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

TITLE: Degrees of tenant isolation for cloud-hosted software services: a cross-case analysis.

YEAR: 2018

Publisher citation: OCHEI, L.C., BASS, J.M. and PETROVSKI, A. 2018. Degrees of tenant isolation for cloud-hosted software services: a cross-case analysis. Journal of cloud computing [online], 7, article ID 22. Available from: https://doi.org/10.1186/s13677-018-0121-8

OpenAIR citation: OCHEI, L.C., BASS, J.M. and PETROVSKI, A. 2018. Degrees of tenant isolation for cloud-hosted software services: a cross-case analysis. Journal of cloud computing, 7, article ID 22. Held on OpenAIR [online]. Available from: https://openair.rgu.ac.uk/

Ochei *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2018) 7:22
https://doi.org/10.1186/s13677-018-0121-8

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Degrees of tenant isolation for cloud-hosted software services: a cross-case analysis

Laud Charles Ochei[1*], Julian M. Bass[2] and Andrei Petrovski[1]

Abstract

A challenge, when implementing multi-tenancy in a cloud-hosted software service, is how to ensure that the performance and resource consumption of one tenant does not adversely affect other tenants. Software designers and architects must achieve an optimal degree of tenant isolation for their chosen application requirements. The objective of this research is to reveal the trade-offs, commonalities, and differences to be considered when implementing the required degree of tenant isolation. This research uses a cross-case analysis of selected open source cloud-hosted software engineering tools to empirically evaluate varying degrees of isolation between tenants. Our research reveals five commonalities across the case studies: disk space reduction, use of locking, low cloud resource consumption, customization and use of plug-in architecture, and choice of multi-tenancy pattern. Two of these common factors compromise tenant isolation. The degree of isolation is reduced when there is no strategy to reduce disk space and customization and plug-in architecture is not adopted. In contrast, the degree of isolation improves when careful consideration is given to how to handle a high workload, locking of data and processes is used to prevent clashes between multiple tenants and selection of appropriate multi-tenancy pattern. The research also revealed five case study differences: size of generated data, cloud resource consumption, sensitivity to workload changes, the effect of the software process, client latency and bandwidth, and type of software process. The degree of isolation is impaired, in our results, by the large size of generated data, high resource consumption by certain software processes, high or fluctuating workload, low client latency, and bandwidth when transferring multiple files between repositories. Additionally, this research provides a novel explanatory framework for (i) mapping tenant isolation to different software development processes, cloud resources and layers of the cloud stack; and (ii) explaining the different trade-offs to consider affecting tenant isolation (i.e. resource sharing, the number of users/requests, customizability, the size of generated data, the scope of control of the cloud application stack and business constraints) when implementing multi-tenant cloud-hosted software services. This research suggests that software architects have to pay attention to the trade-offs, commonalities, and differences we identify to achieve their degree of tenant isolation requirements.

Keywords: Multitenancy, Degree of isolation, Cloud patterns, Global software development, Software development tools, Cloud-hosted software services, Application component, Case study research, Cross-case analysis

*Correspondence: l.c.ochei@rgu.ac.uk
[1]Robert Gordon University, School of Computing and Digital Media, Aberdeen AB10 7QB, UK
Full list of author information is available at the end of the article

Background

Cloud services are applications delivered over the Internet [1] that offer users on-demand access to shared resources such as infrastructures, hardware platforms, and software application functionality [2].

Cloud services provide a dedicated instance of applications to multiple users using multitenancy. Multitenancy allows a single instance of a service to serve multiple users while retaining dedicated configuration information, application data and user management for each tenant [3].

A challenge when implementing multi-tenant services is to ensure that the performance demands and resource consumption of one tenant does not adversely affect another tenant; this is known as *tenant isolation*. Specifically, software architects need to be able to control the required degree of isolation between tenants sharing components of a cloud-hosted application [4].

Varying degrees of tenant isolation is possible, depending on the type of component being shared, the process supported by the component and the location of the component on the cloud application stack (i.e., application level, platform level, or infrastructure level) [3]. For example, the degree of isolation required for a component that cannot be shared due to strict laws and regulations would be much higher than that of a component that has to be reconfigured for some tenants with specific requirements.

We previously conducted separate case studies to empirically evaluate the degree of tenant isolation in three cloud-hosted software development process tools: continuous integration (with Hudson), version control (with File SCM Plugin and Subversion), and bug/issue tracking (with Bugzilla) [5–7]. The case studies allowed us to investigate instances (i.e., evaluating the degree of tenant isolation) of a contemporary software engineering phenomenon within its real-life context using real-world Global Software Development (GSD) tools deployed in a cloud environment [8]. To build an integrated body of knowledge from these individual case studies we decided to perform a case study synthesis. The case study synthesis allows us to extend the overall evidence base beyond the existing case studies. We can thus, make a new whole out of the parts, to provide new insights regarding degrees of tenant isolation.

As a consequence of these goals, we consider three research questions:

1. What are the commonalities and differences in the degrees of tenant isolation for our three case studies?
2. What are the deployment trade-offs for achieving the required degree of tenant isolation for our three case studies?
3. What are the key challenges and recommendations for implementing tenant isolation for our three case studies?

We conducted the case study synthesis by employing a cross-case analysis technique to identify commonalities and differences in tenant isolation characteristics across the previous case studies. The cross-case analysis allows us to triangulate evidence from each case study to build a more substantial body of knowledge on tenant isolation [8, 9]. The cross-case analysis study was carried out in three phases: data reduction, data display, and conclusion drawing and verification. These phases were conducted in an iterative manner during the analysis to reach the conclusion [9].

The main contribution of this article is to provide an explanatory framework and new insights for explaining the commonalities and differences in the design, implementation and deployment of cloud-hosted services, and the trade-offs to consider when implementing tenant isolation [10]. The contributions of this article are summarised as follows:

1. Providing patterns of commonalities and differences across the existing cases studies:

(i) the study revealed five case study commonalities: disk space reduction, use of locking, low cloud resource consumption, customization and use of plugin architecture, and choice of multi-tenancy pattern. Two of these factors have a negative impact on tenant isolation. The degree of isolation is reduced when there no strategy to reduce disk space and customization and plugin architecture is not adopted. In contrast, the degree of isolation improves when careful consideration is given to handling a high workload, locking of data and processes is used to prevent clashes between multiple tenants, data transfer between repositories and selection of appropriate multi-tenancy pattern. (see "Results" section).

(ii) our research reveals five areas of case study differences: size of generated data, cloud resource consumption, sensitivity to workload changes, the effect of the software process, client latency and bandwidth, and type of software process). The large size of generated data, high resource consumption processes, high or fluctuating workload, low client latency, and bandwidth when transferring multiple files between repositories reduces the degree of isolation. The type of software process is challenging because it depends on the cloud resource being optimised. (See "Results" section)

2. Providing a novel explanatory framework for:

(i) mapping the degrees of tenant isolation to different software processes, cloud resources and layers of the cloud application stack (see "Results" and "Analysis" sections).

(ii) explaining the different trade-offs which includes tenant isolation versus (resource sharing, the number of

users/requests, customizability, the size of generated data, the scope of control of the cloud application stack and business constraints) to be considered for optimal deployment of components with a guarantee of the required degree of tenant isolation (see "Results" and "Analysis" sections).

3. Presenting challenges and recommendation for implementing the required degree of tenant isolation. The challenges identified in this study are related to the type and location of components to be shared, customizability of the software tool, optimization of resources to cope with changing workload, hybrid cloud deployment conditions, tagging components with the required degree of isolation, error messages and security challenges during implementation. This study suggests among other things the following recommendations: (i) allowing the software architect to have more control over layers of the cloud infrastructure for configuration and provisioning of resources; (ii) splitting complex software processes into phases and isolation implemented for each phase in turn (ii) categorizing a cloud-hosted service into different aspects that can or cannot be customised (see "Discussion" section).

The rest of the paper is organized as follows - "Related work on degrees of tenant isolation for cloud-hosted services" section gives an overview of related work on degrees of tenant isolation for cloud-hosted services. "Methods" section discusses the cross-case analysis methodology used in this paper and how it fits into the overall case study research process. "Summary of the case studies" section provides a summary of the previous case studies. In "Results" section, presents the results of the cross-case analysis. "Analysis" section is a further analysis of the data produced from the cross-case analysis. "Discussion" section is the discussion of the results of the study. The limitations and validity of the study are presented in "Threats to validity" section. "Conclusions" section concludes the paper with the direction of future work.

Related work on degrees of tenant isolation for cloud-hosted services

Research work on multitenancy has acknowledged that there are varying degrees of isolation tenants accessing a multitenant cloud-hosted service [11–14]. Chong and Carraro discuss three approaches to managing multitenant data, and it is stated that the distinction between the shared data and isolated data is more of a continuum, where many variations are possible between the two extremes [11]. Three multitenancy patterns have been identified which express the degree of isolation between tenants accessing a shared component of an application [3]. These patterns are referred to as shared component, tenant-isolated component and dedicated component. 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.

Wang et al. explored key implementation patterns of data tier multi-tenancy based on different aspects of isolation such as security, customization and scalability [12]. For example, under the resource tier design pattern, the authors identified the following patterns: (i) totally isolated (dedicate database pattern); (ii) partially shared (dedicate table pattern); and (iii) totally shared (share table pattern). These patterns are similar to the shared component, tenant-isolated component and dedicated component patterns at the data tier, respectively [3]. Vengurlekar describes three forms of database consolidation which offer differing degrees of inter-tenant isolation as follows: (i) multiple application schemas consolidated in a single database, multiple databases hosted on a single platform; and (iii) a combination of both [13].

Mietzner et al. described how the services (or components) in a service-oriented SaaS application can be deployed using different multi-tenancy patterns and how the chosen patterns influence the customizability, multi-tenant awareness and scalability of the application [15]. These patterns are referred to as a single instance, single configurable instance and multiple instances. Although this work describes how individual services of a SaaS application can be deployed with different degrees of customizability, we believe that these concepts are similar to different degrees of isolation between tenants.

The three main aspects of tenant isolation are performance, stored data volume and access privileges. For example, in performance isolation, other tenants should not be affected by the workload created by one of the tenants. Guo et al. evaluated different isolation capabilities related to authentication, information protection, faults, administration, etc [16]. Bauer and Adams discuss how to use virtualization to ensure that the failure of one tenant's instance does not cascade to other tenant instances [4].

Walraven et al implemented a run-time enforcement of performance isolation to comply with tenant-specific SLAs over distributed environments for multitenant SaaS using a real-world workflow-based SaaS offering (i.e., an online B2B document processing) executing on a JBoss AS-based private cloud platform [14]. Walraven et al. has also implemented a middleware framework for enforcing performance isolation using a multitenant implementation of a hotel booking application deployed on top of a cluster for illustration [17]. Krebs et al. implemented a multitenancy performance benchmark for a web application based on the TCP-W benchmark where the authors evaluated the maximum throughput and the number of tenants that can be served by a platform [18].

Youngs et al. discussed the decomposition of application functionality into application components and later the summarization of these components to tiers to achieve isolation between tenants [19]. Varia's research work generally motivates why applications should be split into separate components when using the Amazon Web Services on the cloud to guarantee tenant isolation [20, 21]. Varia has also discussed different migration scenarios for existing applications to Amazon Web Services (AWS) or other cloud storage [22].

At the very basic degree of multitenancy, tenants share application components as much as possible which translates to increased utilisation of underlying resources. However, while some application components may benefit from a 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.

A component-based approach to tenant isolation through request re-routing, (COMITRE), was developed in our previous work. This approach was then applied to three case studies that empirically evaluated the varying degrees of isolation between tenants enabled by multitenancy patterns for three different Global Software Development (GSD) tools and associated processes: continuous integration with Hudson, version control with File System SCM plugin, and bug tracking with Bugzilla [5–7]. The GSD tools were deployed to the cloud using different cloud multitenancy patterns which represent varying degrees of isolation between tenants.

The aim of this paper is, therefore, to extend the overall evidence beyond the existing case studies, by synthesizing the findings of the three primary case studies that have been conducted to evaluate the degree of isolation between tenants accessing different cloud-hosted GSD tools deployed using different multitenancy patterns. The findings will provide the commonalities and differences across the existing case studies, and an explanatory framework for mapping degrees of tenant isolation to different software processes and explaining the trade-offs to be considered for optimal deployment of components with a guarantee of the required degree of tenant isolation.

Methods

In this section, we will first present the cross-case analysis method used in this paper and thereafter discuss how the cross-case analysis phase fits into the overall research procedure from the exploratory phase to the development of the explanatory framework.

Cross-case analysis

This study uses cross-case analysis to synthesize the findings of the three primary case studies. In a cross-case analysis, evidence from each primary study is summarised and coded under broad thematic headings, and then summarised within themes across studies with a brief citation of primary evidence. A cross-case analysis was selected because it involves a highly systematic process and allows us to include diverse types of evidence [9, 23].

As shown in Fig. 1, the methodology can be divided into three phases: gathering findings from primary case studies, cross-case analysis and framework development. In the first phase the findings from the three primary case studies are gathered, summarised and pushed into the second phase which is the cross-analysis. In the cross-case analysis phase, we mobilise knowledge from individual case studies, compare and contrast the cases, and in doing so, produce a new knowledge. This knowledge is further refined to form an explanatory framework explaining the commonalities and differences in the design, implementation and deployment of cloud-hosted applications and the trade-offs to consider when implementing tenant isolation [10].

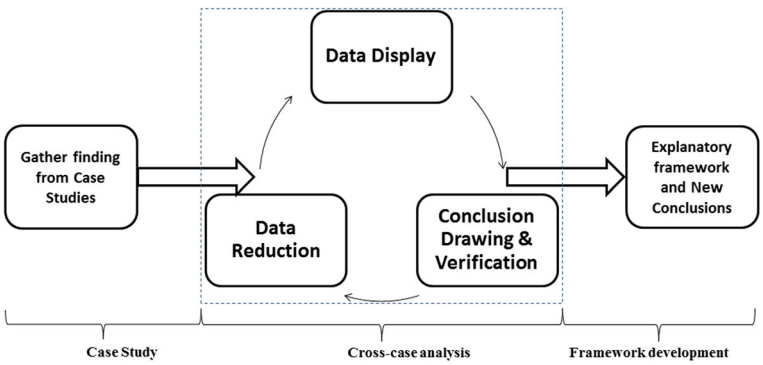

Fig. 1 Cross-analysis technique used for the study

This paper adopts Miles and Huberman's approach for conducting the cross-case analysis. The approach consists of three mains steps: data reduction, data display, and conclusion drawing and verification [9].

Data reduction

This mainly involves the identification of items of evidence in the primary case studies such as the paired sample test, plots of the estimated marginal means of change, discussion of findings and recommendations from previous case studies [9].

Data display

This step involves organising and assembling information that allows the drawing of conclusions using tools such as meta-matrices/tables and cause and relationships graphs. The data display steps will be tackled from two approaches to cross-case comparisons [24]:

(i) Variable oriented approach: This approach focuses on the variables to explain why the cases vary. These variables are related to performance and resource consumption which are known to affect the varying degrees of isolation between tenants.

(ii) Case-oriented approach: This approach focuses on the case itself instead of the variables to explain in what ways the cases are alike. By knowing the aspects in which the cases are alike it is then possible to generalise our findings, for example, to identify factors that appear to lead to a high (or low) degree of tenant isolation with a corresponding effect on resource consumption.

Conclusion drawing

This step involves further refining the above steps to produce conclusions concerning a particular aspect of interest. The outcomes of this step are a summary of: (i) key conclusions from the statistical analysis, and (ii) the recommended patterns for achieving the required degree of tenant isolation.

In summary, the focus of this paper is on a qualitative cross-case analysis of three quantitative case studies. We employed this approach because the context for each of the cases is rather different (for example, the requirements of bug tracking applications are not the same as those of version control applications) and because we are not trying to synthesise results here, but rather analyse the three cases and draw out commonalities and differences

Overall case study research process

The overall research procedure used in this study from the initial selection of the Global Software Development tools and processes up to the development of an explanatory framework (after carrying out cross-analysis of the case studies) is shown in Fig. 2. The overall research process can be regarded as a multimethod method research approach which entails combining different research methods: exploratory study, case study and cross-case analysis. The different research methods were chosen based on the qualities that each research method can contribute to answering our research question. For example, the exploratory study allowed us to carry out an empirical study to find out the type of software process and the supporting tools used in Global Software development projects and also explore the different cloud deployment patterns for deploying services to the cloud. The case study provided us with an in-depth understanding of the software processes and the effect of varying degrees of tenant isolation on the performance and resource consumption of tenants. The cross-case analysis allowed us to accumulate knowledge, compare and contrast cases,

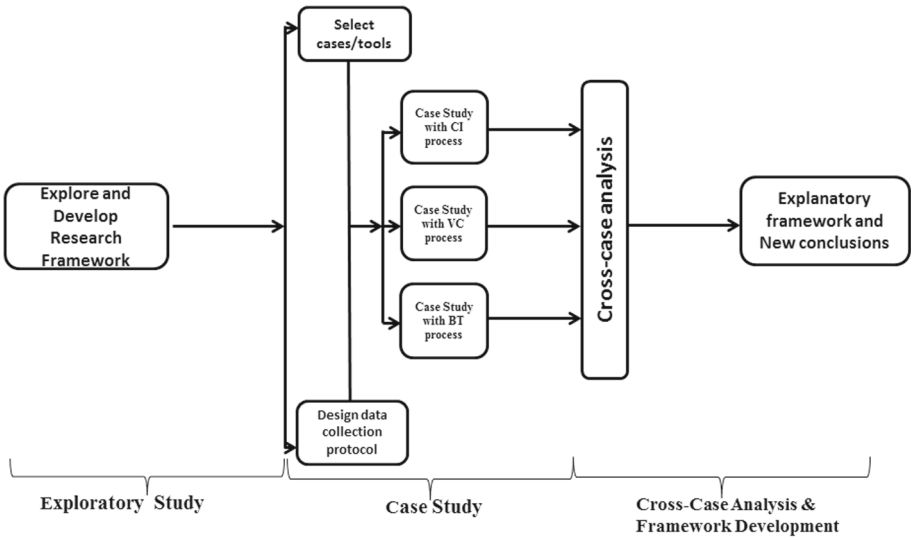

Fig. 2 Overall research process

and in doing so, produce new knowledge such as an explanatory framework for explaining the commonalities and differences in the case studies and the trade-offs to consider when implementing tenant isolation [10].

As shown in Fig. 2, the research process started with an exploratory study which entails reviewing related work on multitenancy and tenant isolation, carrying out empirical studies on widely used software tools, formulating research questions, and developing the scope of study. The next step involved selecting the software tools and supporting processes based on findings of the previous step, and then designing the experimental procedure, data collection and analysis of results of from the primary case studies. The third step, which is the focus of this study, involves carrying out the synthesis of findings of the primary case studies in order to provide an explanatory framework and new insights on tenant isolation.

In the next section, we will provide a summary of the case studies. The summary is important for two main reasons: (i) to summarise the primary case studies and show how the cross-case analysis fits into the overall research process; (ii) to summarise the findings of the case studies.

Summary of the case studies

In this section, we summarise the overall research process with particular reference to how the cases studies were conducted.

Selection of GSD tools and processes

This study used the most-similar technique (i.e., purposive, non-random selection procedure) to select the cases, since random sampling is inappropriate as a selection method [25]. The three GSD tools we selected were inspired by the dataset of GSD tools and processes derived from a previous exploratory study on how to create and use a taxonomy for selecting applicable deployment patterns for cloud deployment of GSD tools. In that study, we derived a set of five GSD tools - JIRA, VersionOne, Hudson, Subversion and Bugzilla [26]. Each of these tools supports different GSD processes such as continuous integration, issue tracking, bug tracking, version control, source code management and agile project management. From this dataset, we then selected three GSD processes that are widely used in Global Software Development: continuous integration with Hudson, version control with FileSystem SCM Plugin, and bug tracking. These GSD processes were chosen for three main reasons: (i) these processes are widely used in GSD; (ii)there are open-source tools and/or plugins that are specifically developed to support them; and (iii) they are flexible to be customize and extended.

We have to point out here that the emphasis of this study is not on the GSD tools or plugins themselves but on the GSD process they support. There are different tools and plugins that can be used to trigger these processes. Our intention was to choose a tool or plugin that is specifically developed to trigger the GSD processes we have selected. For example, Hudson can be used to trigger several GSD processes but it was mainly developed to support continuous integration. Therefore, we wanted to know how these processes and the data they generate impact on tenant isolation with respect to performance and resource utilization.

Conducting case studies

Case studies are well suited for software engineering research since they study contemporary phenomena in its natural context (i.e., real-world open-source GSD tools in our case) [27]. This study is based on three primary case studies (published separately [5–7]) which evaluated empirically the degree of isolation between tenants enabled by multitenancy patterns for Cloud-hosted GSD tools and processes under different cloud deployment conditions. Software tools used for Global Software Development (GSD) are increasingly being deployed on the cloud, and therefore it is essential to properly isolate the code files and processes of tenants so that the required performance, resource utilization, and access privileges of one tenant does not affect other tenants.

Design of the case study

The specific design of the case study is multiple-case design with multiple embedded units of analysis. This case study design represents a form of mixed method research which relies on more holistic data collection strategy for studying the main case but then calls upon more quantitative techniques (in this case, experimentation) to collect data about the embedded unit(s) of analysis [25, 27]. The experiments within the case study will enable us to collect data for evaluating the degree by which the performances of the different deployment architecture/patterns would differ under realistic cloud deployment conditions of GSD tools.

Figure 3 shows the component of the design for the first case study. Case study two and three can be captured using the same diagram by simply replacing *"The Case"* with multi-tenancy deployment patterns for version control system and bug tracking systems respectively. The main context of the study is deployment patterns for cloud-hosted services. Each case study focuses on multitenancy patterns that can be used to deploy services to the cloud. For each case study, there are three units of analysis and each unit of analysis represent each of the three multitenancy patterns (i.e., shared pattern, tenant-isolated pattern, and dedicated pattern). Each multitenancy pattern captures the varying degrees of isolation between tenants.

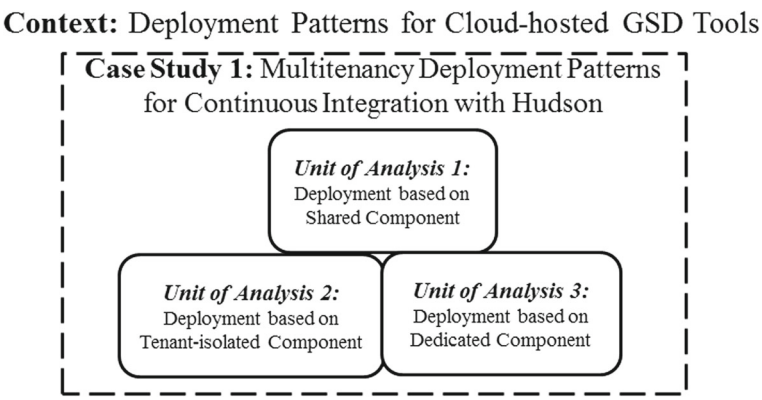

Fig. 3 Multiple-case (embedded) design adopted for the study

The results of the study were analysed using relevant statistically techniques. We adopted the Repeated Measures Design and Two-way Repeated Measures (within-between) ANOVA for the experimental design and statistical analysis respectively [28, 29].

COMITRE: a framework for implementing tenant isolation in the case studies

The implementation of the tenant isolation is done within the framework of COMITRE and its supporting algorithms. We refer the reader to [5, 6, 30] for the architecture, implementation procedure and supporting algorithms for COMITRE that are integrated into the GSD tools to support the implementation of the varying degrees of tenant isolation.

In a nutshell, the Component-based approach to Multitenancy Isolation through Request Re-routing (COMITRE) is an approach for implementing the varying degrees of multitenancy isolation for cloud-hosted services/applications. 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 4 captures the architecture of COMITRE.

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)

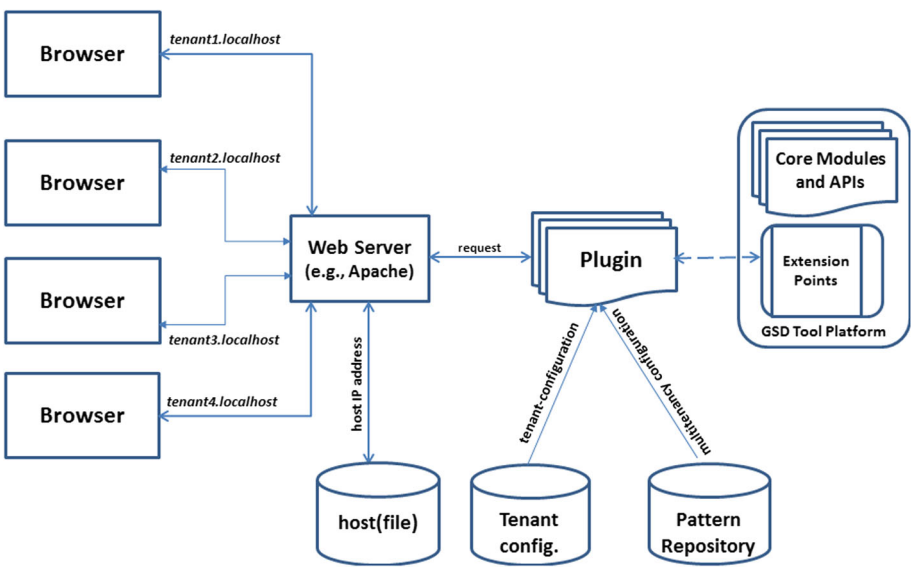

Fig. 4 COMITRE Architecture

or a software component (e.g., plugin) which can be integrated into the GSD tool. Once the request is re-routed to a component and captured, then important attributes of the request can be extracted and configured to reflect the degree of isolation required by the tenant.

Implementation of tenant isolation

In this section, we present a short description of how the three case studies were implemented. The three primary case studies have been published separately, and we refer the reader to [5–7] for details. The case studies are summarised below:

1. *Case Study One - Continuous Integration with Hudson:* Hudson is a continuous integration server, written in Java for deployment in a cross-platform environment [31]. Large organisations such as Apple and Oracle use Hudson for setting up production deployments and automating the management of cloud-based infrastructure [32]. Our main interest is in capturing the isolation of a tenant's data and process during automated build verification and testing, an essential development practice when using a continuous integration system. Tenant isolation was implemented by modifying Hudson using the Hudson's Files-Found-Trigger plugin, which polls one or more directories and starts a build if certain files are found within those directories [33]. This involved introducing a Java class into the plugin that accepts a filename as an argument. During execution, the plugin is loaded into a separate class loader to avoid conflict with Hudson's core functionality. As the build process is being carried out, data is logged into a database every time a change is detected in the file.
2. *Case Study Two - Version Control with File System SCM plugin:* Filesystem SCM plugin can be used to simulate the file system as a source control management (SCM) system by detecting changes such as the file system's last modified date [33]. Our interest was in simulating the process on a local development machine by pointing a build configuration to the locally checked out code and modified files on a shared repository residing on a private cloud. The File System SCM plugin was integrated into Hudson to represent a scenario where a code file is checked into a shared repository for Hudson to build. Tenant isolation was then implemented by modifying this plugin within Hudson. This involved introducing a Java class into the plugin that accepts a file path and the type of file(s) that should be included when checking out from the repository into Hudson workspace. During execution, the plugin is loaded into a separate class loader to avoid conflict with Hudson's core functionality.
3. *Case Study Three - Bug Tracking with Bugzilla:* 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 Bugzilla. Bugzilla maintains a list of *hooks* which represent areas in Bugzilla that an extension can hook into, thereby allowing the extension to perform any required action during that point in Bugzilla's extension [34]. For our experiments, a special extension was written and then *"hooked"* into Bugzilla using the hook named *install_before_final_checks*. This hook allows the execution of custom code before the final checks are done in *checksetup.pl*, and so the COMITRE algorithm was implemented in this hook.

In this study, it is important to note that the use of plugins in the case study experiments is basically a mechanism for customizing, modifying and extending the GSD tools (e.g., Hudson and Bugzilla) to support the implementation of multitenancy architectures and is not a limitation to providing insights about the trade-offs across the different multitenancy patterns. Most open-source tools are easily extensible using plugins by relying on a series of extension points provided to extend its functionality. For example, the extension points provided within Hudson plugin was customised to implement the three multitenancy patterns (i.e., shared component, tenant-isolated component, dedicated component) and thus support multitenancy isolation. During execution, the plugin is loaded into a separate class loader which does not conflict with the core functionality of the GSD tool.

Evaluation of the case studies

The three case studies were evaluated using the same experimental design, setup, procedure and statistical analysis. The evaluation of the case studies is summarised in the sections that follow.

Experimental design

A set of four tenants (T1, T2, T3, and T4) are configured 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. Treatment was created for configuring T1 so that it will experience a high workload. For each experimental run, one of the four tenants (i.e., T1) is configured to experience a demanding deployment condition (e.g., large instant loads) while accessing the application component.

The performance metrics (e.g., response times) and systems resource consumption (e.g., CPU) of each tenant are measured before the treatment (pre-test) and after the treatment (post-test) was introduced. The ***hypothesis*** of the experiment is that the performance and system's resource utilisation experienced by tenants accessing an application component deployed using each multitenancy pattern changes significantly from the pre-test to the post-test.

Based on this information, a two-way repeated measures (within-between) ANOVA was adopted as the experimental design. This experimental design is used when there are two independent variables (factors) influencing one dependent variable [29]. In our case, the first factor is multitenancy deployment pattern, and the second factor is time. Multitenancy pattern is the between factor because our interest is in looking at the differences between the groups using different multitenancy patterns for deployment. Time is the within factor because our interest is in 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.

Experimental setup

The experimental setup consists of a private cloud setup using Ubuntu Enterprise Cloud (UEC). UEC is an open-source private cloud software that comes with Eucalyptus [35]. Figure 5 is the UEC configuration used for the experiment. It has six physical machines - one client node(i.e, the head node) and five server node (i.e., the sub-nodes). The experimental setup is based on the typical minimal Eucalyptus configuration where all user-facing and back-end controlling components (Cloud Controller(CLC), Walrus Storage Controller (WS3), Cluster 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. The guidelines for installing UEC as outlined in [36] were followed to extend the configuration by installing the NC on all the other sub-nodes to achieve scalability. The head node was also used as the client machine since it does not have to be a dedicated machine. Installing UEC is like installing Ubuntu server; the only difference is the additional configuration screens for the UEC components. The hardware configuration of the head node and sub-nodes is summarised in Table 1.

Setup of the UEC used for experiments

Experimental procedure

An abstract summary of the experimental procedure used for all the case studies is captured in Fig. 6. We refer the reader to [5, 6] for specific details of how the GSD tools were modified for each case study, how the processes associated the GSD tools were simulated, and the experimental procedure, so these details will not be repeated here due to space limitations.

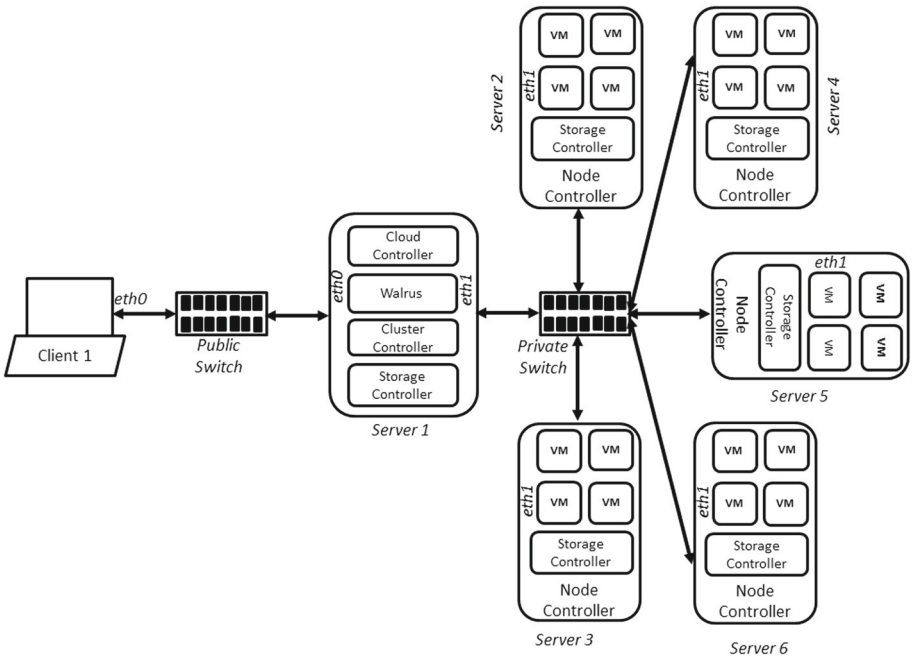

Fig. 5 Setup of the UEC used for experiments

Table 1 Hardware and Network Configuration of the UEC

	HeadNode	Sub-nodes
Hardware settings		
CPU	VT extension, 64 bit, multicore	2 x2 GHz
Memory	4 GB	2 GB
Disk	7200 rpm SATA/SCSI	7200 rpm SATA
Disk Space	80 GB	40
Networking	1 Gbps	1Gbps
Network settings		
Functionality	CLC,WS3,CC,SC	NC
No of NICs	2 (eth0 and eth1)	1(eth0)
IP Addresses	10.85.56.4	10.85.56.5-10.85.56.9
Hostname	nc1	n1, n2,n3,n4,n5
Name Servers	10.12.5.100-10.12.5.102	10.12.5.100-10.12.5.102
Gateway IP	10.85.56.3	10.85.56.3

In a nutshell, the GSD tool used for each case study is modified to support tenant isolation. This involved 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 tenant isolation between tenants, four tenants (referred to as tenant 1, 2, 3, and 4) were configured based on access to the 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 of the GSD tool's functionality that is being accessed.

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. 6 Experimental procedure

A remote client machine was used 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 [37]. To measure the effect of tenant isolation, tenant 1 was configured to simulate a *large instant load* by: (i) increasing the number of requests using the thread count and loop count; (ii) increasing the size of the requests by attaching a large file to it; (iii) increasing the speed at which the requests are sent by reducing the ramp-up period by one-tenth, so that all the requests are sent ten times faster; and (iv) creating a heavy load burst by adding the Synchronous Timer to the Samplers in order to add delays between requests, such that a certain number of requests are fired at the same time. This treatment type can be likened to an unpredictable (i.e., sudden increase) workload [3] and aggressive load [17].

Each tenant request is treated as a transaction composed of all types of request simulated. For example, the HTTP request triggers a build process while the JDBC request logs data into a database which represents an application component that is being shared by the different tenants. The transaction controller is used to group all the samplers in order to get the total metrics (e.g., response) for carrying out the two requests.

The setup values for the experiment are shown in Table 2. It is important to note that since different processes are being simulated using different GSD tools, the setup values (e.g., thread count, loop count, and loop controller count) will vary slightly. To have a balanced basis for comparison, the workload was carefully varied to cope with the private cloud used in such a way that: (i) the number of requests sent by tenant 1 (i.e., the tenant that experiences a very high workload or aggressive load) is two times more, five times heavier, and ten times faster than the other tenants; and (ii) all other tenants regardless of the type of request being simulated sends the same number of requests. Ten iterations for each run were performed and the values reported by JMeter used as a measure for response times, throughput and error%. For system activity, the average CPU, memory, disk I/O and system load usage at a one-second interval were recorded.

The generated workload for the experiments were realistic and appropriate for each of the applications in the case studies. The number and size of requests sent to the application component during the case study experiments were within the limit of the private cloud used (i.e., Ubuntu Enterprise Cloud). This was achieved by carefully varying the setup values to get the maximum capacity of the software process triggered by the GSD tool

Table 2 Setup parameters used in the experiments

Parameters	Values for case study 1	Values for case study 2	Values for case study 3
No of threads	10 for tenant 1 (i.e., the tenant experiencing high load); 5 for all other tenants	2 for all tenants	5 for all tenants
Thread Loop count	2	5 for all tenants	2 for all tenants
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 for tenant 1, and 2 for all other tenants for each type of request (HTTP, BeanShell, FTP upload and FTP download samplers)	20 for tenant 1 and 10 for all other tenants for the HTTP and BeanShell samplers
Ramp-up period	6 seconds for tenant 1, 60 seconds for all other tenants	6 seconds for tenant 1, 60 seconds for all other tenants	6 seconds for tenant 1, 60 seconds for all other tenants
Size of request	1MB for tenant 1, and 200KB for other tenants.	1MB for tenant 1, and 200KB for other tenants.	1MB for tenant 1, and 200KB for other tenants.

(e.g., Hudson's build processes) running on the private cloud before conducting the experiments.

Statistical analysis of the case studies

Based on the information about the experimental design, the two-way Repeated Measures (within-between) ANOVA was adopted for the statistical analysis because (i) it does not require many subjects since each subject would be measured twice. and (ii) it eliminates the difficulty of trying to match subjects perfectly between the different conditions in all respects [38].

The tool used for statistical analysis is SPSS v21. The two-way (within-between) ANOVA was performed first to determine if the groups had significantly different changes from Pre-test to the Post-test. After that, *planned comparisons* were carried out involving the following:

(i) a one-way ANOVA followed by Scheffe post hoc tests to determine which groups showed statistically significant changes relative to the other groups. The dependent variable (simply called "Change"), used in the one-way ANOVA test was determined by subtracting the Pre-test from Post-test values.

(ii) a paired sample test to determine if the subjects within any particular group changed significantly from pre-test to post-test were measured at 95% confidence interval. This would give an indication whether or not the workload created by one tenant affected the performance and resource utilisation of other tenants. The "Select Cases" feature in SPSS was used to select the three tenants (i.e., the T2, T3, T4 that did not experience large instant loads) for each pattern which gives a total of 3 cases for each metrics measured [28, 29].

After the first two steps outlined above, the plots of estimated marginal means were analysed in combination with ANOVA (plus post hoc test) and paired sample test results from SPSS output. These plots are referred to as the "Estimated Marginal Means of Change (EMMC)". Note that the word "Change" refers to the transformed variable used as the dependent variable in the one-way ANOVA. The plot of EMMC is simply a plot of the mean value for each combination of factor level.

Results of the case studies

In this section, we present a summary of results of each case study.

Results for case study 1 - continuous integration

The results of the case study are analysed based on the results of the paired sample t-test shown in Table 3, and supplemented with information from the plots of Estimated Marginal Means of Change(EMMC) [1]. The key used in constructing the paired sample t-test table is as follows: YES - represents a significant change between the metrics measured from pre-test to post -test. NO - represents some level of change which cannot be regarded as significant; no significant influence on the tenants. The symbol "-" implies that the standard error of the difference is zero and hence no correlation and t-test statistics can be produced. This means that the difference between the pre-test and post-test values are nearly

Table 3 Paired sample test analysis for case study 1

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

constant with no chance of variability. Figures 8, 9, 10, 11, 12, 13 and 14 in the Appendix A show the plots of the estimated marginal means of change for the parameters measured[2].

(1) Response times and Error%: Table 3 shows that the response times and error% of tenants did not change significantly except for the dedicated component. The plot of the EMMC revealed that the magnitude of change for response times showed a much larger change for the dedicated component. This is due to the overhead incurred because of opening multiple connections to the database each time a JDBC request is made to a different database. For error%, the magnitude of change was larger for tenants deployed based on the shared component than for other patterns. A possible explanation for this is that there is resource contention since multiple connections are opened while sending requests that log all the data into the same component (i.e., database table) that is being shared. Overall, this causes delay in completion times thereby producing a negative effect on error%.

(2) Throughput: The paired sample test result showed that the throughput changed significantly, implying a low degree of isolation. In this situation, the *shared component* is not recommended for avoiding a situation where requests are struggling to gain access to the same application component, thereby resulting in some request either being delayed or rejected. For a tenant-isolated component and dedicated component, there would not be much change in throughput because requests are not concentrated on one application component but instead are directed to the separate components reserved for different tenants. Throughput can be likened to bandwidth, and so it means that the bandwidth was not sufficiently large to cope with the size, number and frequency of requests sent to the CI system.

(3) CPU and System Load: The paired sample test showed that the CPU consumption of tenants did not change significantly for most patterns except for the *shared component*. 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. Builders are not known to consume much CPU. For example, Hudson does not consume much CPU; a build process can even be setup to run in the background without interfering with other processes [32].

One of the most significant findings of this study is that the system load did not influence any of the patterns. The paired sample test results were similar in all patterns; that is, the standard error difference was the same for tenants (or components) deployed using all the three multitenancy patterns. This result shows that the system load was nearly constant with no variability in the values from pretest to post-test. Therefore, in a real cloud deployment, the system load would not be a problem especially if CPU is reasonably large enough to allow the application to scale well.

(4) Memory: The paired sample test result showed that there was a significant change in memory consumption for all three patterns. Complex and difficult builds are those that are composed of a vast number of modular components including different frameworks, components developed by different teams or vendors, and open source libraries [39]. Compilers and builders consume a lot of memory especially if the build is difficult and complex [32]. In a large project, it is expected that multiple builds will interact with multiple components to create several dependencies and supported behaviour with each other thereby making builds difficult and complex.

(5) Disk I/O: Compilers and builders are known to consume disk I/O especially for I/O intensive builds [32]. The results show that only the shared component showed no significant change in disk I/O usage. This is understandable because multiple transactions are channelled to the same component which would either be delayed or blocked because of sharing the components. Further analysis of the plot of the EMMC confirmed that the magnitude of change for the shared component was the least, and therefore is recommended for builds that particularly involve intensive I/O activity especially when locking is enabled.

Results for case study 2 - version control

The results of the case study are analysed based on the paired sample t-test (shown in Table 4) and supplemented with information from the plots of Estimated Marginal Means of Change (EMMC). Figures 15, 16, 17, 18, 19, 20 and 21 in the Appendix B show the plots of the estimated marginal means of change for the parameters measured.

(1) Response times and Error%: The paired sample test results showed that response times changed significantly

Table 4 Paired sample test analysis for case study 2

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

for most of the patterns. As expected, the plot of the EMMC demonstrated that the magnitude of change for response times was much higher for the shared component and the tenant-isolated component. The results seem to show that there were no long delays that affected the error% rate. The error% showed no significant change based on the paired sample t-test. One aspect where error% (i.e., unacceptably slow response times) is known to have an impact is when committing a large number of files to a repository that is directly based on the native OS file system (e.g., FSFS). Delays usually arise when finalising a commit operation which could cause tenants requests to time out while waiting for a response.

(2) Throughput: The paired sample t-test results show that throughput changed significantly for all the patterns. Further analysis of the plots of the EMMC showed that the magnitude of change for the shared component was much higher than the other patterns. Since locking was enabled, it seems to show that it had an adverse impact on a tenant deployed based on a *shared component*. Therefore, the dedicated component would be recommended for tenants accessing bugs, especially if the bugs are stored in a database with locking enabled.

(3) CPU and System Load: The paired sample t-test showed that CPU changed significantly for all patterns. A possible reason for this is the overhead incurred in transferring data from the shared repository based on FSFS to the database (i.e., MySQL). The plot of the EMMC showed that the magnitude of change in CPU increased steadily across the three patterns with the dedicated component being the most influenced. Therefore, if there is need to avoid high CPU consumption, then the dedicated component is therefore not recommended for version control. This is because storing or retrieving bugs could involve locking or blocking other tenants from accessing a component that is being shared.

Table 4 shows that system load was nearly constant with no chance of variability, and so this means that system load did not influence any of the patterns. Therefore, with a reasonably high-speed network connection and CPU size, there should be no problem with system load when sending data across a shared repository residing in a company's LAN or VPN.

(4) Memory and Disk I/O: Memory consumption changed significantly for all patterns based on the paired sample t-test result. The plot of the EMMC showed that the magnitude of change for the shared component was higher than the other patterns. Therefore, the shared component would not be recommended when there is a need for better memory utilisation. The paired sample t-test revealed that the usage of disk I/O by tenants changed significantly from pre-test to post-test for all the patterns. This is due to the intense frequency of the I/O activities in the disk because of the file upload and download operations. The dedicated component would be recommended since this would allow each tenant to have exclusive access to the component being shared, thereby reducing a possible contention for disk I/O and other resources when the number and frequency of request increase suddenly.

Results for case study 3 - bug tracking

This section presents a summary of the experimental results for case study 3. The results of the paired sample t-test are summarised in Table 5, while the plots of the estimated marginal means of change are shown in Figs. 22, 23, 24, 25, 26, 27 and 28 in the Appendix C.

(1) Response times and Error%: From the plots of the estimated marginal means of change (EMMC), it can be seen that the dedicated component showed a lower magnitude of change in response time, and it is recommended for achieving isolation between tenants accessing bugs in a database with locking enabled. However, the plots of EMMC show that the number of requests with unacceptable response times was much higher for shared components compared to tenant-isolated and dedicated components. This is possibly due to the effect of locking on the database which causes a delay in the time it takes for requests to be committed. Using the dedicated component ensures a high degree of isolation, but with limitations of increased resource consumption (e.g., memory and disk I/O). To address this challenge, it is suggested storing large bug attachments on the disk and then storing the links to these files on the bug database to improve performance, especially when retrieving data.

(2) Throughput: The paired sample test result showed that there was no significant change in throughput for most of the patterns unlike two previous case studies. This result is similar to that of the two previous case studies where throughput was relatively stable. The implication of this is that if the component being shared is

Table 5 Paired Sample Test Analysis for Case Study 3

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

a database, then throughput should not be expected to change drastically. Based on the plot of the EMMC, the shared component would be recommended when bugs are stored in a database with locking enabled.

(3) CPU and System Load: The results of the paired sample test show that there was a significant change in CPU for all the patterns. By analysing the plots of the EMMC, the results show that the dedicated component changed the most and so would not be recommended if optimising CPU usage is a key requirement. As with other case study results, there was no influence on any of the patterns for system load. The plots of EMMC showed that system load increased steadily across the patterns from shared component to dedicated component.

(4) Memory and Disk I/O: The paired sample test for both the memory and disk I/O showed a highly significant difference from pretest to post-test both for memory and disk I/O. For memory, the plot of the EMMC similarly showed that the dedicated component had the highest significant change compared to the other patterns. This is possibly due to running Bugzilla in a *mod_perl* environment, and so using a dedicated component would not be a good option for optimising system resources. It is well known that running Bugzilla in *mod_perl* environment consumes a lot of RAM [34]. The significant change in disk I/O consumption is due to the intense frequency of read/write activities in the database. For disk I/O consumption, having enough storage space would be required, especially if a large volume of bugs with attachments is expected. If a large number of users are expected, then applying disk space saving measures such as purging unwanted error or log files regularly could reduce disk I/O consumption and improve the chance of having a higher degree of isolation.

Results

This section explains how the findings from the three case studies were synthesised. The case synthesis was done using cross-case analysis approach and then complemented with narrative synthesis.

Cross-case analysis

In this section, we present the results of the cross-case analysis that was applied in an iterative manner during the analysis to reach the conclusion.

Data reduction

In our previous case study, much of the data reduction process was already done in the primary case studies. For each case study the following details are presented: (i) the paired test sample test, (ii) plots of the estimated marginal means of change, (iii) discussion of the findings and recommendations for achieving the required degree of isolation between tenants.

For each case study, we presented the paired sample test result plus the plots of the estimated marginal means of change (EMMC). In addition to this data, a table (i.e., Table 6) showing the characteristics of the three cases studies is presented in this paper. Table 6 shows the features that are related to each case study based on a selected set of different aspects of the study.

Data display

This step involves organising and assembling information that allows the drawing of conclusions using tools such as meta-matrices/tables and cause and relationships graphs. The data display steps will be tackled from two approaches to cross-case comparisons: variable -orientated and case-oriented [24].

(A) Variable oriented approach: The data derived at this stage is a table (see Table 7) showing the factors in which the cases vary, to explain why there is variation in the degree of tenant isolation across the cases. It is assumed that factors such as performance, resource utilisation, that are known to affect isolation between tenants were already used to evaluate the three cases independently. These factors are captured in the seven metrics used to evaluate the three cases: response times, error%, throughput, CPU, Memory, disk I/O, and system load. Knowing the various aspects in which the cases vary would enable us to explain the variation in the degrees of tenant isolation for different GSD processes. The synthesis identified five aspects in which the cases vary: size of data generated, the resource consumption of the GSD process, client's latency and bandwidth, supporting task performed, and error messages due to sensitivity to workload changes. These aspects are summarised below.

1. *Size of Data Generated:* One of the most important factors that account for the variation in the degree of tenant isolation is the fact that some GSD tools generate more data than others. For example, several of the problems that occur in version control relate to the fact that version control systems usually create additional copies of files on the repository, especially the ones that use the native operating system (OS) file system directly. This adversely affects performance because these files occupy more disk space than they actually use, and the OS spends a lot of time seeking across many files on the disk [40].
2. *Effect of GSD process on Cloud Resources:* The variation in the degree of tenant isolation can also been explained based on the effect of the particular GSD process on the cloud infrastructure resources. Some GSD processes consume more of a particular resource than others, and so this is bound to affect the degree of tenant isolation required by tenants. As shown in the experiments, continuous integration showed no significant change in CPU consumption

Table 6 Characteristics of the Case Studies

Aspect	Case study 1	Case study 2	Case study 3
Research Aim	To evaluate the degrees of isolation of multitenancy patterns for cloud-hosted continuous integration system	To evaluate the degrees of isolation of multitenancy patterns for cloud-hosted version control system.	To evaluate the degrees of isolation of multitenancy patterns for cloud-hosted bug tracking system
Target process	Continuous integration	Version control	Bug tracking
GSD tool/plugin that can be used to simulate the process	Hudson	Subversion, FileSystem SCM Plugin (integrated in Hudson)	Bugzilla
User-level Process investigated	Automated Build Verification/Testing	Check-in, Check-out, locking	Bug creation with file attachment
Process simulated in JMeter	sending HTTP/HTTPS request to continuous integration server	sending an FTP download file and upload file request to a Version control repository	sending an JDBC Request(an SQL query) to a database, invoking external JMeter APIs and Java classes via BeanShell
Developer community	Eclipse Foundation	Apache Software Foundation	Mozilla Foundation
Implementation Language	Java	Python, Java	Perl, Java
Mechanism for Customization and Extension	Hudson plug-in using Hudson HPI tool	Hook scripts or any program triggered by some repository event (e.g., pre-hooks which run in advance of a repository operation)	Bugzilla Extensions Hooks
Storage/DBMS used (Back end)	MySQL	Postgree SQL, Berkley DB	MySQL, PostgreSQL
Implementation of Tenant Isolation (based on COMITRE)	Easy to implement due to Java programming language familiarity	Fairly simple to implement but files permissions could be an issue	Difficult and challenging due to existing database restrictions/constraints
Key implementation challenges	Insufficient system resources (e.g., memory)	File permission errors	Restrictions of database schema (e.g., file size, maximum open connections)

when used with most of the patterns compared to version control and bug tracking. Under normal conditions, continuous integrations systems being compilers consume huge amounts of memory and disk I/O during high workload. Based on our results, the dedicated component is recommended for performing builds when there is a sudden increase in workload.

3. *Client Latency and Bandwidth:* The variation in the degree of tenant isolation can also be explained based

Table 7 Comparison of different aspects in which the Cases vary

Aspects	Case 1- Continuous integration	Case 2 – Version control	Case 3 – Bug tracking system
Resource consumption	High RAM and Disk I/O consumption (e.g., during the building of files)	Some native OS filesystem format (e.g., FSFS) consumes CPU (e.g., Delification, compressing data). Consumes memory during data caching	CPU and RAM consumption (could consume more CPU depending on runtime library used. Bugzilla consumes huge RAM if mod_perl is enabled), consumes memory during Caching DB transactions
Storage Space	Requires large storage space to store build history	Requires large storage space to store additional copies of data	Limited (except large bug attachments are needed)
Latency and Bandwidth of client accessing the server	Transferring large data size across network; long distance between CI server and SCM server	Compressing data across, Migrating repository, Repository backup, Enabling file locking	Transferring large bug attachments across a network, Enabling Locking on DB transactions
Type of GSD process	Long running build, large number of builds, complex and difficult builds	File locking	Long running DB transactions with support for locking could consume more RAM
Storage format of the backend server	Portable across different OS. Storing massive builds on NFS mount reduces performance.	Some DBMS (e.g., Berkeley DB) might not be portable across different OS	Fairly portable across different OS
Interdependencies with other tools	Depends on Version control server for store archive data	Depend on a CI server to trigger polling before checkout data	Integrated with CI server or other issue tracking systems

on the latency and network bandwidth of the client accessing the GSD tool. If a client with a low bandwidth is trying to access a version control repository, then response time and error% will be negatively impacted. Compressing the data transmitted across the network can boost performance, but the drawback is that it consumes much CPU. The results of case study one (i.e., continuous integration) showed that the magnitude of change for response time was more for the shared component compared to other patterns. This seems to suggest that a CI server (e.g., Hudson) should be configured close to an SCM server when polling a version control repository for changes.

4. *Type of GSD Process and Supporting operations:* There are several conditions associated with a GSD process that can result in different or varying degrees of isolation. Examples of such conditions include running long builds, a large number of builds and complex and difficult builds; and enabling file locking. For instance, a complex and difficult build involving lots of interdependencies will consume more resources (e.g., CPU) than an ordinary check-out process in a version control system.
5. *Error Messages and Sensitivity of workload Changes:* The cases also vary in terms of their sensitivity to workload changes as manifested in the nature and type of error messages produced by the different GSD processes during the implementation of tenant isolation. The experimental results show that when a tenant experiences a high workload, different kinds of error messages are generated depending on the GSD process. The error messages are summarised as follows: for continuous integration, the most common type of error was that of insufficient system resource (e.g., memory); for version control, the common error was that of directory and file permissions; and for bug tracking the common error was database-related errors.

(i) Case-oriented approach: The data derived at this stage is a table (see Table 8) showing the factors that are alike across the cases, and which appear to lead to similar outcomes when evaluating the varying degrees of tenant isolation in cloud-hosted GSD tools. The synthesis identified five aspects in which the cases are alike: a strategy for reducing disk space, locking, low consumption of some system resources, and use of plugin architecture for extending the GSD tool, and aspects of tenant isolation. The various aspects in which the cases are alike are summarised as follows.

1. *Strategy for Reducing Disk Space:* An interesting feature of all the GSD tools is that they have strategies for reducing disk space because of the possibility of the GSD tool generating a large volume of data due to the size, the number of artefact and number of users that may be involved in the project. For instance, CI systems can be configured to discard old builds. Version control systems can use *delification* (i.e., a process for transferring differences between versions instead of complete copies) and *packing* to manage disk space. For a bug tracking system, the error and log files can be purged from the database regularly.
2. *Locking Process:* All the GSD tools implement some form of locking whether at the database level or filesystem level. For example, locking is used internally in version control systems to prevent clashes between multiple tenants operating on the

Table 8 Comparison of different aspects in which the Cases are alike

Aspects	Case 1- Continuous integration	Case 2 – Version control	Case 3 – Bug tracking system
Generation of additional data	Archives the results of all the builds it performs, by default	Creates additional copies of files which occupies space	No additional copies created except bug attachments
Use of Locking	Used to block builds dependencies from starting if an upstream/downstream project is in the build queue	Used to prevent clashes between multiple tenants operating on the same working copy	Used to prevent clashes between multiple tenants trying to access the bug database
Use of back-end Storage	stored data native OS Filesystem directly	Mostly stores data on native OS File system directly (occasionally on database)	DBMS or database library
Use of disk saving strategies	Configure system to discard old builds	Transfer differences between versions instead of complete copies; concatenate files into a single pack	Purge error files and log files
Use of Web Server and Runtime Library	Java Runtime Environment (JRE) and JVM	Apache Portable Runtime (APR)	Mod_perl and mod_cgi
Size of users and project	Multiple developers triggering multiple concurrent builds	Multiple developers access working copy of a project	Multiple developers and testers submitting and corrects bugs
System Load and CPU	Low consumption	low consumption (could be high during delification, data compression)	Average consumption (could be high depending on runtime library used)

same working copy [41]. In Bugzilla, locking is used to prevent conflicts between multiple tenants or programs trying to access the Bugzilla database [34]. In continuous integration, locking can be used to block builds with either upstream or downstream dependency from starting if an upstream/downstream project is in the middle of a build or the build queue [32]. When using a version control system that implements locking, fetching large data remotely and finalising a commit operation can lead to unacceptably slow response times (and can even cause tenants' request to time out), and so having the repository together with the working copy located on your machine is beneficial. The results of case study two recommended a shared component to address the trade-off between resource utilisation and the speed of accessing or completing a version control process (e.g., checking out files from a repository).

3. *Low Resource Consumption due to Workload Changes:* Most GSD tools do not consume much system resources like CPU and memory but can benefit from optimisations when there is a sudden change in workload. For continuous integration, memory and disk I/O will be mostly affected. For Bugzilla, it will be memory especially if locking and database transactions are enabled. For subversion, disk space and disk I/O are the obvious resources that will be most affected. System load and CPU consumption are generally low, and so using any of the patterns would not make much difference. System load showed no influence on all patterns.
4. *Mechanism for customization and Use of Plugin Architecture:* All the GSD tools implement a "plugin architecture" for use in customising, modifying and extending the GSD tool. This means that other programs and components can be easily integrated with it [26]. For example, Hudson is easily extensible using plugins. A series of extension points are provided in Hudson that allows developers to extend its functionality [42]. These extension points are where the GDS tools can be customised to support tenant isolation.
5. *Choice of Multitenancy Pattern for Required Degree of Tenant Isolation:* The results generally showed that performance-related parameters such as response time, %error and throughput had changed significantly for shared pattern compared to system's resources such as CPU, memory, disk I/O and bandwidth. For example, in version control and bug tracking, the dedicated component is recommended to improve response time while the shared component is recommended to improve utilisation of memory and disk I/O. Because of this, the dedicated pattern (which is associated with a high degree of isolation) is recommended to improve performance related parameters while the shared pattern (which is associated with a low degree of isolation) is recommended to improve resource utilisation.

Conclusion drawing

This step involves further refining the above steps to produce conclusions concerning a particular aspect of interest. The outcomes of this step are (i) key conclusions from the statistical analysis, and (ii) the recommended patterns for achieving the required degree of tenant isolation.

(A) Summary of Findings from Statistical Analysis The conclusions presented in this section are based on trends noticed in the statistical analysis performed to answer the hypothesis of the experiment which was to determine how tenants deployed using a particular pattern changed from pre-test to post-test.

1. For most of the GSD tools, the shared component changed significantly for performance-related parameters (e.g., response times, error% and throughput), while the dedicated component changed significantly for system's resource-related parameters (e.g., CPU, memory and disk I/O). As the results show, the shared component would be recommended for improving systems resource consumption while the dedicated patterns would be recommended for improving performance. For example, the dedicated component was recommended to improve resource utilisation in bug tracking and CI systems under similar conditions. This is possibly due to the effect of locking which may have had an adverse impact on tenant isolation.

2. System load is nearly constant and no variability was found in almost all the case study results. A possible explanation for this is that the configuration of the deployed component, the nature of the tasks, and absence of a piled-up task queue for a long time being processed resulted in a reasonably good throughput. In most cases, if the load 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.

3. CPU changed significantly for version control and bug tracking systems, but not for continuous integration. This confirms what is already known about compiler/builders which is that it does not consume much CPU. However, it is important to note that certain operations or settings could increase CPU consumption regardless of the GSD tool used. Examples of such operations include enabling locking, data compression, and moving data between repositories in a different file format (i.e., FSFS).

4. Throughput changed significantly, and this change was relatively stable for most of the patterns in all the three case studies, except for case study three where there was no meaningful change. This may be because the system quickly reached peak capacity and so additional requests

simply do not add to the throughput. Furthermore, the small private cloud used for the experiments may have contributed to this fairly stable but significant change in throughput.

(B) Summary of Recommended Multitenancy Patterns for Deployment Table 9 shows a summary of the recommended multitenancy patterns for achieving isolation between tenants when one of the tenants experiences a high load. These recommended patterns are derived by first checking the paired sample test result and then analysing the plots of the estimated marginal means of change (EMMC) to compare the magnitude of change in each pattern. The key used in constructing the table is as follows: (i) the symbol "✓"means that the pattern is recommended; (ii) the symbol "**x**" means that the pattern is not recommended; and (iii) the symbol "-" implies that there is no difference in effect, and so any of the three patterns can be used.

For example, to ensure performance isolation in CI systems (e.g., regarding response time), the shared component is recommended for performing builds generally, and a dedicated component for performing version control especially when locking is enabled. The results generally showed no meaningful change for system load, and so any of the patterns can be used. For Bugzilla, the dedicated component was recommended to improve performance and the shared component to reduce resource consumption. This is based on our experience with Bugzilla which seems to suggest that bug trackers are very sensitive to increase workload especially if bugs are stored in the database with locking enabled. It was noticed that frequent crashes of the Bugzilla database occurred in our experiments which required recovery, and there were also numerous database errors related to restrictions on the maximum number of allowed queries, connections and packets, etc.

(C) Summary of the Effect of Performance and Resource Consumption on Multitenancy Patterns To further enrich the case study synthesis we provide a condensed summary to explain the effect of performance and resource

Table 9 Recommended Patterns for optimal deployment of components

Case Studies	Aspects of Isolation	Parameters	Shared	Tenant-isolated	Dedicated
Case Study 1- Continuous Integration with Hudson	Performance	Response	✓		
		Error%	✓		
		Throughput		✓	✓
	Resource Consumption	CPU		✓	✓
		Memory			✓
		Disk I/O			✓
		System load	-	-	-
Case Study 2- Version Control with File System SCM Plugin	Performance	Res		✓	✓
		Error	✓		
		Thru			✓
	Resource Consumption	CPU	✓	✓	
		Memory		✓	
		Disk I/O	✓		✓
		System Load	-	-	-
Case Study 3 - Bug Tracking with Bugzilla	Performance	Resp			✓
		Error%		✓	✓
		Throughput	✓		✓
	Resource Consumption	CPU	✓	✓	
		Memory	✓	✓	
		Disk I/O	✓	✓	
		System load	-	-	-

utilisation on tenants deployed based on different multitenancy patterns when one of the tenants experiences a sudden change in workload. The summary provided here is similar to narrative synthesis, which is a textual approach that can be used for condensing and explaining the findings from case studies [9].

(1) Response times and Error%: The case studies results showed that response times and error% did not change significantly for the shared component, and so it is recommended for addressing low latency and the bandwidth requirements of tenants. This suggests that a GSD tool should be configured close to the backend storage. For example, the CI server (e.g., Hudson) should be configured close to the SCM server when polling a version control repository for changes. The performance of tenants with low bandwidth accessing a version control system can be boosted by minimising the size of the network communications (e.g., reducing file size transferred between shared repositories). When committing large files to a repository residing over a network, delays could arise causing requests to time out [41]. For version control systems, the error% (i.e., requests with unacceptably slow response times) could be negatively impacted when committing a large number of files to a repository that is using a native OS file system (e.g., FSFS). Tenants request could time out while waiting for a response due to delays in finalising a commit operation [41].

(2)Throughput: Throughput did not change significantly for most of the patterns. Throughput can be likened to network bandwidth and so when the network is reasonably fast, a significant change in throughput should not be expected for application components deployed to the cloud. When accessing a repository over a slow or low bandwidth network, large data sizes could be compressed to improve throughput and performance, although this could lead to more CPU consumption.

(3) CPU and System Load: The case study results show that most GSD tools do not consume much CPU; consumption only slightly increased for some patterns. Therefore, the key in efficient utilisation of CPU while achieving the required degree of isolation lies in avoiding operations that are likely to increase CPU consumption. For continuous integration systems, a build can be run in the background without affecting other resources or processes, but this could increase if builds are difficult and complex [39]. For version control systems, CPU consumption could increase when moving data from one repository into another (e.g., using *svnadmin dump* and *svnadmin* load subcommands in subversion) or switching from a repository that uses a database (e.g., Berkeley DB or MySQL) to a repository that is based on FSFS file format [34]. Compressing data of large sizes in a bid to improve performance could also consume more CPU. System load was not influenced by any of the patterns, possibly because the number and size of requests did not overload the system to cause any significant change.

(4) Memory: As expected, the experiments showed a highly significant change in memory especially for the CI system, and therefore careful consideration is required especially when dealing with difficult and complex builds. The dedicated pattern would be recommended for achieving a high degree of isolation, for example, during complete integration build. When using bug tracking systems that store bugs in a database, certain runtime libraries could increase memory consumption. For example, Bugzilla consumes huge RAM if used in a *mod-perl* environment.

(5) Disk I/O: The experiments showed a highly significant change in disk I/O consumption especially for the CI system because builders and compilers consume a lot of disk I/O. For version control systems, there would be not much difference if any of the patterns are used, although the dedicated pattern would be recommended for exclusive access to the disk space. A large disk space would be required to cope with additional copies of files when using a version control system, and to cope with the large size and volume of bugs when using a bug tracking system that stores bugs in a database.

Analysis

This section a provides an analysis of the data produced from the cross-case analysis in order to give us further explanation and new insights (i.e., an explanatory framework) into tenant isolation (as illustrated in Fig. 7). Firstly, we present a mapping of different degrees of tenant isolation to the GSD process, the cloud application stack and cloud resources on which the GSD tools are hosted. Secondly, the trade-offs that should be considered when implementing the required degree of tenant isolation are discussed.

Figure 7 is inspired and anchored on the structure of a typical architectural deployment structure which has two main components: the cloud-application (i.e., the component or service to be deployed) and the cloud environment (i.e., the environment on which the process/service is being executed) [43]. In our previous research we used this structure to develop a taxonomy of deployment patterns for cloud-hosted applications that reflect the components of an architectural deployment structure and thereafter applied the taxonomy to position a set of GSD tools [26].

Figure 7 also captures the link between a cloud application process (e.g., continuous integration process), being used in a hybrid deployment (scenario) by utilizing a cloud-hosted environment (e.g., SaaS and PaaS deployment environment). It is a well known fact in software architectures (and in cloud deployment architectures) that most deployment patterns are related and have to be combined with others during implementation, for example,

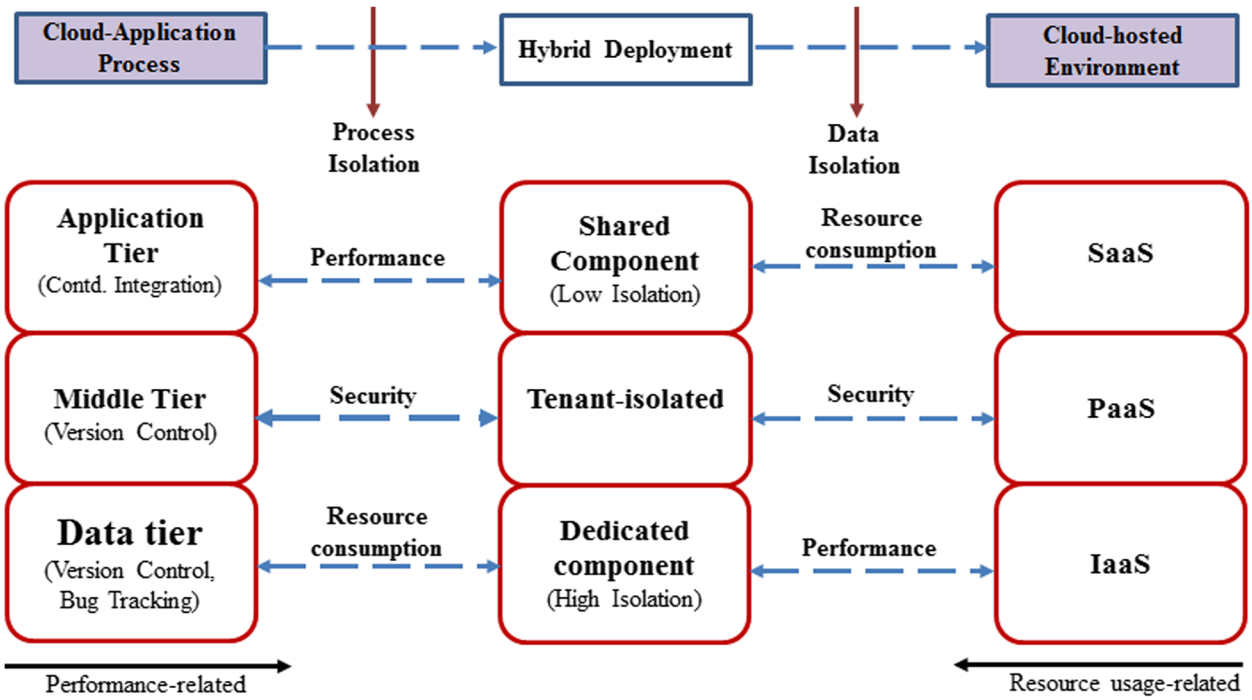

Fig. 7 Mapping of Degrees of Isolation to Cloud-hosted GSD Process and Resources

to address hybrid deployment scenarios, which usually involves integrating processes and data in multiple clouds [43–46].

A hybrid deployment scenario is very common in collaborative GSD projects, where a GSD tool requires multiple cloud deployment environments (or components), each with its own set of requirements. For example, when using Hudson there is usually a need to periodically extract the data it generates to store in an external storage during continuous integration of files. The hybrid data pattern can be used in this scenario to store data of varying sizes generated from Hudson in an elastic cloud and the remainder of the application resides in a static environment. With respect to isolation, the dedicated component pattern implies a high degree of isolation because tenants do not share resources. In other words, resources are dedicated to individual/specific tenants and thus are isolated from each other.

Mapping of tenant isolation to GSD processes and resources

Figure 7 maps the different degrees of tenant isolation to (i) software processes triggered by the cloud-hosted GSD tools; (ii) cloud application stack; and (iii) cloud resources on which the processes are executed. This mapping will serve as a guide to architects to select suitable cloud patterns, cloud layer, and requirements for cloud deployment. As shown in Fig. 7, GSD processes are placed on the left and cloud resources on the right. In this mapping, it is assumed that the ease and flexibility of achieving tenant isolation increase vertically, from top to bottom, and horizontally, from left to right.

(1) Mapping Tenant Isolation to Layers of a Cloud Stack: The mapping in Fig. 7, shows that a high degree of isolation can be achieved on the IaaS layer and vice versa. Therefore, as the required degree of isolation increases, the ability to improve resource consumption reduces when implementing tenant isolation. On the other hand, as the required degree of isolation increases, the ability to improve performance increases. This means that it is better to implement resource sharing or efficient resource utilisation using the shared component and reduce performance interference using a dedicated component.

Depending on the layer of the application stack (i.e., application layer, the platform layer, and infrastructure layer), tenant isolation may be realised differently with associated implications. Assuming the component being shared is a database, implementing the shared component on a bug tracking system at the SaaS layer implies allowing multiple tenants to share a single instance of the bug database. This ensures efficient sharing of cloud resources, but isolation is either very low or not guaranteed at all due to possible performance interference. Implementing a dedicated component at the IaaS layer would require installing the bug database for each tenant

on its own instance of virtual hardware. This guarantees a high degree of isolation but with limitations of high runtime cost and reduction in the number of tenants that can be served.

(2) Mapping Tenant Isolation to GSD Processes: Tenant isolation can be implemented at different levels of a cloud application stack depending on the type of component or process being shared. Due to the way in which software processes interact with an operating system, files system and systems resources, the GSD processes can be mapped to varying degrees of tenant isolation, and hence the application stack. Figure 7 shows a mapping of the three GSD processes to different levels of the cloud application stack. Notice that the GSD processes are placed in the following order from top to bottom: continuous integration, version control and bug tracking. In the mapping, the continuous integration process is placed in the top-bottom to fit into a situation where it is deployed to multiple users using the SaaS model. However, in a hybrid scenario, it is possible to place continuous integration on the middle tier of the cloud application stack (e.g., based on the PaaS deployment model). This scenario is suitable in a case where the continuous integration system is used as a platform to configure and host other programs. The bug tracking system, when used with a database to store bugs, would be placed on the bottom layer.

(3) Mapping Tenant Isolation to Aspects of Isolation: As shown in Fig. 7, the mapping of the different aspects of isolation between tenants is done in the following order: performance, security, resource consumption, from top to bottom for process isolation, and vice versa for data isolation. This means that it is better to use the shared component to improve resource consumption when implementing tenant isolation at the data level. On the other hand, it means that it is better to use the shared component to improve performance related requirements when implementing tenant isolation at the application level.

Again, the chance of implementing the required degree of isolation increases across the mapping in Fig. 7 from left to right for performance-related requirements such as response time and throughput, while it increases from right to left for system's resource related requirements such as CPU and disk I/O usage. Issues related to security, privacy, trust and compliance to regulation can mostly be tackled in a hybrid related fashion. For example, data/bugs generated from a bug tracking system could be stored in a certain location to comply with privacy and legal regulations while the architecture of the GSD tool could be modified to restrict exposure of certain data to users located in regions not considered to be of interest to the owners of the hosted data. Architecting the deployment of a cloud service based on this arrangement can best be tackled using a hybrid approach.

Trade-offs for achieving the required degree of tenant isolation

This section discusses the key trade-offs for consideration when implementing the required degree of tenant isolation for cloud-hosted software processes.

Tenant isolation versus resource sharing

The trade-off between tenant isolation and resource sharing is one of the most important considerations when deploying services to the cloud for guaranteeing tenant isolation. As the degree of isolation increases, the ability to share resources reduces. A low degree of isolation promotes resource sharing in the sense that the component and the underlying cloud resources can be shared with other tenants, thus leading to efficient utilisation of resources. However, there is a price to pay regarding possible performance interference. On the other hand, a high degree of isolation implies duplicating resources for each tenant since sharing is not allowed. This results in high resource consumption and a reduction in the number of users that can access the component. Therefore, if a GSD tool naturally consumes more of a particular resource, then the challenge would be how to avoid certain operations that would further increase the consumption of that resource. For example, continuous integration systems (or builders) consume a lot of memory and disk I/O. As the experiments in case study 1 showed, this consumption could increase much more if locking is enabled for application components deployed based on the dedicated component.

Tenant isolation versus number of users

Another important trade-off to consider is that of tenant isolation versus the number of users. As the degree of isolation increases, the number of users/requests that can access the component reduces. A possible explanation is that as the number of users increases, the physical contention also increases because more requests contend for the available shared resources (e.g., CPU and Disk). Contention either delays or blocks requests, meaning that more time will be spent by requests waiting to use the system's resources. Thus, performance will be impacted negatively leading to a low degree of isolation. This behaviour explains why a larger magnitude of change was noticed for the shared component and tenant-isolated component in case study 1 with continuous integration.

Tenant isolation versus customizability

To implement the required degree of tenant isolation on a GSD tool, some level of customization would have to be done depending on the level where the process or component to be customised resides [47]. The higher the degree of isolation that is required, the easier it is to customise the GSD tool to implement tenant isolation. For example, implementing tenant isolation

for a GSD tool like Hudson via virtualization on the infrastructure level will not be as difficult as implementing it on the application level in terms of the effort, time and skill required. Implementing isolation on the application level would require good programming skills to modify the source code, and also address issues of compatibility and interdependencies between the GSD tool and required plugins and libraries [42]. Each time a multitenant application or its deployment environment changes, then a tedious, complex and maintenance process may also be required.

Tenant isolation versus size of generated data

There is also a trade-off between tenant isolation and the size of data generated by the GSD tool. The more data is generated, the more difficult it is to achieve a higher degree of isolation. For example, most version control systems (e.g., Subversion, File System SCM plugin) create additional copies of files on the shared repository. Over time, these files will occupy disk space thereby adversely affecting the performance experience by tenants. This will lead to a low degree of isolation between tenants since a lot of time would be spent fetching data from the repository that contains numerous unused or unwanted files. The study recommended a dedicated component for exclusive access to disk space, but again this implies significantly increasing the disk space and other supporting resources allocated to each tenant. In Fig. 7, the GSD tools mapped to the lower level of the cloud stack (i.e., version control system and bug tracking) generate the most data. It is important to note that other GSD tools can be configured to generate additional data. For instance, Hudson can be configured to archive artefacts to a repository. Because of this, most the GSD tools have mechanisms for removing unwanted files, thereby saving disk space.

Tenant isolation versus scope of control

Implementing the required degree of tenant isolation to a large extent depends on the "scope of control" of the cloud application stack. The term *cloud application stack* refers to the different layers of resources provided by the cloud infrastructure on which the cloud-hosted service is being hosted. This could either be the SaaS, PaaS or IaaS level [48]. The architect has more flexibility to implement or support the implementation of the required degree of tenant isolation when there is greater "scope of control" of the cloud stack application. In other words, if the scope of control is restricted to the higher level of the cloud stack (i.e., the SaaS) then the architect may only be able to implement a low degree of isolation (e.g., shared component), and vice versa. Therefore, if an architect is interested in achieving a high degree of isolation (e.g., based on the dedicated component), then the scope of control should extend beyond the higher level to the lower levels of the cloud stack (i.e., PaaS and IaaS). This would enable an architect to deploy a GSD tool on the IaaS platform so that exclusive access can be provided to the customer together with all the configuration requirements to support any operation that requires frequent allocation and de-allocation of resources. For example, using a version control system to perform operations that involve moving a repository between different hosts and keeping history would require having file system access in both hosts [40].

Tenant isolation versus business constraints

The trade-offs between tenant isolation and business requirements is a key consideration in architecting the design and deployment of cloud-hosted services. As the degree of isolation increases from top to bottom, the ease and flexibility to implement business requirements that cannot be compensated for at the application level reduces. The shared component, which offers a low degree of isolation, can be used to handle business requirements that can be compensated at the application level. Examples of such business requirements include performance and availability. The architect can easily modify the application architecture of the GSD tool to address this type of requirement.

On the other hand, the dedicated component which offers a high degree of isolation can be used to handle business requirements that cannot be easily compensated. Examples of this type of requirement include legal restrictions and the location and configuration of the cloud infrastructure. For instance, a legal requirement can state that the data hosted in one place (e.g., Europe) by a cloud provider cannot be stored elsewhere (e.g., in the USA). An architect would, therefore, have to map this type of requirement to a cloud infrastructure that directly satisfies this.

Discussion

This section first presents a general discussion of findings from the three case studies with respect to the commonalities and difference identified from the three case studies. Thereafter, it presents some recommendations that can be followed to achieve the required degree of isolation.

Type and location of the application component or process to be shared

The degree of isolation between tenants, to a large extent, depends on the type and location of application component that is being shared. There are different techniques for realising tenant isolation depending on the level of the application stack. On the low level (i.e., IaaS layer) tenant isolation can be achieved by virtualization. On the middle-level (i.e., the PaaS layer), a hypervisor can be used to set up different databases for different tenants. On the

application level (i.e., the SaaS layer), tenant isolation can be implemented by introducing a tenant-id field to tables so that tenants can only access data that is associated with their tenant-id [3].

Previous research on implementing multitenancy assumes two extreme cases of isolation: shared isolation and dedicated isolation which are mostly implemented at the data tier [49–52]. In addition to implementing multitenancy patterns at the data tier, our study also takes into consideration the effect of varying degrees of isolation between tenants. Unlike previous research which focuses more on performance isolation [18, 53], our case studies emphasise the need to consider (i) other aspects of isolation such as resource consumption of tenants, and (ii) the effect of varying degrees of tenant isolation for individual components of a cloud-hosted service under different cloud deployment conditions.

An approach (i.e., COMITRE) has been developed for evaluating the required degree of isolation between tenants. This approach is anchored on shifting the task of routing a request from the server to a separate component (e.g., Java class or plugin) at the application level of the cloud-hosted GSD tool [5]. The approach captures and configures tenants request/transaction so that (i) implementation can be done at different levels (although more flexibly at the application level for optimizing cloud resources); (ii) individual components can be monitored and adjusted to reflect changing workload. This means that the underlying middleware and infrastructure do not necessarily need to be multitenant aware as the isolation is handled on the application level. Another key advantage of using this approach is that it can be used at the application level to optimise the utilisation of the underlying cloud resources in a resource constrained environment, for example, where there are limited CPU, Memory and disk space. The approach can also be extended to work at other layers of the cloud stack (e.g., the PasS layer or IaaS layer). However, the drawback is the effort and skill required in modifying the GSD tool (or cloud service) before implementing COMITRE logic.

Customizability of the GSD tool and supporting process

The cases studies revealed that most global software development (GSD) tools are not implemented using any multitenant architecture, and so they would have to be customised to support the required degree of tenant isolation before deploying them to the cloud. Previous research has applied multitenancy patterns/architectures (and cloud patterns generally) to simple web applications, for example, weblog applications [54, 55]. In addition to applying multitenancy patterns on web applications, this study incorporates into our implementation of the multitenancy patterns key factors that could influence pattern selection such as application processes, workload and resource demands imposed on cloud service and the cloud infrastructure on which the service is hosted.

Customizing a cloud-hosted GSD tool (or any cloud-hosted service) can be very challenging if the tool/service has several components that are being shared. Different application components can be implemented at different levels to address the problem between aspects of the GSD that can be customised with ease and those that cannot. For example, Bugzilla interface can be exposed as an integrated component to different tenants working on other GSD tools like JIRA while the Bugzilla database can be implemented as a dedicated component to ensure proper isolation of bugs belonging to various tenants.

The three case studies we conducted also revealed another major challenge in achieving the required degree of tenant isolation during the customization of the GSD tools (i.e., continuous integration with Hudson, version control with File System SCM plugin and bug tracking with Bugzilla). The GSD tools (and many other cloud-hosted services) can have many inter-dependencies on different levels of the application itself and with other applications, plugins, libraries, etc., deployed with other cloud providers. This could affect the performance and resource consumption of the cloud-hosted system in a way that we did not anticipate and hence the required degree of tenant isolation. There is also a serious risk of using incompatible plugins and libraries required to modify, customise and execute these GSD tools. This could corrupt the GSD tool and stop other supporting programs/processes from running. An easy way to address this challenge on the cloud is to push the implementation of tenant isolation down the lower levels of the cloud stack, where the architect can, for example, install the GSD tool on a PaaS platform. Issues of middleware and methods for customizability of Saas applications have been discussed in [56].

Optimization of cloud resource due to changing workload

The case studies have clearly highlighted the need to optimise the deployment of cloud GSD tools and support processes under different cloud deployment conditions while guaranteeing the required degree of tenant isolation. Under typical configurations, most GSD tools may not consume much cloud resources. However, there is always a real need for optimisation of the system's resource in a situation where there is either under-utilisation of resources or over utilisation of resources (e.g., if the shared application component is overloaded).

As pointed out in the case study involving continuous integration with Hudson, the CPU consumption of tenants changed significantly for the shared component. Therefore, on a private cloud which supports a small number of tenants, the shared component can be used to optimise CPU utilisation. However, there would be no

guarantee of a high degree of isolation between tenants. In CI, builds are known to consume a vast amount of disk I/O, and so a dedicated component can be used for running (i) large number of builds concurrently, and (ii) builds that are too complex [32]. Again if the shared component is used to deploy Bugzilla (where *mod_perl* is enabled) on the same type of cloud infrastructure described above, then minimising RAM consumption would be the main issue of concern, and not CPU [34]. This means that it is better to use a dedicated component for heavy and complex tasks (e.g., builds with multiple dependencies) and a shared component for light tasks. For example, in large projects, multiple builds interact with multiple components to create several inter-dependencies with each other which will consume more resources.

Models used in previous work have focused on minimising the cost of using cloud resources, and meta-heuristics are not used for the optimisation (only in a few cases are simple heuristics used [57]). A model-based decision support system such as the one developed in our previous work can be integrated into the GSD tool (or any cloud-hosted application) in order to optimise cloud resources while guaranteeing the required degree of isolation [58].

Hybrid cloud deployment conditions

This study revealed that there are situations where combining more than one multitenancy pattern is more suitable for implementing isolation between tenants. As the result of case study one involving continuous integration showed, CPU consumption of tenants changed significantly for the shared component; and so, the first stage of the build can use a dedicated component while the second stage of the build can use the shared component. Therefore, in a continuous integration process, builds or commits to a repository could be configured to run concurrently or at regular intervals. Running such builds as a long complete integration build in a slow network environment could take a lot of time and resources. To achieve a high degree of isolation while guaranteeing efficient resource utilization, the integration build can be split into two different stages, so that: (i) the first stage creates a commit build that compiles and verifies the absence of critical errors when each developer commits changes to the main development stream, and (ii) the second stage creates secondary build(s) to run slow and less important tests [59]. This will ensure that secondary builds do not consume many resources and even if they fail, it will not also affect other tenants.

Another area of application may be to handle hybrid deployment scenarios, for instance, integrating data residing on different clouds and static data centres. There are several cloud offerings such as Dropbox and Microsoft's Azure StorSimple that allow customers to integrate a cloud-based storage with a company's storage area network (SAN) [60, 61]. Another scenario that is suitable for combining more than one multitenancy pattern is when different settings are applied concurrently to a particular software process. For example, settings could be applied to vary the frequency with which code files are submitted to a shared repository or lock certain software processes to prevent clashes between multiple tenants. Several other hybrid cloud deployment scenarios can be utilised to guarantee the required degree of tenant isolation[3].

Tagging components with the required degree of isolation

The case studies have shown that a cloud-hosted service may have several interdependencies with components of other services/applications that it is interacting with. It is therefore essential that components designed to use or integrated with a cloud-hosted service be tagged as much as possible when there is need to implement the required degree of tenant isolation. In our previous work, we developed a model that can be used to achieve tagging by mapping our problem to a Multidimensional Multichoice Knapsack Problem (MMKP) instance and associating each component with its required degree of isolation, thus allowing us to monitor and respond to workload changes efficiently [58]. Tagging can be a challenging and cumbersome process and may not even be possible under certain conditions (e.g., in a case where the component is integrated into other services and are not within the control of the customer). Therefore, instead of tagging each component with an isolation value as required, this can also be predicted in a dynamic way. In our previous work [62], an algorithm was developed which learns the features of existing components dynamically in a repository and then uses this information to associate each component with the appropriate degree of isolation. This information is crucial for making scaling decisions and optimisation of resources consumed by the components, especially in a real-time or dynamic environment.

Similar to our approach, many providers implement techniques that capture client transactions/requests and decide what level of isolation is required. However, these approaches do not guarantee the availability and tenant isolation of specific components/individual IT resources (e.g., a particular virtual server or disk storage), but for the offering as a whole (e.g., starting new virtual servers) [63–65]. Our approach is also related in many ways to existing cloud offering such as Amazon's Auto Scaling and EC2 [63, 66] and Microsoft Azure's Web Role [67] where users can specify the different configurations for a component, for example, the number of components that can be deployed for a certain number of requests. However, users cannot tag each component with the required degree of isolation before deployment, as has been proposed in our approach.

Errors and sensitivity to workload interference

Multitenancy can introduce significant error and security challenges in the cloud, especially when implementing varying degrees of isolation between multiple tenants sharing resources. When resources are shared in a multitenant application among multiple tenants, it is very easy for errors associated with one tenant (e.g., due to overload of the tenant or insufficient resource allocated to the tenant) to affect the performance and resource consumption of other tenants.

The type of error messages received from the case studies is a pointer to the key resources to consider in achieving the required degree of tenant isolation. For continuous integration, the error messages are related to insufficient system resources. For example, while implementing tenant isolation with Hudson, the most common error experienced was that of insufficient memory allocation. The cloud infrastructure did not cause this but partly caused by Hudson as it is not very optimised and also by the demands of the continuous integration process. For the version control process, the most common error was that of insufficient memory and file or directory permission issues (e.g., when setting FTP request configurations). This problem becomes more acute when moving the VM image instance (whose file permission had been set on a local machine) to the cloud infrastructure. Therefore, it is necessary to get repository ownership and permission right before conducting the experiments.

For the bug tracking process where bugs are stored in a database, the most common errors are related to resolving database errors, for example, exceeding the limit of file size, query, connections, etc. Therefore, it is necessary to modify the bug database to remove these restrictions. In addition, it would be recommended to modify the configuration of the database to increase the maximum size of a file that can be stored in the database. It may also be necessary to remove restrictions on the maximum number of allowed queries, connections and packets, etc. The bug database running on the VM instance can be quite sensitive to workload changes depending on the size, the volume of bugs, and the bug database isolation level.

The case studies also emphasised the need to carefully vary the frequency with which large instant bugs are submitted concurrently to a database when support for locking is enabled. Locking, in this case, is used to prevent conflicts between multiple tenants attempting to access a bug database. This type of scenario is very important in *distributed bug tracking* in which some bug trackers such as Fossil and Veracity, are either integrated with or designed to use distributed version control systems or continuous integration systems, thus allowing bugs to be generated automatically and added to the database at varying frequencies [68, 69].

The case studies seem to suggest some of the global software development processes are more sensitive to workload interference than others. By workload, we mean the impact of tenant requests to a cloud-hosted service that results in processing load, communication traffic, or data to be stored. For example, bug tracking (with Bugzilla) was susceptible to increased workload especially if locking is enabled for the bug database. It was noticed that there were numerous database related errors and frequent crashes of the Bugzilla MySQL database when there is increased workload which required recovery. This impacts on the required degree of isolation because it forces the software architect to deploy the cloud service using a dedicated pattern in order to avoid performance interface. However, implementing components of the cloud-hosted GSD tool as dedicated components reduces the degree of sharing between tenants and hence increases running cost per tenant.

In a production environment, it would be recommended to use a cloud storage (e.g., Amazon S3, Google Cloud Storage, Azure Blob etc.) instead of a relational database especially if frequent workload changes are expected. The focus for many cloud storage offerings is the need to handle very large amounts of data that are globally distributed and whose structure can be easily adjustable to new requirements quickly and flexibly [20, 22]. There are several cloud patterns can be used to minimise workload interference in multitenant applications. For example, in [70], several cloud computing solution patterns are presented for handling application workloads such as applications with highly variable workloads in public clouds, and workload spikes with cloud burst.

Volume of application data generated

The case studies reveal that the volume of application data has a significant impact on the performance of the application which in turn affects the required degree of tenant isolation. In our experiments, for example, continuous integration and version control processes, generated more data than the bug tracking system. This adversely affects performance because these files occupy more disk space than they use, and the operating system spends a lot of time seeking across many files on the disk.

It has been argued that the shared component is better for reducing resource consumption while the dedicated component is better in avoiding performance interference [3, 71]. However, as the experimental results show, there are certain software development processes where that might not necessarily be so, for example, in version control, where additional copies of the files are created in the repository (especially the ones that use the native operating system (OS) filesystem directly), thus consuming more disk space [41]. Over time, performance begins to degrade as more time is spent searching across many

files on the disk. Therefore, we recommend that in order to implement the required degree of tenant isolation for a cloud-hosted application/service, there should be careful consideration of how to deploy certain applications associated with software processes that create additional copies of files as part of the software process (e.g., version control and continuous integration) and other applications that do not (e.g., bug tracking).

Threats to validity

The validity of case study research can be evaluated using four key criteria: construct validity, internal validity, external validity, and reliability [25]. Construct validity has been achieved by first conducting a pilot study, and after that three case studies using the same experimental procedure and analysis. The results of the study including the plots of estimated marginal means and the statistical results of the three case studies are compared and analysed to ensure consistency. Construct validity was further increased by maintaining a clear chain of evidence from the primary case studies to the synthesised evidence, including the approach for implementing tenant isolation, experimental procedure, and statistical analysis. Furthermore, the validity of the synthesised information has been increased by involving the authors of the primary case studies in reviewing the case study synthesis.

Internal validity has been achieved by precisely distinguishing the units of analysis and linking the analysis to a frame of reference about the degrees of isolation between tenants as identified in the literature review. This frame of reference is based on the fact that the varying degrees of tenant isolation are captured in three multitenancy patterns: shared component, tenant-isolated component and dedicated component. The case studies are carried out one after the other; each was done with a space of about a three-month interval. Before the next study was done, the cloud infrastructure was shut down, previous data erased and then the infrastructure started again.

External validity has been achieved by using multiple case studies design and comparing the evidence gathered from the three case studies. Furthermore, statistical analysis (i.e., paired sample t-test) has been used across the three case studies to evaluate the degree of isolation. It should be stated that the findings and conclusions of this study should not be generalised to small size software tools and processes, especially the ones that are not mature and stable. This study applies to cloud-hosted GSD tools (e.g., Hudson) for large-scale distributed enterprise software development projects.

Reliability is achieved by replicating the same experimental procedure (based on applying COMITRE) in the three case studies. Due to the small size of the private cloud used for the experiment, the setup values (e.g., the number of requests and runs for each case study experiment) are carefully varied to get the maximum capacity of the simulated process before conducting the experiments. The case study synthesis combined two approaches: narrative synthesis and cross-case analysis, thus allowing us to gain synergies, harmonise weaknesses and assess the relative strengths of each approach. On the transparency of the case study, all the information derived from the case studies is easily traceable, and the whole process is repeatable. The authors had access to the raw data which gave them the opportunity to go deeper in their synthesis. This means that the case studies' report was synthesised at the right level of abstraction and granularity.

Conclusions

In this research, we have conducted a cross-case analysis of findings from three case studies (i.e., continuous integration with Hudson, version control with FileSystem SCM Plugin and Bug tracking with Bugzilla) that empirically evaluated the degrees of tenant isolation between tenants for components of a cloud-hosted software service. This paper has contributed to literature by: (i) providing empirical evidence on the varying degrees of tenant isolation based on case studies of cloud-hosted GSD tools and associated processes, and (ii) providing commonalities and differences in the case studies as well as an explanatory framework for explaining the trade-offs to consider when implementing tenant isolation.

Our research revealed that: (i) certain software processes (e.g., long-running builds) can be split into phases and different degrees of isolation then implemented for each phase to resolve conflicting trade-offs; (ii) customization of a cloud-hosted tool/service is a key requirement for achieving the required degree of tenant isolation and so during implementation consideration has to be given to those aspects that can be customised and those that cannot be customised; and (iii) implementing the required degree of tenant isolation also depends on the "scope of control" of the cloud application stack (i.e., ability to access one or more layers of cloud infrastructure): the greater the scope, the more flexible an architect can implement or support the implementation of the required degree of isolation.

The cross-case analysis revealed five case study commonalities: disk space reduction, use of locking, low cloud resource consumption, customization and use of plug-in architecture, and choice of multi-tenancy pattern. The degree of isolation is reduced when there no strategy to reduce disk space and customization and plug-in architecture is not adopted. The degree of isolation improves when careful consideration is given to how to handle a

high workload, locking of data and processes is used to prevent clashes between multiple tenants, data transfer between repositories and selection of appropriate multitenancy pattern. The study also revealed five differences: size of generated data, cloud resource consumption, sensitivity to workload changes, the effect of the software process, client latency and bandwidth, and type of software process). The large size of generated data, high resource consumption processes, high or fluctuating workload, low client latency, and bandwidth when transferring multiple files between repositories reduces the degree of isolation. The type of software process is challenging because it depends on the cloud resource being optimised.

A further contribution of this paper is an explanatory framework for (i) mapping the tenant isolation to different GSD processes, cloud resources and layers of the applications stack (ii) explaining the different trade-offs to be considered for optimal deployment of components with a guarantee of the required degree of tenant isolation. The case study synthesis identified six trade-offs that should be considered while implementing tenant isolation: tenant isolation versus (resource sharing, the number of users/requests, customizability, the size of generated data, the scope of control of the cloud application stack and business constraints). For example, some level of customization would have to be done depending on the layer where the process or component associated with the cloud-hosted service resides: the higher the required degree of isolation for components located on the lower layer, the easier it is to implement tenant isolation. While virtualization can be easily used to implement isolation at the infrastructure level, significant effort, time and good programming skills would be required at the application level to modify the source code, and address issues of compatibility and interdependencies between the GSD tool and several required plugins and libraries.

Based on the required degree of multitenancy isolation, a comparative analysis will be carried out on the performance and resource consumption of each code factoring task to the base system during every proposed improvement before deploying it to the cloud. Another interesting option would be to conduct case studies with other cloud deployment scenarios and indicators which could affect multitenancy isolation. Some scenarios to be explored include the effect of different file system formats (e.g., version control systems like Subversion can be affected by the type of file system format used to store artefacts), the number and size of data generated or stored, and concurrent running processes.

Endnotes

[1] The word "Change in the acronym EMMC refers to the dependent variable used for paired sample t-test.

[2] The symbol CS1 used in Figs. 8, 9, 10, 11, 12, 13 and 14 in the Appendix A stands for Case Study 1.

[3] Fehling et al. describe several cloud patterns that are suitable for deploying cloud services in a hybrid fashion

Appendix A: Plots of estimated marginal means of change for the case study 1

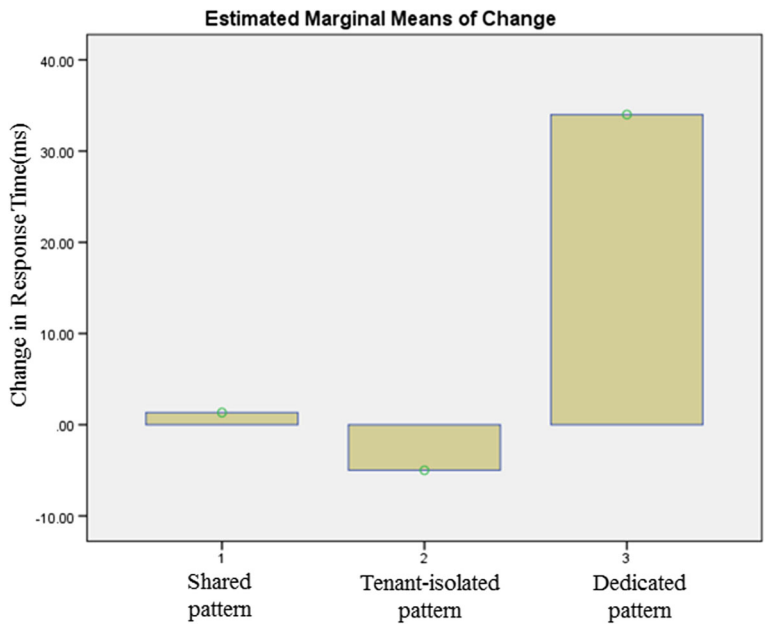

Fig. 8 ResTime Changes [CS1]

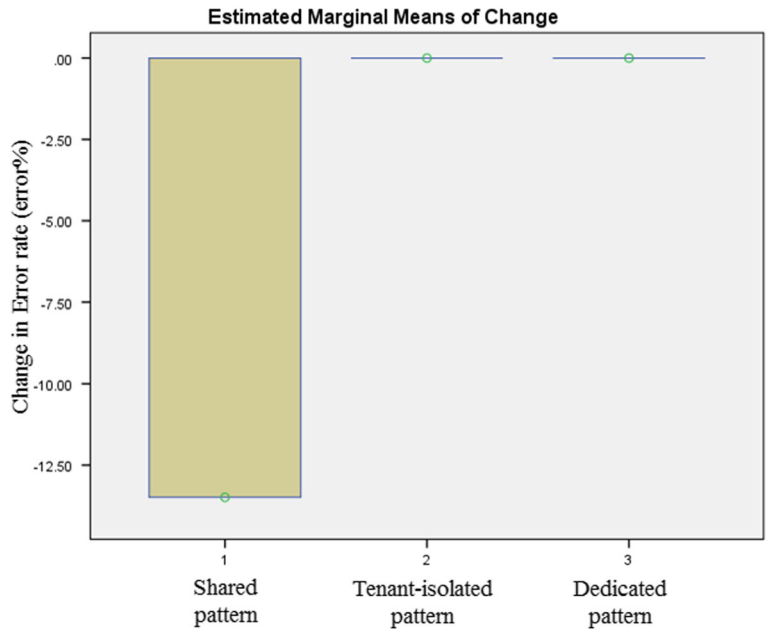

Fig. 9 Error% Changes [CS1]

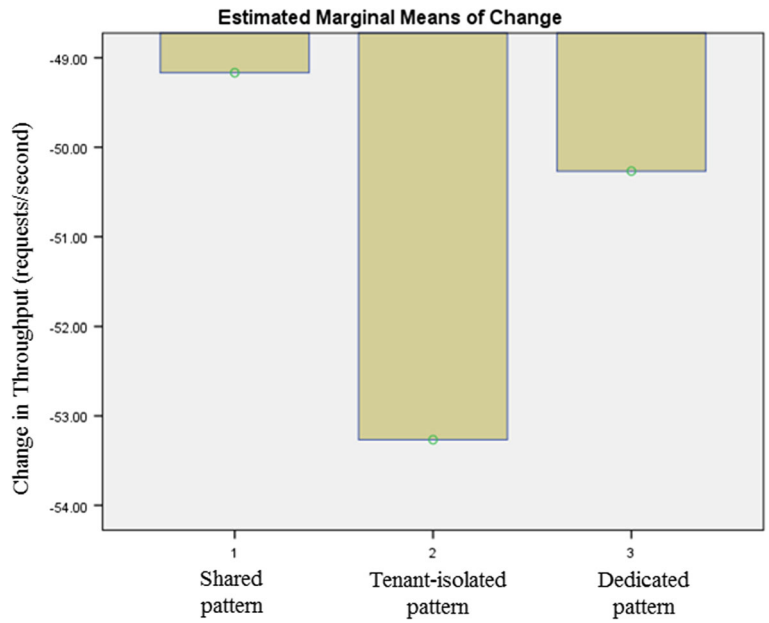

Fig. 10 Throughput Changes [CS1]

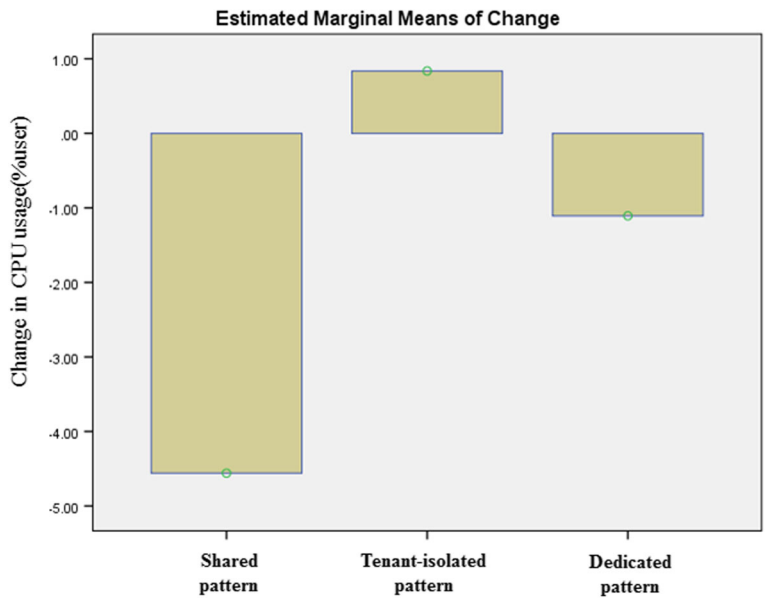

Fig. 11 Changes in CPU [CS1]

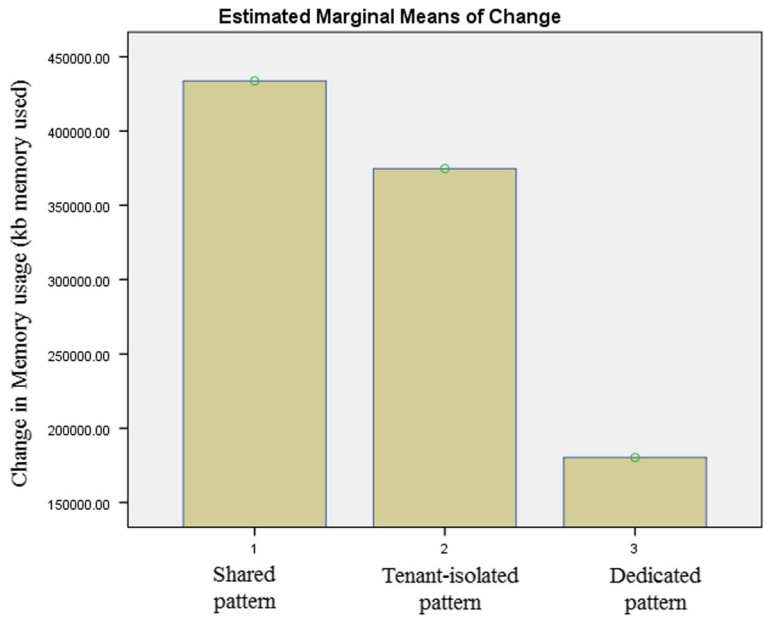

Fig. 12 Changes in Memory [CS1]

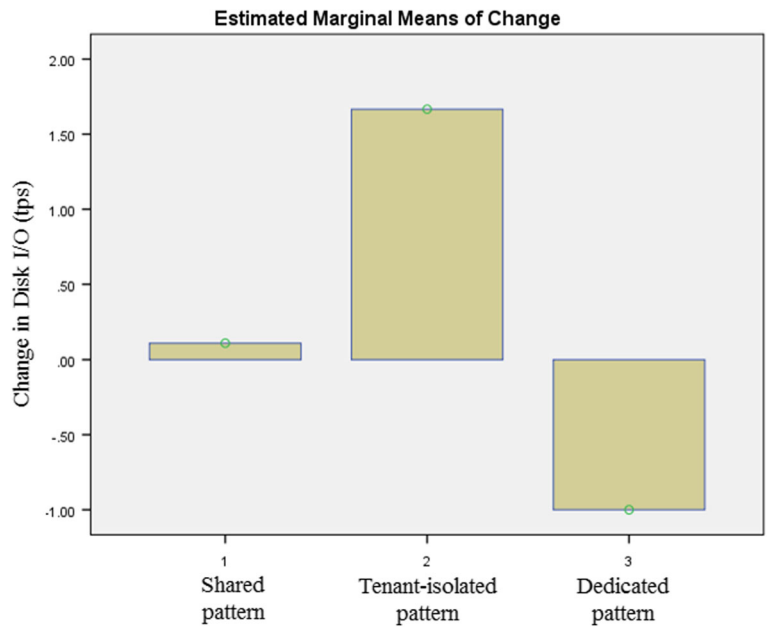

Fig. 13 Changes in Disk I/O [CS1]

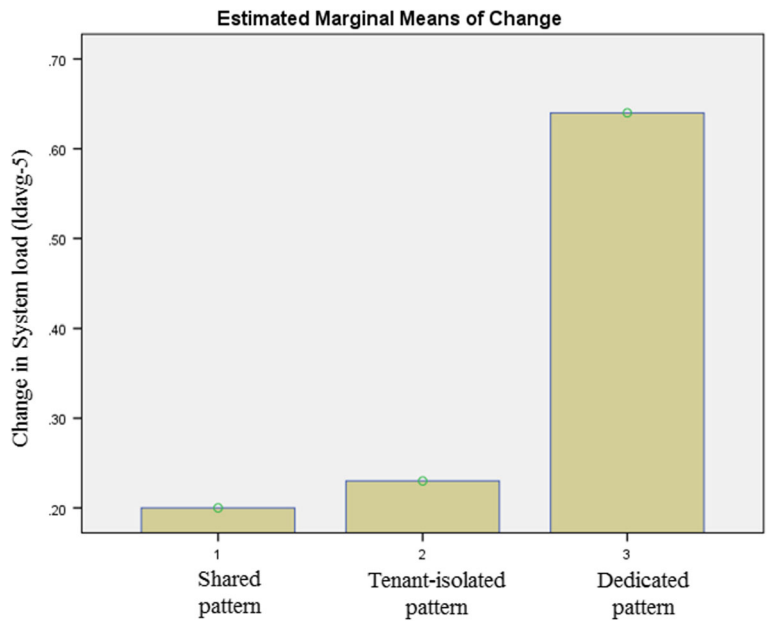

Fig. 14 System Load [CS1]

Appendix B: Plots of estimated marginal means of change for the case study 2

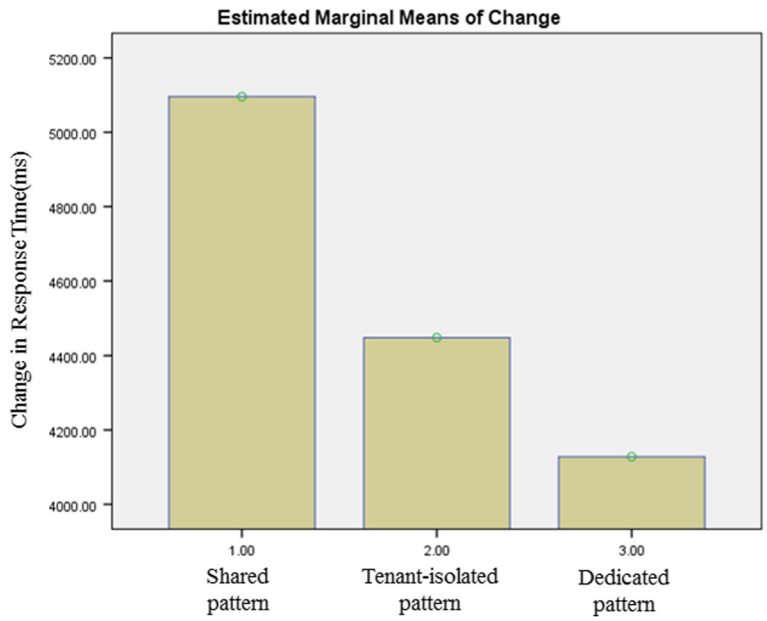

Fig. 15 Changes in Response Times [CS2]

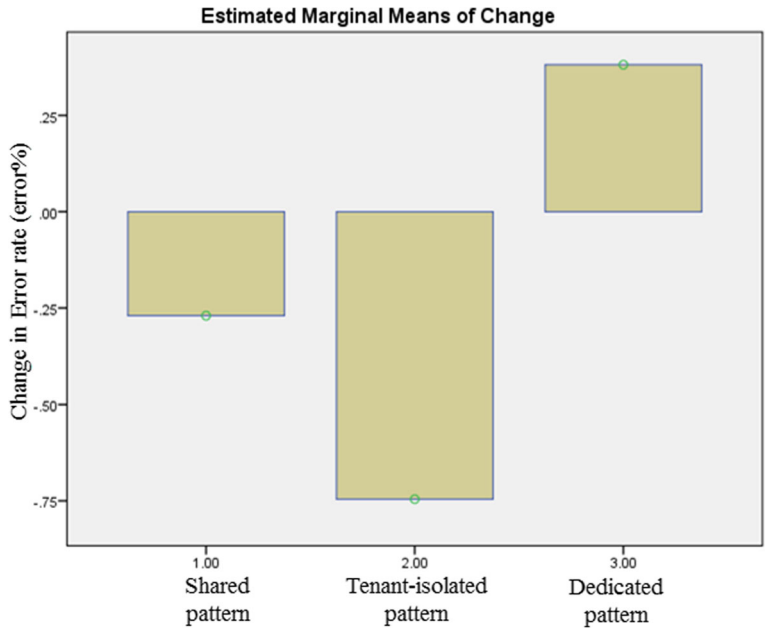

Fig. 16 Changes in Error% [CS2]

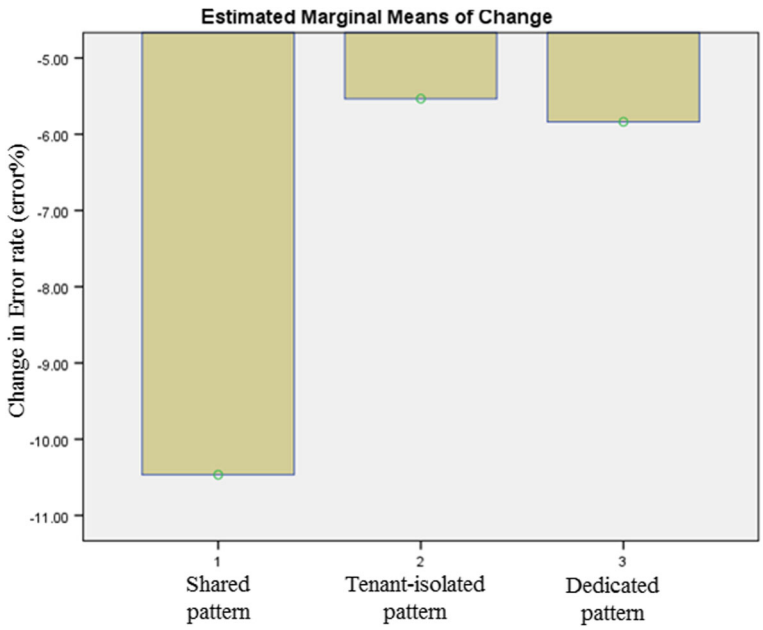

Fig. 17 Changes in Throughput [CS2]

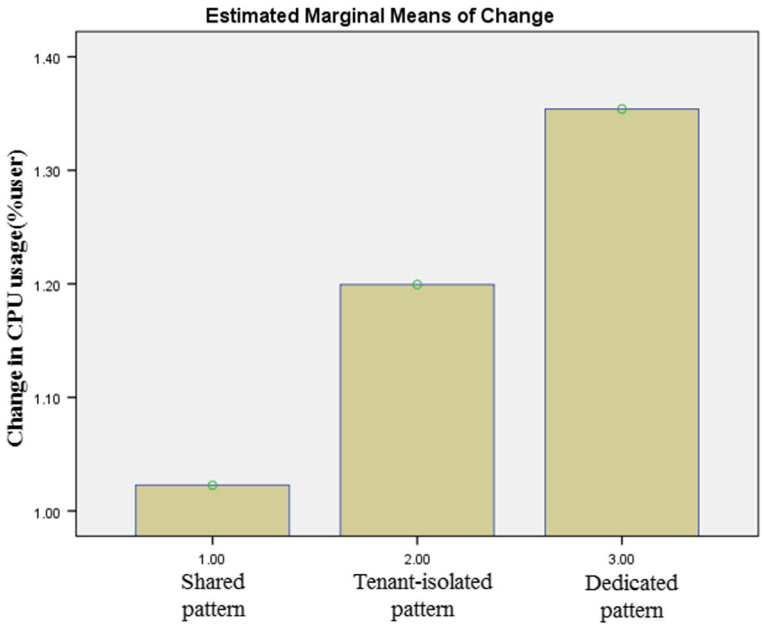

Fig. 18 Changes in CPU [CS2]

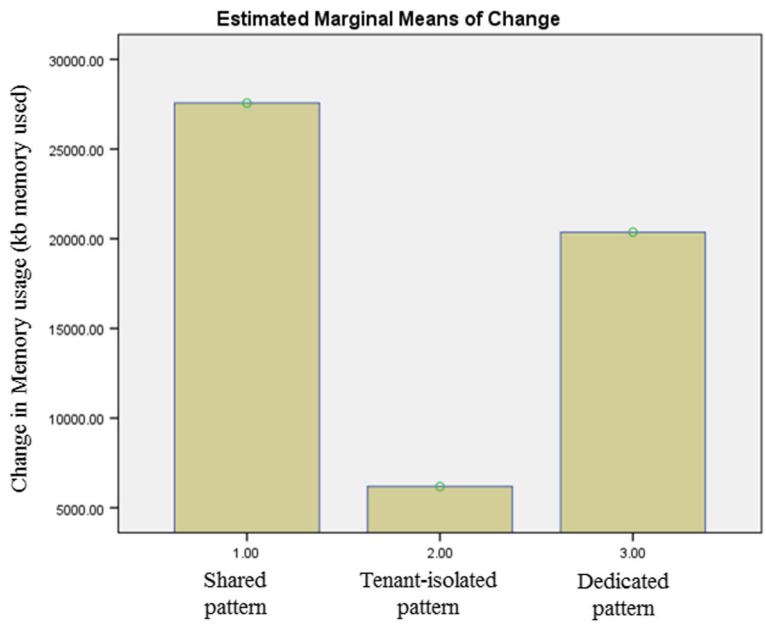

Fig. 19 Changes in Memory [CS2]

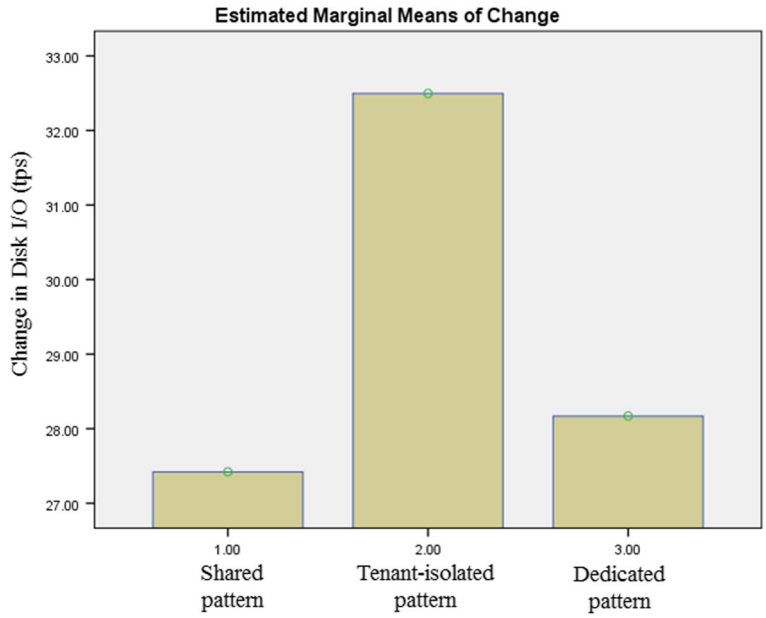

Fig. 20 Changes in Disk I/O [CS2]

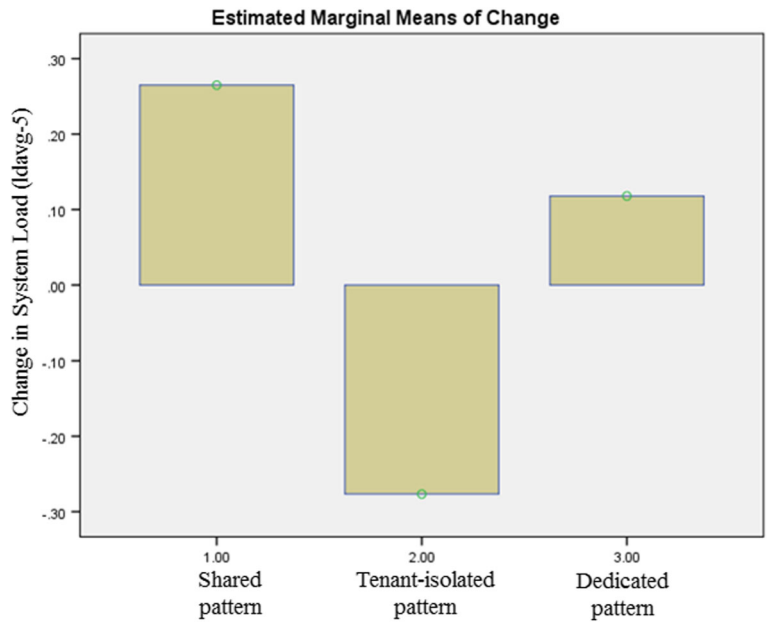

Fig. 21 Changes in System Load [CS2]

Appendix C: Plots of estimated marginal means of change for the case study 3

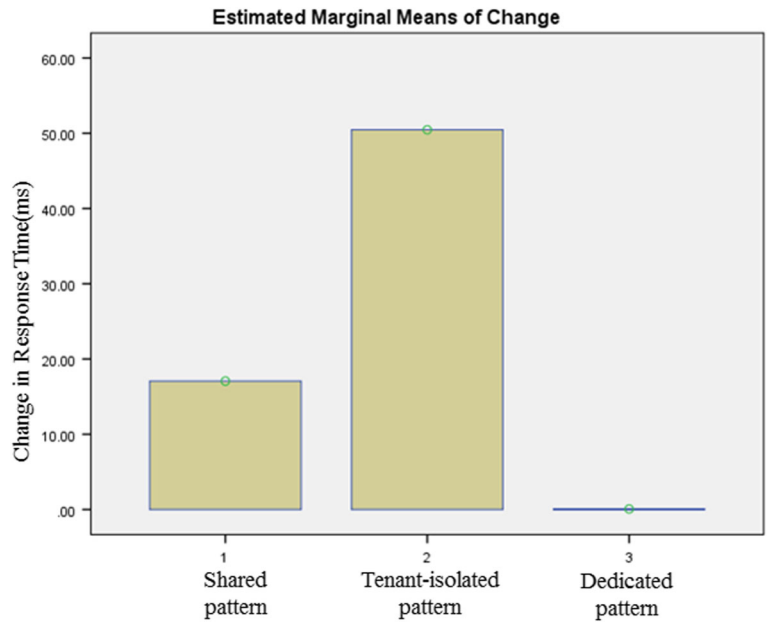

Fig. 22 Changes in Response Times [CS3]

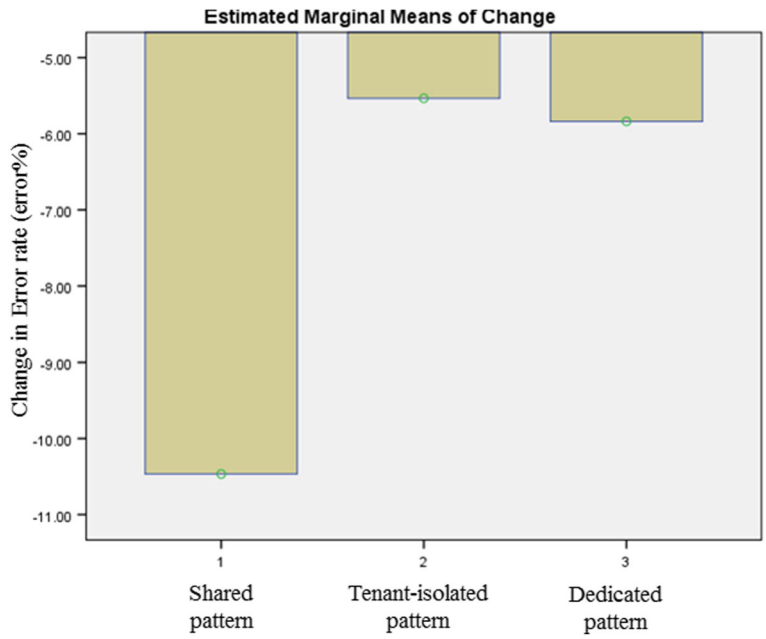

Fig. 23 Changes in Error% [CS3]

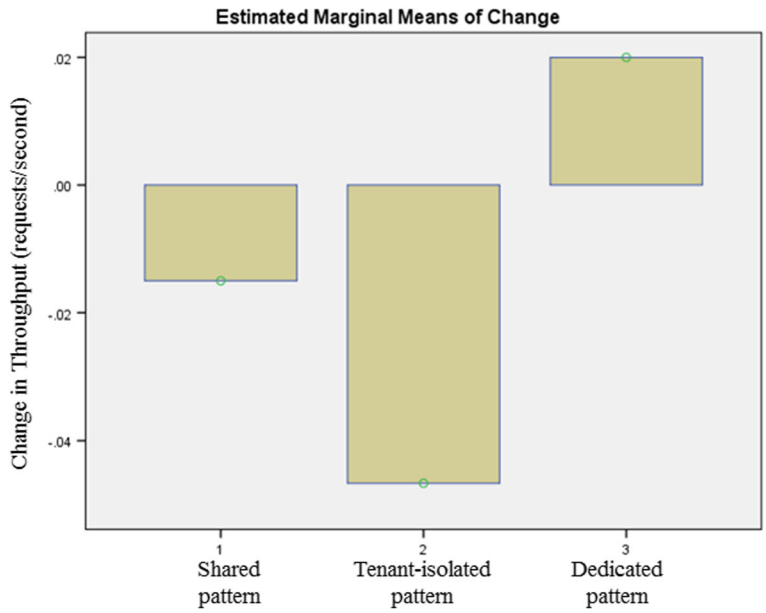

Fig. 24 Changes in Throughput [CS3]

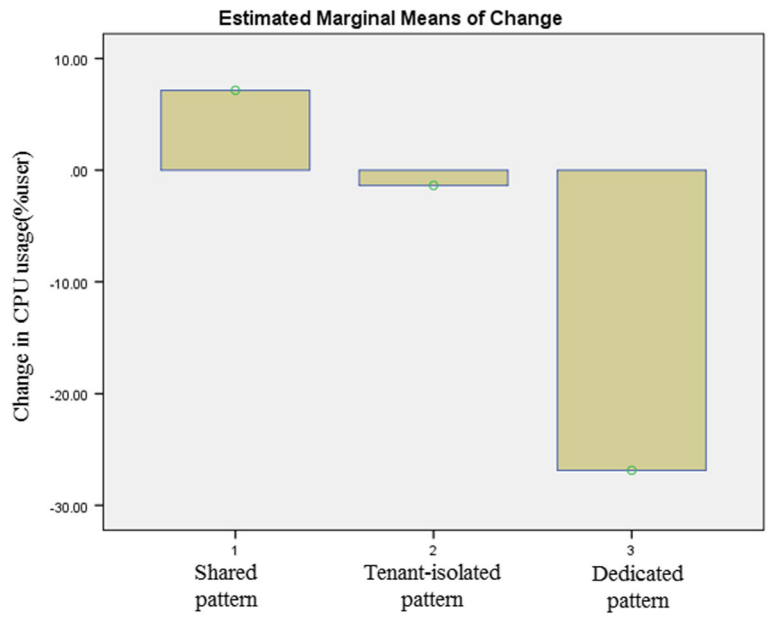

Fig. 25 Changes in CPU [CS3]

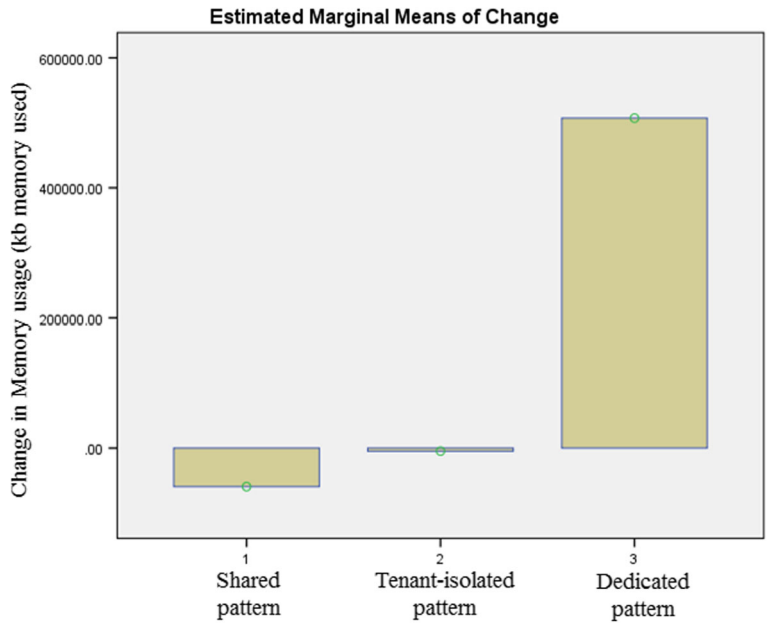

Fig. 26 Changes in Memory [CS3]

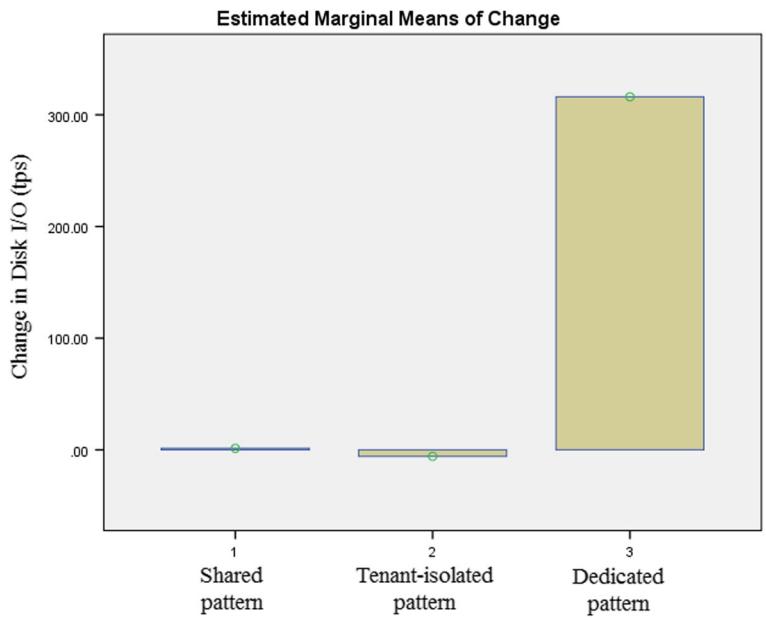

Fig. 27 Changes in Disk I/O [CS3]

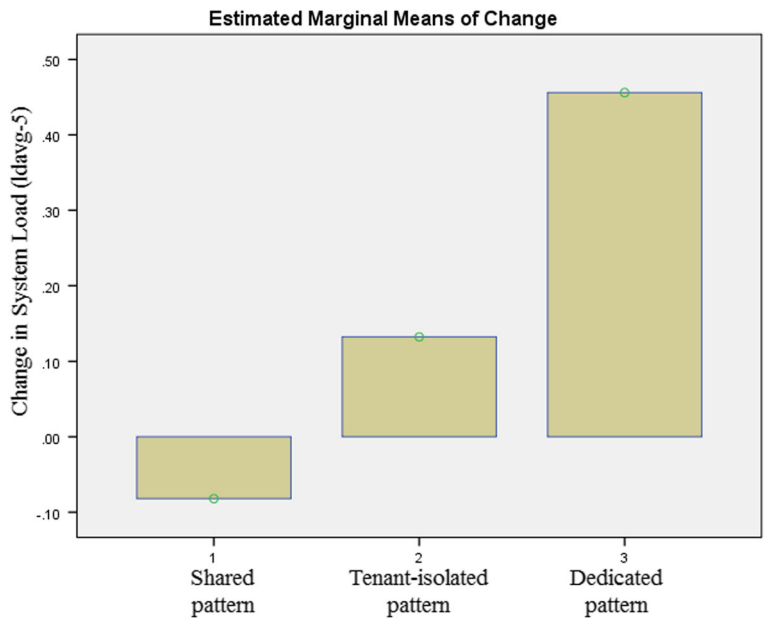

Fig. 28 Changes in System Load [CS3]

Abbreviations
The key abbreviations used in this paper are: COMITRE: Component-based approach to tenant isolation through request re-routing; EMMC: Estimated marginal means of change; GSD: Global software development; MMKP: Multichoice multidimensional knapsack problem; SCM: Source configuration management; SAR: System activity report

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

Funding
No formal funding was received for this research, but the research was supported by the Tertiary Education Trust Fund (TETFUND), Nigeria and Robert Gordon University, Aberdeen, UK.

Availability of data and materials
Not applicable.

Authors' contributions
LCO is the main author in this research paper. JB supervised and reviewed the case studies and synthesis of findings from the case studies. AP contributed to the literature review and general organization of the paper. All authors read and approved the final manuscript.

Authors' information
1. *Dr. Laud Charles Ochei* holds PhD from Robert Gordon University, Aberdeen, United Kingdom. His research interests is in software engineering, distributed systems, cloud computing, and Internet of things. He has published several research papers in International Conferences and Journals.
2. *Dr. Andrei Petrovski* is a Reader in Computational Systems at Robert Gordon University, Aberdeen, United Kingdom. His primary research interests lie in the field of Computational Intelligence (CI) - particularly, in the application of CI heuristics (such as Genetic Algorithms and Particle Swarm Optimisation) to single- and multi-objective optimisation problems. Andrei has an interest in computer-assisted measurements, virtual instrumentation, and sensor networks.
3. *Dr. Julian Bass* is a Senior Lecturer at University of Salford, UK. His research interest are in software development for large-scale systems focusing on multi-national teams and using modern lean and agile methods. He also has interests in deployment architectures used in cloud-hosted software services and leading KTP with Add Latent Ltd to develop and deploy cloud-hosted asset management applications for their major clients in the energy and utility sectors.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Robert Gordon University, School of Computing and Digital Media, Aberdeen AB10 7QB, UK. [2]University of Salford, School of Computing, Science and Engineering, Salford M5 4WT, UK.

Received: 22 June 2018 Accepted: 26 October 2018
Published online: 17 December 2018

References
1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672
2. Khazaei H, Misic J, Misic VB (2012) Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems. Parallel Distrib Syst IEEE Trans on 23(5):936–943
3. Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P (2014) Cloud Computing Patterns. Springer, London
4. Bauer E, Adams R (2012) Reliability and availability of cloud computing. Wiley, New Jersey
5. Ochei LC, Bass J, Petrovski A (2015) Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools. In: 2015 International Conference on Cloud and Autonomic Computing (ICCAC). IEEE. pp 101–112. https://ieeexplore.ieee.org/abstract/document/7312145/
6. Ochei LC, Petrovski A, Bass J (2015) Evaluating degrees of isolation between tenants enabled by multitenancy patterns for cloud-hosted version control systems (vcs). Int J Intell Comput Res 6(3):601–612
7. Ochei LC, Bass J, Petrovski A (2016) Implementing the required degree of multitenancy isolation: A case study of cloud-hosted bug tracking system. In: 13th IEEE International Conference on Services Computing (SCC 2016). IEEE
8. Runeson P, Host M, Rainer A, Regnell B (2012) Case study research in software engineering: Guidelines and examples. Wiley, New Jersey
9. Cruzes DS, Dybå T, Runeson P, Höst M (2015) Case studies synthesis: a thematic, cross-case, and narrative synthesis worked example. Empir Softw Eng 20(6):1634–1665
10. Cruzes DS, Dybå T (2011) Research synthesis in software engineering: A tertiary study. Inf Softw Technol 53(5):440–455
11. Chong F, Carraro G (2006) Architecture strategies for catching the long tail. technical report, microsoft. [Online https://msdn.microsoft.com/en-us/library/aa479069.aspx]. Accessed Oct 2018
12. Wang ZH, Guo CJ, Gao B, Sun W, Zhang Z, An WH (2008) A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In: IEEE International Conference on e-Business Engineering. IEEE. pp 94–101. https://ieeexplore.ieee.org/abstract/document/4690605/
13. Vengurlekar N (2012) Isolation in private database clouds. Oracle Corporation. [Online https://www.oracle.com/technetwork/database/database-cloud/]. Accessed Oct 2018
14. Walraven S, De Borger W, Vanbrabant B, Lagaisse B, Van Landuyt D, Joosen W (2015) Adaptive performance isolation middleware for multi-tenant saas. In: Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on. IEEE. pp 112–121. https://ieeexplore.ieee.org/abstract/document/7431402/
15. Mietzner R, Unger T, Titze R, Leymann F (2009) Combining different multi-tenancy patterns in service-oriented applications. In: Proceedings of the 2009 IEEE International Enterprise Distributed Object Computing Conference (edoc 2009). IEEE. pp 131–140. https://ieeexplore.ieee.org/abstract/document/5277698/
16. Guo CJ, Sun W, Huang Y, Wang ZH, Gao B (2007) A framework for native multi-tenancy application development and management. In: Proceedings of the 2007 IEEE International Conference on ECommerce Technology and the IEEE International Conference on Enterprise Computing, E-Commerce, and EServices. IEEE. pp 551–558. http://doi.ieeecomputersociety.org/10.1109/CEC-EEE.2007.4
17. Walraven S, Monheim T, Truyen E, Joosen W (2012) Towards performance isolation in multi-tenant saas applications. In: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing. ACM. p 6
18. Krebs R, Wert A, Kounev S (2013) Multi-tenancy performance benchmark for web application platforms. In: Web Engineering. Springer, Berlin. pp 424–438. https://link.springer.com/chapter/10.1007/978-3-642-39200-9_36
19. Youngs R, Redmond-Pyle D, Spaas P, Kahan E (1999) A standard for architecture description. IBM Syst J 38(1):32–50
20. Varia(c) J Cloud architectures. Amazon Web Services (AWS). [Online http://www.truecloudcosts.com/Docs/Amazon%20-%20Cloud%20Architectures.pdf]. Accessed Nov 2018
21. Varia(a) J (2014) Migrating your existing applications to the cloud: a phase-driven approach to cloud migration. Amazon Web Services (AWS). Online https://d1.awsstatic.com/whitepapers/cloud-migration-main.pdf. Accessed Nov 2018
22. Varia(b) J (2014) Architecting for the cloud: best practices. Amazon Web Services (AWS). Online https://d1.awsstatic.com/whitepapers/AWS_Cloud_Best_Practices.pdf. Accessed Oct 2018
23. Cruzes DS, Dybå T (2010) Synthesizing evidence in software engineering research. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM. p 1. https://dl.acm.org/citation.cfm?id=1852788
24. RAGIN C (1997) Turning the tables: How case-oriented research challenges variable-oriented research. Comparative Social Research 16:27–42
25. Yin RK (2014) Case Study Research: Design and methods, 4th ed, Vol. 5. Sage Publications, Inc., California

26. Ochei LC, Bass JM, Petrovski A (2015) A novel taxonomy of deployment patterns for cloud-hosted applications: A case study of global software development (gsd) tools and processes. Int J Adv Softw 8(3-4): 420–434
27. Runeson P, Host M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164
28. Sheridan JC, Ong C (2011) SPSS version 18.0 for Windows-Analysis without anguish. Wiley, Milton
29. Verma JP (2015) Repeated Measures Design for Empirical Researchers. Wiley, New Jersey
30. Ochei LC, Bass JM, Petrovski A (2018) A framework for achieving the required degree of multitenancy isolation for deploying components of a cloud-hosted service. Int J Cloud Comput 7(3/4):248–281
31. Hudson (2018) Hudson - continuous integration server. Eclipse Foundation Project. [Online https://www.eclipse.org/hudson/]. Accessed Nov 2018
32. Manfred Moser M, O'Brien T (2011) The Hudson book. Oracle, Inc., USA. Oracle, Inc., California. Online https://www.eclipse.org/hudson/the-hudson-book/book-hudson.pdf. Accessed Nov 2018
33. Hudson (2018) Files found trigger. [Online https://plugins.jenkins.io/files-foundtrigger]. Accessed Jan 2018
34. Bugzilla (2015) The bugzilla guide - 4.0.18+ release. The Mozilla Foundation. Oracle. [Online https://www.bugzilla.org/docs/4.0/en/pdf/Bugzilla-Guide.pdf/]. Accessed Nov 2018
35. Johnson D, Kiran M, Murthy R, Suseendran R, Yogesh G (2016) Eucalyptus beginner's guide - uec edition. Eucalyptus Systems, Inc. [Online https://www.csscorp.com/eucauecbook]. Accessed Feb 2017
36. Pantić Z, Babar MA (2012) Guidelines for building a private cloud infrastructure. IT University of Copenhagen, Denmark. ITU Technical Report Series (TR-2012-1530). https://pure.itu.dk/portal/en/publications/guidelines-for-building-a-private-cloud-infrastructure(ba9de07f-75c3-46e6-b749-6a97e94561ad).html. http://130.226.142.177/wp-content/uploads/2012/05/Guidelines-to-BuildingPrivateCloud-Infrastructure-Technical-Report.pdf
37. Erinle B (2013) Performance Testing with JMeter 2.9. Packt Publishing Ltd, Birmingham
38. Field A (2013) Discovering statistics using IBM SPSS statistics. Sage Publications Ltd, London
39. Electric-Cloud (2018) Build automation: Top 3 problems and how to solve them. Electric Cloud, Inc. [Online https://electric-cloud.com/plugins/build-automation]. Accessed Nov 2018
40. Subversion (2018) Working copy metadata storage improvements (client). The Apache Software Foundation. [Online https://wiki.eclipse.org/Hudson-ci/writing-first-hudson-plugin]. Accessed Nov 2016
41. Subversion (2018) Working copy metadata storage improvements (client). The Apache Software Foundation. O'Reilly, California. Online https://subversion.apache.org/docs/releasenotes/1.7. Accessed Nov 2018
42. Hudson (2018) Hudson-ci/writing-firsthudsonplugin. Eclipse. [Online https://wiki.eclipse.org/Hudson-ci/writing-first-hudson-plugin]. Accessed Nov 2018
43. Bass L, Clements P, Kazman R (2013) Software Architecture in Practice, 3/E. Elsevier, Cambridge
44. Sharma A, Kumar M, Agarwal S (2015) A complete survey on software architectural styles and patterns. Procedia Comput Sci 70:16–28
45. Schmerl B, Kazman R, Ali N, Grundy J, Mistrik I (2017) Managing trade-offs in adaptable software architectures. In: Managing Trade-Offs in Adaptable Software Architectures. Elsevier, Cambridge. pp 1–13
46. Furda A, Fidge C, Barros A, Zimmermann O (2017) Reengineering data-centric information systems for the cloud–a method and architectural patterns promoting multitenancy. In: Software Architecture for Big Data and the Cloud. Elsevier, Cambridge. pp 227–251. https://www.sciencedirect.com/science/article/pii/B9780128054673000223
47. Khan MF, Mirza AU, et al. (2012) An approach towards customized multi-tenancy. Int J Mod Educ Comput Sci 4(9):39
48. Badger L, Grance T, Patt-Corner R, Voas J (2012) Cloud computing synopsis and recommendations. NIST Spec Publ 800:146
49. Chong F, Carraro G, Wolter R (2017) Multi-tenant data architecture. Microsoft Corporation. [Online https://msdn.microsoft.com/en-us/library/aa479086.aspx]. Accessed 15 Feb 2017
50. Vanhove T, Vandensteen J, Van Seghbroeck G, Wauters T, De Turck F (2014) Kameleo: Design of a new platform-as-a-service for flexible data management. In: Network Operations and Management Symposium (NOMS), 2014 IEEE. IEEE. pp 1–4
51. Schneider M, Uhle J (2013) Versioning for software as a service in the context of multi-tenancy. University of Potsdam, Hasso-Plattner-Institute, Potsdam, Germany, Tech. Rep. http://www.freenerd.de/assets/VersioningSaas_SchneiderUhle.pdf
52. Schiller O (2015) Supporting multi-tenancy in relational database management systems for oltp-style software as a service application. Ph.D. dissertation, University of Stuttgart, Germany. https://doi.org/10.18419/opus-3589. https://elib.uni-stuttgart.de/handle/11682/3606
53. Krebs R (2015) Performance isolation in multi-tenant applications. Ph.D. dissertation, Karlsruhe Institute of Technology, Germany. https://se.informatik.uni-wuerzburg.de/fileadmin/10030200/user_upload/dissKIT_BW.PDF
54. Moyer C (2012) Building Applications for the Cloud: Concepts, Patterns and Projects. In: Pearson Education, Inc, Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116. Addison-Wesley Publishing Company, Boston
55. Homer A, Sharp J, Brader L, Narumoto M, Swanson T (2014) Cloud Design Patterns(Corbisier R, ed.). Microsoft, Washington
56. Walraven S, Van Landuyt D, Truyen E, Handekyn K, Joosen W (2014) Efficient customization of multi-tenant software-as-a-service applications with service lines. J Syst Softw 91:48–62
57. Aldhalaan A, Menascé DA (2015) Near-optimal allocation of vms from iaas providers by saas providers. In: Cloud and Autonomic Computing (ICCAC), 2015 International Conference on. IEEE. pp 228–231
58. Ochei LC, Petrovski A, Bass JM (2016) Optimizing the deployment of cloud-hosted application components for guaranteeing multitenancy isolation. In: Information Society (i-Society), 2016 International Conference on. IEEE. pp 77 – 83. International Conference on Information Society (i-Society 2016)
59. Fowler M (2006) Continuous integration. houghtWorks, Inc. [Online https://www.martinfowler.com/....html]. Accessed 16 May 2018
60. Microsoft(a) (2018) Storsimple: An enterprise hybrid cloud storage solution. Microsoft Inc. [Online https://azure.microsoft.com/en-gb/services/storsimple/]. Accessed 10 May 2018
61. Ziembicki D (2014) Microsoft System Center Integrated Cloud Platform. Microsoft Press, Washington
62. Ochei LC, Petrovski A, Bass J (2016) An approach for achieving the required degree of multitenancy isolation for components of a cloud-hosted application. In: 4th International IBM Cloud Academy Conference (ICACON 2016). IBM Cloud Academy, IBM Corporation, New York. https://www.ibm.com/solutions/education/cloudacademy/us/en/cloud_academy_conference_2016.html
63. Amazon(a) (2017) Amazon elastic compute cloud (ec2) documentation. Amazon Web Services, Inc. [Online https://aws.amazon.com/documentation/ec2/]. Accessed 17 Feb 2017
64. Poddar R, Vishnoi A, Mann V (2015) Haven: Holistic load balancing and auto scaling in the cloud. In: Communication Systems and Networks (COMSNETS), 2015 7th International Conference on. IEEE. pp 1–8
65. Kim M, Mohindra A, Muthusamy V, Ranchal R, Salapura V, Slominski A, Khalaf R (2016) Building scalable, secure, multi-tenant cloud services on ibm bluemix. IBM J Res Dev 60(2-3):8–1
66. Amazon(b) (2017) What is auto scaling? Amazon Web Services, Inc. [Online http://docs.aws.amazon.com/autoscaling/]. Accessed 8 Mar 2017
67. Microsoft(b) (2016) Introducing microsoft azure. Microsoft Corporation. [Online https://azure.microsoft.com/]. Accessed 13 Sept 2016
68. German DM, Adams B, Hassan AE (2016) Continuously mining distributed version control systems: an empirical study of how linux uses git. Empir Softw Eng 21(1):260–299
69. Corbet J, Kroah-Hartman G, McPherson A (2013) Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. The Linux Foundation. [Online https://www2.thelinuxfoundation.org/]. Accessed 16 May 2018
70. Doddavula SK, Agrawal I, Saxena V (2013) Cloud computing solution patterns: Infrastructural solutions. In: Cloud Computing: Methods and Practical Approaches. Springer, London. pp 197–219
71. Krebs R, Momm C, Kounev S (2014) Metrics and techniques for quantifying performance isolation in cloud environments. Sci Comput Program 90:116–134