aging-related bugs represents the ratio of the number of reports for aging-related bug to the total number of valid reports. The bug rates may be affected by the amount of activities (updates in a certain time period) in the project. Both Hadoop MapReduce and Cassandra are extensively active projects and their bug rates are higher than the others. The ratios of aging-related bugs in the all five projects are observed around at 0.01 (from 0.4% to 1.4%).

1. *Types of aging-related bugs*

Next, the aging-related bugs are categorized by the resources affected by aging in long time execution. Software objects such as file descriptors, database connections, data objects and threads are also considered as software resources which should have a finite limit of their amounts. The violation of such limitation often causes a memory leak problem in its execution environment. As summarized in TABLE II, amongst all the resource categories, file descriptors are the most affected resources by aging-related bugs. Many bug reports of Hadoop MapReduce and Cassandra alert possible memory leaks due to the faults in closing file descriptors. Similarly, connections used in software such as database connections or network connections are likely to be unreleased even after the completion of their missions. Such unreleased connections often cause a memory leak. Although the aging-related bugs largely belong to these two categories, fixing such bugs are

TABLE I. AGING-RELATED BUGS FOUND IN THE FIVE OSS PROJECTS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Project period  (months) | The total number  of valid bug reports | The number of  aging-related bugs | Bug rate  (per month) | Ratio of aging-related  bugs to the total reports |
| Hadoop  MapReduce | 70 | 1802 | 21 | 25.74286 | 0.011653718 |
| Cassandra | 35 | 1605 | 17 | 45.85714 | 0.0105919 |
| memcached | 39 | 179 | 1 | 4.589744 | 0.005586592 |
| Eucalyptus | 37 | 540 | 8 | 14.59459 | 0.014814815 |
| Xen | 79 | 1612 | 8 | 20.43038 | 0.004956629 |

TABLE II. THE NUMBER OF AGING-RELATED BUGS IN RESOURCE CATEGORIES

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | File descriptors | Connections/ Pipes | Data objects/ Variables | Threads | External resources | Unknown | Others |
| Hadoop  MapReduce | 5 | 1 | 8 |  | 3 | 2 | 2 |
| Cassandra | 10 | 4 | 2 |  |  | 1 |  |
| memcached |  |  | 1 |  |  |  |  |
| Eucalyptus |  | 1 | 3 | 3 |  |  | 1 |
| Xen | 2 | 1 | 1 |  |  | 3 | 1 |

TABLE III. RESOLUTION STATUSES OF THE ALL REPORTS AND AGING-RELATED BUGS IN THE FIVE OSS PROJECTS

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | All bugs | | | | Aging-related bugs | |
| Resolution  Status | Fixed or  resolved | Unresolved | Won't Fix | Cannot  reproduce | Fixed or  resolved | Unresolved |
| Hadoop  MapReduce | 957 | 779 | 41 | 25 | 13 | 8 |
| Cassandra | 1442 | 48 | 37 | 78 | 17 | 0 |
| memcached | 146 | 9 | 24 | - | 1 | 0 |
| Eucalyptus | - | - | - | - | 8 | 0 |
| Xen | 858 | 641 | 115 | - | 5 | 3 |

relatively easy once the unreleased object is identified. In fact, most of the bug reports in these categories are resolved in a short period. Besides the file descriptors and connections, data objects and thread handlers also confront the risk of leakage due to the missing operations to release.

In some aging-related bugs in Hadoop MapReduce affect the external resources such as the number of files in a directory, the number of zombie processes, the objects in external software components. We categorize these resource types as external resources as they persist independently of the software execution. Compared to the first four categories, this type of aging-related bugs is difficult to find by verification of single software component. Since the aging phenomena caused by this type of aging-related bugs depend on the configuration of the execution environment, the limited reproducibility of the aging becomes a problem.

1. *Resolutions of aging-related bugs*

Once the root-cause of software aging is identified, the developers work around the bug to create a patch. When the patch is tested and the removal of the problem is confirmed, the bug report is closed as resolved status. The resolution statuses of aging-related bugs at 31st January 2012 are summarized in TABLE III. The data for Eucalyptus is not presented due to the inaccessibility to the Launchpad which

is discarded after employing the new bug tracking system (in June 2012). The status “Cannot reproduce” is used only in JILA-based system (i.e, Hadoop MapReduce and Cassandra). This status maybe categorized into “Unresolved” or “Won’t Fix” in the other projects.

Hadoop MapReduce and Xen have surprising numbers of unresolved issues. Although such undesirable numbers of unresolved problems can be accounted by the length of the projects in a sense, the immaturities of the software should be alerted to the users. In contrast, the number of unresolved bugs in Cassandra is considerably small instead of its high bug rate as seen in [TABLE I.](#_bookmark0) In particular, all of the aging- related bugs in Cassandra are in the fixed status at this time. Although the high bug rate indicates the immaturity of the source code, the small number of unresolved bugs implies the well-organized bug tracking community. All of the aging-related bugs in memcached and Eucalyptus are also in the fixed status.

For Hadoop MapReduce, periodic clean-up of unnecessary objects are adopted as a tentative or a permanent solution to aging-related bug instead of removing the bug. Periodic clean-up is also used for a tentative measure in the other projects until the corresponding bug-fix is available. The solution relying on the garbage collection in Java virtual machine (JVM) can also be considered as an approach using

periodic clean-up. Some aging-related bugs in Java based projects are fixed by replacing an object reference with a weak reference to that object. Since Java objects having only weak references become the target of garbage collection of JVM, unnecessary objects are pushed to be cleared by garbage collection.

Periodic clean-up approaches usually require the configuration parameter which specifies the limitation to trigger clean-up operation. However, such configuration parameter faces the risk of user's mis-configuration. If the limitation is not set within the available resource capacity, the software suffers from software aging and eventually fails due to the depletion of resources. Careful configuration of the limitation is particularly important in the execution environments that allow flexible resource allocation such as

1

0.9

0.8

0.7

0.6

0.5

0.4

0.3

0.2

0.1

0

0

100

200

300

400

500

600

700

800

900

1000

1100

1200

1300

1400

1500

1600

1700

1800

1900

2000

days

All bugs

Aging-related bugs

virtual machines as discussed in [[5].](#_bookmark11)

A question related to the resolution of aging-related bug is whether the time to fix the aging-related bug is longer than that for all the bug reports. We investigate the time to fix the bugs (TTFx) from the time when the bugs are initially reported to the time when the problems are resolved. [Figure](#_bookmark1) [1](#_bookmark1) shows the distributions of TTFx for all the bugs and the aging-related bugs in Hadoop MapReduce. The TTFx distribution for all the bugs is obtained from the set of bug reports in resolved status. As can been seen, most of the resolved bugs are fixed within 1000 days. The distribution of TTFx for aging-related bugs is obtained from the 13 aging- related bugs in resolved status (including one applying periodic clean-up as a permanent solution). Contrary to our expectation, most of aging-related bugs are fixed in a shorter time than the TTFx for all the bugs. Since there are many unresolved issues in Hadoop MapReduce, it may contain more complex type of bugs which take longer time to fix than the aging-related bugs. In addition, we cannot neglect the impact of the time to identify the root-cause of the software aging which may not be counted in the TTFx. Some short TTFx for aging-related bug reports are accounted for by the quality of the reports which describes the suspicious root-cause and further provides a suggested software patch.

The TTFx distributions in Cassandra are computed as shown in [Figure 2.](#_bookmark2) Compared to Hadoop MapReduce, reported bugs are fixed promptly as 65% of the bugs are resolved in ten days and more than 99% of the bugs are resolved in 300 days. In particular, all of the aging-related bugs are completely removed within 250 days. Looking at the distribution of TTFx for aging-related bugs, we can see the silent period from 20 days to 140 days where the bug-fix event is not frequent as for all the bugs. Basically the delays of the bug-fixes in this period are caused by missing status updates rather than the difficulties of the bug-fix. These bugs could have been updated as resolved status earlier. Therefore, with only from this observation, we can hardly conclude that the aging-related bugs have longer TTFx than the others.

TTFx distributions for the other three projects are not computed because the number of aging-related bugs is small. We focus on Hadoop MapReduce and Cassandra in the following subsection as well.

Figure 1. Distributions of time to fix the bugs in Hadoop MapReduce

1



All bugs

Aging-related bugs

0.9

0.8

0.7

0.6

0.5

0.4

0.3

0.2

0.1

0

0 100 200 300 400 500

days

Figure 2. Distributions of time to fix the bugs in Cassandra

1. *Trends of aging-related bugs*

Finally, we investigate the trend of aging-related bugs by counting the number of bugs per month from the initial bug report. [Figure 3](#_bookmark3) shows the bug curve for Hadoop MapReduce by plotting the accumulated number of bugs from April 2006 to January 2012. Although many users download and install the current version of Hadoop MapReduce, the software is not matured in terms of software reliability characterized by the bug curve. The recent sharp increasing trend of the number of bugs may be caused by the rapid increase of the number of users. As the software becomes popular, the number of users increases and hence the detection of software bugs is accelerated.

According to the trend of the number of bugs, the ratio of aging-related bugs is changed as shown in [Figure 4.](#_bookmark4) As can be seen, aging-related bugs constantly exist around at 1% (0.01) of the total number of bugs during the observed period. It is not appropriate to conclude that the aging-related bugs have an original trend over the trend of the total number of bugs. However, we confirm that aging-related bugs are contained in Hadoop MapReduce at a certain ratio of the total number of bugs in most of the development lifecycle.

The increasing trend of the number of bugs is also observed in Cassandra from March 2009 to January 2012 as shown in [Figure 5.](#_bookmark5)

2000

1800

1600

1400

1200

1000

800

600

400

200

0

The total number of bugs

1800

1600

1400

1200

1000

800

600

400

200

0

The total number of bugs

2006/04 2007/04 2008/04 2009/04 2010/04 2011/04

Figure 3. The bug curve for Hadoop MapReduce

The ratio of aging-related bug

0.016

0.014

0.012

0.01

0.008

0.006

0.004

0.002

0

2006/04 2007/04 2008/04 2009/04 2010/04 2011/04

Figure 4. The ratio of aging-related bugs in Hadoop MapReduce

Although the official release of Cassandra 1.0 was announced in October 2011, the number of bug reports increases constantly. The ratio of aging-related bugs is observed as shown in [Figure 6.](#_bookmark6) Similar to Hadoop MapReduce, aging-related bugs are found in the most ranges of the period by about 1 % of the total bugs. As some aging- related bugs are reported after the release of Cassandra 1.0, a number of users can confront the issues caused by aging- related bugs.

1. DISCUSSION

The bug classification method used in our investigation may not perfectly cover all the latent aging-related bugs and it will bias the analysis. First, we intentionally include the unresolved reports in the scope of the investigation because remaining unresolved issues are quite common in OSS projects and are considered as the risk of the users as seen in Section III-C. However, the uncertainty of the classification of the unresolved issues might bring a misreading of the amount of aging-related bugs. Second, the chosen key-words for filtering the bug reports may not be enough to filter the aging-related reports. Our classification method relies on the description of the reports and hence we may overlook some aging-related bug reports due to the incomplete descriptions and/or the insufficient set of key words. We should remark that this is the first attempt to extract the aging-related bug reports from the OSS's bug tracking system and the method can be improved considering the potential biases discussed above.

2009/3 2009/9 2010/3 2010/9 2011/3 2011/9

Figure 5. The bug curve for Cassandra

The ratio of aging-related bug

0.03

0.025

0.02

0.015

0.01

0.005

0

2009/3 2009/9 2010/3 2010/9 2011/3 2011/9

Figure 6. The ratio of aging-related bugs in Cassandra

Reproducibility of bugs is an important aspect of bug tracking process especially in OSS projects. We observe that the bug reports are broadly separated into two cases; i) the case where the fault in the source code is specified or ii) the case where the actual fault is unknown. In the former case, developers can easily fix the bug and commit the corresponding patch promptly. The reporter of the bug sometime prepares the patch at the same time as posting the report. In the latter case, however, the developers who try to fix the reported bug need to reproduce the bug in their own environment. The task to reproduce the problem is sometime cumbersome because it is not easy to create the exact same execution environment as that of the bug reporters. We find many bug reports which remain unresolved status or closed without fix due to the difficulty of reproducing the problem. The difficulty sometime is caused by the insufficient description about the reporter’s system configuration. Fixing aging-related bugs also suffer from the limited reproducibility as they highly depend on the execution environment and require long time to confirm the problem.

To improve the reproducibility of aging-related bugs, it is important to characterize the aging phenomena associated with workloads. The framework to characterize workload- aging relationship [[8],](#_bookmark14) the techniques to understand the aging trends in a short period of time such as accelerated degradation tests [[9]](#_bookmark15) and accelerated life tests [[10]](#_bookmark16) are potentially useful for this purpose. These techniques are not designed for the tools available in OSS projects where a number of developers worldwide share the source code and

run the software in the individual environments. However, leveraging these techniques to provide a common testing tool and introducing the process to share the configuration and test results of the tool, the reproducibility of aging-related bugs might be greatly improved.

Although the terminology such as "software aging", "aging-related bug" and "software rejuvenation" have been used in many research literature, such terminology are seldom used in the software development community as we did not find them in the reports of aging-related bugs. This implies that there is a split among research works and software development in practice. Due to the gap in the terminology, the outcomes from research works may not be utilized effectively in practical software development projects. Developing a common tool for testing aging-related bugs in OSS development might help to overcome the gap.

1. RELATED WORK

The most closely related work to the presented study is the bug investigation report of JPL/NASA projects [[4].](#_bookmark10) The report categorizes 520 of unique software faults found in 18 missions into Bohrbug, non-aging-related Mandelbug, or aging-related bug and the proportions of the aging-related bugs over the software mission periods are investigated. The categorization is based on the textual description, while we resort to key word search for filtering only aging-related bugs and reading the description subsequently. The observed proportions of aging-related bug in cloud-oriented OSS projects (around at 0.1) are relatively smaller than 0.044 which is resulted from the JPL/NASA projects (23 software bugs are categorized into aging-related bugs from 520 bugs [[4]](#_bookmark10)). While their study focused on the reports only in the operational phase, the bug reports investigated in our study include the reports in the development phase as well. This might be a cause of the difference in the proportion of aging- related bugs. Besides the analysis of the proportion, we also study the distribution of the time to fix the aging-related bugs. Studies on the bug reports of OSS projects have been also carried out for mobile OSes [[6]](#_bookmark12) and for several applications [[7].](#_bookmark13) According to [[7],](#_bookmark13) 5-14% of the faults are only triggered by transient conditions, while they are not further categorized into aging-related bugs or others. The bug studies for mobile OSes also counts the number of transient or intermittent bugs by 2-10% [[6]](#_bookmark12) which might include aging-related bugs. In contrast to these works, our investigation focuses on the cloud-oriented OSS projects and

shows the characteristics of aging-related bugs in detail.

Some of the aging-related bugs discussed in this paper have been already presented in the literature [[11][12]](#_bookmark17)[[13].](#_bookmark19) Kourai et al. reported the aging bugs in Xen hypervisor and proposed a fast rejuvenation technique to mitigate the software aging in VMM [[11].](#_bookmark17) Aging behavior caused by aging-related bugs in Xen is also presented in [[12].](#_bookmark18) Araujo et al. pointed out the aging issue in Eucalyptus and apply periodic rejuvenation to counter the software aging [[13].](#_bookmark19) Although our investigation somewhat overlaps these reports, we cover more wide-range of aging-related bugs and show the statistics and the trends.

1. SUMMURY

Aging-related bugs in five OSS projects for cloud computing system are investigated from the public bug reports on the bug tracking systems. Aging-related bugs are found in all the five OSS projects and the ratios of aging- related bugs range from 0.4% to 1.4%. The ratio tends to be constant in the software development lifecycle. From the observations of the resource categories affected by aging, file descriptors are the major causes of the software aging. Some aging-related bugs affect the external resources of the software and thus they are difficult to be detected in the software unit test. The tool assistance for reproducing the aging phenomena in a replicated site might be helpful to locate and fix such complex type of aging-related bugs in OSS projects.

REFERENCES

1. M. Grottke, R. Matias Jr., and K. S. Trivedi, The fundamentals of software aging, In Proc. of the 1st Int'l Workshop on Software Aging and Rejuvenation (WoSAR2008), pp.1-6, 2008.
2. F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows,

T. Chandra, A. Fikes, and R. Gruber, Bigtable: a distributed storage system for structured data. In Proc. of Symposium on Operating System Design and Implementation (OSDI 2006), pp. 205-218, 2006.

1. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud- computing system, UCSD Tech. Rep. 2008-10.
2. M. Grottke, A. P. Nikora, and K. S. Trivedi, An empirical investigation of fault types in space mission system software, In Proc. of Int'l Conf. on Dependable Systems and Networks (DSN 2010), pp. 447-456, 2010.
3. F. Machida, J. Xiang, K. Tadano and Y. Maeno, Software life- extension: a new countermeasure to software aging, accepted in the 23rd Int'l Symp. on Software Reliability Engineering (ISSRE2012).
4. A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, Characterizing failures in mobile OSes: a case study with Android and Symbian, In Proc. of Int'l Symp. Software Reliability Engineering (ISSRE2010), pp. 249- 258, 2010.
5. S. Chandra and P. M. Chen, Whither generic recovery from application faults? A fault study using open-source software, In Proc. of Int'l Conf. on Dependable Systems and Networks (DSN2000), pp. 97-106, 2000.
6. A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, Workload characterization for software aging analysis, In Proc. of Int'l Symp. on Software Reliability Engineering (ISSRE 2011), pp. 240-249, 2011.
7. R. Matias Jr., Pedro A. Barbetta, K. S. Trivedi, and P. J. Freitas Filho, Accerelated degradation tests applied to software aging experiments, IEEE Transactions on Reliability, vol. 59, no. 1, pp. 102-114, 2010.
8. R. Matias Jr., K. S. Trivedi, and P. R. M. Maciel, Using accelerated life tests to estimate time to software aging failure, In Proc. of Int'l Symp. Software Reliability Engineering (ISSRE2010), pp. 211-219, 2010.
9. K. Kourai and S. Chiba, Fast software rejuvenation of virtual machine monitor, In IEEE Trans. on Dependable Sec. Comput. vol. 8, no. 6, pp. 839-851, 2011.
10. F. Machida, J. Xiang, K. Tadano, and Y. Maeno, Combined server rejuvenation in a virtualized data center, In Proc. of Int'l Conf. on Autonomic Trusted Computing, 2012.
11. J. Araujo, R. Matos, P. Maciel, F. Vieira, R. Matias Jr., and K. S. Trivedi, Software rejuvenation in Eucalyptus cloud computing infrastructure: a method based on time series forecasting and multiple thresholds, In Proc. of the 3rd Int'l Workshop on Software Aging and Rejuvenation (WoSAR2011), 2011.