OpenAIR @RGU
ROBERT GORDON UNIVERSITY ABERDEEN

This publication is made freely available under GREEN open access.

AUTHOR(S): OCHEI, L.C., PETROVSKI, A. and BASS, J.M.

TITLE: Evaluating degrees of tenant isolation in multitenancy patterns: a case study of cloud-hosted version control system (VCS).

YEAR: 2015

Publisher citation: OCHEI, L.C., PETROVSKI, A. and BASS, J.M. 2015. Evaluating degrees of tenant isolation in multitenancy patterns: a case study of cloud-hosted version control system (VCS). In Proceedings of the 2015 international conference on information society (i-Society), 9-11 November 2015, London, UK. Piscataway, NJ: IEEE [online], pages 59-66. Available from: http://dx.doi.org/10.1109/i-Society.2015.7366859

OpenAIR citation: OCHEI, L.C., PETROVSKI, A. and BASS, J.M. 2015. Evaluating degrees of tenant isolation in multitenancy patterns: a case study of cloud-hosted version control system (VCS). In Proceedings of the 2015 international conference on information society (i-Society), 9-11 November 2015, London, UK. Piscataway, NJ: IEEE, pages 59-66. Held on OpenAIR [online]. Available from: https://openair.rgu.ac.uk

Publisher copyright statement:

This is the AUTHOR ACCEPTED version of proceedings originally published by IEEE and presented at 2015 international conference on information society (i-Society), 9-11 November 2015, London, UK. (ISBN 9781908320483; eISBN ; ISSN).

©2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

OpenAIR takedown statement:

Section 6 of the "Repository policy for OpenAIR @ RGU" (available from http://www.rgu.ac.uk/staff-and-current-students/library/library-policies/repository-policies) provides guidance on the criteria under which RGU will consider withdrawing material from OpenAIR. If you believe that this item is subject to any of these criteria, or for any other reason should not be held on OpenAIR, then please contact openair-help@rgu.ac.uk with the details of the item and the nature of your complaint.

This publication is distributed under a CC BY-NC-ND 4.0 license.

https://creativecommons.org/licenses/by-nc-nd/4.0

OpenAIR at RGU

Digitally signed by OpenAIR at RGU
DN: cn=OpenAIR at RGU, o=Robert Gordon University, ou=Library, email=publications@rgu.ac.uk, c=GB
Date: 2016.05.26 14:34:05 +01'00'

Evaluating Degrees of Tenant Isolation in Multitenancy Patterns: A Case Study of Cloud-hosted Version Control System (VCS)

Laud Charles Ochei, Andrei Petrovski
School of Computing Science and Digital Media
Robert Gordon University
Aberdeen, United Kingdom
Emails: {l.c.ochei,a.petrovski}@rgu.ac.uk

Julian M. Bass
School of Computing, Science and Engineering
University of Salford
Manchester, United Kingdom
Email: J.Bass@salford.ac.uk

***Abstract*—One of the key concerns of implementing multitenancy (i.e., serving multiple tenants with a single instance of an application) on the cloud is how to enable the required degree of isolation between tenants, so that the required performance of one tenant does not affect other tenants. There is little research which provides empirical evidence on the required degree of isolation between tenants under different cloud deployment conditions. This paper applies COMITRE (COmponent-based approach to Multitenancy Isolation Through request RE-routing) to empirically evaluate the degree of isolation between tenants enabled by multitenancy patterns for cloud-hosted Version Control System (VCS). We implemented three multitenancy patterns (i.e., shared component, tenant-isolated component, and dedicated component) by developing a multitenant component using the FileSystem SCM plugin integrated within Hudson. The study revealed that dedicated component provides the highest degree of isolation between tenants (compared to shared component and tenant-isolated component) in terms of error% (i.e., the percentage of errors with unacceptably slow response times) and throughput. System load of tenants showed no variability, and hence did not influence the degree of tenant isolation for all the three multitenancy patterns. We also provide some recommendations to guide an architect in implementing multitenancy isolation on similar VCS tools like Subversion and CVS.**

Keywords*—Multitenancy, Degree of Isolation, Tenant, GSD-tools, Cloud Deployment Pattern.

I. Introduction

One of the key challenges of implementing multitenancy is how to enable tenant isolation (hereafter referred to as *multitenancy isolation*) between tenants sharing components of an application, for example, cloud-hosted application [1][2]. As software tools are increasingly being deployed on the cloud for software development, there is need to properly isolate tenant's code files and processes so that the required performance, stored volume, and access privileges of one tenant does not affect other tenants.

There are varying degrees of isolation between tenants when sharing application components. For example, special configurations of individual tenants, and laws and corporate regulations may imposed a higher degree of isolation between tenants sharing a particular component. The challenge for a cloud deployment architect would be how to select the right multitenancy patterns(or combinations of patterns) to resolve the trade-offs between the required performance, systems resources and access privileges at different levels of a cloud-hosted application.

Motivated by this problem, this paper applies COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing (COMITRE) [3] to empirically evaluate the degree of isolation between tenants enabled by multitenancy patterns under different cloud deployment conditions. Fehling et al [1], captured the degree of isolation between tenants in three multitenancy patterns, and also proposed that the degree of isolation between tenants is the main difference between these patterns. However, these patterns have never been evaluated to measure the actual degree of tenant isolation for applications within the domain of cloud-hosted VC systems such as Subversion, CVS, and Perforce. Version control is a key software development practice used to support teams involved in Global Software development [4][3] [5].

The research question this paper addresses is: **"How can we evaluate the degree of isolation between tenants enabled by multitenancy patterns for cloud-hosted Version Control System"**. By evaluating the degrees of multitenancy isolation, we mean comparing the effect of performance (e.g., response times) and resource utilization (e.g., CPU) on tenants deployed based on different multitenancy patterns when one of the tenants experiences high workload. Three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) were implemented by exposing the functionality of each pattern as a plugin integrated with Hudson deployed on a private cloud (i.e., Ubuntu Enterprise Cloud-UEC). Thereafter, we evaluated the degree of isolation for each pattern both at the process isolation and data isolation levels, as it affects tenants interaction with Version control system.

The main contributions of this paper are:

1. Applying COMITRE to implement multitenancy isolation for cloud-hosted version control system.
2. Evaluating empirically the degree of isolation between tenants enabled by multitenancy patterns under different cloud deployment conditions.
3. Presenting recommendations and best practice guidelines to guide a cloud deployment architect when implementing multitenancy isolation on a cloud-hosted Version Control system.

The rest of the paper is organized as follows - section two gives an overview of the basic concepts related to deployment patterns for Cloud-hosted GSD tools, with particular reference to multitenancy patterns, and tenant isolation. In section three, we discuss the research methodology including GSD tool selection and the development of an approach for implementing multitenancy isolation. Section four presents the evaluation which covers the case study, experimental setup and procedure. In section five, we present the results of the study and then discuss the implications of the results in section six. The recommendations and limitations of the study are detailed in section seven and eight respectively. Section nine concludes the paper with future work.

II. Multitenancy Patterns for Cloud-hosted GSD Tools.

A. Cloud-hosted GSD Tool and Software Processes.

Definition 1: **Global Software Development.** Global Software Development means the splitting of the development of the same software product or service among globally distributed sites [6].

Definition 2: **Cloud-hosted GSD tools.** "Cloud-hosted GSD tools" are collaboration tools used to support GSD processes in a cloud environment [5]. We adopt the: (i) NIST Definition of Cloud Computing to define properties of cloud-hosted GSD tools; and (ii) ISO/IEC 12207 as a classification frame for defining the scope of a GSD tool. Three examples of widely used Global software development processes are: continuous integration, version control and issue/error tracking [4] [5]. In the next section, we will discuss about version control which is the focus of this paper.

B. Relevance of Version Control Process in Global Software Development

Definition 3: **Version Control.** Version control is the process of tracking incremental versions of files and, in some cases, directories over time, so that specific versions can be recalled later [7]. In Global software development, version control systems are being relied upon as a communication medium for developers in a software development team. For example, viewing past revisions and changesets is a valuable tool to see how your project has evolved and for reviewing teammates code.

There are two main categories of version control systems: **centralized** (e.g., Subversion) and **distributed** (Git and Mercury). This paper focuses on the centralized version control system, which works in a client and server relationship. That is, the repository is located in one place and provides access to many clients. It can be likened to a scenario where an FTP client connects to an FTP server. All changes and commits by users are sent and received from the central repository.

C. Cloud Deployment Patterns for Multitenancy Isolation

Definition 4: **Cloud deployment patterns.** "Cloud deployment patterns" are architectural patterns which embodies decisions as to how elements of the cloud application will be assigned to the cloud environment where the application is executed [5]. The notion of *Cloud deployment pattern* is similar to the concept of (architectural) deployment patterns [8], cloud computing patterns [1]. Architectural and design patterns have long been used to provide known solutions to a number of common problems facing a distributed system [9, 8].

Definition 5: **Multitenancy isolation.** We define "Multitenancy isolation" as a way of ensuring that the required performance, stored data volume and access privileges of one tenant does not affect other tenants accessing the component/functionality of a shared application component.

Definition 6: **Application Component.** We present an informal definition of an "Application Component" as an encapsulation of a functionality that is shared between multiple tenants. An application component could be a communication component (e.g., message queue), data handling component (e.g., databases, tables), processing component (e.g., load balancer), or a user interface component (e.g., AJAX).

D. Evaluating Degree of Multitenancy Isolation

Multitenancy isolation can be captured in three main cloud patterns: shared component (i.e., tenants share the same resource instance, and are unaware of other tenants), tenant-isolated component (tenants share the same resource and their isolation is guaranteed) and dedicated component (i.e., tenants do not share resource, though each tenant is associated with one instance (or certain number of instances) of the resource) [1].

The three main aspects of tenant isolation are: performance, stored data volume and access privileges [1]. For example, in performance isolation, other tenants should not be affected by the workload created by other tenants. Any of the three multitenancy patterns can be used to achieve varying degrees of isolation between tenants. The dedicated component gives the highest degree of isolation but at a high running cost and high resource consumption. The shared component gives the lowest degree of isolation but allows for better resource sharing leading to better resource utilization.

The lack of performance guarantee (i.e., performance isolation) is one of the major challenges facing users of cloud-hosted applications [10]. Guo et al [11] evaluated different isolation capabilities related to authentication, information protection, faults, administration etc. A closely related work to ours is that of Walraven et al [12] where they implemented a middleware framework for enforcing performance isolation. He used a multitenant implementation of a hotel booking application deployed on top of a cluster for illustration. Krebs et al [13] implemented a multitenancy performance benchmark for web application based on the TCP-W benchmark. Other works related to multitenancy isolation can be found in [2] [14] [15]

Krebs et al [16] acknowledged that performance related issues are often caused by a minority of tenants with high workloads. The focus of this paper is providing empirical evidence of the effect of performance and resource utilization on other tenants due to high workload created by one of the tenants. We implemented multitenancy component using the FileSystem SCM plugin integrated into Hudson in a real cloud environment. The implementation represents a typical cloud deployment of a version control process based on multitenancy patterns.

III. Methodology

A. Selecting the GSD Tools and Software Processes

There are several software processes that have been found to have the most impact on Global Software Development. Examples of three key processes are: continuous integration, source/version control and issue/bug tracking [5, 17]. We conducted an empirical study in a previous study to select three open-source GSD tools to represent these software processes: Hudson [18], Subversion [7] and Bugzilla [19]. The empirical study was conducted to find out: (1) the type of GSD tools used in large-scale distributed enterprise software development projects; and (2) what tasks/software processes they utilize the GSD tools for. See Ochei et al [5] and Bass [17] for details. This paper focuses on applying our approach (i.e., COMITRE) to implement multitenancy in a version control system.

B. 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 can be seen in Ochei et al [3].

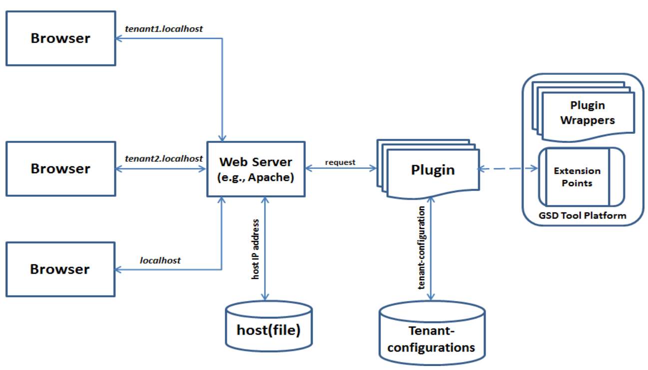

Figure 1. COMITRE Architecture

C. Validating the Implementation of Multitenancy Isolation

We validate our approach (i.e., COMITRE) for implementing multitenancy isolation both in theory and in practice. We first validated each multitenancy pattern in theory as follows: (i) carefully analyzed the class diagrams and description of the implementation of the three multitenancy pattern as presented by Fehling et al [1] and other related sources [20] [21]; (ii) systematically cross-checked our implementation against that proposed by other researchers; and (iii) Examined that our implementation is compliant with how clients (i.e., tenants) access a multitenant component.

We also demonstrate the practicality of our approach by applying it to implement the three multitenancy patterns on FileSystem SCM plugin integrated within Hudson., a widely used open-source GSD tool for continuous integration. Experts and researchers in the field of cloud deployment patterns and Global Software Development have confirmed that the implementation of multitenancy isolation together with the output represents the behaviour of tenants interacting with a shared functionality/component of a cloud-hosted application.

IV. Evaluation

In the following, we present the experimental setup and the case study we have used in this study. This paper uses the File System SCM plugin to illustrate the version control process because we want to simulate the process on a local development machine. Specifically, we want to point the build configuration to the locally checked out code and modified files on a shared repository residing on a private cloud. Filesystem SCM plugin can be used to simulate the file system as a source control management (SCM) system by detecting changes such as the file system's last modified date [22]. This plugin can be integrated into several GSD tools: continuous integration systems (e.g., Hudson), version control systems (e.g., perforce, Git) and error/issue tracking system (e.g., JIRA). Another plugin that can be used within Hudson for this experiment is SVNKit, a Java software library for working with Subversion.

In our experiments, we integrated this plugin into Hudson because we are assuming a scenario where a code file is checked into a shared repository for Hudson to build. We implemented multitenancy isolation by modifying this plugin within Hudson. This involved introducing a Java class into the plugin that accepts a file path and the type of file(s) that should be included when checking out from the repository into Hudson workspace. During execution, the plugin is loaded into a separate class loader to avoid conflict with Hudson's core functionality.

A. Experimental Setup and Procedure

1) Experimental Setup: The experimental setup consist of a private cloud setup using Ubuntu Enterprise Cloud (UEC), an open-source private cloud software that comes with Eucalyptus. The private cloud consist of six physical machines-one headnode and five sub-nodes based on the typical minimal Eucalyptus configuration [23].

2) Experimental Design: 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 IV D for details of the two scenarios). In addition, we also created a treatment for configuring T1 (see section IV D 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 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, as previously used by Ochei et al [3]. The ***aim of the experiment*** is to evaluate the degrees of isolation of multitenancy patterns for cloud-hosted Version Control system. 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) Experimental Procedure: In our experiments, we implemented multitenancy isolation, by modifying the Filesystem SCM plugin integrated within Hudson for handling tenant's request to its shared component. A summary of the experimental procedure we adopted can be seen in Ochei et al [3].

A typical version control process during Global Software Development involves a combination of continuous integration (i.e., building a code file), checkouts (i.e., file download), checkins (i.e., file upload), and updating and synchronizing files with the latest version from the repository. A detailed experimental procedure considered in this paper translates into the following steps:

1. The first step is to put a new file to the repository for the first time. To achieve this, we used the HTTP request sampler in JMeter to send request to Hudson to trigger a build. Within Hudson, we used the "Execute Shell" feature to execute a shell script. This shell script simply selects the initial contents of a MySQL database (i.e., used here to represent a shared data handling component) and then outputs it into two separate files (referred to as *file1* and *file2*). The first file (i.e., *file1*) represents the local working copy and the second file (i.e., *file2*) represent the main copy.

2. The second step is to check out the copy of the new file to the local machine. To implement this in JMeter, we used the FTP request sampler and then selected the get (RETR) to download the file from the repository. In effect, this action downloads *file1* from the repository into a local machine and saves it as *file3*.

3.The third step involves making changes to the file by inserting records into the Mysql database and then outputting the latest content to the local working copy. To simulate this we used a BeanShell Sampler in JMeter to invoke a custom Java class. This Java class is specifically written to insert records into Mysql database and then updates *file3* with the latest content of the database.

4.The last step is to checkin *file3* back into the repository with a timestamp message ("Row added at 2015-01-01-00.00.01"). To implement this in JMeter, we again used the FTP request sampler and then selected the put (STOR) to upload the file to the repository and appends the content to *file2*.

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 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 request are sent by reducing the ramp-up period by one-tenth 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 request are fired at the same time. This treatment type is similar to unpredictable (i.e., sudden increase) workload [1] and aggressive load [12].

Each tenant request is treated as a transaction composed of the three types of request: HTTP request, FTP request, and File I/O operation. JMeter Transaction controller was introduced to take the aggregate measurement of all the requests involved in the end-to-end action sequence of the scenario. The setup values for experiment are as follows: (1) No of threads = 2; (2) Thread Loop count = 5; (3) Loop controller count = 4 for tenant 1, and 2 for all other tenants for each type of request sent (i.e., HTTP request, Beanshell, and FTP request samplers); (4) Ramp-up period of 6 seconds for tenant 1 and 60 seconds for all other tenants; and (5) Total number of expected requests = 480. With this setup, it means tenant 1 sends two times the number of requests of the other tenants, and also 10 times faster to simulate an aggressive load. We performed 10 iterations for each run and used the values reported by JMeter as a measure for response times, throughput and error%. The error% is computed as the percentage of the total number of request (i.e., in the end-to-end sequence of version control process) whose response time is unacceptably slow and above which the request is considered a failure. Statistically, this translates to response time greater than the upper bound of the 95% confidence interval of the average response time of all requests. For system activity, we reported the average CPU, memory, disk I/O and system load usage at one-second interval.

B. Case Studies of Multitenancy Isolation

We present two case studies to evaluate the effect of multitenancy isolation at both data level and process level during an automated version control process. Fig. 2 captures the architecture of multitenancy Isolation at the data level. For multitenancy isolation at the process isolation, the component that is being shared is a lock object [3]. The two case studies are explained as follows:

Case Study 1 - Data Isolation during Version Control: To achieve this, we configured the data handling component in a way that isolates the data of different tenants (see Fig. 2). In our experiments, this case study maps to Scenario 1-Variation in request arrival rate. It represents a case where there is variation in the frequency with which code changes are committed to the source code to trigger a build process. To simulate this behaviour in JMeter, we simply add the Gaussian Random Timer to the Samplers.

Case Study 2 - Process Isolation during Version Control: To achieve this, we introduce the concept of database isolation level which is used to control the degree of locking that occurs when inserting data into a database [24]. In our experiments, this case study maps to Scenario 2-Lock duration. It represents a case where a tenant that first accesses an application component locks (or blocks) it from other tenants until the transaction commits. To simulate this behaviour, we used 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).

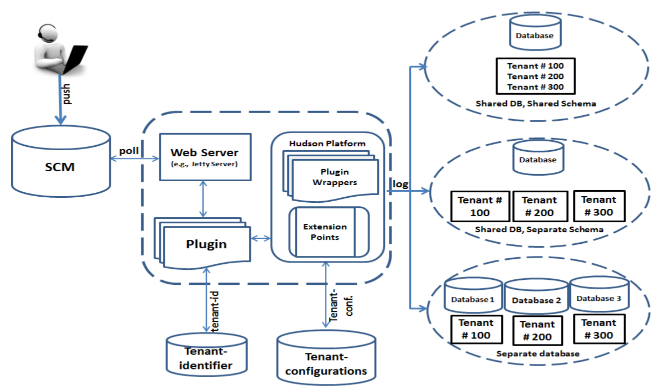

Figure 2. Multitenancy Data Isolation Architecture

V. Results

We first performed the 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:

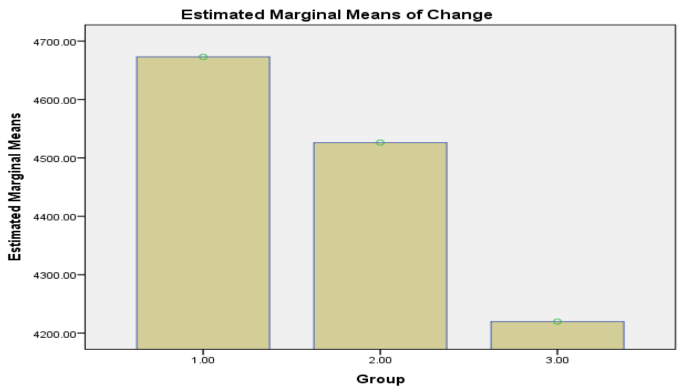

Figure 3. Changes in response time for each pattern relative to other patterns-1

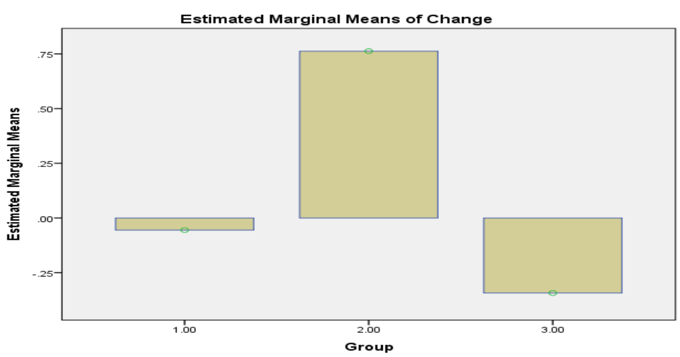

Figure 4. Changes in error% for each pattern relative to other patterns-1

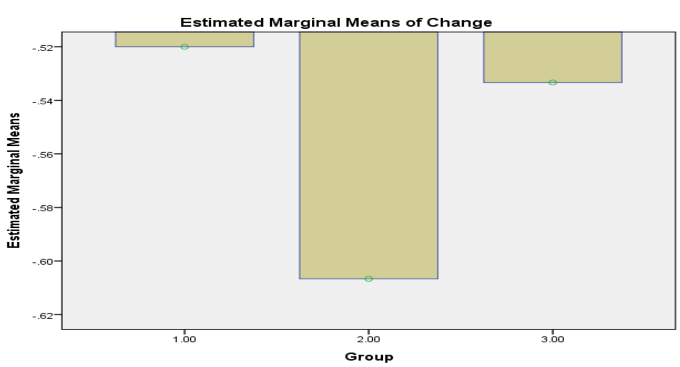

Figure 5. Changes in throughput for each pattern relative to other patterns-1

(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.
(ii) a paired sample test to determine if the subjects within any particular group changed significantly from pre-test to post-test 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 and for each deployment scenario giving a total of 6 cases for each metrics

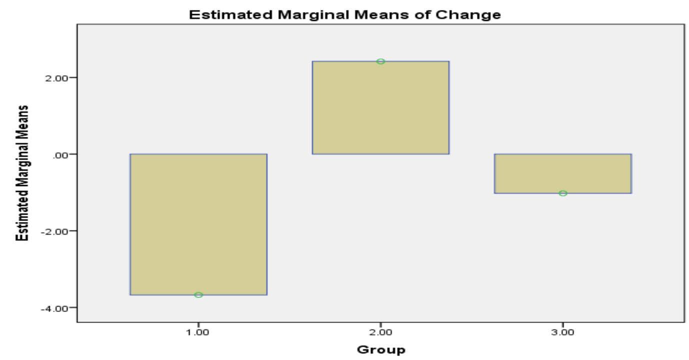

Figure 6. Changes in CPU for each pattern relative to other patterns-1

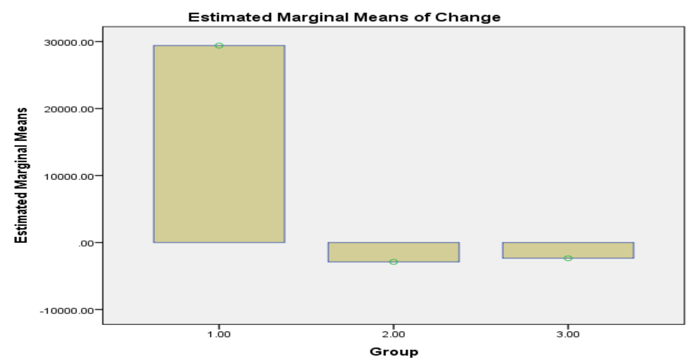

Figure 7. Changes in memory for each pattern relative to other patterns-1

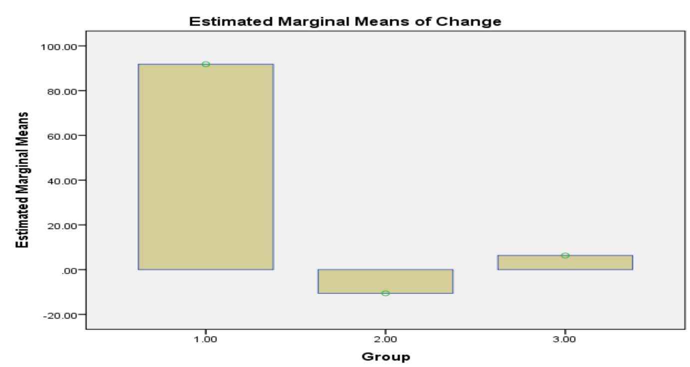

Figure 8. Changes in disk I/O for each pattern relative to other patterns-1

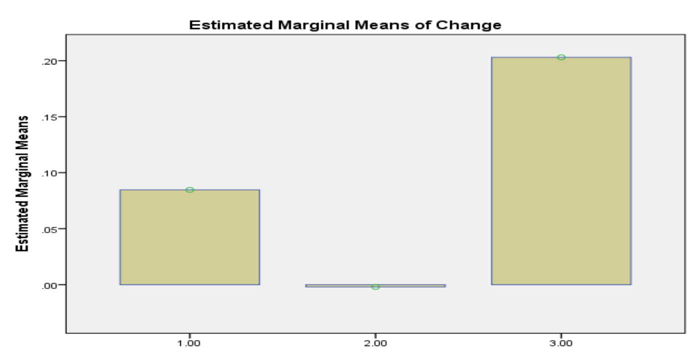

Figure 9. Changes in system load for each pattern relative to other patterns-1

that was measured.

To answer the questions outlined above, we analyzed the plots of estimated marginal means of change shown in Fig. 3 to Fig. 9 in combination with ANOVA (plus post hoc test) and paired sample test results from SPSS output. *The quasi-independent variable is nominally scaled in SPSS, and so we changed the interpolation line to a bar chart to give a meaningful interpretation of the result*. 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 between the metrics from pre-test 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 values are nearly constant with no chance of variability. *Due to space limitations, we show only the Estimated Marginal Means of Change of the measured parameters for scenario 1*.

(1) Response times: The Post hoc test revealed that none of the patterns showed significant change relative to the other patterns. However, Table 1 (i.e., the paired sample t test) showed that the response times of tenants changed significantly from pre-test to post test for all the patterns, except tenant-isolated under scenario 2. As tenant-isolated component is in the middle most times the results are either close to shared component or dedicated component. As expected, the plot of the estimated marginal means of change shows that response times for shared component and tenant-isolated component changed significantly the most for tenants exposed to the deployment conditions of both scenarios.

(2) Error%: The patterns did not show significantly different changes from pre to post test. The post hoc test showed that that the groups did not change significantly in comparison to the other groups. The paired t-test also showed that tenants did not also change significantly from pre test to post test under all the scenarios.

(3) Throughput: The statistical analysis showed that the patterns had significantly different changes from Pre to Post for tenants exposed to only the deployment conditions of scenario 1. None of the patterns showed a significant change relative to the other patterns for scenario 2 (i.e., effect of lock duration). Further analysis using the Post hoc test showed that there was no significant difference between the Shared component and the dedicated component. However, the paired sample t-test revealed that the throughput of tenants changed significantly from pre-test to post test for all the patterns in both scenarios.

(4) CPU and Memory usage: The statistical analysis of CPU showed that the patterns had significantly different changes from Pre to Post for both scenarios. The paired sample t-test also showed that the CPU of tenants changed significantly from pre-test to post test for all the patterns.

For memory, none of the patterns showed a significant change relative to each other. The paired sample t-test revealed that the memory of tenants changed significantly from pre-test to post test only for shared component.

(5) Disk I/O: The statistical analysis of disk IO showed that the patterns had significantly different changes from Pre to Post. Further analysis using the Post hoc test showed that the change the shared component showed was not significant in comparison to the change the dedicated component showed. The paired sample t-test revealed that the disk I/O of tenants changed significantly from pre-test to post test for all the patterns under all the scenarios.

(6) System Load: Table 1 showed that system load (measured as one-minute load average reported by SAR-ldavg) did not show any variability in the values from pre-test to post-test. This is similar to the result obtained in Ochei et al [3] where the authors evaluated the degrees of multitenancy isolation for cloud-hosted continuous integration using Hudson.

VI. Discussion

(1) Response times: The results show that while none of the patterns changed significantly in comparison to the other patterns, the tenants within all the groups (i.e., the patterns) changed significantly from pre-test to post-test when one of the tenants is exposed to large instant workloads during version control. From Fig. 3, one would recommend dedicated component for carrying out version control process since it is the least influenced among the three patterns. That is, we do not recommend using shared component and tenant-isolated component to improve response time.

(2) Error%: Based on Fig. 4, the error% of tenants accessing the shared patterns changed the least among the three other patterns for both scenarios. One would therefore recommend the shared component when there is low bandwidth or slow network connection. The most expensive part of a typical version control system is retrieving data (e.g., FTP downloads) from a shared repository [7]. The response times of key individual components of the end-to-end action sequence of the version control process such as FTP upload and FTP download may have contributed to the extremely slow response times for tenant-isolated and dedicated component. It may be challenging to know what can be regarded as a very slow or extremely slow response times. Bauer and Adams [2] recommends that the maximum acceptable service latency (i.e., response time) should be 10-20 times greater than the 50th percentile for users of a cloud-hosted application. To further avoid high response times which could lead to other forms of errors, users of subversion, a widely used version control system, are advised to access the shared repository using accounts setup using svnserve or Apache HTTP server with the right ownership and file permissions [7].

(3) Throughput: The plot of the estimated marginal means of change from Fig. 5 showed a negative change. This means that the throughput of other tenants actually decreased in response to an increase in response times when one of the tenants is exposed to large instant loads. We therefore would recommend dedicated component for tenants accessing a shared application component since the estimated marginal means of change remained unchanged in both scenarios in comparison to the other patterns.

(4) CPU and Memory usage: Fig. 6 and Fig. 7 shows that the magnitude of change in CPU consumption for scenario 1 was not consistent, although it was slightly more for shared component than the other patterns. For scenario 2, response times increased steadily across the three patterns with the dedicated component being the most influenced. The dedicated component is therefore not recommended as the multitenancy pattern of choice for applications involved in a process that may lock/block other tenants from accessing a shared application component. For memory, the magnitude of change

TABLE I. PAIRED SAMPLES TEST ANALYSIS OF TENANT ISOLATION

Pattern	Response times	Error%	Throughput	CPU	Memory	Disk I/O	System Load
Scenario 1							
Shared	YES	NO	YES	YES	YES	YES	YES
Tenant-isolated	YES	NO	YES	YES	NO	YES	-
Dedicated	YES	NO	YES	YES	NO	YES	-
Scenario 2							
Shared	YES	NO	YES	YES	YES	YES	-
Tenant-isolated	NO	NO	YES	YES	YES	YES	YES
Dedicated	YES	NO	YES	YES	YES	YES	-

for the shared component was clearly more than the other three patterns. Therefore we would not recommend the shared component when using a memory intensive applications or there is a need for a better memory utilization.

(5) Disk I/O: The change in disk I/O consumption for tenant-isolated component and dedicated component was nearly the same for tenants accessing the shared application component deployed under scenario 1. Therefore there would be no much difference if either of them is used. The change in disk I/O consumption for shared component and dedicated component was also nearly the same for tenants accessing a shared application component deployed under scenario 2. Although, there was no significant difference between the shared and dedicated component we would still recommended the dedicated component since each tenant would have exclusive access to the shared application component, thereby reducing contention and high disk I/O consumption rate.

(6) System Load: From Table 1, it can be seen that system load showed no chance of variability, especially for dedicated component. This means that system load did not influence any of the patterns when tenants were exposed to all the deployment conditions considered in this study. This implies that with a reasonably high-speed network connection, there should be no problem with system load when a version control system is used to send data across a shared repository residing in a company's LAN or VPN.

VII. Recommendations

Based on the experience we have gathered while working with cloud-hosted GSD tools and consulting with experts on a number of software projects, we present in the following various options within a version control system that could be explored to implement multitenancy isolation at the file based level. In addition we also present factors that could influence the degree of isolation between tenants.

Most version control systems (e.g., Subversion) recognizes the existence of a system-wide configuration area. This gives system administrators the ability to establish defaults for all users on a given machine. The first time the svn command-line client is executed, it creates a per-user configuration area. On Unix-like systems, this area appears as a directory named .subversion in the user's home directory. This feature can be used to implement a low - medium degree of isolation between tenants based on, for example, shared component or tenant-isolated component.

In subversion, unversioned files resulting from program compilation can be excluded using Subversion global-ignores (i.e., a whitespace-delimited list of names of files and directories not displayed unless they are versioned). Examples of default values are: *.o *.lo *.la *.al .libs *.so *.so.[0-9]* *.a . A Similar feature named 'Enable Filtering" in the File System SCM plugin can be used to either include or exclude certain files (in the form of wildcard) while uploading or downloading to the repository. This feature can be used to implement a very high degree of isolation using the dedicated component.

The following factors could influence the degree of multitenancy isolation and care should be taken to avoid them when implementing multitenancy in a version control system:

(1) It is less safe when a version control system is used with a repository storage through a shared filesystem. For example, in Subversion it is safe as a single server-process running as one user.

(2) Most version control systems (e.g, subversion and Git) store additional copies of data on the local machine, which can be an issue for large projects or files or if developers work on multiple branches simultaneously. There are features within most version control systems that can help to save disk space. For example, the "Discard old builds" feature could be used to limit the number of builds allowed to be remain in the system. The "Clear Workspace" feature on the File System SCM plugin can be used to delete all existing files/sub-folders in workspace before checking-out.

VIII. Limitations of the Study

The study used (an open-source) FileSystem SCM plugin to trigger the version control process. This means that the focus is not on a particular version control tool but on the software development process (i.e., version control). The number of requests sent to the application component was within the limit of the private cloud used (i.e., Ubuntu Enterprise Cloud). Therefore, the results of this study applies to private clouds and should not be generalized to large public clouds.

In this study, multitenancy isolation was implemented on the application level of the cloud stack by capturing the tenant-id associated with requests and re-routing them to different components configured for each tenant. This approach is very useful in a resource constrained environment where duplicating the deployment of the VM instance for each tenant is costly, for instance in terms of time, bandwidth and resource consumption (i.e., using a large number or size of VM instance).

This study assumes that a small number of users sends multiple request across the network; it would be interesting to replicate this study in a large private cloud infrastructure (using other version control tools like Subversion) to investigate the effect of a large number of users. The most common challenge while conducting experiments was that of insufficient memory and file or directory permission issues (e.g., when setting FTP request configurations). This problem becomes more acute when moving the VM image instance (whose file permission had been set on a local machine) to the cloud infrastructure. Therefore it is necessary to get repository ownership and permissions right before conducting the experiments.

IX. Conclusion and Future Work

In this paper, we have applied COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing), to contribute to literature on multitenancy isolation for cloud-hosted Version Control Systems by showing how to evaluate the degree of isolation between tenants enabled by multitenancy patterns.

We implemented three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) by modifying the FileSystem SCM Plugin (integrated within Hudson) and deploying it as a Virtual Machine (VM) instance to the Ubuntu Enterprise Cloud (UEC) private cloud. The study revealed that dedicated component provides the highest degree of isolation between tenants (compared to shared component and tenant-isolated component) especially with respect to error% (i.e., the percentage of errors with unacceptably slow response times) and throughput. Response times, CPU and memory consumption had the most negative impact on tenant isolation when exposed to demanding deployment conditions (e.g, large instant loads) for all the multitenancy patterns. System load did not influence tenants accessing components deployed based on all of the multitenancy patterns. The study recommends that during version control, the shared application component should reside in a repository on a cloud infrastructure with a high-speed connection and a reasonably large CPU and memory size.

We also plan to apply COMITRE to a case study involving an error/issue tracking system (e.g., Bugzilla) in a robust cloud infrastructure. Thereafter, we will carryout a cross-case analysis involving a comparison of the commonalities and differences in the processes found in several case studies. This will result in a pattern selection framework for deploying cloud-hosted GSD tools.

Acknowledgment

This research was supported by the Tertiary Education Trust Fund, Nigeria, and Robert Gordon University, UK.

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] L. Ochei, A. Petrovski, and J. Bass, "Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools," 2015 IEEE International Conference on Cloud and Autonomic Computing (ICCAC).

[4] J. Portillo-Rodriguez, A. Vizcaino, C. Ebert, and M. Piattini, "Tools to support global software development processes: a survey," in *Global Software Engineering (ICGSE), 2010 5th IEEE International Conference on*. IEEE, 2010, pp. 13–22.

[5] L. Ochei, A. Petrovski, and J. Bass, "Taxonomy of deployment patterns for cloud-hosted applications: A case study of gsd tools," 2015, seventh International Conference on Cloud Computing, Grids, and Virtualization (CLOUD COMPUTING 2015).

[6] F. Lanubile, "Collaboration in distributed software development," in *Software Engineering*. Springer, 2009, pp. 174–193.

[7] B. Collins-Sussman, B. Fitzpatrick, and M. Pilato, *Version control with subversion (For Subversion 1.7: Compiled from r4561)*. O'Reilly, 2011.

[8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, 3/E*. Pearson Education India, 2013.

[9] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, p. 120, 1995.

[10] bitcurrent. Bitcurrent cloud computing survey 2011. [Online: accessed in October 2014 from http://www.bitcurrent.com/download-file/?fid=2476].

[11] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, "A framework for native multi-tenancy application development and management," in *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on E-Commerce Technology*. IEEE, 2007, pp. 551–558.

[12] 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.

[13] R. Krebs, A. Wert, and S. Kounev, "Multi-tenancy performance benchmark for web application platforms," in *Web Engineering*. Springer, 2013, pp. 424–438.

[14] F. Chong and G. Carraro. Architecture strategies for catching the long tail. technical report, microsoft. [Online: accessed in February 2015 from https://msdn.microsoft.com/en-us/library/aa479069.aspx].

[15] IEEE. Cloud profiles working group (cpwg). [Online: accessed in February 2015 from http://standards.ieee.org/develop/wg/CPWG-2301...WG.html].

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

[17] J. Bass, "How product owner teams scale agile methods to large distributed enterprises," *Empirical Software Engineering*, pp. 1–33, 2014.

[18] M. Moser and T. O'Brien. The hudson book. Oracle, Inc., USA. [Online]. Available: http://www.eclipse.org/hudson/the-hudson-book/book-hudson.pdf

[19] Bugzilla.org. The bugzilla guide. [Online: accessed in October 2014 from http://www.bugzilla.org/docs/].

[20] MSDN. Multi-tenant data architecture. Microsoft Corporation. [Online]. Available: https://msdn.microsoft.com/en-gb/library/hh534480.aspx

[21] O. D. Concepts. Introduction to the multitenancy architecture. [Online]. Available: docs.oracle.com/database/121/CNCPT/cdbovrvw.htm

[22] Hudson. Files found trigger. [Online: accessed in October 2014 from http://wiki.hudson-ci.org//display/HUDSON/Files+Found+Trigger].

[23] D. Johnson, M. Kiran, R. Murthy, R. Suseendran, and G. Yogesh. Eucalyptus beginner's guide - uec edition. [Online: accessed in April, 2015 from http://www.csscorp.com/eucauecbook].

[24] Oracle. Oracle database concepts 10g release 1 (10.1). Oracle Corporation. [Online]. Available: http://docs.oracle.com/cd/B1203701/server.101/.../toc.htm