OCHEI, L.C. and EJIOFOR, C.I. 2018. Evaluating the effect of locking on multitenancy isolation for components of cloud-hosted services. Advances in science, technology and engineering systems journal, 3(3), pages 92-99. Available from: https://doi.org/10.25046/aj030312

Evaluating the effect of locking on multitenancy isolation for components of cloud-hosted services.

OCHEI, L.C., EJIOFOR, C. I.

2018

Advances in Science, Technology and Engineering Systems Journal Vol. 3, No. 3, 92-99 (2018)

www.astesj.com

Special Issue on Multidisciplinary Sciences and Engineering

ASTESJ

ISSN: 2415-6698

Evaluating the effect of Locking on Multitenancy Isolation for Components of Cloud-hosted Services

Laud Charles Ochei[*,1], Christopher Ifeanyichukwu Ejiofor[2]

[1]School of Computing and Digital Media, Robert Gordon University, United Kingdom
[2]Department of Computer Science, University of Port Harcourt, Nigeria, christopher.ejiofor@uniport.edu.ng

ARTICLE INFO

Article history:
Received: 13 April, 2018
Accepted: 11 May, 2018
Online: 31 May, 2018

Keywords:
Multitenancy
Tenant Isolation
Locking
Cloud-hosted Service
Cloud Patterns
Software Process
Bug Tracking

ABSTRACT

Multitenancy isolation is a way of ensuring that the performance, stored data volume and access privileges required by one tenant and/or component does not affect other tenants and/or components. One of the conditions that can influence the varying degrees of isolation is when locking is enabled for a process or component that is being shared. Although the concept of locking has been extensively studied in database management, there is little or no research on how locking affects multitenancy isolation and its implications for optimizing the deployment of components of a cloud-hosted service in response to workload changes. This paper applies COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing) to evaluate the impact of enabling locking for a shared process or component of a cloud-hosted application. Results show that locking has a significant effect on the performance and resource consumption of tenants especially for operations that interact directly with the local file system of the platform used on the cloud infrastructure. We also present recommendations for achieving the required degree of multitenancy isolation when locking is enabled for three software processes: continuous integration, version control, and bug tracking.

1. Introduction

Multitenancy (that is, an architectural practice of using a single instance of a service to serve multiple tenants) is a notable feature in many cloud-hosted services. Multiple users are usually expected to access a shared functionality or resource and so there is need to ensure that processes and data associated with a particular tenant and/or component does not affect others [1]. We refer to this concept as *multitenancy isolation*. Multitenancy isolation is a way of ensuring that the performance, stored data volume ad access privileges required by one tenant and/or component does not affect other tenants and/or components [1][2].

There are different or varying degrees of multitenancy isolation. For example, a higher degree of isolation would be imposed on a component that cannot be shared due to strict regulations than for a component that can be shared with minimal reconfiguration. A high degree of isolation implies that there is little or no interference between tenants when they are accessing a shared functionality/process or component of a cloud-hosted service, and vice versa. We can achieve a high degree of isolation by duplicating a component (and its supporting resources) exclusively for one tenant.

One of the conditions that can influence the degree of isolation is when locking is enabled for the functionality/process or component that is being shared. Locking is a well-known concept used in database management to prevent data from being corrupted or invalidated when multiple users try to read or write to the database [3]. Any single user can only modify items in the database to which they have applied a lock that gives them exclusive access to the record until the lock is released. The concept of locking in database management is closely related to multitenancy isolation in the sense that both of them are used to prevent multiple users from performing conflicting operations on a shared process or component and can also be implemented at different or varying degrees. Despite this similarity, there is little or no research on how locking affects multitenancy isolation and its implications for optimizing the deployment for components of a cloud-hosted service in response to workload changes.

*Corresponding Author: Laud Charles Ochei, Robert Gordon University
E-mail : l.c.ochei@rgu.ac.uk

Motivated by this problem, this paper applies COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing) to evaluate the impact of enabling locking for a shared process or component of a cloud-hosted application. This paper addresses the following research question: "How can we evaluate the required degree of multitenancy isolation when locking is enabled on a shared process or component of a cloud-hosted service?" To the best of our knowledge, this study is the first to apply an approach for implementing the required degree of multitenancy isolation for a shared process or component of a cloud-hosted service when locking is enabled and to analyse its impact on the performance and resource consumption of tenants.
In this study, we implemented multitenancy isolation based on three multitenancy patterns (i.e., shared component, tenant-isolated component, and dedicated component) to analyse the effect of the different degrees of isolation on performance and resource consumption of tenants when one of the tenants is exposed to high workload. The experiments were conducted using a cloud-hosted continuous integration system using Hudson as a case study deployed on a UEC private cloud. The results showed that when locking is enabled, it can have a significant effect on the performance and resource consumption of tenants especially for operations that interact directly with the local file system of the operating system or platform used on the cloud infrastructure.

The main contributions of the paper are:

1. Applying the COMITRE approach to empirically evaluate the required degree of multitenancy isolation for cloud-hosted software services when locking is enabled.
2. Presenting how locking is used in three different software processes (i.e., continuous integration, version control and bug tracking) to achieve multitenancy isolation, and its implication for optimal deployment of components.
3. Presenting recommendations and best practice guidelines for achieving multitenancy isolation when locking is enabled.

The rest of the paper is organised as follows: Section two discusses the relevance locking to multitenancy isolation for cloud-hosted services. Section three is the methodology, and Section four presents the results and discussion. The recommendations and limitations of the study are detailed in Section five and six respectively. Section seven concludes the paper with future work.

2. Relevance of Locking on Multitenancy Isolation for Cloud-Hosted Services

Multitenancy is an important cloud computing property where a single instance of an application is provided to multiple tenants, and so would have to be isolated from each other whenever there are workload changes. Just as multiple tenants can be isolated, multiple components being accessed by a tenant can also be isolated. We define "Multitenancy isolation" in this case as a way of ensuring that the required performance, stored data volume and access privileges of one component does not affect other components of a cloud-hosted application being accessed by tenants.

When a component of a cloud-hosted application receives a high workload and there is little or no possibility of a significant influence on other tenants, we say that there is a high degree of isolation and vice versa. The varying degrees of multitenancy isolation, can be captured in three main cloud deployment patterns: (i) dedicated component, where components cannot be shared, although a component can be associated with either one tenant/resource or group of tenants/resources; (ii) tenant-isolated component, where components can be shared by a tenant or resource instance and their isolation is guaranteed; and (iii)shared component, where components can be shared with a tenant or resource instance and are unaware of other components.

Assuming that there is a requirement for a high degree of isolation between components, then components have to be duplicated for each tenant which leads to high resource consumption and running cost. A low degree of isolation may also be required, in which case, it might reduce resource consumption, and running cost, but there is a possibility of interference when workload changes and the application does not scale well.

Most of the widely used Global Software Development processes like continuous integration (for example, Hudson), version control (for example, with Subversion) and bug tracking (for example, with Bugzilla) implement some form of locking whether at the database level or filesystem level. In continuous integration for instance, locking can be used to block builds with either upstream or downstream dependencies from starting if an upstream/downstream project is in the middle of a build or in the build queue. Again, locking operations are also used in version control systems (e.g., subversion) and bug tacking systems (e.g., bugzilla) [3] [4] [5].

There are several research work on multitenancy isolation such as [6], [7] and [8]. However, none of these works have focused on the effect of locking on multitenancy isolation for components of a cloud-hosted service.

3. Evaluation

In the following, we present the experimental setup and the case study we have used in this study.

3.1. Applying COMITRE to Implement Multitenant Isolation

We applied COMITRE to evaluate multitenancy Isolation in a Version Control system. Fig. 1 shows the structure of COMITRE. It captures the essential properties required for the successful implementation of multitenancy isolation, while leaving large degrees of freedom to cloud deployment architects depending on the required degree of isolation between tenants. The actual implementation of the COMITRE is anchored on shifting the task of routing a request from the server to a separate component (e.g., Java class or plugin) at the application level of the cloud-hosted

GSD tool. The full explanation of COMITRE plus the step-by-step procedure and the algorithm that implements it is given in [9].

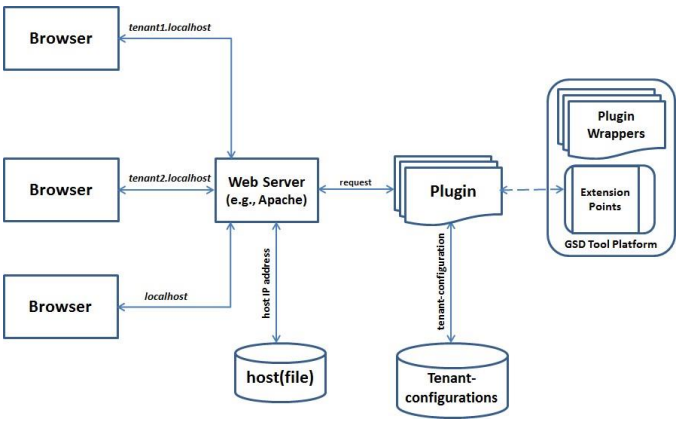

Fig. 1. COMITRE Architecture

We used a case study to evaluate the effect of tenant isolation at the data level during automated build verification/testing process for an application that logs every operation into a database in response to a specific event such as detecting changes in a file. To achieve this, we used Hudson's Files Found Trigger plugin, which polls one or more directories and starts a build if certain files are found in those directories [10]. Multitenancy isolation was implemented by modifying Hudson. This involved introducing a Java class into the plugin that accepts a filename as argument. During execution, the plugin is loaded into a separate class loader to avoid conflict with Hudson's core functionality [11].

To simulate multitenancy isolation at the data level when locking is enabled, we configured the data handling component in a way that isolates the data of different tenants (see Fig. 2). This is related to the concept of (i) locking is used in version control systems (e.g., Subversion) process to prevent clashes between multiple tenants operating on the same working copy of a file; and (ii) database isolation level which is used to control the degree of locking that occurs when multiple tenants or programs are attempting to access a database used by a cloud-hosted application. Most bugs/issue tracking applications (e.g., Bugzilla, ITracker, JIRA) use a database to store bugs [12]. Therefore, a tenant that first accesses an application component locks (or blocks) it from other tenants until the transaction commits.

3.2 Experimental Design and Statistical Analysis

A set of four tenants (T1, T2, T3, and T4) are configured into three groups to access an application component deployed using three different types of multitenancy patterns (i.e., shared component, tenant-isolated component, and dedicated component). Each pattern is regarded as a group in this experiment. We also created two different scenarios for all the tenants (see section 4.3 for details of the two scenarios). In addition, we also created a treatment for configuring T1 (see section 4.2 for details of the treatment). For each group, one of the four tenants (i.e., T1) is configured to experience a demanding deployment condition (e.g., large instant loads) while accessing the application component. Performance metrics (e.g., response times) and systems resource consumption (e.g., CPU) of each tenant are measured before the treatment (pre-test) and after the treatment (post-test) was introduced.

Based on this information, we adopt the Repeated Measures Design and Two-way Repeated Measures (within between) ANOVA for the experimental design and statistical analysis respectively. Experiments using repeated measures design make measurements using only one group of subjects, where tests on each subject are repeated more than once after different treatments [13]. The ***aim of the experiment*** is to evaluate the effect of locking on multitenancy isolation for components of cloud-hosted services. The ***hypothesis*** we are testing is that the performance and system's resource utilization experienced by tenants accessing an application component deployed using each multitenancy pattern changes significantly from the pre-test to the post test.

3.3 Experimental Setup and Procedure

The experimental setup consists of a private cloud setup using Ubuntu Enterprise Cloud (UEC). UEC is an open-source private cloud software that comes with Eucalyptus. The private cloud consists of six physical machines- one headnode and five sub-nodes. We used the typical minimal Eucalyptus configuration where all user-facing and back-end controlling components (Cloud Controller(CLC), Walrus Storage Controller, Cloud Controller (CC), and Storage Controller (SC)) are grouped on the first machine, and the Node Controller (NC) components are installed on the second physical machine. In our experiment, we installed NCs on all the other machines in order to achieve scalability for this configuration.

We use a remote client machine to access the GSD tool running on the instance via its public IP address. Apache JMeter is used as a load balancer as well as a load generator to generate workload (i.e., requests) to the instance and monitor responses. A file is pushed to a Hudson repository to trigger a build process that executes an Apache JMeter test plan configured for each tenant. Each instance is installed with SAR tool (from Red Hat *sysstat* package) and Linux du command to monitor and collect system activity information. Every tenant executes its own JMeter test plan which represents the different configurations of the multitenancy patterns.

To simulate multitenancy at the data level using JMeter, we use the JMeter Beanshell sampler to invoke a custom Java class that runs a query that sets the database transaction isolation level to SERIALIZABLE (i.e., the highest isolation level). To measure the effect of tenant isolation, we introduce a tenant that experiences a demanding deployment condition. We configured tenant 1 to simulate a *large instant load* by: (i) increasing the number of the requests using the thread count and loop count; (ii) increasing the size of the requests by attaching a large file to it; (iii) increasing the speed at which the requests are sent by reducing the ramp-up

period by onetenth, so that all the requests are sent ten times faster; and (iv) creating a heavy load burst by adding the Synchronous Timer to the Samplers in order to add delays between requests, such that a certain number of the request are fired at the same time. This treatment type is similar to unpredictable (i.e., sudden increase) workload and aggressive load.

Each tenant request is treated as a transaction composed of the 2 types of request: HTTP request and JDBC request. HTTP request triggers a build process while JDBC request logs data into the database which represents an application component that is being shared by the different tenants. Transaction controller was introduced to group all the samplers in order to get a total metrics (e.g., response) for carrying out the two requests. Figure 5 shows the experimental setup used to configure the test plan for the different tenants in Apache JMeter.

The initial setup values for experiment are as follows: (1) No of threads = 10 for tenant 1 (i.e., the tenant experiencing high load), and 5 for all other tenants; (2) Thread Loop count = 2; (3) Loop controller count = 10 for HTTP requests of tenant 1, and 5 for all other tenants; 200 for JDBC requests of tenant 1, and 100 for all other tenants; (4) Ramp-up period of 6 seconds for tenant 1 and 60 seconds for all other tenants; and (5) Estimated total number of expected requests = 250 for HTTP requests and 2500 for JDBC requests. This means that in each case the tenant experiencing high load receives two times the number of requests received by each of the other tenants. In addition, the requests are sent 10 times faster to simulate an aggressive load.

We performed 10 iterations for each run and used the values reported by JMeter and System activity report (SAR). The following system metrics were collected and analysed:

(i) CPU Usage: The %user values (i.e., the percentage of CPU time spent) reported by SAR were used to compute the CPU usage.
(ii) System load: We used the one-minute system load average reported by SAR.
(iii) Memory usage: We used the kbmemused (i.e., the amount of used memory in kilobytes) recorded by SAR.
(iv) Disk I/O: The disks input/output volume reported by SAR was recorded.
(v) Latency: The 90% latency reported by JMeter.
(vi) Throughput: We used the average throughput reported by JMeter.
(vii) Error %: The percentage of request with errors reported by JMeter.

4 Results

In this section, we discuss how the experimental results were analysed. We first performed A two-way (within-between) ANOVA to determine if the groups had significantly different changes from Pre-test to Post-test. Thereafter, we carried out *planned comparisons* involving the following: (i) a one-way ANOVA followed by Scheffe post hoc tests to determine which groups showed statistically significant changes relative to the other groups. The Dependent variable used in the one-way ANOVA test was determined by subtracting the Pre-test from Post-test values.

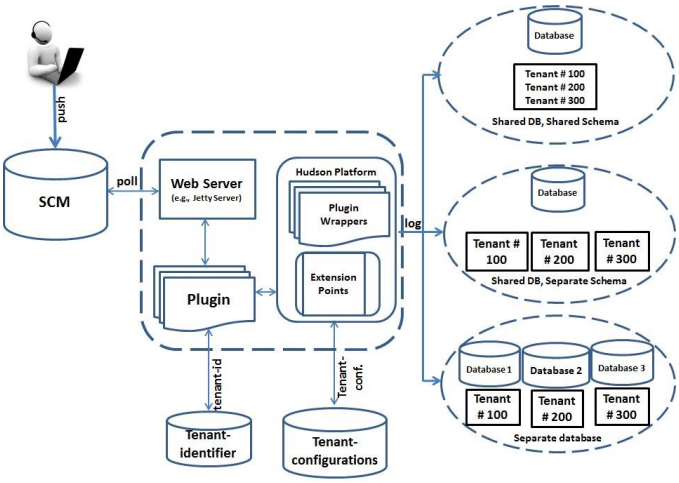

Fig. 2. Multitenancy Data Isolation Architecture

(ii) a paired sample test to determine if the subjects within any particular group changed significantly from pre-test to posttest measured at 95% confidence interval. This would give an indication as to whether or not the workload created by one tenant has affected the performance and resource utilization of other tenants. We used the "Select Cases" feature in SPSS to select the three tenants (i.e., the T2,T3,T4 that did not experience large instant loads) for each pattern.

Table 1 summarizes the effect of Tenant 1 (i.e., the tenant that experiences high load) on the other three tenants (T2, T2, T4). The key used in constructing the table is as follows: YES - represents a significant change in the metrics from pretest to post -test. NO - represents some level of change which cannot be regarded as significant; no significant influence on the tenants. The symbol "-" implies that the standard error of the difference is zero and hence no correlation and t-test statistics can be produced. This means that the difference between the pre-test and post-test values are nearly constant with no chance of variability. In the following, we present a brief discussion the findings of the study based on the estimate of the marginal means of change and paired sample test for scenario 1 and scenario 2.

(1) Response times and Error%: The paired sample test result shows that the response times of tenants changed significantly only for the dedicated pattern. A further analysis of the EMMC showed that the dedicated pattern had a much larger magnitude of change than all the other patterns. The Error% showed that there was no significant change in the tenants within any of the patterns; there was either no significant difference or no variability.

(2) Throughput: The results of the paired sample test showed that the tenants within all the patterns changed significantly from pre-

test to post-test. The shared component showed the smallest magnitude of change based on the plots of the EMMC.

(3) CPU: The plots of the EMMC showed that the shared component had the largest magnitude of change. The other two patterns were nearly the same. The paired sample test showed that shared component was the only pattern that changed significantly.

(4) Memory: The plot of the EMMC showed that the shared component changed showed the smallest magnitude of changed. We noticed an interesting trend in the sense that magnitude of change decreased steadily from the shared component to the dedicated component. The paired sample test showed that tenants deployed based on all the patterns changed significantly.

(5) Disk I/O: The paired sample test showed that there was no significant change between the tenants deployed based on the shared pattern. The plots of the Estimated Marginal Means of changed (EMMC) confirmed that the shared component changed the least.

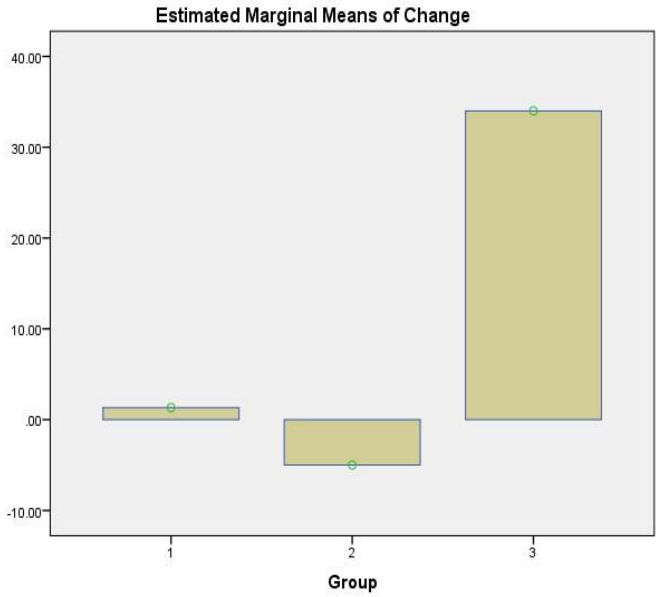

Fig. 3. Changes in response time

(6) System Load: The paired sample test showed that there was no significant influence on the system load for all the patterns. This means that even when locking is enabled the system load is not likely to change much.

5 Discussion

(1) CPU: The results showed that the CPU did not change significantly, except for the shared component. This implies that apart from the shared component, the degree of isolation was high. Therefore, we can say that although locking for enabled, there appears to be little or no influence in terms of resource consumption. This is understandable because Hudson, like many builders, do not consume much CPU.

(2) System Load: As the results show, the system load of the tenants showed either a nearly constant magnitude of change or no chance of variability. This means that even when locking is enabled, there may be no significant change in the system load as long as the size of the processor is large enough to cope of the number of piled-up requests.

(3) Memory: Builders are well known to consume a lot of memory, especially when handling difficult and complex builds. As the results showed, there was a significant difference between the tenants for all the patterns when locking was enabled. Overall, this means that there was a low degree of isolation between the tenants. In terms of the magnitude of change, the plots of EMMC showed the largest magnitude of change while dedicated component was the smallest. This implies that while the shared component is not recommended to minimize performance, but it may be used optimize the memory usage. On the other hand, the dedicated component can be used to avoid performance interference.

(4) Disk I/O: Compilers and builders generally consume a lot of disk I/O and it interacts directly with the operating system or the filesystem of the cloud platform used. As shown in the paired sample test result, tenants deployed based on shared component did not change significantly, implying a high degree of isolation. Therefore, when locking is enabled on an application component that is shared while carrying out I/O intensive builds, then the shared component would be recommended. The plots of the EMMC, confirms this position in the sense that the shared component showed the smallest magnitude of change out of the three patterns.

Table I. Paired Samples Test Analysis of Multitenancy Isolation When Locking is enabled

Pattern	Response times	Error%	Throughput	CPU	Memory	Disk I/O	System Load
Shared	No	No	Yes	Yes	Yes	No	-
Tenant-isolated	No	-	Yes	No	Yes	Yes	-
Dedicated	Yes	-	Yes	No	Yes	-	-

(5) Response times and Error%: The results show that the dedicated component had the largest magnitude of change for response times, while the reverse was the case for error% which had the largest magnitude of change for the shared component. This means that the shared component would not be recommended for preventing performance interference. It also shows that there would be a high possibility of requests timing out for tenants deployed based on shared component than for other tenants. A possible explanation for this is that requests can be delayed or blocked while trying to gain access to the shared application component.

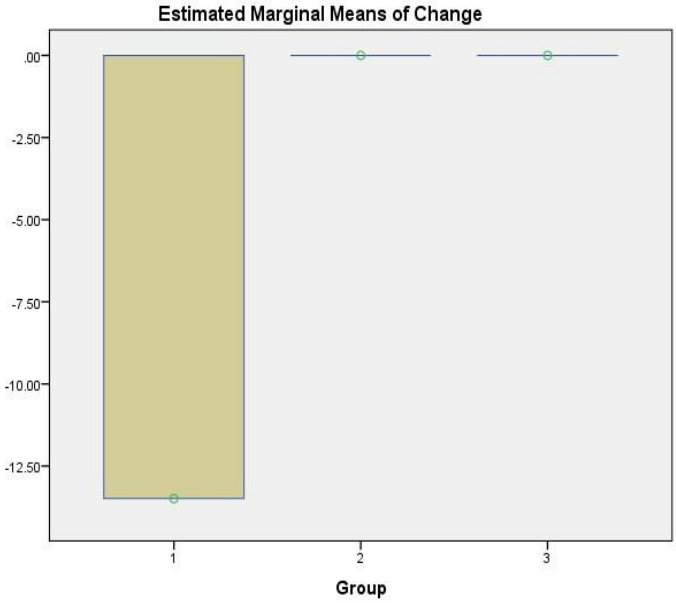

Fig. 4. Changes in error%

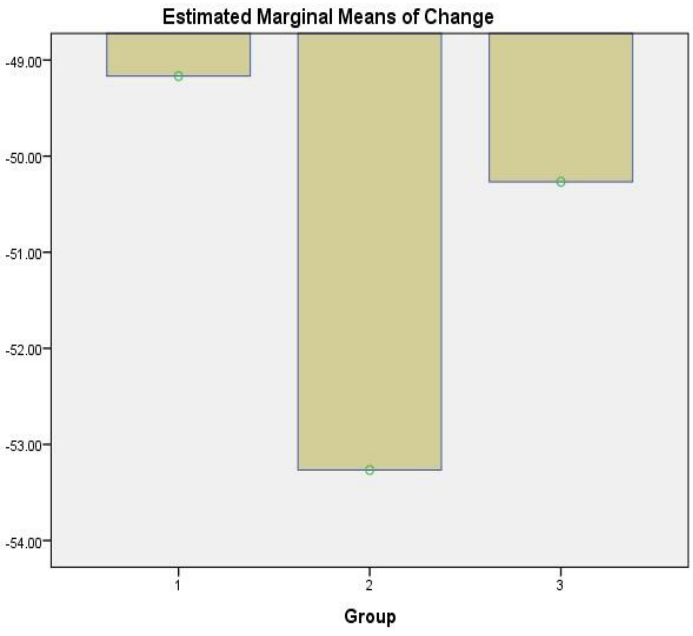

Fig. 5. Changes in throughput

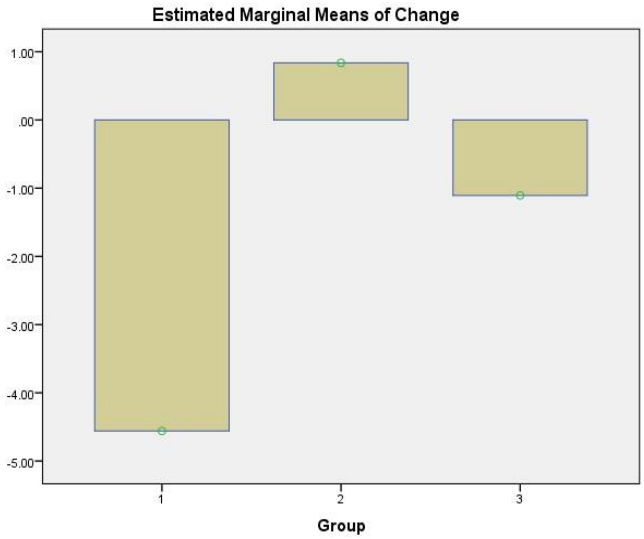

Fig. 6. Changes in CPU

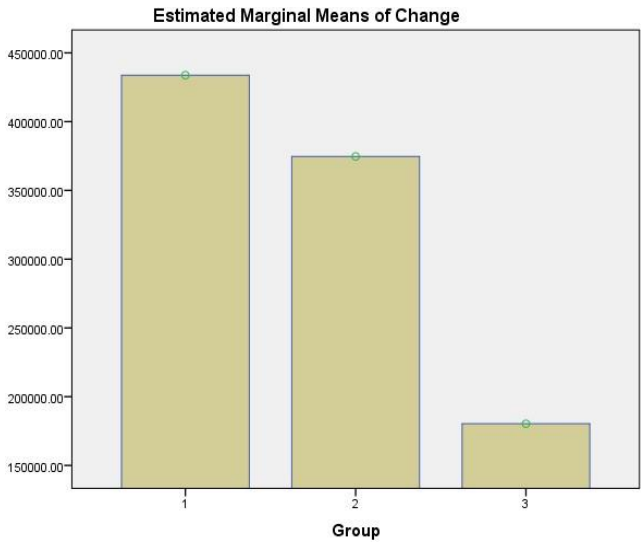

Fig. 7. Changes in memory

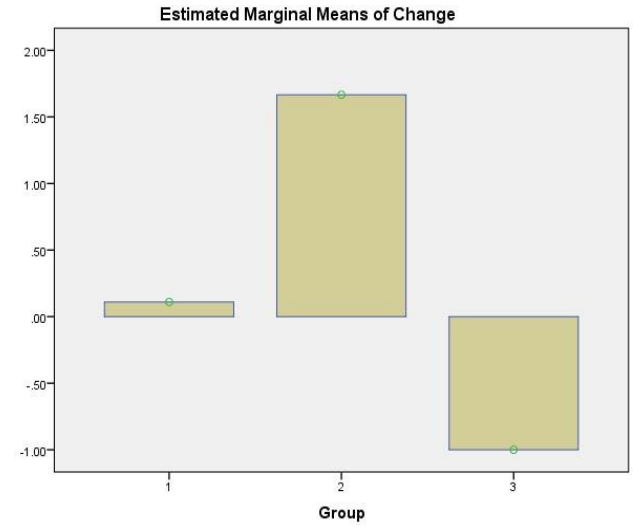

Fig. 8. Changes in disk I/O

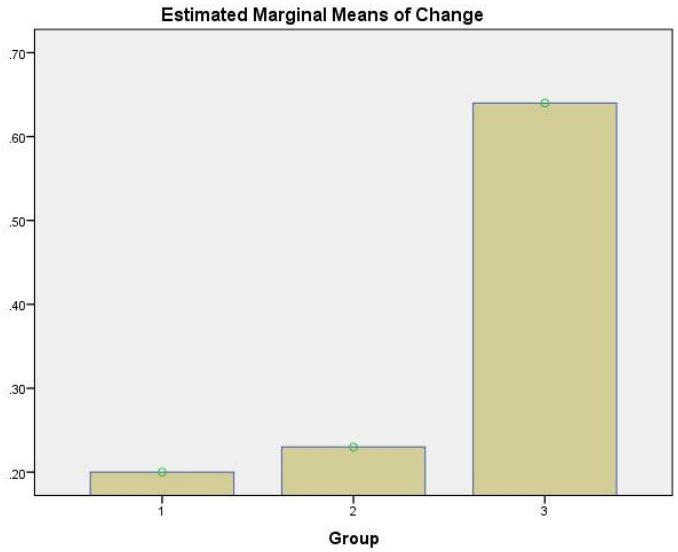

Fig. 9. Changes in system load

6. Recommendations and Limitations

The experimental results show that locking could have a significant effect on multitenancy isolation. Running a complete integration build in a slow network environment could take a lot of time and resources. To achieve the required degree of isolation, we recommend splitting the integration build into different stages and implement separate multitenancy patterns for each phase. For example, we could (i) creating a commit build that compiles and verifies the absence of critical errors when each developer commits changes to the main development stream based; and (ii) creating a secondary build(s) to run slower and less important tests. This study assumes that a small number of tenants send multiple requests to an application component deployed on a private cloud. The number of requests sent to the application component configured within Hudson was within the limit of the UEC private cloud used. Therefore, the results of this study should not be generalized to large public clouds.

7. Application of Locking on Cloud-hosted Software Development Tools and Associated Processes

A well-managed locking strategy is required to deal with real-time tightly synchronized/consistency-critical cloud applications such as such graph processing, financial applications, and real-time enterprise analysis applications. These cloud-hosted applications rely heavily on key software development processes such as continuous integration, version control and bug/issue tracking to build, test, and release software faster and more reliably.

Lock management in a multitenant cloud-hosted application is essential because if an architect misses placing a lock where required, then safety is violated. In contrast, if an architect inserts unneeded locks in a cloud-hosted application, then the performance of the system suffers due to the unnecessary synchronizations [14]. In the following, we discuss how locking is used in three important types of software development processes, and some recommendations to follow regarding achieving the required degree of multitenancy isolation.

7.1. Locking in Continuous Integration process

Locking is a very important operation in a typical continuous integration process. For example, in Hudson, it is used to block builds dependencies from starting if an upstream or downstream project is in the build queue. One implication of this is that if there is a presence of piled-up requests/builds on the queue, then the system load is likely to be affected. This was not the case in the experiments and so the system load was nearly constant with no chance of variability.

We recommend that in order to optimize resources that support a cloud-hosted service while at the same time guaranteeing multitenancy isolation, the architect should avoid certain operations lock processes for a long time, especially when there is either limited resources or frequent workload changes. Such operations include carrying out difficult and complex builds (i.e., builds that have many interdependencies with other programs or systems), and (ii) running a large number of builds concurrently.

7.2. Locking in Version Control process

Locking (similar to the “reserved checkouts” mechanism) is used internally in version control process (e.g., in Subversion) to achieve mutual exclusion between users to avoid clashing commits or to prevent clashes between multiple tenants operating on the same working copy. A Version control system can be setup to use a database as its backend. For example, it is common for architects to setup subversion to store data in a Berkeley DB database environment. When this is the case, locking can be used internally by the Berkeley DB to prevent classes between multiple processes and programs trying to access the database.

With respect to multitenancy isolation, when multiple tenants are accessing a shared version control repository, it implies a shared component is being used for deployment. Under this situation, it is possible for fatal errors or interruptions to occur which can prevent a process from having the chance to remove the locks it has placed in the database. While implementing dedicated component deployment would be an obvious solution to avoid such interferences, one would have to go a step further when working with networked repository. This could involve putting in place an off-site backup strategy, and shutting down server programs (e.g., Apache HTTP server) from accessing or attempting to access the repository.

When using a version control system such as subversion that implements locking, fetching large data remotely and finalizing a commit operation can lead to unacceptably slow response times and can even cause tenants request to time out. Therefore, having the repository together with the working copy located on your machine is beneficial. It is also important to note that file locking

along with data compression are some of the operations that could consume resources, especially when accessing a shared repository from a client with a slow network and low bandwidth.

7.3. Locking in Bug tracking process

A bug tracking system is used to keep track of reported software bugs in software development projects. A major component of a bug tracking system is the storage component that records facts about known bugs. Depending on the type of storage component used to store bugs, locking can be used to prevent multiple tenants trying to access the bug data store.

Most bug and issue tracking systems (e.g., Bugzilla and JIRA) use a database to store bugs. Enabling locking on the bug database, for example, can also increase resource consumption (e.g., CPU, memory), especially when running long transactions, running complex transactions concurrently or transferring large bug attachments across a slow network connection.

8. Conclusion and Future Work

In this paper, we have presented the effect of locking on multitenancy isolation for components of a cloud-hosted service to contribute to literature on multitenancy isolation and cloud deployment of application components. The study revealed that when locking is enabled for components of a cloud-hosted service, it can have a significant impact on the performance and resource consumption of tenants especially for operations that interact directly with the local file system (e.g., FAT, NTFS, GoogleFS**,** HFS+) of the platform on which the service is hosted. One option we have recommended is to split a software process (e.g., a long build process) into separate phases and then implement different degrees of isolation for each phase.

We plan to apply our approach to implementing multitenancy isolation for a cloud-hosted service in a distributed scenario where locking is enabled for all or some of the components at different of the cloud stack. For example, in distributed bug tracking some bug trackers like Fossil and Veracity are either designed to use (or integrated with) distributed VC or CI systems, thus allowing bugs to be created automatically and inserted to the database at varying frequencies.

References

[1] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns*. Springer, 2014.

[2] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[3] B. Collins-Sussman, B. Fitzpatrick, and M. Pilato, *Version control with subversion*. O'Reilly, 2004.

[4] M. Moser and T. O'Brien. The Hudson book. Oracle, Inc., USA. Online: accessed in November, 2017 from http://www.eclipse.org/hudson/the-hudsonbook/book-hudson.pdf.

[5] Bugzilla.org. The bugzilla guide. The Mozilla Foundation. [Online: accessed in November, 2017 from http://www.bugzilla.org/docs/.

[6] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications." *CLOSER*, vol. 12, pp. 426–431, 2012.

[7] S. Strauch, V. Andrikopoulos, F. Leymann, D. Muhler, "Esbmt: Enabling multi-tenancy in enterprise service buses," *CloudCom*, vol. 12, pp. 456–463, 2012.

[8] S. Walraven, T. Monheim, E. Truyen, and W. Joosen, "Towards performance isolation in multi-tenant saas applications," in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*. ACM, 2012, p. 6.

[9] L. C. Ochei, J. Bass, and A. Petrovski, "Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools," in *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 2015, pp. 101–112.

[10] Hudson. Apache software foundation. [Online: accessed in January 2017 from http://wiki.hudsonci.org//display/HUDSON/Files+Found+Trigger].

[11] L. C. Ochei, A. Petrovski, and J. Bass, "Evaluating degrees of isolation between tenants enabled by multitenancy patterns for cloud-hosted version control systems (vcs)," *International Journal of Intelligent Computing Research*, vol. 6, Issue 3, pp. 601 – 612, 2015.

[12] Serrano, N. and Ciordia, I., 2005. Bugzilla, ITracker, and other bug trackers. *IEEE software*, *22*(2), pp.11-13.

[13] Verma, J.P., 2015. Repeated measures design for empirical researchers. John Wiley & Sons.

[14] Demirbas, M., Tasci, S. and Kulkarni, S., 2012, July. Maestro: A cloud computing framework with automated locking. In *Computers and Communications (ISCC), 2012 IEEE Symposium on* (pp. 000833-000838). IEEE.