Implementing the Required Degree of Multitenancy Isolation: A Case Study of Cloud-hosted Bug Tracking System

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*—Implementing the required degree of isolation between tenants is one of the significant challenges for deploying a multitenant application on the cloud. In this paper, we applied 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 a cloud-hosted Bug tracking system using Bugzilla. The study revealed among other things that a component deployed based on dedicated component offers the highest degree of isolation (especially for database transactions where support for locking is enabled). Tenant isolation based on performance (e.g., response time) favoured shared component (compared to resource consumption (e.g., CPU and memory) which favoured dedicated component). We also discuss key challenges and recommendations for implementing multitenancy for application components in cloud-hosted bug tracking systems with guarantees for isolation between multiple tenants.**

Keywords*—*Multitenancy, Degree of Isolation, GSD tools, Cloud Patterns, Bug tracking

I. Introduction

Software tools used for Global Software Development (GSD) projects are increasingly being deployed on the cloud [1] [2] [3]. As these tools are used by multiple users/tenants, there is need to ensure that proper isolation of both the data (e.g., source code files) and processes (e.g., builds) associated with these tools. Therefore the challenge for an architect would be to: (i) implement multitenancy so that a single instance of an application/component is used to serve multiple tenants [4][5]; and (ii) implement the required degree of isolation between tenants so that the performance, resource utilization and access rights of other tenants is not affected by one of the tenants accessing the component or functionality of a shared application component [6] [4].

There are varying degrees of isolation that can be implemented for a cloud-hosted GSD tool. For example, certain laws and regulations would impose a much higher degree of isolation between tenants accessing an application component than if the same component needed some re-configuration. Therefore, an architect has to resolve the trade-offs between the required degree of isolation, and the performance, resource consumption and access privileges at different levels of an application when implementing multitenancy isolation.

Motivated by this problem, this paper evaluates the degree of isolation between tenants enabled by multitenancy patterns using a bug tracking system as a case study in order to resolve these trade-offs under different cloud deployment conditions. This study is inspired by the work of Fehling *et al.* [4] where the authors captured the degrees of multitenancy in three cloud deployment patterns: shared component, tenant-isolation component and dedicated component. They also suggested that the varying degrees of isolation between tenants is the main factor that can be used to distinguish between these cloud patterns. However, the various cloud deployment conditions are often unknown which offers the required degree of isolation, and the implication of specific cloud resources like CPU, memory and disk I/O. In addition, these patterns have never been evaluated empirically to determine the practicality of determining the degree of isolation between tenants for applications in software engineering domain. This study is one of the three primary case studies on applying our novel approach, COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing) to empirically evaluate the degree of isolation between tenants enabled by multitenancy patterns within the context of cloud-hosted GSD tools (in this case bug tracking system) under different cloud deployment conditions.

We evaluate the degrees of multitenancy isolation by comparing the effect of resource utilization (e.g., CPU and memory) and performance (e.g., response times and error%) on tenants deployed based on different multitenancy patterns (i.e., shared component, tenant-isolated component, and dedicated component) when one of the tenants experiences a demanding deployment conditions (e.g., a sudden increase in 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 bug tracking system"**. Multitenancy isolation introduces significant security and performance challenges in the cloud depending on the location of the functionality to be shared on the cloud application stack, and the required degree of isolation

between the tenants. For example, if one of the tenants on the network is malicious, it can cause performance degradation and denial of service to other tenants [7].

To implement multitenancy (based on multitenancy patterns), the Bugzilla database was reconfigured in a way that isolates ***bugs*** from different tenants (see Fig. 2). The degree of isolation was then evaluated for each pattern at two levels: process isolation and data isolation; as it affects tenants interaction with the bug tracking system. The overarching result of the study is that a component deployed based on dedicated component offers the highest degree of isolation (especially for database transactions where support for locking is enabled). Multitenancy isolation based on performance (e.g., response time) favoured shared component (compared to resource consumption (e.g., CPU and memory) which favoured dedicated component).

The main contributions of this paper are:
1. Applying an enhanced COMITRE algorithm to: (i) empirically evaluate the required degree of multitenancy isolation between tenants enabled by multitenancy patterns for a cloud-hosted bug tracking system; and (ii) compare how well different multitenancy patterns perform under different cloud deployment conditions.
2. Presenting recommendations and best-practice guidelines for implementing the required degree of multitenancy isolation for cloud-hosted bug tracking systems, and their implications for optimizing performance and cloud resources (e.g., RAM, CPU, disk space) based on different cloud deployment scenarios.

The rest of the paper is organized as follows - Section II discusses the concept of multitenancy isolation as it affects implementing the required degree of isolation between tenants using cloud-hosted bug tracking system. In Section III, we discuss the methodology including GSD tool selection, and application of COMITRE algorithm to implement and evaluate the required degree of multitenancy isolation. Section IV presents the results and then discusses the implications of the results in Section V. The recommendations and limitations of the study are detailed in Section VI and VII, respectively. Section VIII concludes the paper with future work.

II. Multitenancy Patterns for Deploying Cloud-hosted Bug Tracking System using Bugzilla.

In this section, we discuss the concept of multitenancy isolation on the bug tracking system within the context of Global software development.

A. The Role of a Bug Tracking System in Global Software Development (GSD)

In recent times, software tools used for Global Software Development have been moving to the cloud. We call these tools *Cloud-hosted GSD tools*. Examples of these software tools are Hudson (used for continuous integration), Subversion (used for version control) and Bugzilla (used for bug tracking).

Bug tracking (or issue tracking) is the process of keeping track of reported software bugs or issues in software development projects. This paper focuses on Bugzilla, a web-based general-purpose bug tracker and testing tool, originally developed and used for the Mozilla project [8]. Other examples of bug tracking tools are JIRA, ITracker, Rational ClearQuest, and TrackStudio. *Bug tracking*, as used in this paper, also includes issues and enhancements to an application and not only restricted to error-related data such as stack traces and log files. However, we do not include task registry, which is more related to the function of a project management system [9].

The main component of a bug tracking system is the database that stores bugs and attachments, which require isolation. Attachments are usually added to compliment the process of submitting a bug. Developers are encouraged to use attachments instead of comments especially for large chunks of ASCII data, such as trace, debugging output files, or log files [8]. These attachments have to be isolated as bugs can be assigned to different teams members for resolution.

B. Evaluating Degree of Multitenancy Isolation

We define "Multitenancy isolation" as a way of ensuring that the performance, stored data volume and access privileges required by one tenant does not affect other tenants accessing the component or functionality of a shared application component [10] [11]. There are three multitenancy patterns which express the degree of isolation between tenants accessing an application component: shared component, tenant-isolated component and dedicated component [4]. The dedicated component represents the highest degree of isolation whereas shared component represents the lowest. The degree of isolation for tenant-isolated component would be in the middle.

There are three main aspects of tenant isolation, namely, performance, stored data volume and access privileges [4]. In performance isolation for example, one tenant is not supposed to be affected by the workload generated by other tenants. Guo et al [12] evaluated different isolation capabilities related to authentication, information protection, faults, administration etc. Bauer and Adams [5] discussed how to use virtualization to ensure that the failure of one tenants' instance does not cascade to other tenants' instances. The work of Walraven et al [13] is closely related to ours. By using a multitenant implementation of a hotel booking application deployed on top of a cluster, the authors implemented a middleware framework for enforcing performance isolation. Krebs et al [14] implemented a multitenancy performance benchmark for web application based on the TCP-W benchmark which was used to evaluate where the maximum throughput and the amount of tenants that can be served by a platform. Other works related to multitenancy isolation can be seen in [15] [16] [17].

The focus of this paper is to evaluate the degree of isolation between tenants enabled by multitenancy patterns, and thus provide empirical evidence of their effects on performance, resource utilization and access privileges on other tenants due to high workload created by one of the tenants. In our work, we implemented multitenancy isolation by reconfiguring the Bugzilla database to support varying degree of isolation between tenants. In addition, our evaluation is done in a real cloud environment. The application used for our evaluation is within the domain of software engineering, to emulate a typical software development process. Furthermore, we also deployed

Bugzilla to the cloud based on the three different types of cloud multitenancy patterns.

III. Methodology

This study sits within the framework of a cross-case analysis methodology which will be used later to synthesis the results of the three case studies to draw conclusions and implication for implementing the required degree of isolation between tenants. Therefore, this study adopts the same methodology used in two previous case studies to allow for transparency and ease of evaluation of evidence.

A. Selecting the GSD Tools and Software Processes

A previous work conducted to find out the type of GDS tools (and associated tasks) used in large scale distributed enterprise software development projects produced a dataset of five GSD tools: JIRA, VersionOne, Hudson, Subversion and Bugzilla (see Ochei et al [3] and Bass [18] for details). Based on this dataset, three software processes which have the highest impact on Global Software Development were selected: continuous integration (CI), version control (VC) and issue/bug tracking. Two of these software processes have been used previously in two separate case studies to empirically evaluate the degrees of multitenancy isolation based on our novel approach for implementing multitenancy isolation (i.e., COMITRE) [10] [11]. In this paper, we focus on applying COMITRE to implement the required degree of multitenancy isolation in a bug tracking system.

B. Applying COMITRE to Evaluate the Degrees of Multitenancy Isolation in Bugzilla

In a nutshell, COMITRE's implementation relies on shifting the task of routing a request from the web server to a separate component (e.g., a Java class or plugin) at the application level of a cloud-hosted GSD tool. Figure 1 captures the structure of COMITRE while the logic used to implement it shown in Algorithm 1. We have presented the full explanation of COMITRE (including its step-by-step procedure) in Ochei et al. [10] [11].

This paper applies an improved version of the algorithm (i.e., in terms of adding more details to achieve generality and applicability in different environments) used to implement COMITRE for evaluating the degrees of isolation between tenants for a particular multitenancy pattern. Assuming the required isolation level of tenants is set to 1, 2, and 3 for the three different multitenancy patterns, the logic can then be summarized as follows: (i) if the isolation level is 1, then a tenant can access the created component irrespective of where it is located; (ii) if the isolation level is 2, the tenant is first authenticated, and then assigned a tenantID. This ID is linked with a specific configuration of the tenant which is then used to adjust the behavior of the created component; and (iii) if the isolation level is 3, then the created component is tagged as unsharable, and so is dedicated exclusively for one tenant.

Bugzilla was modified using the recommended *Bugzilla Extension* mechanism. Extensions can be used to modify either the source code or user interface of Bugzilla, which can then be distributed to other users and re-used in later versions of the software. Bugzilla maintains a list of *hooks* which represent

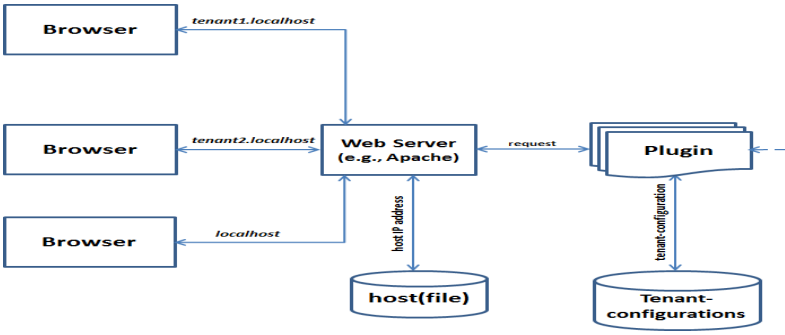

Fig. 1. Architectural diagram of COMITRE approach

areas of Bugzilla that an extension can hook into, thereby allowing the extension to perform any required action during that point in Bugzilla's extension. For our experiments, we developed a special extension and then "'hooked"' it into Bugzilla using the hook called *install_before_final_checks*. This hook can be used to execute any custom logic before the final checks are carried out by *checksetup.pl*, and so we implemented COMITRE logic in this hook [8]. The two main processes we wanted to capture in Bugzilla are: (i) creating a bug, and (ii) adding an attachment specific to a bug. Creating a simple bug with attachment in Bugzilla requires access to three main tables: bugs, attachments, and attach_data. Most bug tracking systems like JIRA and Bugzilla use a database to store bugs/issues created by users during the software development process. To simulate this process in Apache JMeter, we use the JMeter Beanshell sampler to invoke two separate custom Java classes that run a query that: (i) inserts multiple bugs with attachments into the Bugzilla database concurrently, and (ii) sets the database transaction isolation level to SERIALIZABLE (i.e., the highest isolation level) during bug creation with attachment.

C. Evaluation

A set of four tenants (T1, T2, T3, and T4) were configured into three groups (i.e., the three multitenancy patterns) to access an application component deployed based on three different multitenancy patterns. We created three different scenarios for all the tenants to evaluate the effect of multitenancy isolation at both data and process levels. We describe the scenarios as follows: (i) scenario 1- concurrent release of requests, represents a case where large instant bugs submitted concurrently to the database; (ii) scenario 2- variation in inter-arrival times of requests, represents a case where there is variation in frequency with which bugs are submitted to the database; and (iii) scenario 3- enabling support for locking, represents the use of locking to prevent conflicts between multiple tenants attempting to access a bug database. Fig. 2 shows a sample architecture of multitenancy isolation involving three tenants at the data level. For multitenancy isolation at the process level, the component that is being shared is a lock object [10] [11]. At the process level, the component that is being shared is a lock object [10] [11]. The above scenarios are very important in the so called *distributed bug tracking* in which some bug trackers such as Fossil and Veracity, are either integrated with or designed to use distributed VCS or CI systems, thus allowing bugs generated automatically and

Algorithm 1 COMITRE Algorithm

```
INPUT: tenantRequest, tenantConf-file, isolationLevel
OUTPUT: multApplFunctn
Get tenantID from incoming request
tenantConf ← null
share ← true
Select tenantData from tenantConf-file
if tenantData is found then
    tenantConf ← tenantData
end if
Create defaultApplFunctn
multApplFunctn ← defaultApplFunctn
if tenantConf is not null then
    if isolationLevel = 1 then
        Create tenantApplFunctn
    else if isolationLevel = 2 then
        Authenticate tenantID
        Create tenantApplFunctn
        Adjust tenantApplFunctn with tenantID
    else if isolationLevel = 3 then
        Create tenantApplFunctn
        share ← false
    end if
    multApplFunctn ← tenantApplFunctn
end if
return multApplFunctn
```

added to the database at varying frequencies.

In order to measure the effect of isolation between tenants, we configured one of the four tenants to represent a *treatment* (i.e., T1 in the case) for each group to experience a large instant loads while accessing the application component. Measurements (e.g, response time, CPU usage) were then taken before the treatment (pre-test) and after the treatment (post-test). Apache JMeter is used to simulate large instant loads as follows: (i) increasing the number of requests using the *thread count* and *loop count*; (ii) attaching a large file to the request to increase its size; (iii) increasing the speed of sending requests by reducing the ramp-up period by 10 percent so that requests are sent faster.; and (iv) creating a heavy load burst using the Synchronous Timer so that a certain number of request are fired at the same time. This type of configuration can be likened to unpredictable (i.e., sudden increase) workload [4] and aggressive load [13].

The experiments were conducted on a UEC (Ubuntu Enterprise Cloud) private cloud based on a typical minimal Eucalyptus configuration. The values for the experimental setup are as follows: (1) Number of threads = 5; (2) Thread Loop count = 2; (3) Loop controller count = 20 for tenant 1, and 10 for all other tenants; (4) Ramp-up period: 6 seconds for tenant 1 and 60 seconds for all other tenants; and (5) Size of bug attachment = 1MB for tenant 1 and 200KB for all other tenants. With this setup, the requests sent by tenant 1 are two times more, five times heavier, and ten times faster, than the other tenants. Ten iterations were performed for each run and the values reported by JMeter were taken as measure for response times, throughput and error% (i.e., the percentage of the total number of requests whose response time is unacceptably slow and above which the request is considered a failure). The system activity report (SAR) tool was used to report the average memory, CPU, disk I/O and system load.

As in previous case studies, we performed three main statistical test (i.e., two-way ANOVA, one-way ANOVA followed by Scheffe post hoc, paired sample test) to show whether or not the performance and resource utilization of other tenants have been affected by the workload generated by one of the tenants [11]. For example, the paired sample test was used to determine if the subjects within any particular group changed significantly from pre-test to post-test measured at 95% confidence interval.

The ***aim of the experiment*** is to evaluate the degrees of multitenancy isolation for cloud-hosted bug tracking system. The experimental hypothesis states 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***. A summary of the experimental procedure we adopted can be seen in Ochei et al [10] [11].

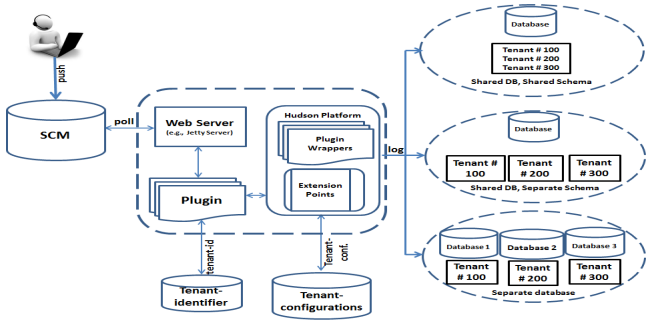

Fig. 2. Multitenancy Data Isolation Architecture

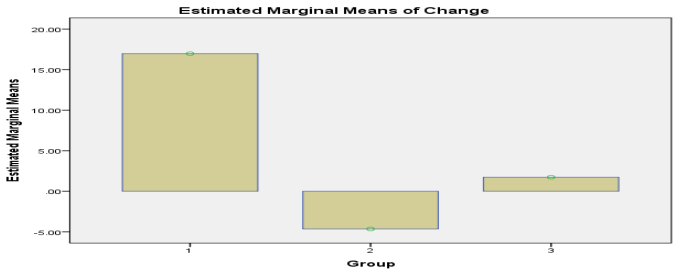

Fig. 3. Changes in response time for each pattern relative to other patterns-1

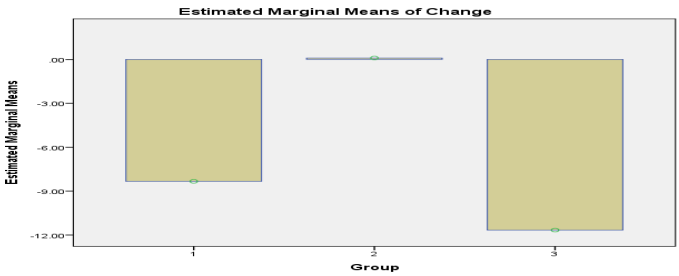

Fig. 4. Changes in error% for each pattern relative to other patterns-1

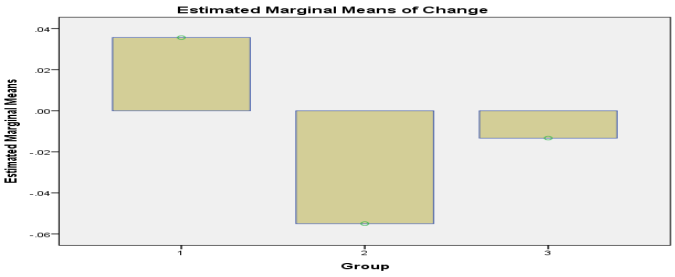

Fig. 5. Changes in throughput for each pattern relative to other patterns-1

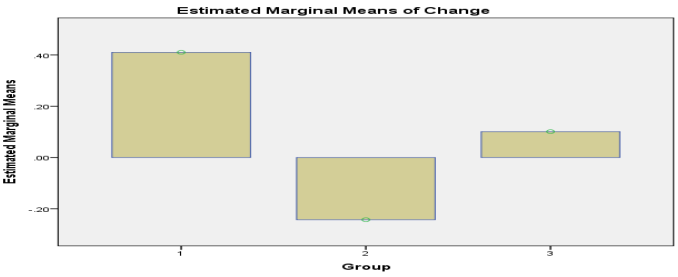

Fig. 9. Changes in system load for each pattern relative to other patterns-1

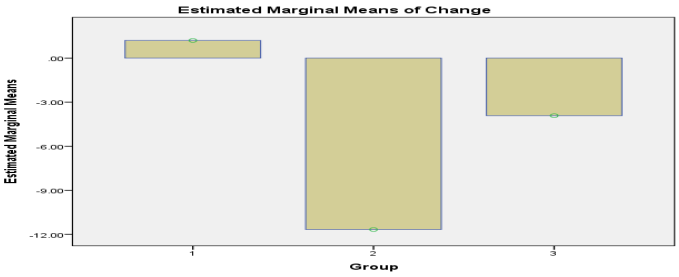

Fig. 6. Changes in CPU for each pattern relative to other patterns-1

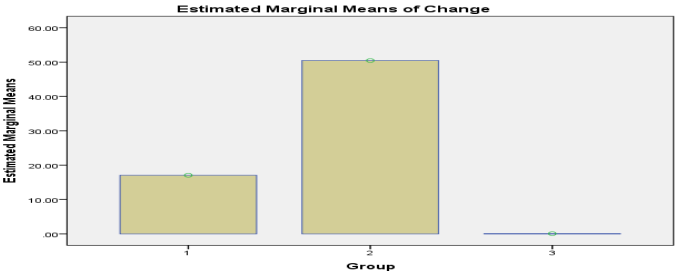

Fig. 10. Changes in response time for each pattern relative to other patterns-3

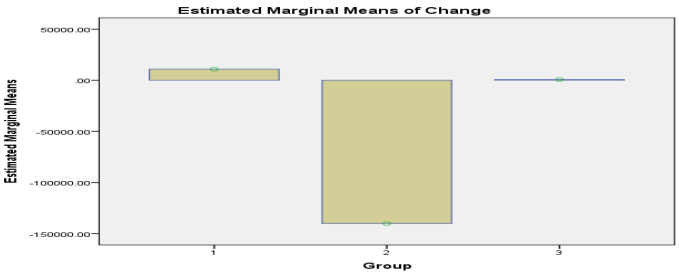

Fig. 7. Changes in memory for each pattern relative to other patterns-1

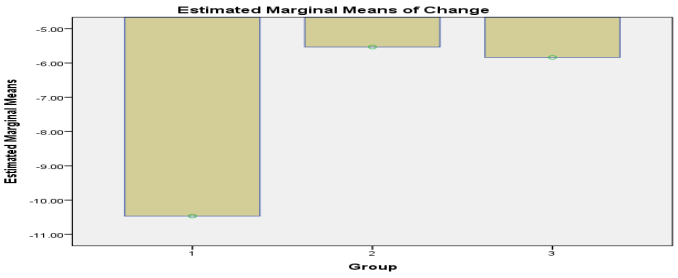

Fig. 11. Changes in error% for each pattern relative to other patterns-3

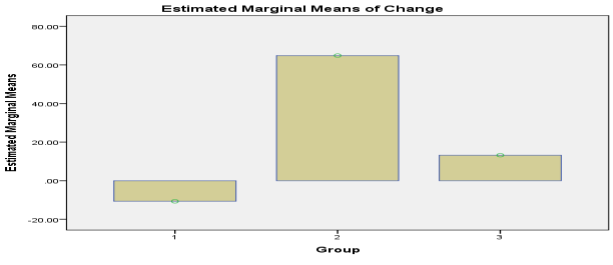

Fig. 8. Changes in disk I/O for each pattern relative to other patterns-1

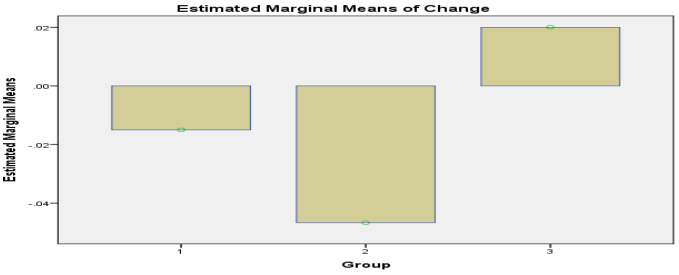

Fig. 12. Changes in throughput for each pattern relative to other patterns-3

IV. Results

The experimental results were analyzed using a combination of plots of estimated marginal means of change (EMMC) derived from ANOVA (plus post hoc test) and paired sample test results from SPSS. *Due to space limitations, we show only EMMC for scenario 1 and 2 as shown in Fig. 3 to Fig. 16).* Table 1 summarizes the effect of T1 (which experiences large instant loads) on the other three tenants (T2, T2, T4). The notation used in Table 1 are: (i) YES - significant change;(ii) NO - no significant change; and (iii) the symbol "-" means no chance of variability. A summary of the results based on the paired sample test and the estimated marginal means of change is presented as follows:

(1) Response times and Error%: The post hoc test showed that there was significant difference between shared and tenant-isolated component and between tenant-isolated and dedicated component. From plots of EMMC, it is clear that dedicated component showed the lowest magnitude of change in response time, and so it is recommended for achieving isolation between tenants accessing bugs database. The post hoc test results for error% was similar to that of response times. The plots of EMMC shows that the number of requests with unacceptably

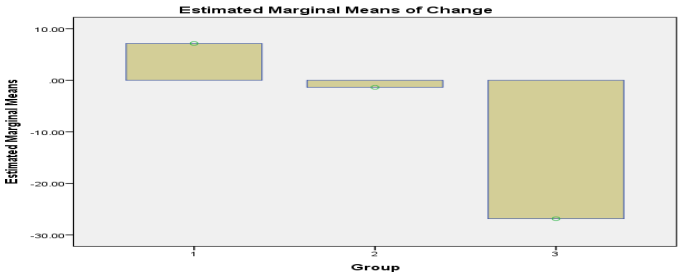

Fig. 13. Changes in CPU for each pattern relative to other patterns-3

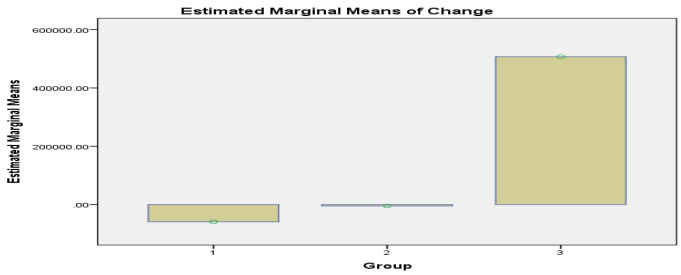

Fig. 14. Changes in memory for each pattern relative to other patterns-3

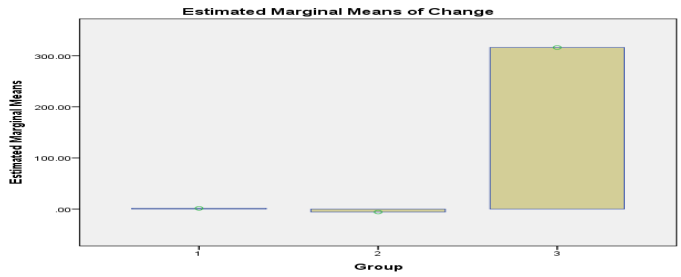

Fig. 15. Changes in disk I/O for each pattern relative to other patterns-3

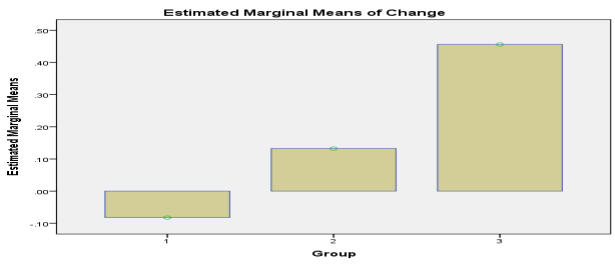

Fig. 16. Changes in system load for each pattern relative to other patterns-3

slow response times was much higher for shared components (compared to tenant-isolated and dedicated components). This is due to the effect of locking on the database which causes delay on the time it takes for a request to be committed.

(2) Throughput: There was no significant change based on the paired sample test. In fact, this result is very similar to that of the other two scenarios, where the throughput was fairly stable. This seems to suggest that if the application component being shared is a database, then we should not expect throughput to change drastically.

(3) Memory and Disk I/O: The post hoc test and paired sample test confirmed that there was a highly significant change in both memory and disk usage across the three patterns, and the paired sample test for both the memory and disk I/O showed a highly significant difference from pre-test to post test. The plot of the EMMC similarly showed that the dedicated component had the highest significant change compared to the other patterns, and so should not be used to run Bugzilla on runtime-libraries that by nature consumes much memory. For example, it is well known that mod_perl support in Bugzilla consumes a lot of RAM [8]. For disk I/O consumption, having enough storage space would be required, especially if we expect large volume of bugs with attachments.

(4) CPU and System load: The paired sample test showed that CPU usage changed significantly for all the patterns. By analyzing the plots of the EMMC, the results show that the dedicated component changed the most and so would not be recommend for optimizing CPU usage in addition to achieving multitenancy isolation. As with other case study results, there was no influence on the any of the patterns. Interesting, the plots of EMMC showed an increasing trend from shared component to dedicated component.

V. Discussion

(1) Response time, Error% and Throughput: The results showed that dedicated component is generally highly recommended to improve performance, while the reverse holds for throughput. The implication is that using a dedicated component will consume more memory and CPU. To address this challenge, we suggest storing large bug attachments on disk and then store the links to these files on the bug database. This can help to improve performance. Also when transferring large bug attachments across a low network bandwidth, compressing the data could improve speed and throughput.

(2) Disk I/O: The results showed that disk I/O changed significantly in terms of disk consumption in all the patterns and for most of the scenarios. The only exception was for scenario 3 where we would not recommend the dedicated component when locking is enabled. Based on this information, we conclude that there would be no meaningful difference in the disk I/O consumption which may be slightly on the high side. It would be necessary to have enough disk space, especially if we expect large volume of bugs with attachments. Also if error files are setup (i.e., errorlog in Bugzilla) then these files should be purged from time to time to save disk space.

(3) Memory and CPU: Bug trackers are not known to consume much memory and CPU. However, there are a few bug tracking related operations that could affect memory and CPU consumption. The first is the type of runtime time library used to support web server running the bug tracker. For example,if you are running Bugzilla under mod_perl, then using a dedicated component would not be a good option for optimizing memory, especially when locking is enabled on the bug database. It is well known that mod_perl support in Bugzilla consumes a lot of RAM [8]. Compressing the size of large bug attachments could improve performance but the shortcoming is that it will consume much CPU.

(4) System load: The results showed that there was no chance of variability, and there is no influence on any of the patterns. A possible explanation for this is that the configuration of the running VM instance, the nature of the tasks, and absence of piled-up task queue for a long time being processed resulted in a reasonably good throughput. In most cases, if the load

TABLE I. PAIRED SAMPLES TEST ANALYSIS OF TENANT ISOLATION FOR SCENARIO 1, 2 AND 3

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

average is less than the total number of processors in the system, this suggests that the system is not overloaded and so it is assumed that nothing else influences the load average.

VI. Recommendations and Limitation

In this section, we discuss the limitations of the study and also provide some recommendations that will help in implementing the required degree of multitenancy isolation for bug tracking systems. A summary of recommended multitenancy patterns for realizing isolation between tenants based on different scenarios is shown in Table 2. For example, we would recommend using shared component to improve disk I/O consumption and the dedicated component to improve response time when locking is enabled on the bug database.

Volume of bug data: Bug trackers, unlike version controls systems, do not generate additional copies of files. A large disk size would be required to accommodate the volume of bugs generated, if dealing with a large user base. Moreover, the submitted bugs may also be associated with large file attachments which could weigh down the database. To address this problem, large files/attachments could be stored directly on the disk while the file path to the attachments are stored in the bug database. The error or log files (e.g., errorlog in Bugzilla) could also be purged regularly to reduce disk space.

Optimizing the cloud resources: Bug trackers do not consume much resources like CPU, and memory. However, they could consume more CPU depending on runtime library used. For example, Bugzilla consumes huge RAM if mod_perl is enabled. The results of the experiments also shows that Bugzilla consumes a significant amount of memory if transactions in the bug database are cached.

Sensitivity to workload interference: Our experience with Bugzilla may suggest that bug trackers are sensitive to increase workload especially if locking is enabled for the bug database. We noticed frequent crashes of Bugzilla database in our experiments which required recovery. There are also numerous database related errors. We recommend increasing the maximum size of file that can be stored in the database. It may also be necessary to remove restriction on the maximum number of allowed queries, connections and packets etc.

Clients accessing bug database with low latency and bandwidth: If a client with low bandwidth is accessing the bug database it may be necessary to compressed large bug attachments before moving the data across the network. However, there is a price to pay in term in terms of high CPU utilization.

Implementing multitenancy isolation for bug trackers on different layers of the application stack: Depending on the layer of the application stack, multitenancy isolation for the bug database may be realized differently with associated implications. For example, implementing the shared component on the SaaS layer ensures efficient sharing of cloud resources, but isolation is either very low or not guaranteed at all. Implementing dedicated component on the IaaS layer would require installing the bug database for each tenant on its own instance of virtual hardware. This increases the runtime cost and limits the number of tenants that can be served.

With regards to limitations of the study, this work applies to open-source cloud-hosted bug tracking that use relational databases to store bugs/issues. Due to limitations on the capacity of the private cloud (i.e., Ubuntu Enterprise Cloud), we used large instant requests to create a high workload within the limit of the private cloud. Therefore, the results is applicable to private clouds and not be generalized to large public clouds. This study assumes that a small number of users send multiple requests; it would be interesting to replicate this experiments in a large private cloud infrastructure to investigate the effect of large number and size of users.

One of the most challenging aspects of the study was resolving database related errors, for examples, exceeding limit of file size, query, connections etc. Therefore it is necessary to modify the bug database to remove these restrictions. The bug database running on the VM instance can be quite sensitive to workload changes depending on the size, volume of bugs, and bug database isolation level, and so it is important to carefully vary the number requests that would cope with the size of the cloud infrastructure used before running the experiments.

VII. Conclusion and Future Work

In this paper, we have implemented multitenancy by applying COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing), to contribute to literature on multitenancy isolation for cloud-hosted Bug

TABLE II. Recommended Multitenancy Patterns for optimizing performance and resource utilization

Case Studies	Aspects of Isolation	Factors	Scenario 1			Scenario 2			Scenario 3		
			Sh	*Te*	*De*	*Sh*	*Te*	*De*	*Sh*	*Te*	*De*
Bug tracking with Bugzilla	Performance	Res Time			✓		✓	✓			✓
		Thru	✓	✓	✓		✓	✓	✓		✓
	Security	Error	✓	✓		✓	✓			✓	✓
	Resource utilization	CPU	✓				✓	✓	✓	✓	
		Memory			✓	✓		✓	✓	✓	
		Disk I/O	✓	✓	✓	✓	✓	✓	✓	✓	
		System Load	-	-	-	×	-	-	-	-	-

Tracking System by showing how to implement the required degree of isolation between tenants enabled by multitenancy patterns. Three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) were implemented by reconfiguring Bugzilla database and deploying it as a Virtual Machine (VM) instance to a private cloud.

The study revealed that for transactions on bug database where support for locking is enabled, performance isolation between tenants (e.g., in terms of response time) can be improved with dedicated component while isolation with resource consumption (e.g., CPU and memory) can be improved with shared component. We also presented a summary of recommended multitenancy patterns and their implications for cloud-hosted bug tracking systems. The study recommends that during bug tracking, the storage space should be reasonably large enough to accommodate bugs with large attachments. To save disk space, bugs can be stored directly on disk while the file paths to the such bugs are stored in the database tables.

We plan to investigate how locking is used in three different GSD processes (i.e, continuous integration, version control and bug tracking) to prevent clashes between multiple tenants when trying to access a shared functionality or application component, and its implication for optimizing the deployment of the application component. Furthermore, we also plan to compare and contrast the three case studies. In the future, we will develop a decision support model for optimizing the deployment of application components of cloud-hosted software services while guaranteeing multitenancy isolation.

ACKNOWLEDGMENT

This research was supported by the Tertiary Education Trust Fund (TETFUND), Nigeria and IDEAS Research Institute, Robert Gordon University, UK.

REFERENCES

[1] R. Buyya, J. Broberg, and A. Goscinski, *Cloud Computing: Principles and Paradigms*. John Wiley & Sons, Inc., 2011.

[2] M. A. Chauhan and M. A. Babar, "Cloud infrastructure for providing tools as a service: quality attributes and potential solutions," in *Proceedings of the WICSA/ECSA 2012 Companion Volume*. ACM, 2012, pp. 5–13.

[3] L. C. Ochei, J. M. Bass, and A. Petrovski, "A novel taxonomy of deployment patterns for cloud-hosted applications: A case study of global software development (gsd) tools and processes," *International Journal On Advances in Software.*, vol. 8, numbers 3 and 4, pp. 420–434, 2015.

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

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

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

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

[8] Bugzilla.org. The bugzilla guide. The Mozilla Foundation. [Online: accessed in November, 2015 from http://www.bugzilla.org/docs/.

[9] N. Serrano and I. Ciordia, "Bugzilla, itracker, and other bug trackers," *Software, IEEE*, vol. 22, no. 2, pp. 11–13, 2005.

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

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

[12] 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.

[13] 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.

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

[15] S. Walraven, B. Lagaisse, and W. Joosen, "Application-level multi-tenancy: the promise and pitfalls of shared-everything architectures," 2014, distriNet Research Group.

[16] 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.

[17] R. Krebs and M. Loesch, "Comparison of request admission based performance isolation approaches in multi-tenant saas applications." in *CLOSER*, 2014, pp. 433–438.

[18] J. Bass, "How product owner teams scale agile methods

to large distributed enterprises," *Empirical Software Engineering*, pp. 1–33, 2014.