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

TITLE: Evaluating degrees of multitenancy isolation: a case study of cloud-hosted GSD tools.

YEAR: 2015

Publisher citation: OCHEI, L.C., BASS, J.M. and PETROVSKI, A. 2015. Evaluating degrees of multitenancy isolation: a case study of cloud-hosted GSD tools. In Proceedings of the 2015 international conference on cloud and autonomic computing (ICCAC 2015), 21-25 September 2015, Cambridge, MA, USA. Piscataway, NJ: IEEE [online], pages 101-112. Available from: http://dx.doi.org/10.1109/ICCAC.2015.17

OpenAIR citation: OCHEI, L.C., BASS, J.M. and PETROVSKI, A. 2015. Evaluating degrees of multitenancy isolation: a case study of cloud-hosted GSD tools. In Proceedings of the 2015 international conference on cloud and autonomic computing (ICCAC 2015), 21-25 September 2015, Cambridge, MA, USA. Piscataway, NJ: IEEE, pages 101-112. Held on OpenAIR [online]. Available from: https://openair.rgu.ac.uk/

Evaluating Degrees of Multitenancy Isolation: A Case Study of Cloud-hosted GSD Tools

Laud Charles Ochei
School of Computing Science and Digital Media
Robert Gordon University
Aberdeen, United Kingdom
Email: l.c.ochei@rgu.ac.uk

Julian M. Bass
School of Computing Science and Digital Media
Robert Gordon University
Aberdeen, United Kingdom
Email: j.m.bass@rgu.ac.uk

Andrei Petrovski
School of Computing Science and Digital Media
Robert Gordon University
Aberdeen, United Kingdom
Email: a.petrovski@rgu.ac.uk

***Abstract*—Multitenancy is an essential cloud computing property where a single instance of an application serves multiple tenants. Multitenancy introduces significant challenges when deploying application components to the cloud due to the demand for different degrees of isolation between tenants. At the very basic degree of isolation, tenants still share application components as much as possible. However, while some components may benefit from low degree of isolation between tenants, others may need a higher degree of isolation, for instance, in a situation where a component is too critical to be shared, or needs to be configured specifically for individual tenants. This paper describes COMITRE (COmponent-based approach to Multitenancy Isolation Through request RE-routing) to empirically evaluate the degree of isolation between tenants enabled by three multitenancy patterns (i.e., shared component, tenant-isolated component, and dedicated component) for cloud-hosted Global Software Development (GSD) tools. We developed a multitenant component for each multitenancy pattern, integrated it within Hudson, and then compared their impact on different tenants. The study revealed among other things that a component deployed based on shared component offers a lower degree of tenant isolation (than tenant-isolated component and dedicated component) when one of the tenants is exposed to a demanding deployment condition (e.g, large instant loads). We also provide some recommendations to guide an architect in implementing multitenancy isolation on a set of GSD tools: Hudson, Subversion and Bugzilla.**

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

I. Introduction

Multitenancy is an essential cloud computing property in which a single instance of an application/component serves multiple tenants. One of the key concerns 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]. Multitenancy isolation ensures that the performance, stored data volume and access privileges required by one tenant does not affect other tenants [3] [4].

As software tools are increasingly being deployed on the cloud, there is need to ensure proper isolation of both tenant's data (e.g., code files) and processes (e.g., builds) associated with these tools. At the very basic degree of multitenancy, tenants share application components as much as possible which translates to increased utilization of underlying resources. However, while some application components may benefit from low degree of isolation between tenants, other components may need a higher degree of isolation because the component may either be too critical or needs to be configured very specifically for individual tenants because of their unique deployment requirements. Again, tenant-specific requirements, such as laws and corporate regulations, may even further increase the degree of isolation required between tenants. The challenge therefore, for a cloud deployment architect would be how to resolve the trade-offs between the required performance, systems resources and access privileges at different levels of an application when opting for one (or combinations) of the multitenancy patterns for cloud deployment of software tools.

Motivated by this problem, this paper evaluates the degree of isolation between tenants enabled by multitenancy patterns for cloud-hosted software tools in order to resolve these trade-offs under different cloud deployment conditions. We are inspired by the work of Fehling et al [1], where the author captured the degree of multitenancy in three cloud patterns: shared, tenant-isolation and dedicated components; and also suggested that the degree of isolation between tenants is the main factor that can be used to distinguish between these patterns. However, we do not know the various deployment conditions which offers the required degree of isolation. In addition, these patterns have never been empirically evaluated to measure the actual degree of tenant isolation for applications in software engineering domain. This study focuses on cloud-hosted Global Software Development (GSD) tools, for example, Hudson [5]. To the best of our knowledge, this study is the first to present a Component-based approach to Multitenancy Isolation through Request Re-routing (COMITRE) and then apply it to empirically evaluate the degree of isolation between tenants enabled by multitenancy patterns within the context of cloud-hosted GSD tools under different cloud deployment conditions.

By evaluating the degrees of multitenancy isolation, we mean comparing the effect of performance (e.g., response times and error%) and resource utilization (e.g., CPU and memory usage) on tenants deployed based on different multite-

nancy patterns (i.e., shared component, tenant-isolated component, and dedicated component) when one of the tenants experiences a demanding deployment conditions (e.g., large instant loads). The research question this paper addresses is: **"How can we evaluate the degree of isolation between tenants enabled by multitenancy patterns for cloud-hosted GSD tools"**. Multitenancy isolation introduces significant security and performance challenges in the cloud depending on the location of the functionality to be shared, and the required degree of isolation between the tenants. For example, if one of the tenants on the network is malicious, it can cause denial of service and performance degradation to other tenants [6].

We implemented three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) by exposing the functionality of each pattern as a plugin integrated with a GSD tool deployed on a private cloud. Thereafter, we evaluated the degree of isolation for each pattern at two levels: process isolation and data isolation, as it affects tenants interaction with GSD tool. The overarching result of the study is that the degree isolation between tenants using a shared component are significantly lower than for tenant-isolated component and dedicated component. However, **the disk I/O consumption rate rises faster (under heavy load) for tenants using a shared component than in other multitenancy patterns**.

The main contributions of this paper are:
1. Presenting COMITRE, a novel approach for implementing multitenancy isolation for cloud-hosted applications.
2. Demonstrating the practicality of the approach by applying it to: (i) empirically evaluate the degree of isolation between tenants enabled by multitenancy patterns for a cloud-hosted GSD tool; and (ii) compare how well different multitenancy patterns perform under different cloud deployment conditions.
3. Presenting recommendations and best practice guidelines for implementing multitenant isolation on a selected set of GSD tools: Hudson and Subversion and Bugzilla.

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 Tools and Software Processes.

Definition 1: **Global Software Development (GSD).** GSD means the splitting of the development of the same software product or service among globally distributed sites [7].
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 [8]:
(1) Continuous Integration (CI): CI is a development practice that requires developers to integrate source code into a shared repository several times. Each check-in is then verified by an automated build, allowing teams to detect problems early. Hudson is a widely used GSD tool used for continuous integration. It is written in Java for deployment in a cross-platform environment. Hudson is hosted partly as Eclipse Foundation project and partly as a Java.NET project. It has a rich set of plugins making it easy to integrate with other software tools. Organizations such as Apple and Oracle use Hudson for setting up production deployments and automating the management of cloud-based infrastructure [9].
(2) 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. A widely used GSD tool for version control is Subversion [10]. Subversion implements a centralized repository architecture whereby a single central server hosts all project metadata. This facilitates distributed file sharing [11].
(3) Issue/Bug Tracking: Bug tracking is the process of keeping track of reported software bugs or issues in software development projects. Examples of widely used error and bug tracking systems are Bugzilla and JIRA. Bugzilla is a web-based general-purpose bug tracker and testing tool originally developed and used for the Mozilla project [12]. JIRA is a bug tracking, issue tracking and project management software tool. JIRA products (e.g., JIRA Agile, JIRA Capture) are available as a hosted solution through Atlassian OnDemand, which is SaaS cloud offering. JIRA is built as a web application with support for plugin architectures and an API that allows developers to integrate JIRA with third-party applications such as Eclipse, IntelliJ IDEA and Subversion [13].

B. Cloud Deployment Patterns for Multitenancy Isolation

Definition 3: **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 [6], cloud computing patterns [1], cloud architecture patterns [14], and cloud design patterns [15]. Architectural and design patterns have long been used to provide known solutions to a number of common problems facing a distributed system [16, 6].

Definition 4: **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 or functionality of a shared application component. Multitenant isolation can be captured in three main cloud patterns [1]:
(1) Shared component: Tenants share the same resource instance, and may not be aware that it is being used by other tenants.
(2) Tenant-isolated component: Tenants share the same resource but their isolation is guaranteed.
(3) Dedicated component: Tenants do not share this resource.

However, each tenant is associated with one instance (or certain number of instances) of the resource.

Definition 5: **Application Component.** We present an informal definition of "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), processing component (e.g., load balancer), or a user interface component (e.g., AJAX).

C. Evaluating Degree of Multitenancy Isolation

The three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) expresses the degree of isolation between tenants accessing a shared component of an application. The shared component represents the lowest degree of isolation between tenants while the dedicated component represents the highest. The degree of isolation between tenants accessing a tenant-isolated component would be in the middle.

The three main aspects of tenant isolation [1] 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. Guo et al [17] evaluated different isolation capabilities related to authentication, information protection, faults, administration etc. Bauer and Adams [2] discusses how to use virtualization to ensure that the failure of one tenants instance does not cascade to other tenant instances. A closely related work to ours is that of Walraven et al [18] where the authors implemented a middleware framework for enforcing performance isolation. They used a multitenant implementation of a hotel booking application deployed on top of a cluster for illustration. Krebs et al [19] implemented a multitenancy performance benchmark for web application based on the TCP-W benchmark where the authors evaluated the maximum throughput and the amount of tenants that can be served by a platform. Other works related to multitenancy isolation can be found in [20] [21].

The focus of our work is evaluating the degree of isolation between tenants enabled by multitenancy patterns. Specifically, we are interested in providing empirical evidence of the effect of performance and resource utilization on other tenants due to high workload created by one of the tenants. In our work, we implemented multitenancy as a component integrated into an open source Global Software Development (GSD) tool. In addition, our evaluation is done in a real cloud environment. The application we used for our evaluation is within the domain of software engineering, to emulate a typical software development process. Furthermore, we deployed our GSD tool to the cloud using cloud multitenancy patterns.

III. Methodology

A. Selecting the GSD Tools and Software Processes

Three main software processes have been found to have the most impact in Global Software Development: continuous integration, source/version control management and issue/bug tracking [5, 22]. In this study, we have selected three open-source GSD tools to represent these software processes: Hudson, Subversion and Bugzilla. These GSD tools were selected based on an empirical study 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 [5] for details. This paper focuses on applying our approach (i.e., COMITRE) to implement multitenancy in Hudson, a widely used GSD tool for continuous integration.

B. COMITRE: A Component-based approach to Multitenancy Isolation through Request Re-routing

In the following, we present COMITRE an approach for implementing multitenancy isolation for cloud-hosted applications. COMITRE can be seen as an abstract format that allows the implementation of multitenancy isolation in various ways. 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. Figure 1 captures the structure of COMITRE. The approach summarizes to the following steps:

Step 1: Define the structure of the tenant request- The structure of the tenant identifier has to be clearly defined. The tenant identifier can be in various forms such as an IP address, port number, request header, or a query string attached to the request. Once the format of the tenant-identifier has been chosen, we then use this to define the structure of a typical tenant request. For example, when using a load generator like Apache JMeter, the tenant-identifier can be sent as a parameter along with the request which will appear as a query string. In our paper, the structure of the HTTP request looks like this: 172.19.1.2:8080/FileTrigger1/build?delay=0sec?tenant1=1.

Step 2: Configure Server to re-route tenant request to application- The next step is to configure the web server to re-route the sever request to a component of the application. This can be done in two ways: (i) programmatically in Java or even in bash shell script; (ii) manually entering the tenant identifier into the host file (/etc/hosts/) so that the request of all the tenants points to the same IP address of the localhost (usually 127.0.0.1).

Step 3: Create configuration for each multitenancy pattern - We then create the configuration of each multitenancy pattern (i.e., shared component, tenant-isolated component and dedicated component) and also a default configuration that can be assigned to every tenant in case a matching tenant-identifier was not found in the tenant-conf file. These configurations map to different degrees of isolation between tenants.

Step 4: Tenant Identification and Resolution- This is a two-step process: (i) Capture the incoming request (e.g., http, ftp, JDBC request); (ii) Extract the tenant-identifier from the request and use it to resolve the tenant.

Step 5: Configure tenant-specific information- Based on this tenant-identifier and its associated information, a specific configuration is created for each tenant. The configuration includes information such as tenant-identifier, tenant request, required degree of isolation, and the application component that is to be accessed by multiple tenants.

Step 6: Select matching tenant configuration from the list of tenants in the tenant configuration file- The selected tenant configuration is returned, otherwise the default tenant configuration is returned if matching tenant is not found.

Step 7: Send viewable response to the user - The last step

is to present the viewable response to the user. This response is the multitenant component that has been adjusted based on the tenant-specific configuration.

The actual implementation of the COMITRE is anchored on shifting the task of routing a request from the server to a separate component at the application level of the cloud-hosted GSD tool. For example, this component could be a program component (e.g., Java class file) or a software component (e.g., plugin) which can be integrated into the GSD tool. The logic that is implemented in the component is shown in Algorithm 1.

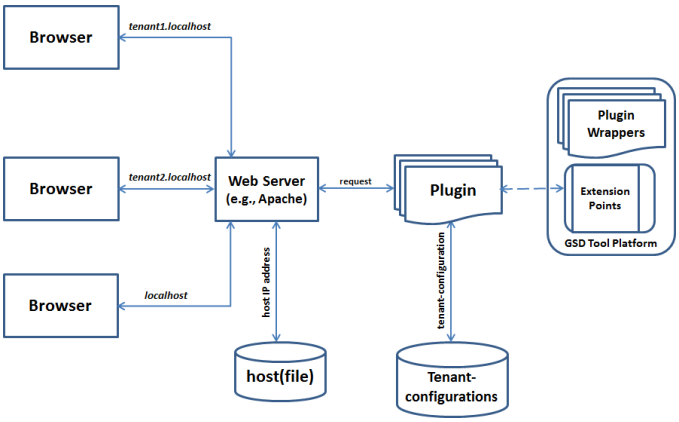

Fig. 1. COMITRE Architecture.

Algorithm 1 COMITRE Algorithm

1: **INPUT:** tenantRequest, tenantConf-file
2: **OUTPUT:** multApplFunctn
3: Get tenant identifier from incoming request
4: *tenantConf* ← null
5: Select tenantData from *tenantConf-file*
6: **if** tenantData is found **then**
7: *tenantConf* ← tenantData
8: **end if**
9: Create defaultApplFunctn
10: Create tenantApplFunctn
11: *multApplFunctn* ← defaultApplFunctn
12: **if** tenantConf is not null **then**
13: *multApplFunctn* ← tenantApplFunctn
14: **end if**
15: **return** multApplFunctn

The input to the algorithm is tenant request and tenant configuration file, while the output is the multitenant functionality/component that is shared among the different tenants. This input could be a text file or a database that contains among other things the tenant-identifier, the default functionality of the applications as well as the functionality that is to be exposed to the different tenants. Each tenant-specific data has to be configured (either manually or programmatically) before the request for each tenant is sent to the application. Another option, could be to update the "hosts" file (i.e., typically found in the "/etc/hosts" folder on Ubuntu) and add entries for the IP addresses of other tenants to point to the default IP address of the host. The algorithm begins by capturing the tenant identifier from an incoming request (e.g., http, ftp, JDBC request). The tenant identifier could be a query string attached to the URL of each request or an IP address. Tenant-specific data for each tenant is selected from the configuration file and mapped to the tenant request which is then used to adjust the behaviour of the functionality or component that is being shared. If tenant configuration is not found, then the default functionality is returned.

C. Validating the Implementation of Multitenancy Isolation

We validate our approach (i.e., COMITRE) for implementing multitenancy isolation both in theory and in practice. Each application of the approach to a specific multitenancy pattern will result in a differently looking behaviour of the component that is being shared among the different tenants, but all applications of the approach will share a common set of desired properties.

Each multitenancy pattern was validated in theory by following the implementation proposed by Fehling et al [1]:
(i) we carefully analyzed the sketch of the architecture proposed for the three multitenancy patterns, the description of the patterns and their behaviour after implementation.
(ii) we systematically cross-checked our implementation against other implementations of multitenancy architectures, and also examined that our implementation is compliant with how tenants access a multitenant component.

From implementation standpoint, Fehling et al's [1] explanation of row-based isolation (i.e., tenants with unique tenant-Id's sharing the same database and table) and table-based isolation (i.e., tenants sharing the same database, but having different tables) reflects shared component and tenant-isolated component respectively. This implementation is similar to other well-known implementations of multi-tenant (data) architecture [23].

We also demonstrate the practicality of our approach by applying it to implement the three multitenancy patterns on 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.

A. Case Studies of Multitenancy Isolation for a Continuous Integration System

We present two case studies, that focus on implementing multitenancy at both the process and data levels of a cloud-hosted application. This entails introducing a process and data-access component to Hudson so that the data and processes of different tenants are handled in an isolated fashion. Figure 2 and 3 captures the architectures of both implementations.

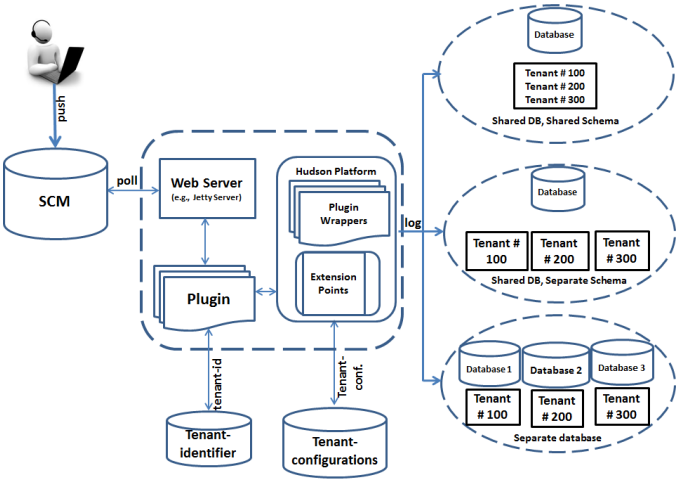

Fig. 2. Multitenancy Isolation Architecture for Cloud-hosted Applications.

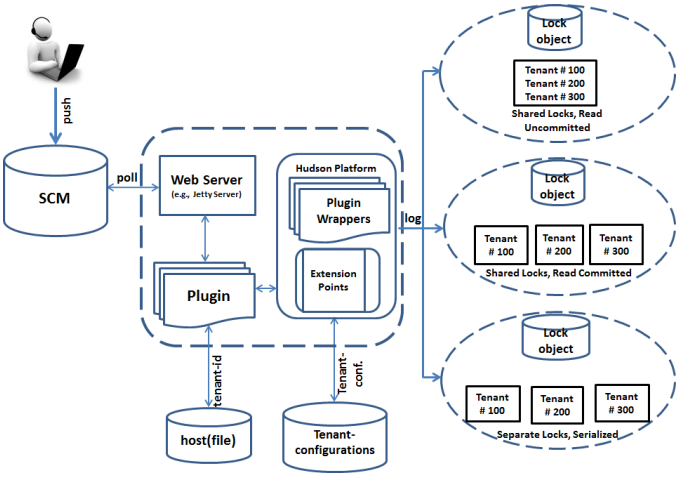

Fig. 3. Multitenancy Isolation Architecture for Cloud-hosted Applications.

1) Case Study 1 - Isolating Tenant Data during Automated Build Verification/Testing Process with Hudson: This case study is used 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 within those directories [24]. We implemented multitenancy isolation 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.

B. Case Study 2 - Isolating Tenant Process during Automated Build Verification/Testing Process with Hudson

This case study is used to evaluate the effect of tenant isolation at the process 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 introduce the concept of database isolation level which is used to control the degree of locking that occurs when selecting or updating data in a database. We configured the database component of the application to the highest isolation level: SERIALIZABLE level, to evaluate the impact of the lock duration where locks on data are held until transaction completes [25].

C. Experimental Setup

The experimental setup consist of a private cloud setup using Ubuntu Enterprise Cloud (UEC). UEC is an open-source private cloud software that comes with Eucalyptus [26]. 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.

Aim of the Experiment: The aim of the experiment is to evaluate the degrees of isolation of multitenancy patterns for cloud-hosted GSD tools.

Sub-Question 1: Performance experienced by tenants accessing an application component deployed using each multitenancy pattern changes significantly from the pre-test to the post test.

Sub-Question 2: 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.

(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 three different scenarios (i.e., treatments) for configuring T1 (see section IV D for details of the three scenarios). For each group, one of the four tenants (i.e., T1) is configured to send large instant loads to the application component. Performance metrics (e.g., response times and throughput) and systems resource consumption (e.g., CPU, memory) are measured for each pattern before the treatment (Pre Test) and after the treatment was introduced.

Based on this information, we adopt a two-way repeated measures (within-between) ANOVA. This experimental design is used when there are two independent variables (factors) influencing one dependent variable. In our case, the first factor is multitenancy deployment pattern, and the second factor is time. Multitenancy pattern is the between factor, because we are looking at the differences between the groups using different mutltitenancy patterns for deployment. Time is the within factor, because we are measuring each group twice (pre-test and post-test). The data view of our experimental design is composed of a Group column that indicates which of the three groups the data belongs to, and 2 columns of actual data, one for the Pre test and one for the Post Test.

D. Experimental Procedure

A summary of the experimental procedure is outlined in Figure 6. In the following, we describe the procedure for the experiment in more detail in this section. We modified a GSD tool to support multitenancy isolation. This entails developing a plugin and integrating it with the GSD tool so that it can be accessed by different tenants. The GSD tool is then bundled as a VM image, and uploaded to a private cloud with a typical minimal UEC configuration.

To evaluate the degree of multitenancy isolation between tenants, we configured four tenants (referred to as tenant 1, 2, 3, and 4) based on accesses to functionality/component of the GSD tool that is to be served to multiple tenants. Accesses to this functionality is associated with a tenant-identifier that is attached to every request. Based on this identifier, a tenant-specific configuration is retrieved from the tenant configuration file and used to adjust the behaviour the GSD tool's functionality that is being accessed.

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.

All the tenants simultaneously send requests to the GSD tool according to its own JMeter test plan. To measure the effect of tenant isolation we introduce a tenant that experiences intense or aggressive workload. There are three scenarios that we are simulating for this tenant:
Scenario1-Large instant load: This scenario is used to illustrate the effect of large instant load on other tenants. To simulate this behaviour, all of the request sent by tenant are released at once. We also added the Synchronous Timer to the Samplers and then reduce the ramp-up period by one-tenth so that all the requests are sent ten times faster. The scenario is similar to unpredictable (i.e., sudden increase) workload [1] and aggressive load [18].
Scenario 2 -Variation in request arrival rate: This scenario 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.
Scenario 3 -Lock duration: This scenario illustrates 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 in JMeter, we set the transaction isolation level to TRANSACTION-SERIALIZABLE.

All other tenants experience the same normal load which is set to just below the maximum capacity of the system determined separately through repeated test runs. In our paper, we limit the normal load to 100 requests per tenant. For each test run, the same number of request is sent by all the tenants except the one that is experiencing large intense and aggressive load. This means that the total number of requests for each run is spread over the different tenants. The time to load all the requests is kept constant at 60 seconds.

The following system metrics were collected and analyzed:

(i) CPU Usage: The %user values (i.e., 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.

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 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.

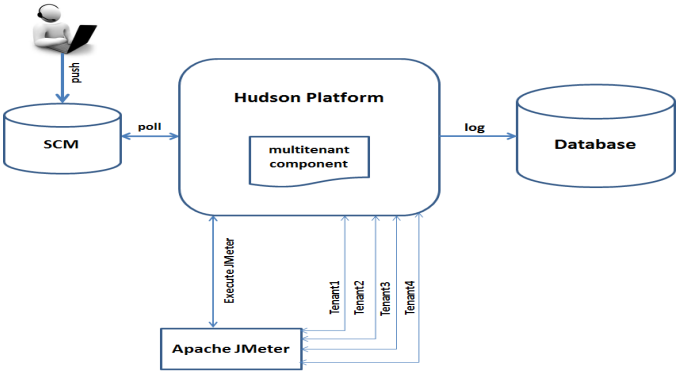

Fig. 4. Experimental Setup

We performed 10 iterations for each run and used the values reported by JMeter as a measure for response times, throughput and error%. For system activity, we reported the average CPU, memory, disk I/O and system load usage at one-second interval. 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.

1) Prepare the Private Cloud for the Test Run
   a) Create an Ubuntu Virtual Machine Image
   b) Install the modified GSD tool on the image
   c) Upload the Image to UEC
   d) Launch the instance and SSH to the instance
2) Execute the Test Run
   a) Start the GSD tool and view it on a browser
   b) Start JMeter load test on the GSD tool
   c) Start instance monitoring with SAR tool
   d) Stop test run after all responses received
3) Collect Results
   a) Export JMeter and SAR result to text file
   b) Clear previous JMeter and SAR results
   c) Reboot instances for next test run
   d) Repeat step 2 for more runs

Fig. 5. Experimental Procedure

V. Results

In this section we report the results of the experiments. Four set of results are presented here, each with a different scope and view of the degree of isolation between tenants enabled by three multitenancy patterns for all the scenarios considered.

A. Preliminary Analysis of Response Times and Disk I/O:

Results for response times and disk I/O are presented here for scenario 1 which reflects a case where one of the tenants experiences large instant workload that is significantly higher than the other tenants. The Response Time graphs of figure 6,7, and 8 illustrates how the response times varies over a period of time in response to a sharp increase in the workload of one of the tenants. This graph is shown only for tenant 4. From the graph , we can see that response times for all the patterns are the almost the same. A closer look at the graphs shows that the response times for dedicated component is fairly stable over longer period of time than the other tenants.

We also show in a bar chart the disk I/O of tenants accessing an application component for all patterns (Figure 9, 10 and 11). For dedicated component, the disk I/O was nearly zero in scenario 1. For scenario 2, the disk I/O is higher for shared component than the other patterns. Lastly, for scenario 3, the disk I/O is again nearly zero for dedicated pattern, while that of tenant-isolated and dedicated component were much higher but almost at the same level.

We have not gained enough information by analyzing only the response time graphs and the bar charts. In the first instance, it would be difficult and time consuming to examine the response time graph and bar charts of each tenant in the experiment (i.e., about 36 cases). Even if we were able to do that, the result would still not be very helpful since the result is not relative to the other patterns. Apart from this, we do not also know how the performance and resource consumption of each tenant has changed if compared to when all of the tenants were exposed to the same workload. To address this problem, we need a robust experimental design that would produce results that would give a broad view of all tenants in each pattern under all the scenarios considered. To this end, we performed the one-way ANOVA followed by Scheffe post hoc tests. The dependent variable used for the one-way ANOVA is a transformed variable termed "Change", which represents the change values from Pre-Test to Post-test. Further details are captured in the next section.

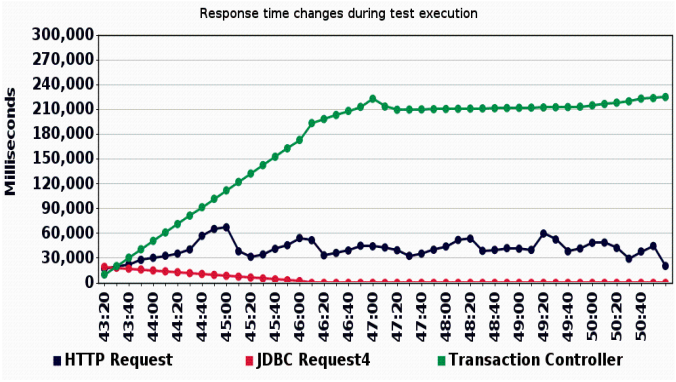

Fig. 6. Tenant 4 accessing a Shared Component

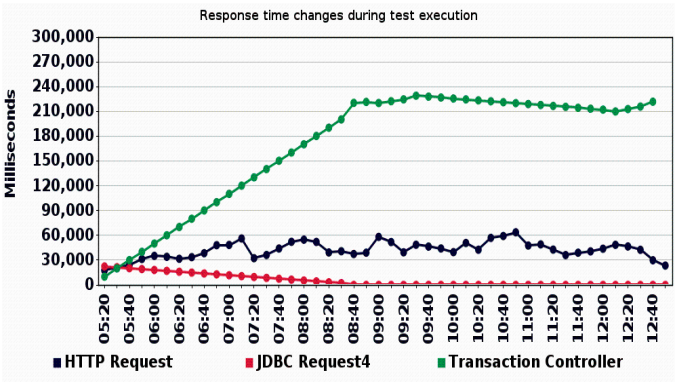

Fig. 7. Tenant 4 accessing a Tenant-isolated Component

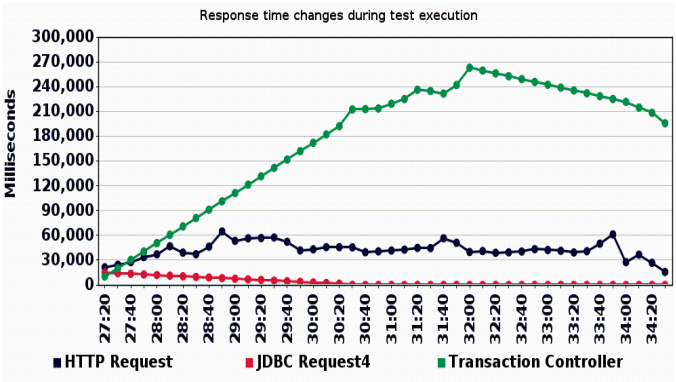

Fig. 8. Tenant 4 accessing a Dedicated Component

B. Analyzing Changes in Tenant's Performance and Resource Consumption from Pre-Test to Post-Test

In order to gain further insight into the behavior of the tenants across all the patterns, we carried out planned com-

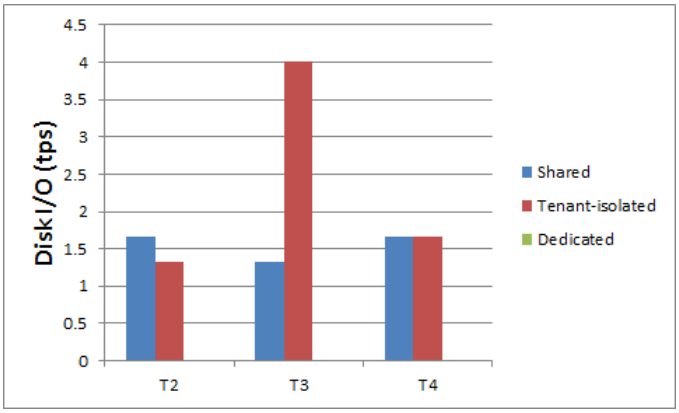

Fig. 9. Disk I/O of tenants for scenario 1

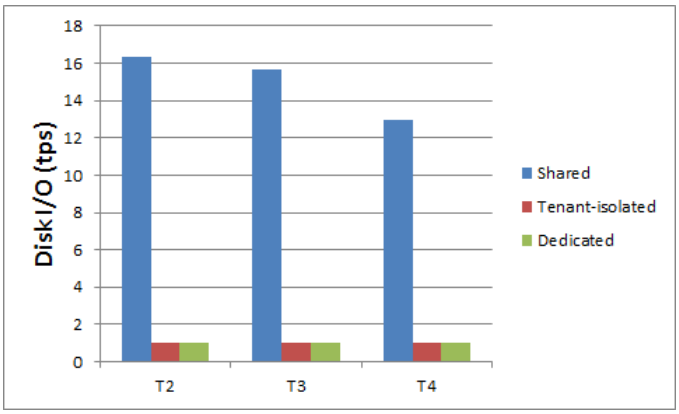

Fig. 10. Disk I/O of tenants for scenario 2

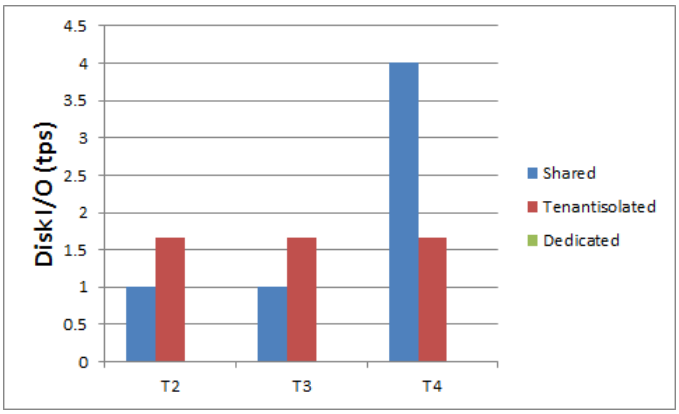

Fig. 11. Disk I/O of tenants for scenario 3

parisons based on the plots of the estimated marginal means of change derived from Scheffe Post hoc test and thereafter the paired t-test. The Dependent variable in the Scheffe post hoc test was determined by subtracting the Pre-test from Post-test values. We used the "Select Cases" feature in SPSS which allowed us to perform calculations only on the data we select. In our case, we selected the three tenants (T2,T3,T4) for each pattern and for each deployment scenario giving a total of 9 cases for each metrics that was measured.

The statistical test showed that there is a highly significant difference between the Pre and Post values for all the metrics considered in this study: response times, error%, throughput, CPU, memory, disks I/O and system load. This significant difference holds when all the groups are averaged together. This is not very helpful, because we are interested in not just knowing how tenants accessing components deployed based on these patterns changed relative to each other, but interested in knowing where the statistical significant difference lies with respect to the rate of change. This is very important considering the fact that some patterns will be more influenced than others. In addition, we also want to know if the subjects (i.e., the tenants) within any particular pattern improved significantly from the pre-test values to the post test values. 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.

To answer this question, we analyze the plots in Figure 12 to Figure 18. These figures show the plot of the mean values for each combination of factor levels. That is, it shows how each pattern performed on the Pre and Post Test. We focus on the three tenants (i.e., T2, T3, and T4) that did not experience the high intense load.

(1) Response times: From Figure 12 it is clear that the tenants accessing a dedicated component (especially for scenario 3) have a much higher response times than both shared component and dedicated component. This is due to the overhead incurred as a result of opening multiple connections to the database each time a JDBC request is made. Shared component has the lowest response times out of the three patterns. This is obvious as few connections are opened to the database which logs all the data into the same table.

(2) Throughput: The throughput is fairly stable for all the patterns except for shared component. This is because all the tenants are struggling to gain access to the same application component, so some requests are either delayed or refused. For tenant-isolated component and dedicated component, the throughput is reduced due to the fact that requests are not concentrated on one application component but instead are directed to the separate components reserved for different tenants.

(3) Error%: The percentage of request with errors is fairly stable for most of the patterns except of tenant 4 accessing a shared component. The interesting thing is that the error% rate of tenant 4 accessing a shared component for all scenarios is very high. A possible explanation for this is that as the number of tenants accessing a shared component increases, the error rate also increases.

(4) CPU usage: The CPU consumption for tenants accessing the shared component especially for scenario 3 is the lowest of all the patterns. The plot of figure 15 also shows that tenants accessing the dedicated component consumes more CPU than all other patterns.The least CPU consumption was for tenants accessing the shared component when tenant 1 is exposed to experimental setting of scenario 3.

(5) Memory: The memory consumption of the tenants changed significantly for all the three patterns. However, by analyzing the plot in Figure 16, the change was significant

when scenario 3 was applied to tenant 1. The lowest change was for tenants accessing the dedicated component.

(6) Disk I/O: The disk I/O consumption of tenants accessing the shared component was very high when tenants were exposed to deployment conditions in scenario 2. The disk I/O for both tenant-isolation component and dedicated component were almost at the same level.

(7) System Load: System load had the least impact of all the metrics measured in the experiment. It can be seen from the plot of Figure 18 that spikes (or the rise and fall in the curve) are consistent for all the patterns throughout all the deployment scenarios. The tabular paired samples test (Table I) further confirms this position. The standard error difference is the same for tenants components deployed using all the three multitenancy patterns.

Based on the above analysis, we summarize the results of the study as follows: Shared component offers the least degree of isolation between tenants, while there was no significant difference between tenant-isolated component and dedicated component for most of the metrics measured.

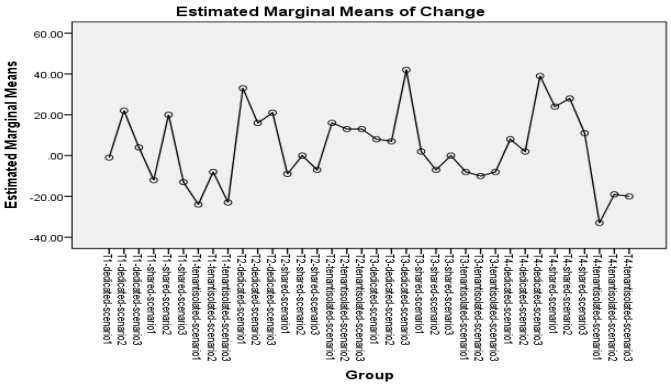

Fig. 12. Changes in response times for each pattern relative to other patterns

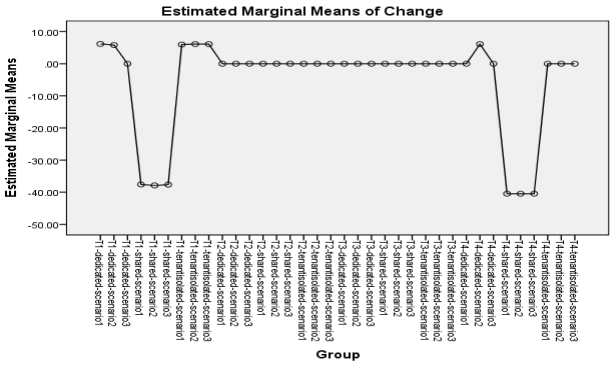

Fig. 13. Changes in error% for each pattern relative to the other patterns

VI. Discussion

In this section we will address the experimental questions we asked in section IV C. Table 1 summarizes the effect of Tenant 1 (i.e., the tenant that experiences high load) on the other tenants (i.e., T2, T2, T4). Specifically we are interested

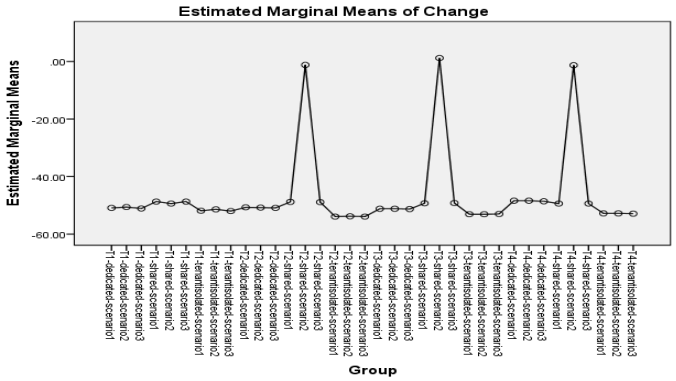

Fig. 14. Changes in throughput for each pattern relative to the other patterns

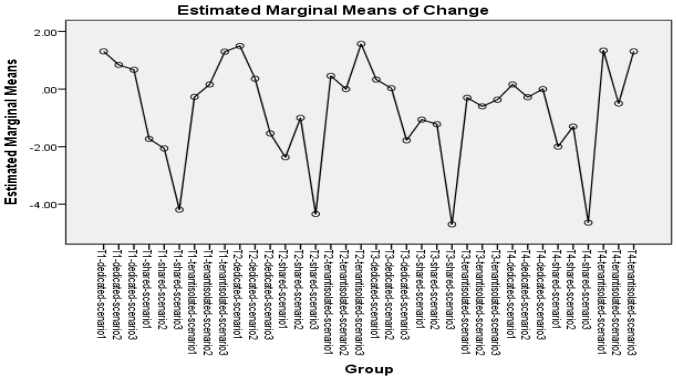

Fig. 15. Changes in CPU usage for each pattern relative to the other patterns)

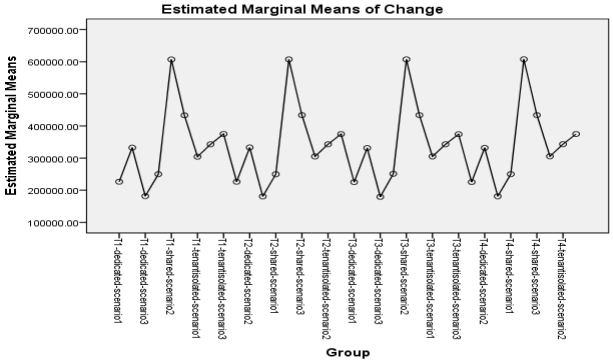

Fig. 16. Changes in memory for each pattern relative to the other patterns

in knowing if T2, T3 and T4 within a particular group (e.g., shared components for scenario 1) changed significantly from pre-test to post test measured using Paired Samples T-test at 95% confidence interval. 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; that is, no significant influence on other 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.

TABLE I. PAIRED SAMPLES TEST ANALYSIS OF TENANT ISOLATION

Pattern	Response times	Error%	Throughput	CPU	Memory	Disk I/O	System Load
Scenario 1							
Shared	NO	NO	YES	YES	YES	YES	-
Tenant-isolated	NO	-	YES	NO	YES	NO	-
Dedicated	NO	-	YES	NO	YES	-	-
Scenario 2							
Shared	NO	NO	NO	YES	YES	YES	-
Tenant-isolated	NO	-	YES	NO	YES	-	-
Dedicated	NO	NO	YES	NO	YES	-	-
Scenario 3							
Shared	NO	NO	YES	YES	YES	NO	-
Tenant-isolated	NO	-	YES	NO	YES	YES	-
Dedicated	YES	YES	YES	NO	YES	-	-

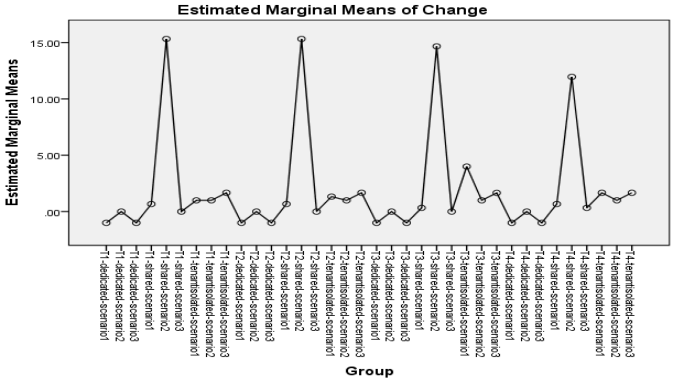

Fig. 17. Changes in disks for each pattern relative to the other patterns

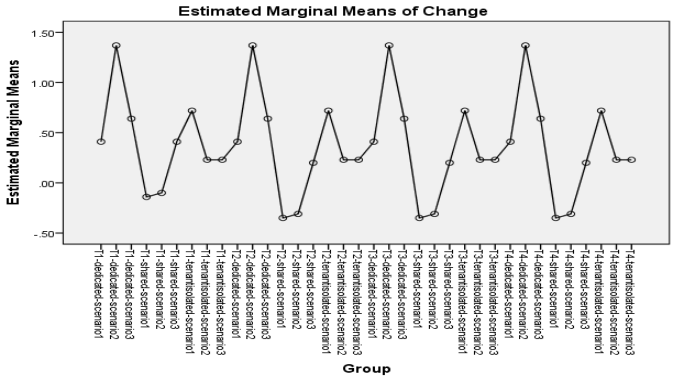

Fig. 18. Changes in system load for each pattern relative to the other patterns

(1) System Load and CPU: One of the most significant findings of this study is that the system load did not influence any of the patterns for all the scenarios considered. The results showed no variability in the values from pre-test to post test. This implies that in real cloud deployment, architects should not focus so much on the system load (for example, in a case where one of the tenants suddenly experiences a large instant load). CPU consumption of tenants changed significantly only for shared component for all the scenarios considered. Therefore, once a reasonable CPU size (e.g., multiple CPUs or a multi-core CPU) is used, there should be no problem in performing builds. Hudson for example does not consume much CPU; a build process can even be setup to run in the background without interfering with other processes [9].

(2) Disk I/O: Builds processes are known to consume disk I/O especially for I/O intensive builds [9]. As the results show, there was no difference or variability in the values for disk I/O usage for the dedicated component. This is understandable, considering the fact that each tenant has a dedicated component where transactions are channeled to without requiring multiple connections that may either be delayed or blocked as a result of sharing the components. Therefore for builds that are particularly I/O intensive, the shared component and tenant-isolated components are not recommended. Shared component showed a highly significant difference between the pre-test and post-test scores.

(3) Memory: Compilers and builders generally consumed a lot of RAM especially if the build is difficult and complex. There was a significant change in memory consumption for tenants accessing components deployed on the three patterns under all the scenarios considered. By analyzing the plots of the estimated marginal means of change, we observe that shared component under scenario 2 had the highest memory consumption implying a low degree of isolation. The plot also shows that dedicated component recorded the least change with respect to all the deployment conditions.

(4) Throughput: The throughput also changed significantly for all the patterns especially for shared component. Therefore when continuous integration systems like Hudson are configured to poll a repository for changes or to log data into a repository then dedicated component should be used.

(5) Response times and Error%: The only pattern that has changed significantly with respect to response times and percentage of requests with errors is the dedicated component. The plot (Figure 13) shows a large section where the estimated marginal means of change for error% remained stable. Based on this information, we recommend the use of shared component and tenant-isolated component when builds are configured to automatically publish artifacts to a shared repository.

VII. RECOMMENDATIONS

Based on the experience we have gathered while working with cloud-hosted GSD tools and consulting with experts on a

TABLE II. CONDITIONS THAT INFLUENCE DEGREE OF MULTITENANCY ISOLATION

Bug/issue tracking (with Bugzilla)	Continuous integration (with Hudson)	Version Control (Subversion)
(1) Requires more hardware to support large user base and number of bugs. (2) Performance can be improved by enabling mod_perl module. (3) Works well with SQL-like databases, e.g., MySQL and PostgreSQL and SQLite, but Oracle has several known issues with Bugzilla.	(1) Builds that generate large disk I/O activity, and difficult to compile consume system resources. (2) Running large number of builds concurrently could also consume more resources. (3) Using NFS mount to store output when running massive builds will result in performance degradation. (4) Allowing old builds to consume disk space. Enabling the "Discard old builds" feature can be used to resolve this condition.	(1) Subversion is less safe when used with a repository storage through a shared filesystem. It is safe as single server-process running as one user. (2) Subversion stores 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 simulatenously.

number of large scale enterprise software projects, we present in Table 1 a short list of factors that could influence the degree of isolation (in relation to performance) between tenants for the following GSD tools: Hudson, Subversion and Bugzilla. Table 2 shows a summary of components within Hudson, Subversion and Bugzilla (and in similar GSD tools) that can be explored to implement multitenancy isolation at the file based level.

VIII. Limitations of the Study

The study is used for open-source GSD tools. This is obvious considering the fact that we want to modify the source code of an existing GSD tool. 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. This study assumes that a small number of users send multiple request; it would be interesting to replicate this study in a large private cloud infrastructure to investigate the effect of a large number of users. Hudson itself is not very optimized, and so the most common error we had was that of insufficient memory allocation. This was not caused by the cloud infrastructure but by Hudson itself, so it is necessary to properly vary the setup values to get the maximum capacity of Hudson build processes running on the private cloud before conducting experiments.

IX. Conclusion and Future Work

In this paper, we have created a novel approach termed COMITRE- a Component-based approach to Multitenancy Isolation through Request Re-routing, to contribute to literature on multitenancy isolation for cloud-hosted Global Software Development (GSD) Tools by showing how to evaluate the degree of isolation between tenants enabled by multitenancy patterns. Cloud deployment architects will benefit from our proposed approach to implement multitenancy for existing cloud-applications, in particular open-source Global Software Development (GSD) tools.

This is the first study that presents an approach (i.e., COMITRE) for implementing multitenancy isolation and applying it to evaluate the degree of isolation between tenants enabled by multitenancy patterns for a cloud-hosted GSD tool at both the process level and data level. We implemented three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) by modifying Hudson and deploying it as a Virtual Machine (VM) instance to the Ubuntu Enterprise Cloud (UEC) private cloud.

The study revealed that shared component provides the lowest degree of isolation between other tenants when one of the tenants is exposed to demanding deployment conditions (e.g., large instant loads). There was no significant difference between the implementation of tenant-isolated component and dedicated component for a small number of build processes. The study concludes that when code files are checked into a shared repository at a low frequency to trigger a build process, then a high degree of isolation (in terms of response times) is expected both for tenant-isolated component and dedicated component. For shared component, the degree of isolation is lower which means that it is more prone to performance effect when exposed to high load.

We plan to carryout more experiments with other scenarios such as: (1) running more than one build concurrently and (2) executing complex builds or varying sizes of build scripts/codes. We also plan to carry out more case studies with a version control system (e.g., Subversion) and 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 the case studies which will then lead to a framework for selecting cloud deployment patterns for deploying GSD tools to the cloud.

Acknowledgment

This research was supported by the Tertiary Education Trust Fund (TETFUND), Nigeria and IDEAS Research Institute, 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] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications." *CLOSER*, vol. 12, pp. 426–431, 2012.
[4] S. Strauch, V. Andrikopoulos, F. Leymann, D. Muhler *et al.*, "Esbmt: Enabling multi-tenancy in enterprise service buses." *CloudCom*, vol. 12, pp. 456–463, 2012.

TABLE III. IMPLEMENTATION OPTIONS FOR MULTITENANCY ISOLATION FOR CLOUD-HOSTED GSD TOOLS

Multitenancy Patterns	Bug/Issue Tracking	Continuous Integration	Version Control
Shared Component	Can be configured using the localconfig file. This file contains the default settings for a number of Bugzilla parameters	Can be implemented using the global config file. global.jelly is a file generated automatically by Maven, and its the Jelly Script file to produce the global configuration option.	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.
Tenant-Isolated Component	The localconfig file which contains the default settings can be loaded in order to change the database type and password for a user (e.g., $ db_driver and $ db_pass).	Can be implemented using config.jelly, a Jelly script file generated automatically by Maven to produce the configuration option specific for the job.	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.
Dedicated Component	When Bugzilla combines all comments on a single bug into a field for full-text searching, its size could be more than the default size of items (i.e., 1MB). The MySQL configuration file located in: /etc/my.cnf can be edited to allow for insertion of large attachments into Bugzilla database.	Pre- and post-build actions (e.g., SCM, archiving files) can be enabled on the configuration page to add a special functionality to Hudson jobs. For example, a repository path could be dedicated for archiving files after a successful build operation.	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 .

[5] L. Ochei, J. Bass, and A. Petrovski, "Taxonomy of deployment patterns for cloud-hosted applications: A case study of global software development tools," *IARIA*, 2015.

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

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

[8] 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.

[9] 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

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

[11] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno, "Collaboration tools for global software engineering," *Software, IEEE*, vol. 27, no. 2, pp. 52–55, 2010.

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

[13] Atlassian.com. Atlassian documentation for jira 6.1. Atlassian, Inc. [Online]. Available: https://www.atlassian.com/software/jira/

[14] B. Wilder, *Cloud Architecture Patterns*, 1st ed., R. Roumeliotis, Ed. O'Reilly Media, Inc., 2012.

[15] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, *Cloud Design Patterns*, R. Corbisier, Ed. Microsoft, 2014.

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

[17] 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*. IEEE, 2007, pp. 551–558.

[18] 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.

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

[20] 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].

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

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

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

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

[25] Oracle. Oracle database concepts 10g release 1 (10.1). Oracle Corporation. [Online]. Available: http://docs.oracle.com/cd/B1203701/server.101/b10743/toc.htm

[26] 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].