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

TITLE: A framework for achieving the required degree of multitenancy isolation for deploying components of a cloud-hosted service.

YEAR: 2018

Publisher citation: OCHEI, L.C., PETROVSKI, A. and BASS, J.M. 2018. A framework for achieving the required degree of multitenancy isolation for deploying components of a cloud-hosted service. International journal of cloud computing [online], 7(3-4), pages 248-281. Available from: https://doi.org/10.1504/IJCC.2018.095399

OpenAIR citation: OCHEI, L.C., PETROVSKI, A. and BASS, J.M. 2018. A framework for achieving the required degree of multitenancy isolation for deploying components of a cloud-hosted service. International journal of cloud computing, 7(3-4), pages 248-281. Held on OpenAIR [online]. Available from: https://openair.rgu.ac.uk

A Framework for Achieving the Required Degree of Multitenancy Isolation for Deploying Components of a Cloud-hosted Service

Laud Charles Ochei

School of Computing and Digital Media,
Robert Gordon University,
Aberdeen AB10 7QB, United Kingdom.
Email: l.c.ochei@rgu.ac.uk
* Corresponding author

Andrei Petrovski

School of Computing and Digital Media,
Robert Gordon University,
Aberdeen AB10 7QB, United Kingdom.
Email: a.petrovski@rgu.ac.uk

Julian M. Bass

School of Computing, Science, and Engineering,
University of Salford,
Salford M5 4WT, United Kingdom.
Email: J.Bass@salford.ac.uk

Abstract: When a cloud offering is provided to multiple users/tenants, multitenancy isolation has to be implemented. While several approaches exist for implementing multitenancy, little attention has been paid to implementing the required degree of isolation since there are varying degrees of isolation that can be implemented for each tenant. This paper presents a framework for achieving the required degree of isolation between tenants accessing a cloud offering so that the required performance, resource utilization and access privilege of one tenant does not affect other tenants when there are workload changes. The framework is composed of two main constituents (i) Component-based approach to Multitenancy Isolation through Request Re-routing (COMITRE), (ii) an optimization model for providing optimal solutions for deploying components of a cloud-hosted service. We demonstrate using a case study of (i) a Cloud-hosted Bug Tracking System and (ii) a synthetic dataset, that the required degree of multitenancy isolation can be achieved, while at the same time providing optimal solutions for deploying components of a cloud-hosted service. We also provide challenges and recommendations for implementing the framework on different layers of the cloud stack.

Keywords: Multitenancy, Degree of Isolation, Cloud-hosted service, Bug Tracking System, Global Software Development tools, Components, Optimal solution, Optimization Model.

Reference to this paper should be made as follows: Ochei, L.C., Bass, J.M. and Petrovski, A. (2018) ‘A Framework for Achieving the Required Degree of Multitenancy Isolation for Deploying Components of a Cloud-hosted Service’, *International Journal of Cloud Computing*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Dr. Laud Charles Ochei holds a PhD in Computing from Robert Gordon University, Aberdeen, United Kingdom. He has a broad range of research and software development experience in various academic and industry collaborations. His research interests are in software engineering, distributed systems, Internet of Things, and cloud application architectures. He is also interested in developing novel approaches for deploying cloud-hosted services to guarantee multitenancy isolation. He has published several research papers in peer-reviewed International Conferences and Journals.

Dr Andrei Petrovski is a Reader at Robert Gordon University, Aberdeen, United Kingdom. His research expertise includes computational modelling, optimisation and decision support. He is also interested in practical application of cloud and mobile computing, as well as computer-assisted measurements. He uses these technologies for improving versatility of information systems and for adding intelligent features to these systems. This has led to the publication of over 60 peer-reviewed papers including a book chapter, journal and conference articles in the field of Computational Intelligence, Control Systems, Medical Informatics and Decision Support. Application domains of his research include oil and gas industry, medicine and health informatics, electrical power systems.

Dr Julian Bass is a Senior Lecturer at the University of Salford, Manchester, UK. He has published over one hundred peer reviewed conference papers and journal articles. His interests include cloud application architectures, software development processes for large-scale projects and ICTs for internal development. He is a Senior Editor for the Electronic Journal on Information Systems in Developing Countries. Dr Bass is a Chartered Engineer and Fellow of BCS, the Chartered Institute for IT.

This paper is a revised and expanded version of ‘An approach for achieving the required degree of multitenancy isolation for components of a cloud-hosted application’ presented at the 4th International IBM Cloud Academy Conference (ICACON 2016), University of Alberta, Edmonton, Canada, June 2-3, 2016.

1 Introduction

In recent times, Global Software Development (GSD) tools used for software processes such continuous integration (CI), version control (VC) and bug tracking, are increasingly being deployed on the cloud (Chauhan and Babar, 2012) (Ochei, Bass and Petrovski, 2015*a*). For example, large companies like Apple and Oracle are using software tools like Hudson to setup deployments and automate the management of cloud-based infrastructure (Moser and O’Brien, 2016). As these software tools are used by multiple tenants/users, there is a requirement for implementing multitenancy isolation, which entails isolating tenants’ data and processes so that the performance, resource utilization and access privilege of one tenant does not affect other tenants. Multitenancy isolation is especially required in a scenario where one of the tenants accessing a cloud-hosted service or a component of cloud-hosted service suddenly receives a high workload in response to certain changes in the deployment conditions of the cloud-hosted service (see Figure 1).

In architecting the design and deployment of multitenant cloud-hosted services, tenants may require different degrees of isolation. Examples of such services include business software such as CRM, software development tools such as Hudson, office applications such as Google docs, web-based email, photo sharing, etc. At the very basic degree of multitenancy, tenants would be able to 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 not shareable due to certain laws and regulations.

The sharing of application components and the underlying resources between tenants has the potential of improving the efficient utilization of resources and reducing the cost of running the application. However, this sharing reduces the degree of isolation between tenants because other tenants may experience performance degradation if one of the tenants experiences a high load. This means that a high degree of isolation (e.g., a component offering critical functionality) is important for avoiding performance interference, but leading to high resource consumption and running cost while a low degree of isolation (e.g., a component that requires minimal reconfiguration) promotes resource sharing but is more prone to performance interference when workload changes. Therefore an architect has the task of implementing not just varying degrees of multitenancy isolation but also of resolving these conflicting trade-offs in order to achieve the required degree of multitenancy isolation.

Motivated by this problem, this paper presents a framework that can be used to achieve the required degree of isolation between tenants. The research question addressed in this paper is: **"How can we achieve the required degree of isolation as well as optimal solutions for deploying multiple components (or tenants) associated with a multitenant cloud-hosted service"**.

We implemented three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) by modifying Bugzilla (hosted on a private cloud) in a way that isolates the data of different tenants. In addition, we also developed a model for providing optimal solutions for deploying components of the cloud-hosted service. This model was extensively evaluated using a synthetic testbed to show that the obtained solutions are optimal and can be used to deploy components of service to the cloud.

This paper is an extension of the previous work by Ochei et al. (Ochei et al., 2016) where the authors presented an approach for achieving the required degree of multitenancy isolation for components of a cloud-hosted application. The previous work focused on using a cloud-hosted bug tracking system (i.e., Bugzilla) as a case study to demonstrate the applicability of the approach in a scenario involving multiple tenants submitting bugs at varying frequency to the bug database. We extend our previous work by first developing an optimization model for providing optimal solutions for deploying components of a cloud-hosted service, and then combining it with the approach developed in the previous work to produce a framework for achieving the required degree of multitenancy isolation for deploying components of a cloud-hosted service.

This study aims to show that we can achieve the required degree of isolation while providing empirical evidence of the evaluation of varying degrees of multitenancy isolation for cloud-hosted software tools. The main contributions of this paper are:

1. Presenting a framework composed of two key constituents for achieving the required degree of multitenancy isolation for a particular tenant:

(i) Component-based approach to Multitenancy Isolation through Request Re-routing (COMITRE) for implementing varying degrees of isolation between tenants.
(ii) an optimization model for providing optimal solutions for deploying components of a cloud-hosted service.
2. Extensively evaluating and demonstrating the applicability of each component of the framework in a case study involving both a real-world application (i.e., cloud-hosted bug tracking system) and a synthetic testbed.
3. Presenting recommendations in terms of (i) suitable (multitenancy) cloud patterns, and (ii) different implementation options on the different layers of the cloud stack, in order to achieve multitenancy isolation.

The rest of the paper is organized as follows - Section two presents an overview of cloud architectures for achieving multitenancy isolation. Section three discusses the conflicting trade-offs for consideration in order to achieve the required degree of isolation. Section four presents the framework for achieving the required degree of multitenancy isolation including COMITRE and the optimization model. Section five is the evaluation of the approaches. Section six presents the results and discussion. The recommendations and limitations of the study are detailed in Sections seven and eight respectively. Related work is presented in nine and Section ten concludes the paper with future work.

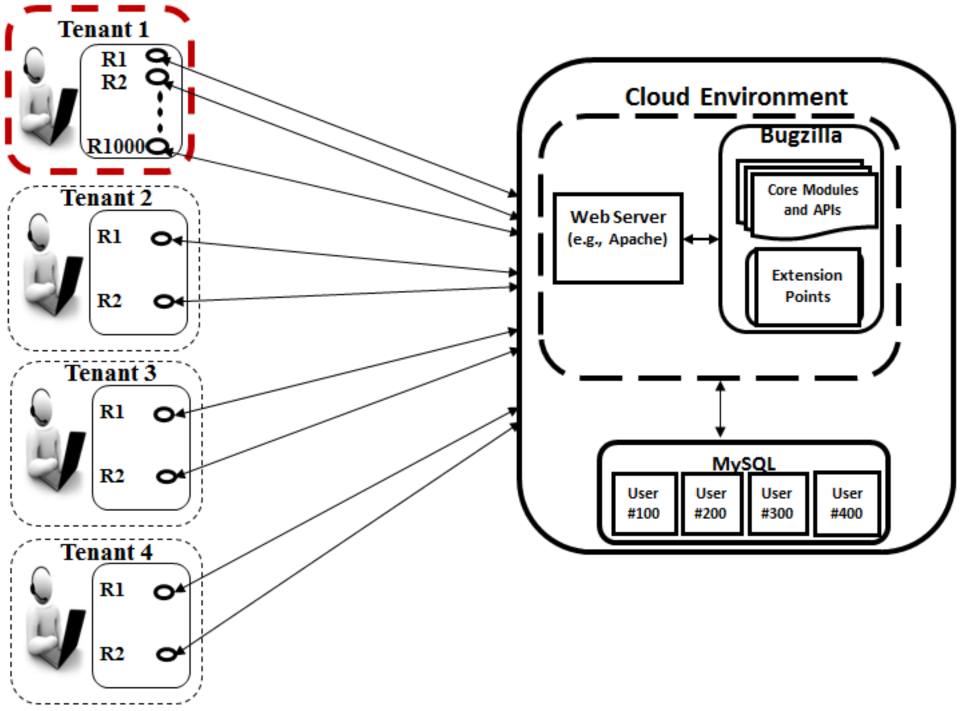

Figure 1. One of the tenants sends high workload to the multitenant application.

2 Cloud Architectures for Achieving the Required Degree of Multitenancy Isolation

Architectural and design patterns have long been used by software architects to provide known solutions to a number of common problems facing a distributed system (Bass et al., 2013). The architecture of a system determines whether or not the system's required quality attributes such as performance, availability and security will be satisfied (Bass et al., 2013) (Vlissides et al., 1995). In the cloud computing environment, a cloud pattern represents a well-defined format for describing a suitable solution to a cloud-related problem. This paper uses the term "'Cloud deployment pattern" instead of "Cloud patterns" to focus on a class of architectural patterns that embody decisions as to how elements of the cloud application will be assigned to the cloud environment where the application is executed.

In our previous work (Ochei, Bass and Petrovski, 2015*a*), we developed a taxonomy for guiding a cloud deployment architect to focus on a particular architectural deployment component. This could either be on the: (i) cloud-hosted environment, to map the business requirement of an organization to cloud properties that cannot be changed (e.g., location and ownership of the cloud infrastructure), or (ii) cloud-hosted application, to mitigate certain cloud properties that can be compensated at an application level (e.g., improving the availability of a component integrated into a cloud-hosted GSD tool). The framework and the associated algorithms that are presented in this paper apply to multitenancy patterns, which target the cloud-hosted application at the application level, and so are implemented almost at runtime.

When multiple tenants are accessing components of a cloud application, an architect has to implement "*Multitenancy isolation*" to ensure that the performance, stored data volume and access privileges required by one tenant does not affect other tenants accessing the component or functionality of a shared application component (Ochei, Bass and Petrovski, 2015*b*) (Ochei, Petrovski and Bass, 2015). Three multitenancy patterns which express the degree of isolation between tenants accessing an application component have been presented in (Fehling et al., 2014). These patterns are: shared component, tenant-isolated component and dedicated component. The dedicated component represents the highest degree of isolation whereas the shared component represents the lowest. The degree of isolation for a component deployed based on a tenant-isolated component would be somewhere in the middle. This is the case with hybrid deployment scenarios, where the distinction between the shared component and the dedicated component is not binary. Instead, it is more of a continuum, with many possible variations between the two extremes.

Multitenancy isolation is influenced by three main aspects of a system (Fehling et al., 2014): performance, resource utilization, and accessibility to the functionality and data. Our approach can be applied as an element of differentiation from several cloud perspectives such as price, performance and confidential data. For example, the dedicated component pattern can be implemented at the application level of platform level to provide the highest degree of isolation for a tenant whose confidential data cannot be accessed or shared with other tenants throughout the execution of the application.

3 Conflicting Trade-offs in Achieving Multitenancy Isolation

When implementing multitenancy, users may require varying or different degrees of isolation between components. A high degree of isolation between components may be required to avoid interference, but this usually leads to high resource consumption and running cost per component. A low degree of isolation promotes the sharing of components, thus leading to low resource consumption and running cost, but with a high possibility of performance influence when the workload changes and the application does not scale up/down.

Therefore, the challenge is how to determine an optimal solution in the presence of trade-offs between two or more conflicting objectives (Martens et al., 2010) (Legriel et al., 2010). To resolve this trade-off, the problem is modelled as a multi-objective optimisation problem. Many multi-objective optimisation problems result in a trade-off situation that involve losing some quality of one objective function in return for gaining quality in some of the other objective functions (Martens et al., 2010) (Legriel et al., 2010; Garg et al., 2012). In our case, we either lose resource sharing capabilities to gain a higher degree of isolation when implementing a dedicated component or suffer an increase in performance interference to gain the ability to share resources, reduce running cost and resource consumption, and target a large number of users when implementing a shared component.

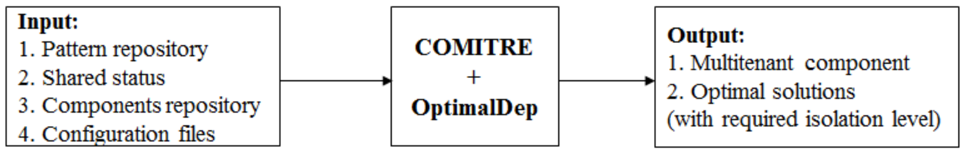

Figure 2. Framework as part of an input-process-output model

4 A Framework for Achieving the Required Degree of Multitenancy Isolation

The framework is first captured in Figure 2 as part of an input-process-output (IPO) model, an approach widely used in systems analysis and software engineering for describing the basic structure of a service or process (Grady, 1995; Bahill and Madni, 2017). In our case, this model represents a cloud-hosted service that can be designed to use or integrate with several components and/or other services. In using the IPO model, the framework receives inputs from a user (e.g., pattern repository, component repository, a configuration file), carries out some analysis and optimization, and returns the results (e.g., a multitenant component with the required degree of isolation, optimal solutions). Thereafter, we use a (procedural) flowchart model in Figure 3 to illustrate the overall flow of tasks in using the constituents of the framework to achieve the required degree of multitenancy isolation for deploying components of a cloud-hosted service.

4.1 COMITRE: Component-based approach to Multitenancy Isolation through Request Re-routing

We describe an improved version of an algorithm (i.e., Algorithm 1) that can be used to achieve the required degree of isolation for a particular application component that is accessed by multiple tenants. This algorithm has to be executed within the COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing) architecture. In a nutshell, COMITRE is anchored on shifting the task of routing

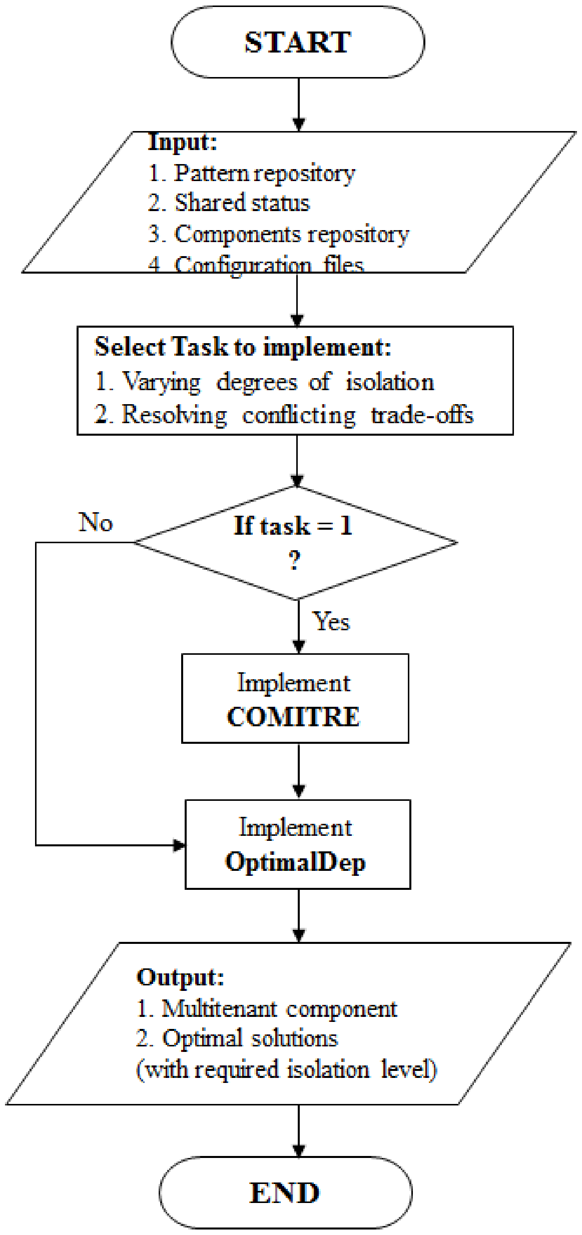

Figure 3. A procedural flowchart for illustrating the framework

a request from the web server to a separate component (e.g., Java class or plugin) at the application level of a cloud-hosted GSD tool. The structure of COMITRE is shown in Figure 4, and the full explanation of COMITRE plus the step-by-step procedure was presented in Ochei et al. (Ochei, Bass and Petrovski, 2015*b*) (Ochei, Petrovski and Bass, 2015).

One of the main inputs to *Algorithm 1* is the required isolation level for the deployed component. This can be set to either 1, 2 or 3 to represent the shared component, tenant-

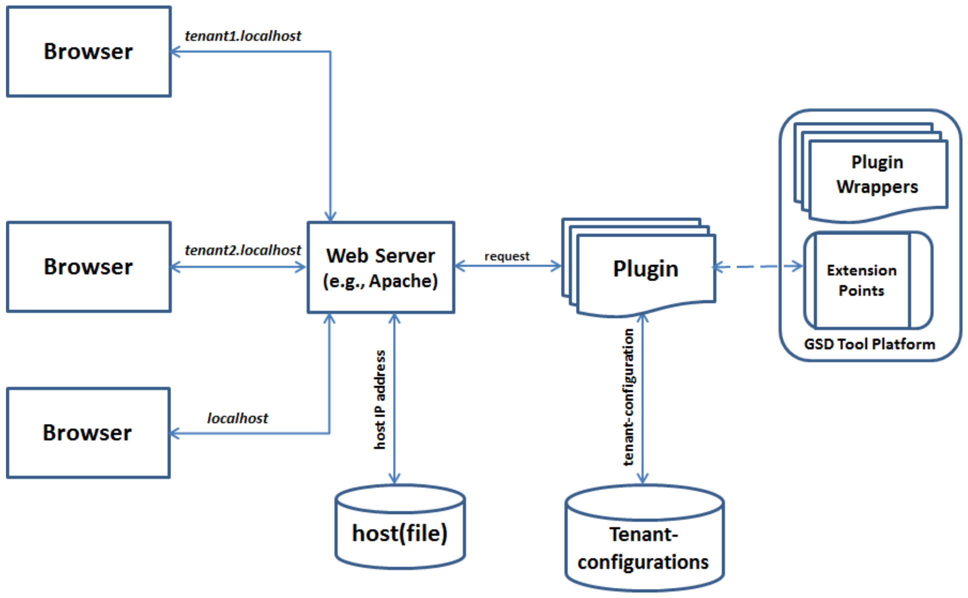

Figure 4. Architectural diagram of COMITRE approach

isolated component, and dedicated component, respectively. If the isolation level is either 1 or 2, then the created component can be shared with other tenants and can be accessed by all tenants irrespective of where it is located. However, for isolation level set to 2, the tenant has to be authenticated first and then assigned a unique tenantID, which is used to adjust the behaviour of the created component. If the isolation level is 3, then the created component is marked as not to be shared with others, and so is dedicated exclusively for one tenant.

Algorithm 1 assumes that the architect specifies the required isolation level for the component. However, in a cloud environment such decisions have to be taken in almost real-time, and so we also need an algorithm that can determine which isolation level is best for a component or functionality to be created. *Algorithm 2* presents an algorithm to determine the isolation level for a tenant application function based on an existing application component on the cloud infrastructure. In line 4, if *sharedStatus* is true, then the isolation level is set to either 1 or 2, otherwise, control is transferred to line 14-15 to assign the isolation level 3. The first type of information is whether or not the application component is similar in functionality or configuration to existing ones. This is captured in line 5-8 of *Algorithm 2*, where similar components are searched for in the component repository. Assuming that components with similar configurations are found, then the isolation level is set to 1 in line 10, otherwise, it is set to 3.

Algorithm 1: *COMITRE Algorithm*

```
1: INPUT: tenantRequest, tenantConf-file, isolationLevel
2: OUTPUT: multApplFunctn
3: Get tenantID from incoming request
4: tenantConf ← null
5: sameConf ← false
```

```
6: share ← true
7: Select tenantData from tenantConf-file
8: if tenantData is found then
9:     tenantConf ← tenantData
10: end if
11: Create defaultApplFunctn
12: multApplFunctn ← defaultApplFunctn
13: if tenantConf is not null then
14:     if isolationLevel = 1 then
15:         sameConf*** ← true
16:         Create tenantApplFunctn
17:     else if isolationLevel = 2 then
18:         Authenticate tenantID
19:         Create tenantApplFunctn
20:         Adjust tenantApplFunctn with tenantID
21:     else if isolationLevel = 3 then
22:         Create tenantApplFunctn
23:         share ← false
24:     end if
25:     multApplFunctn ← tenantApplFunctn
26: end if
27: return multApplFunctn
```

Algorithm 2: *IsolationLevel Algorithm*

```
1: INPUT: compRepository, shareStatus
2: OUTPUT: isolationLevel
3: sameConf ← false
4: if shareStatus = true then
5:     Search compRepository for comp. with similar conf.
6:     if similar compConf is found then
7:         sameConf ← true
8:     end if
9:     if sameConf = true then
10:        isolationLevel = 1
11:    else
12:        isolationLevel = 2
13:    end if
14: else
15:     isolationLevel = 3
16: end if
17: return isolationLevel
```

4.2 OptimalDep: A Model for Providing Optimal Solutions to Deploy Components of a Cloud-hosted Service

For a cloud-hosted service that can be designed to use or be integrated with several components in N different groups, and with m resource constraints, the problem of providing optimal solutions that guarantee multitenancy isolation can be mapped to a 0-1 multichoice multidimensional knapsack problem (MMKP) (Martello and Toth, 1990; Kellerer et al., 2004).

4.2.1 *Description of the Optimal Deployment Problem*

Suppose there are N groups of components $(c_1,...,c_N)$ with each having l_i $(1 \leq \text{i} \leq N)$ components that can be used to design (or integrate with) a cloud-hosted application. Each application component is associated with: (i) the required degree of isolation between components (I_{ij}); (ii) the arrival rate of requests to the component λ_{ij}; (iii) the service demand of the resources supporting the component D_{ij}; (iv) the average number of requests that can be allowed to access the component Q_{ij} and (v) m resources which are required to support the component, $r_{ij}^{\alpha} = r_{ij}^1, r_{ij}^2,..., r_{ij}^m$. The total amount of available resources in the cloud required to support all the application components is $R = R^{\alpha}$ $(\alpha = 1,..., m)$. The objective of an MMKP is to pick exactly one component from each group for a maximum total value of the collected items, subject to m resource constraints of the knapsack (Yu et al., 2007; Akbar et al., 2006).

Concerning our problem, the goal is to deploy components of a cloud-hosted service by selecting one component from each group to meet the resource constraints of the system and maximise the optimal function $\mathcal{G}$. There are unique features in our problem that lend to solving it using an MMKP and an open multiclass problem. For example, the resources supporting each component are mapped to the resources required by the object in MMKP and are also mapped to the service centres of each class in the open multiclass QN.

The optimization problem faced by a cloud architect for deploying components of a cloud-hosted application due to workload changes is thus expressed as follows:

$$\text{Maximize } \mathcal{G} = \sum_{i=1}^{N} \sum_{j \in C_i} g_{ij}.a_{ij}$$

subject to

$$\sum_{i=1}^{N} \sum_{j \in C_i} r_{ij}^{\alpha}.a_{ij} \leq R^{\alpha}(\alpha = 1, 2, ..., m) \tag{1}$$

$$\sum_{j \in C_i}^{N} a_{ij} = 1$$

$$a_{ij} \in 0, 1(i = 1, 2, ..., N), j \in C_i$$

where a_{ij} is set to 1 if component j is selected from group C_i and 0 otherwise. Note that a_{ij} represents the components selected for deployment. Each component is associated with an isolation value, the number of requests allowed to access it, and values for the amount of resources it requires to operate (i.e., cpu, ram, disk and bandwidth size). The notation $r_{ij} = r_{ij}^1, r_{ij}^2,..., r_{ij}^m$, is the resource consumption of each application component j from group C_i. The total consumption of all resources r_{ij}^{α} of all application components must be less than the total amount of resources available in the cloud infrastructure $R = R^{\alpha}$ $(\alpha = 1,..., m)$.

Given an isolation value of a component I, and the average number of requests Q, that can be allowed to access the component:

$$g_{ij} = (w1 \times I_{ij}) + (w2 \times Q_{ij}) - (w3 \times P_{ij})$$

The penalty, P, for violating the resource constraints is given as:

$$P_{ij} = \sum_{j=1}^{m} Rj^{max} \left\{ 0, \left(\frac{R_j - R_j^{max}}{R_j^{max}} \right) \right\}^2 \qquad (2)$$

where w1,w2,w3 are the weights for isolation value (w1=100), number of requests(w2=1) and penalty(w3=0.1). The weights are chosen based on problem-specific knowledge so that more importance or preference is given to the isolation value and number of requests which are parameters to be maximised in our model. The degree of isolation, I_{ij}, for each component, is set to either 1, 2, or 3 for *shared component, tenant-isolated component and dedicated component*, respectively. It is important to note that the value of I_{ij} is not a continuum, and so it represents the three distinct categories of multitenancy patterns (i.e., shared pattern, tenant-isolation pattern and dedicated pattern) which can be used to deploy components to the cloud.

4.2.2 System Model and Assumptions

This research assumes that the cloud-hosted system is made up of multiple components of the same tenant (e.g., user, team or department of a company) on the same underlying cloud infrastructure. The components may be classified into different groups (e.g., storage components, processing components), where some components may either have the same functionality, and hence can be shared with other tenants or are exclusively dedicated to some tenants or group of tenants. Each application component requires a certain amount of resources of the underlying cloud infrastructure to support the number of requests it receives. When there are workload changes (e.g., a component receiving a high load) the goal of the model is to select components for deployment to maximize the degree of isolation between components by ensuring that they behave as if they were components of different tenants and, thus, are isolated from each other; and (ii) maximizes the number of requests allowed to access the component without having the total resources used to exceed the available resources (see Figure 5).

To calculate the number of requests, Q_{ij} that can be allowed to access the component, an open multiclass QN model has to be solved (Menasce et al., 2004) for each component using the arrival rate of each class of requests, and the service demands of each resource required to support the component (i.e., CPU, RAM, Disk capacity, and Bandwidth).In order to solve the QN model, it is important to make the following assumptions:
(i) a component is deployed to support a single cloud application, and so cannot support different applications or applications at different system requirements; (ii) requests sent to a component have significantly different behaviour whose arrival rate is independent of the system state; (iii) the service demands at the CPU, RAM, Disk, and Bandwidth that support each component are known or can be easily measured by either the SaaS provider or the SaaS customer; (iv) the resources supporting each component are enough to handle the magnitude of new incoming requests as the workload changes. This ensures that there is no overload when all components are functional.

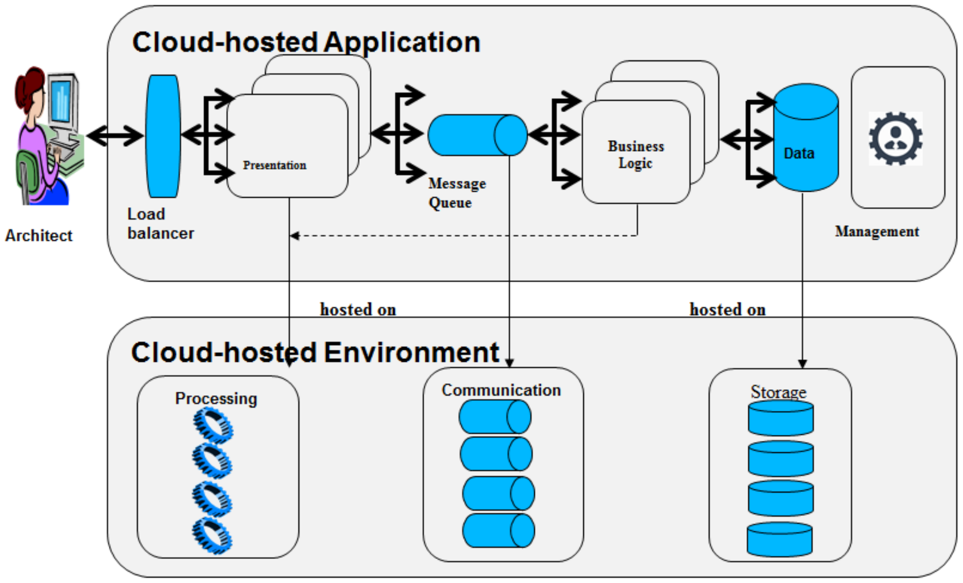

Figure 5. System Model of a Cloud-hosted Service with multiple groups of components

4.2.3 Description of OptimalDep Algorithm

A high-level description of the optimalDep algorithm is as follows: when a request arrives indicating a change in workload, the algorithm uses the open multiclass QN model to determine for each class, the queue length (i.e., the average number of requests allowed to access a component) as a function of the arrival rates (i.e., λ) for each class (lines 7-14). The average number of requests is used to update the properties of each component (i.e., mmkpFile) (line 15). Then the metaheuristic search is run to obtain the optimal solution for deploying the component with the highest degree of isolation and the highest number of requests allowed per component (line 17). This algorithm assumes the optimal solution is the one that guarantees the maximum degree of isolation and the highest number of requests allowed to access the components and the whole cloud-hosted service. Clearly, the algorithm can be extended to work for the required degree of isolation by including the isolation value (i.e., isolation value 1, 2 or 3), as an input parameter both in the OptimalDep algorithm and in the metaheuristics to search for and extract components that correspond to the required degree of isolation.

Note that the algorithms described in this chapter are different from the autoscaling algorithms offered by IaaS providers like *Amazon* and existing optimisation models proposed for use by SaaS providers such as *Salesforce.com* (Aldhalaan and Menascé, 2015*b*). Saas providers may be able to monitor and estimate to a certain degree the performance and resources utilization of applications components integrated within applications running on VMs that they have rented out to SaaS customers. However, SaaS providers do not know the required degree of isolation of each application component (e.g., components that offer critical functionality), the number of available components to be deployed, and the number and capacities of resources required to support each component. In some cases, it may also be necessary to associate a particular user/request to certain components or group of components to guarantee the required degree of isolation. These

Table 1 An example of optimal Component Deployment

	GROUP 1		GROUP 2		GROUP 3	
	Item 1	Item 2	Item 1	Item 2	Item 1	Item 2
Initial State						
Isolation	1	2	2	3	2	1
No. of Req.	0	0	0	0	0	0
Item Resources: (CPU,MEM,DSK,BDW)	8,6,3,3	9,3,9,9	4,1,2,6	2,6,1,6	7,9,4 ,6	2,5,1,7
Service Demands: (CPU,MEM,DSK,BDW)	0.25,0.23, 0.22,0.20	0.25,0.23, 0.22,0.20	0.25,0.23, 0.22,0.20	0.25,0.23, 0.22,0.20	0.25,0.23, 0.22,0.20	0.25,0.23, 0.22,0.20
Request to increase workload from 0 to 3.7req/min						
No. of Req. (updated)	5.20	5.20	5.20	5.20	5.20	5.20
Current Solution						
Solution Format = (F/ I/ Q)						
Soln 1: 515.6/5/15.60	✓		✓		✓	
Soln 2: 415.6/4/15.60	✓		✓			✓
Soln 3: 615.6/6/15.60	✓			✓	✓	
Soln 4: 515.6/5/15.60	✓			✓		✓
Soln 5: 615.59/6/15.60		✓	✓		✓	
Soln 6: 515.59/5/15.60		✓	✓			✓
Soln 7: 715.59/7/15.60		✓		✓	✓	
Soln 8: 615.59/6/15.60		✓		✓		✓

details are only available to SaaS customers (e.g., a cloud deployment architect) since they own the components and are also responsible for deploying and managing the components to the cloud.

4.2.4 OptimalDep Algorithm Example

The following example shows the different solutions evaluated by *optimalDep* combined with the SA(Greedy) algorithm to find an optimal solution to the optimization problem. Every time there is a change in the workload, the optimalDep algorithm finds a new optimal solution for deploying components with the highest degree of isolation and the highest number of supported requests (see Table 1).

Let us assume that there are three groups of components (N=3) that can be designed to use (or integrate with) a cloud-hosted service and each component has a maximum number of requests that can be allowed to access it without having a negative impact on the degree of isolation between components of the cloud-hosted service. Each component is supported by four main resources: CPU, RAM, Disk capacity and bandwidth. The service demands for CPU=0.25, RAM=0.23, disk=0.22, bandwidth=0.2, while the maximum capacity of each of these resources is 20.

When a request arrives indicating a change in workload (i.e., in our case, this means an arrival rate between 0 to 3.7 req/min), an open Multiclass Queuing network is solved to find the average number of requests that can access the components. The ninth row shows the updated problem instance with the current number of requests (i.e., 5.2) that can access the components in each group. The updated problem instance is then solved with the metaheuristic and the state with the highest optimal function value is returned. Solution 1 (in row twelve) shows the optimal value of 515.6 for selecting a solution that deploys the first component from all the groups. This solution results in an optimal value of 515.6 (isolation value=500; and number of request=15.60). Note that no component can be selected for deployment and hence no changes can be effected on the cloud environment until the search is over and a better solution is found.

Up to this point, all the solutions have been evaluated and only the solution with the optimal value is returned as the optimal solution. In this example, the optimal solution with the highest fitness value is solution 7 with a utility value of 715.60. Note that this example assumes a fixed service demand for all components in each group. In an ideal situation, components would have different service demands. This would lead to different values for the number of requests, thus further opening up different options for the selection of an optimal solution.

4.2.5 Metaheuristic Algorithm for Solving the Model

The optimisation problem described in the previous section and then mapped to an MMKP is an NP-hard problem which has been known to have a feasible state space that grows in a combinatorial way (Yu et al., 2007) (Chipperfield et al., 1999). The number of feasible states for our optimal component deployment problem is given by the following expression:

$$\left\{\binom{l}{j}\right\}^{N} \tag{3}$$

Equation 2 above represents the number of ways for selecting one component (*j* items) from each a group (made of up l items) out of several (N) groups of components to integrate with or designed to use a cloud-hosted application when workload changes in a particular time interval. Thus in response to the workload changes, the number of ways of selecting one component (i.e., *j*=1) each from twenty groups (i.e., N=20) containing ten items in each group (i.e., *l*=10) will result in approximately 10.24 x 10^{12} states. Depending on the number of times and frequency with which the workload changes, the number of states could grow very large at a much faster rate.

Therefore, an efficient heuristic is needed to find an optimal solution to the optimisation problem, which must be solved by the decision support system and provided to the SaaS customer (or a cloud deployment architect) in almost real-time. Algorithm 4 shows a simple metaheuristic named *SA(Greedy)* based on simulated annealing that can be utilized with *OptimalDep* (see line 17 of Algorithm 3). The algorithms for *optimalDep* and *SA(Greedy)* are presented as Algorithm 3 and Algorithm 4, respectively. A high-level description of these algorithms is provided below:

Algorithm 3: *OptimalDep Algorithm*

```
1: optimalDep (workloadFile, mmkpFile)
2: optimalSoln ← null
3: Accept workload from SaaS users
4: Load workloadFile, mmkPfile; populate global variables
5: repeat
6:     /*Compute No. of req. using QN Model*/
7:     for $i \leftarrow 1, NoGroups$ do
8:         for $i \leftarrow 1$,GroupSize do
9:             Calculate Utilization
10:            Calculate No. of req.
11:            Calculate Total No. of req.
```

```
12:         Store fitValue, Isol, qLength of optimal soln.
13:       end for
14:     end for
15:     Update the mmkpFile with qLength
16:     /*Run Metaheuristic*/
17:     SA(Greedy)( )
18:     /*Display optimal solution for deployment*/
19: until no more workload
20: Return (optimalSoln, fitValue, Isol, qLength)
```

The ***SA(Greedy)*** algorithm combines simulation annealing and a greedy algorithm to find an optimal solution to our optimization problem which has been modelled as an MMKP. The algorithm loads the MMKP problem instance and then populates the global variables (i.e., arrays of varying dimensions that store the values of isolation, and the average number of requests, and component resource consumptions). A simple cooling schedule is used which is expressed as:

$$T_t = T_0 - \eta t \tag{4}$$

Algorithm 4: *SA(Greedy) Algorithm*

```
1: SA(Greedy) (mmkpFile, N)
2: Randomly generate $N$ solutions
3: Set initial temperature $T_0$ to st. dev. of all optimal values
4: Create greedySoln $a^1$ with optimal value $g(a^1)$
5: optimalSoln $\leftarrow$ $g(a^1)$
6: bestSoln $\leftarrow$ $g(a^1)$
7: for $i \leftarrow 1, N$ do
8:     Create neighbouring soln $a^2$ with optimal value $g(a^2)$
9:     Mutate the soln $a^2$ to improve it
10:    if $a^1 < a^2$ then
11:        bestSoln $\leftarrow a^2$
12:    else
13:        if random[0,1) < exp(-($g(a^2)$ - $g(a^1)$)/T) then
14:            $a^2 \leftarrow$ bestSoln
15:        end if
16:    end if
17:    $T_{i+1}$ = $T_0 - \eta t$ /*see Equation 4*/
18: end for
19: optimalSoln $\leftarrow$ bestSoln
20: Return (optimalSoln)
```

The above cooling schedule (equation 4) is linear and which means that T decreases every t iterations by an amount ηt (Karasakal and Köksalan, 2000). Since the introduction of this linear cooling schedule shown in Equation 4 by the authors in (Kirkpatrick et al., 1983), it has been widely used in several optimization models relying on simulated annealing (Zoraghi et al., 2012; Huang, 2003). In the above cooling schedule, the variable η is computed as follows:

$$\eta = \left(\frac{T_0}{max(t)}\right) \tag{5}$$

Our strategy for setting the initial temperature T_0 is to randomly generate a number of solutions equal to the size of the number of groups in the problem instance, before the simulated annealing algorithm runs, and then to set the initial temperature T_0 to the standard deviation of all the randomly generated optimal solutions (line 2-4). Another option could be to set T_0 to the standard deviation of a set of solutions from a heuristic whose initial solution was generated randomly. In line 4, a greedy solution is then created as an initial solution. The simulated annealing process improves the greedy solution and provides the optimal solution for deploying components to the cloud (line 5-19).

A simple dry run of the algorithm for the instance C(20,20,4) is as follows: 20 optimal solutions are randomly generated and then the standard deviation of all the solutions is computed. Assuming this value is 5.26, the T_0 is set to 5. At the first iteration, $g(a^2)$ = 151634.9773 and $g(a^1)$ = 151535.7984 and the current temperature then becomes 4.999995. At the next iteration, the current temperature is expected to reduce further (see equation 5). After five iterations, the algorithm constructs an initial/first solution with $g(a^1)$ = 151732.4362, a current/second random solution with $g(a^2)$ = 151733.9821 and with a current temperature of 4.999975. The solution a^2 will replace a^1 with probability, P =exp(-(1.5459)/4.999975)=0.7340, because $g(a^2) > g(a^1)$. In lines 13 to 15, a random number between 0 and 1 (i.e., rand = 0.0968) is generated, and since *rand < 0.7340*, a^2 replaces a^1 and the algorithm continues with a^2. Otherwise, the algorithm continues with a^1. At the next iteration, the temperature T is reduced which now becomes T_6 = 4.99997 (line 17). The iteration continues until N (i.e., the number of iterations set for the algorithm to run) is reached, and so the search converges with a high probability to the optimal solution. Another variant of the simulated annealing (i.e., ***SA(Random)***) is possible; that is, instead of constructing an initial greedy solution, a random solution is simply generated and then passed to the simulated annealing process to become the initial solution.

5 Evaluation

In this section, we present the evaluation of each component of the framework including the case study implementation, experimental design and procedure.

5.1 Case Study 1: Evaluating Varying Degrees of Multitenancy Isolation

In this section, we discuss the selection of the case study application (i.e., a cloud-hosted bug tracking system), its implementation to support multitenancy isolation and experimental design and procedure.

5.1.1 Implementing Multitenancy Isolation on Bugzilla

An empirical study conducted in previous work revealed some common software tools (and associated software processes) used in large-scale distributed enterprise global software development projects: JIRA, VersionOne, Hudson, Subversion and Bugzilla (see details in (Ochei, Bass and Petrovski, 2015*a*)). In this study, we use Bugzilla, a widely used cloud-hosted bug and issue tracking system.

To demonstrate the practicality of our approach, we modified Bugzilla to support multitenancy at the database level. We implemented our algorithm in a special extension we created and then "‘hooked"’ it into Bugzilla using the hook named *install_before_final_checks*. This particular hook allows the execution of custom code before the final checks are done in *checksetup.pl* (Bugzilla, 2016).

5.1.2 Experimental Procedure for Case Study 1

In order to evaluate the algorithm, we performed an experiment to evaluate the effect of isolation between tenants in a scenario where there is variation in the inter-arrival times of requests. This scenario is applicable in *distributed bug tracking* in which some bug trackers like Fossil and Veracity are either designed to use (or integrated with) distributed VC or CI systems, thus allowing bugs to be created automatically and inserted into the database at varying frequencies. We configured JMeter BeanShell sampler to send multiple requests to the bug database. The experimental procedure outlined (Ochei, Bass and Petrovski, 2015*b*) and (Ochei, Petrovski and Bass, 2015) was followed in this study.

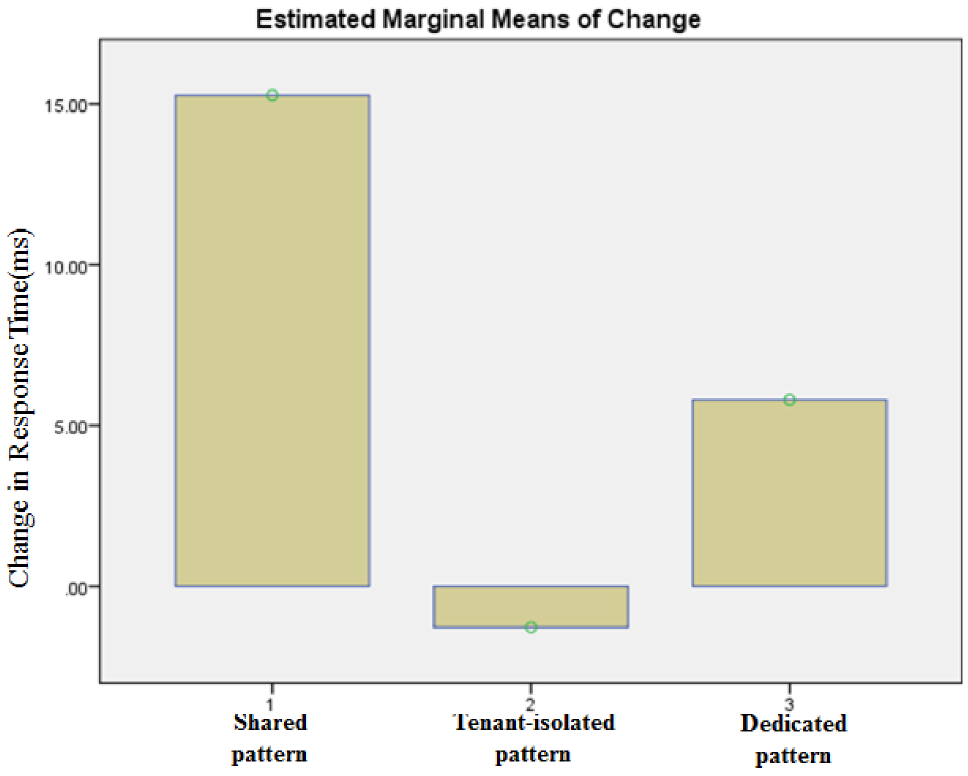

Figure 6. Changes in response time for each pattern relative to other patterns-2

5.2 Case Study 2: Optimizing the Deployment of Components of a Cloud-hosted Service

In this section, we discuss the selection of the case study application, its implementation to support multitenancy isolation and experiment design, setup and procedure.

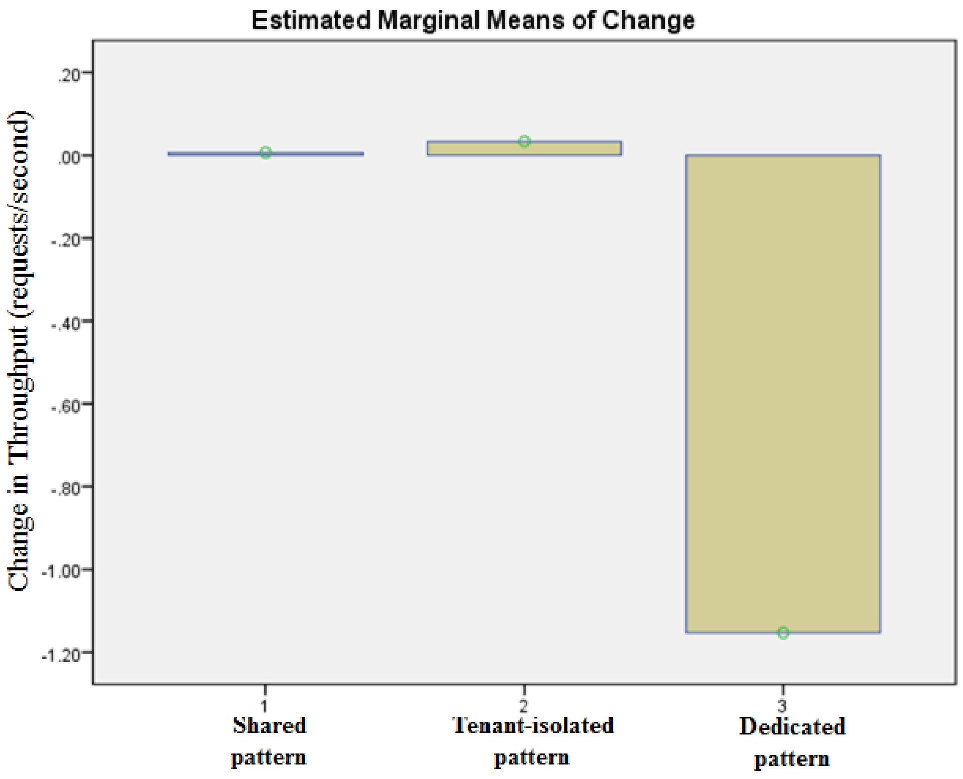

Figure 7. Changes in throughput for each pattern relative to other patterns-2

5.2.1 Dataset and Instance Generation

The dataset used for simulation experiments to illustrate the provision of optimal solutions to deploy components was based on a simulation testbed. There are two datasets used in this study: the MMKP instance file and the workload file.

(a) *MMKP Instance file:* Due to the unique nature of our problem, the multichoice multidimensional knapsack (MMKP) instances used in the experiments were randomly generated and not based on a publicly available dataset of MMKP instance. However, the instance was generated based on a standard approach widely used in literature (Parra-Hernandez and Dimopoulos, 2005; Cherfi and Hifi, 2010).

(b) *Workload file:* Workload file contains the values that are used to simulate the workload offered to the system. The key values it contains are the arrival rate of requests and the service demands of each resource supporting the components.

Several problem instances of various sizes and densities were randomly generated. After that, these instances were solved using each variant of the metaheuristic. Two categories of instance were generated and tailored on the instances widely cited in literature: (i) OR benchmark Library (Beasley, 1990) and other standard MMKP benchmarks (Cherfi and Hifi, 2010; Hifi et al., 2004), and (ii) the new irregular benchmarks used by Shojaei et al. (Eckart and Marco, n.d.). These benchmarks are usually used for single objective problems. This benchmark format was modified and extended to conform to a multiobjective case by associating each component with two different profit values: isolation values and the

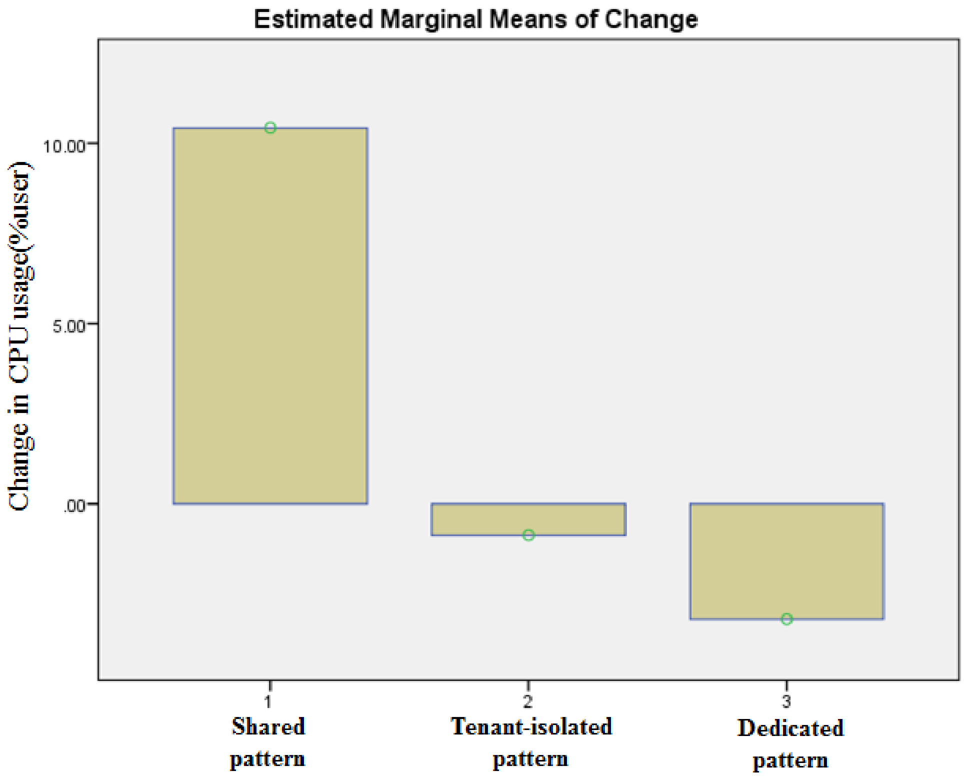

Figure 8. Changes in CPU for each pattern relative to other patterns-2

average number of requests (Zitzler and Thiele, 1999). Values of the dataset were generated as follows:

(1) Values for different degrees of isolation were randomly generated in the interval [0.05,0.25].

(2) Values for resource consumption were randomly generated in the interval [1,9].

(3) Values for service demand were randomly generated in the interval [0.05,0.25].

(4) Values workload were randomly generated following a Poisson distribution in the interval [1,5].

(5) Values for resource capacities generated were generated by setting it to half of the maximum possible resource consumption (Cherfi and Hifi, 2010).

5.2.2 Applicability of the Generated Instances to Real-life Cloud Deployment Scenario

The MMKP problem instances represent a repository of components configuration that can be used to deploy components designed to use (or integrate with) a cloud-hosted service. A component could be a database, database table, a message queue, VM or even Docker container. It is also important to note that although the weight values (i.e., the resource consumption of the components) generated in the MMKP instance may appear to be in

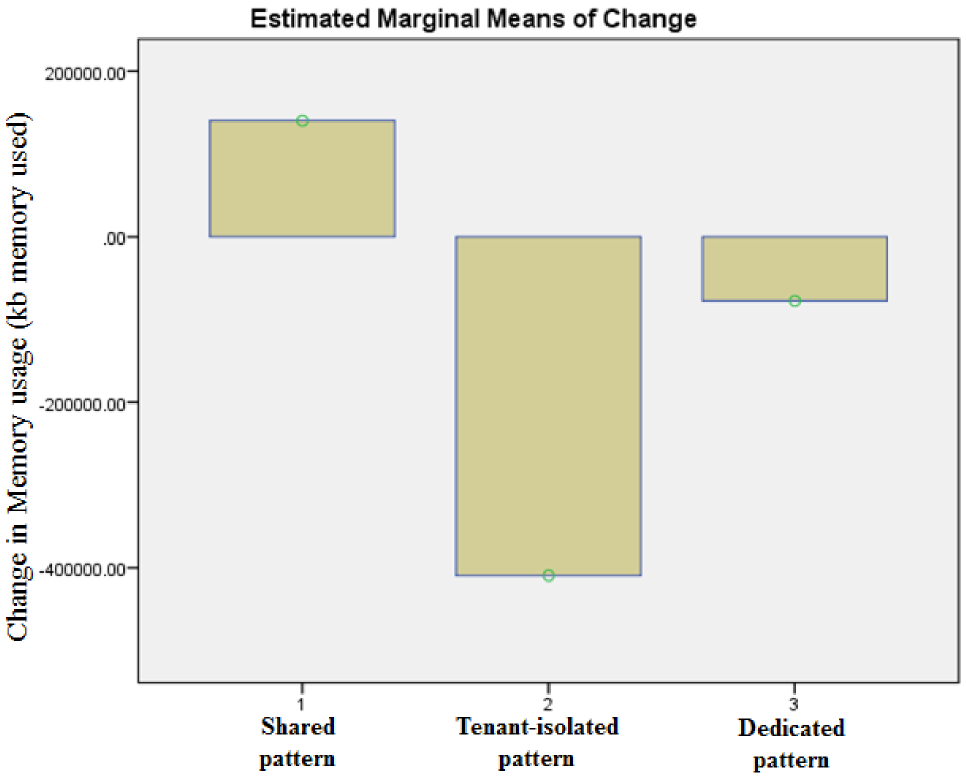

Figure 9. Changes in memory for each pattern relative to other patterns-2

the same interval, in reality these values could be normalised (or transformed) to represent different resources units of the components.

As an illustration, one of Amazon's EC2 instance types, named *"'compute optimized (c4.xlarge model)"'*, has the following specification: 4 vCPU, 8 GiB of memory, EBS-optimized only storage (which is similar to an IOPS provisioned on an Amazon Elastic Block store volume (EBS)) and 750 Mbps of dedicated EBS bandwidth (Amazon, 2016). An Amazon EBS can be created with Provisioned IOPS SSD(io1) volumes up to 16 TiB in size. So assuming the weights of a component on a generated MMKP instance is given as [4, 8, 8, 8], this specification could easily be transformed to the actual specification of the above named Amazon EC2 instance using this normalisation format: [CPU, RAM, DISK/2, BANDWIDTH/100]. This means that this particular component is supported with 4 virtual CPUs, 8GB of memory, 8 TB of disk space and 8 Mbps of bandwidth.

5.2.3 Experimental Procedure for Case Study 2

The problem/MMKP instances used for our experiments were generated as described in the previous section. The instance generating program and the algorithms were written using Java programming with Netbeans IDE 7.3.1. All experiments have been carried out on the same computation platform, which is a Windows 8.1 running on a SAMSUNG Laptop with an Intel(R) CORE(TM) i7-3630QM at 2.40GHZ, with 8GB memory and 1TB swap space on the hard disk. Table 2 shows the parameters used for the experiments. Each instance is

Table 2 Parameter values used in the experiments.

Open Multiclass QN Model	**Value**
λ (offered load)	[0,4]
Isolation Value	[1,2,3]
No. of Requests	[1,10]
Resource consumption	[1,10]
Service Demands	[0.15, 0.24]
SA(Greedy) Algorithm	
No of Iterations	N=1000000
No. of Runs	20
Temperature	T_0 = st. dev of N randomly generated solutions (N=no. of groups)
Cooling Schedule	$T_t = T_0 - \eta t$ (see equation 3 and 4)

tested with a workload associated with it. The exhaustive search algorithm was incapable of solving large instances. This was because of the low memory of the used machine. And so a small MMKP instance, C(4,5,4) was used for the evaluation and comparison of the algorithms.
Aim of the experiment: The aim of the experiment is to evaluate the performance (i.e., regarding obtained solution quality, robustness, scalability and computational effort) of the different variants of the metaheuristic when integrated into the model-based decision support system (i.e., optimalDep).

6 Results and Discussion

In this section, we will first present the results of the case study that empirically evaluates the varying degrees of isolation and thereafter the case study that provides optimal solutions for deploying components of a cloud-hosted service.

6.1 Results on Evaluating Varying Degrees of Isolation

We analyzed the experimental results based on the plots of estimated marginal means of change (EMMC) in combination with ANOVA (plus post hoc test) and paired sample test results from SPSS output. The purpose of the statistical test was to determine if tenants deployed using a particular pattern changed significantly, thus giving an indication as to whether or not the workload created by one tenant affected the other tenants. Further details can be seen in (Ochei, Bass and Petrovski, 2015*b*) (Ochei, Petrovski and Bass, 2015). *Due to space limitations, we only discuss the results and show the EMMC diagrams for response time, throughput, CPU and memory* (see Figure 6 - Figure 9).

(1) Response time, Throughput: The results showed that response time changed significantly, and so an architect should expect a low degree of isolation when a component is deployed using a shared component. The tenant-isolated component and dedicated component offered a higher degree of isolation. As expected, throughput changed significantly for dedicated component (compared to the other patterns). The implication for an architect is that when there is variation in the frequency with which bugs are submitted

to the database, the shared component cannot be used to improve performance.
(2) CPU and Memory: The paired sample test showed that there was a significant change in both CPU and Memory usage. Bug trackers are not known to consume much CPU and Memory. However, some operations such as compression of large bugs files and attachments could cause high consumption (Bugzilla, 2016). Therefore when there is variation in the frequency with which bugs are submitted to the bug database, then the dedicated component can be used to optimize the CPU and RAM consumption more efficiently than the other patterns.

6.2 Results on Optimal Solutions for Deployment

This section presents a comparison of solutions obtained from the optimalDep algorithms.

6.2.1 Comparison of Solutions obtained from optimalDep Algorithm with the Optimal Solution

The solutions obtained from the optimalDep algorithm (when running either with SA(Random) or SA(Greedy)) are compared with the optimal solutions obtained by running the OptimalDep algorithm with an exhaustive search for a small problem size. The quality of the optimal solutions was measured in terms of the percent deviation from the optimal solution. The instance used is C(4,5,4) because it was small enough to cope with the requirements of the machine. The workload (i.e., the arrival rate) for each component was randomly generated between 0.0 and 4.

The results are summarised in Table 3. Each row of the first column shows a different workload with an arrival rate ranging from 2.7-3.9. The second column shows the optimal function variables as (OP/IV/RV), which stands for the value of the optimal function, isolation value, and the number of allowed requests, for the optimal solution. The third and fourth columns show the optimal function variables as (OP/FEval), which stand for the value of the optimal function and the number of function evaluations to attain the optimal solution.

As shown in Table 4, all the four variants of the metaheuristic produced results that were the same as the optimal solution for all workloads. This means that the four variants of the metaheuristic attained a 100% success rate and 0% percent deviation. The similarity seen in the results may be due to the small size of the instance. This small size was chosen to cope with the machine used for the experiments which could not solve problem instances larger than C(4,5,4) due to limitations in its hardware requirements (i.e., CPU and RAM). Notice that the number of function evaluations required to produce the optimal solution for the greedy variations of the algorithm is 0. This is due in part to the small size of the MMKP instance, and the fact that some effort has already been put in to produce the greedy solution and so the optimal solution is attained very quickly with little or no computational effort in terms of the number of function evaluations.

6.3 Comparison of Solutions obtained from optimalDep algorithm with the Target Solution

As an optimal solution could not be obtained with large instances (e.g., C(500,20,4)), the results were compared to a target solution as proposed by (Talbi, 2009). In our case, the target solution represents a requirement defined by a decision maker on the quality of the solutions to obtain. This is expressed as:

$$\begin{aligned} TargetSoln &= Ivalue + Qvalue \\ &\text{where the } \textit{Ivalue} \text{ is given as:} \\ Ivalues &= ((n \times max(I) \times w1) \\ &\text{and the } \textit{Qvalue} \text{ is given as:} \\ Qvalue &= (0.05 \times (n \times max(Q) \times w2))) \end{aligned} \tag{6}$$

where *n* is the number of groups, *max(I)* is the maximum isolation value, *max(Q)* is the maximum possible number of requests (calculated based on the upper limit of the arrival rate) and *w1* assigned to *I* and *w2* is the weight assigned to the *Q*. This equation, when used to compute the target solution of C(4,5,4) with an arrival rate of 2.7 req/sec gives 1219.2, which is very close to the optimal solution shown in Table 4. The target solution for all instance sizes ranging from C(10,20,4) to C(1000,20,4) is shown in Table 6 and 7. The optimal values obtained for each instance were the same for all the variants of the metaheuristic. The rest of the experiment was conducted with an arrival rate of 3.9 requests per second.

It should be noted that the simulation ran for 1000000 function evaluations in order to be able to attain the best possible solution for the algorithm. Therefore, the success rate would be expected to be nearly 100%, with the corresponding performance rate since the optimal solution would have converged. Because of this, this study extends the evaluation to cover scenarios where there is: (i) limited resource or a need to optimise available resources while providing optimal solutions; and (ii) limited time to provide optimal solutions, for example, when the algorithm can run for only 1000 iterations.

6.3.1 Measuring the Quality of Solutions

The quality of the solutions was measured in terms of the percent deviation from the target solution (see equation 3.3). As shown in Table 5, the percent deviation for all the variants of the metaheuristic was the same. It was noticed that the percent deviation of solutions is lower when the number of components per group is high. For example, the percent deviation for C(500,5,4) is 3.5 when the number of components is 5 and the percent deviation of C(500,20,4) and then 1.49 when the number of components is increased to 20. This means that the quality of solutions is a function of the number of components per group. The more choices of a particular type of component there are, the better the chance of obtaining an optimal configuration. This is particularly important for large open-source projects that are either designed to use a large number of components within the cloud-hosted service or be integrated with several components residing in other locations.

6.3.2 Measuring the Robustness of the Solutions

Robustness refers to how sensitive the solutions are, against small deviations in the input data or other parameters; the lower the variability, the better the robustness(Talbi, 2009). The standard deviation was used as a measure of this variability. Table 6 and Table 7 show the standard deviation for several instances of varying sizes and densities in the variable, Opt/Std. For example, 0.27 is the value for standard deviation for C(1000,20,4). It was observed that the standard deviation for instances with size m=20 was more than that of m=5.

This seems to suggest that when there are more components in a group, the robustness of the solutions will be low when the *optimalDep* is run either with SA(Random) and SA(Greedy). This means that metaheuristic based on simulated annealing was less stable and robust for instances with large densities (i.e., $m > 5$), especially as the size of the instance increases (i.e., $n > 100$).

6.3.3 Measuring the Computational Effort

The estimated execution time required by each variant of the metaheuristic to reach the target solution for different instance sizes was also computed. The estimated execution time is presented in Table 8. Each row of the first column shows a different problem/instance size ranging from C(10,20,4) to C(1000, 20, 4). The second, fourth, and sixth columns show the mean execution times for obtaining a greedy solution, random solution and optimal value from a randomly generated solution, respectively. The third column, fifth, and seventh column show the standard deviation of the mean execution times for obtaining a greedy solution, random solution and optimal value, respectively. Columns eight, nine, ten, and eleven show the execution times for reaching the target solution for each of the variants of the metaheuristic.

As expected, Table 8 shows that the average execution times for producing the initial greedy solution is larger than that of the random solution. The execution time required to produce an initial greedy solution is 400 times in most of the cases over that of random solutions. However, because the average number of function evaluations required by the metaheuristic that starts with greedy solutions (i.e., SA(Greedy)) is far less than those that start with random solutions. Thus, the overall execution time of SA(Greedy) is still less than that of SA(Random). Therefore, the variants of the metaheuristic that start with the greedy solution used less computational effort regardless of the variant of the metaheuristic.

6.3.4 Summary of Results

The results of the study can be summarised as follows:
(i) Percent deviation for all variants was nearly the same. For large instances, percent deviation of variants based on greedy solutions was smaller and more stable.
(ii) Standard deviation of solutions were higher (which implies a low robustness) for instances with large densities (i.e., the number of components per group), especially when the size of the instance is larger than 100.
(iii) Metaheuristics that started with greedy solutions attained a 100% success rate much faster and used less execution time than those that started with random.
(iv) Small instance size had no significant effect on robustness and quality of solutions. However, as with large instance sizes, the variants of the metaheuristics that start with a greedy solution required fewer function evaluations to reach the target solution.
(v) Instances with more components per group had less percent deviation, hence a higher chance of producing better quality solutions

The implication of the results are as follows: The benefit of our model-based decision support system is in monitoring, evaluating, adjusting and deploying components of cloud-hosted service (especially for large-scale projects) for guaranteeing multitenancy isolation when there are workload changes. For large-scale cloud-hosted services, running the model-based decision support system with a metaheuristic whose initial solution starts with a greedy solution (compared to random solutions) can significantly boost the quality and robustness of the solutions produced. The results show that solutions from the metaheuristic

Table 3 Comparing the SA(Greedy) with optimal solution

Workload(λ)	**Optimal**	**SA(Rand)**	**SA(Greedy)**
2.7	1220.8/12/20.8	1220.8/41	1220.8/0
2.9	1225.69/12/25.69	1225.69/51	1225.69/0
3.1	1232.38/12/32.38	1232.38/60	1232.38/0
3.3	1242.14/12/42.14	1242.14/38	1242.14/0
3.5	1257.99/12/57.99	1257.99/41	1257.99
3.7	1289.77/12/89.77	1289.77/32	1289.77/0
3.9	1415.09/12/215.09	1415.09/18	1415.09/0

Table 4 Average performance on different instance sizes (m=5, m=20)

Instance Size	**SA(rn)**	**SA(gr)**	**Instance Size**	**SA(rn)**	**SA(gr)**
C(10,5,4)	7.64	7.64	C(10,20,4)	1.38	1.38
C(20,5,4)	0.7	0.7	C(20,20,4)	1.98	1.98
C(30,5,4)	1.44	1.44	C(30,20,4)	0.09	0.09
C(40,5,4)	1.34	1.34	C(40,20,4)	1.39	1.39
C(50,5,4)	3.38	3.38	C(50,20,4)	1.4	1.4
C(60,5,4)	1.99	1.99	C(60,20,4)	2.04	2.04
C(70,5,4)	0.96	0.96	C(70,20,4)	1.03	1.03
C(80,5,4)	4.08	4.08	C(80,20,4)	1.36	1.36
C(90,5,4)	1.62	1.62	C(90,20,4)	2.01	2.01
C(100,5,4)	5.03	5.03	C(100,20,4)	2.11	2.11
C(200,5,4)	3.79	3.79	C(200,20,4)	1.48	1.48
C(300,5,4)	5.22	5.22	C(300,20,4)	1.13	1.13
C(400,5,4)	3.7	3.7	C(400,20,4)	1.28	1.28
C(500,5,4)	3.53	3.53	C(500,20,4)	1.48	1.48
C(600,5,4)	3.36	3.36	C(600,20,4)	1.25	1.25
C(700,5,4)	3.78	3.78	C(700,20,4)	1.24	1.24
C(800,5,4)	3.84	3.84	C(800,20,4)	1.43	1.43
C(900,5,4)	3.44	3.44	C(900,20,4)	1.11	1.11
C(1000,5,4)	4.28	4.28	C(1000,20,4)	1.16	1.16
AVG	3.32	3.32	AVG	1.39	1.39
STD	1.66	1.66	STD	0.45	0.45

based on simulated annealing were more stable and robust when applied to small instances. Metaheuristics that started with greedy solutions were more scalable and require fewer function evaluations to reach the target solution when compared to metaheuristics that start with random solutions.

7 Related Work

The varying degrees of multitenancy isolation based on three multitenancy patterns and different aspects of isolation are described in (Fehling et al., 2014). In (Guo et al., 2007) different isolation capabilities were evaluated for authentication, information protection,

Table 5 Optimal values and standard deviation of different instances(m=5)

Instance	Target Solution	SA(Rand)	SA(Greedy)
C(10,5,4)	3048	2815.09/0.0	2815.09/0.0
C(20,5,4)	6096	6053.26/1.5E-4	6053.26/1.50
C(30,5,4)	9144	9012.30/0.0	9012.30/0.0
C(40,5,4)	12192	12028.67/0.0	12028.67/0.0
C(50,5,4)	15240	14725.40/0.0	14725.40/0.0
C(60,5,4)	18288	17923.88/0.0	17923.88/0.0
C(70,5,4)	21336	21130.88/7.3E-4	21130.89/7.3E-4
C(80,5,4)	24384	23389.81/0.00	23389.81/0.00
C(90,5,4)	27432	26987.22/0.0	26987.22/3.5E-4
C(100,5,4)	30480	28945.60/0.00	28945.60/0.00
C(200,5,4)	60960	58647.47/0.01	58647.47/0.01
C(300,5,4)	91440	86662.77/0.02	86662.77/0.02
C(400,5,4)	121920	117405.15/0.04	117405.14/0.05
C(500,5,4)	152400	147023.73/0.09	147023.77/0.07
C(600,5,4)	182880	176734.98/0.10	176734.94/0.10
C(700,5,4)	213360	205301.49/0.12	205301.44/0.14
C(800,5,4)	243840	234472.51/0.16	234472.44/0.16
C(900,5,4)	27432	264882.74/0.20	264882.83/0.18
C(1000,5,4)	304800	291762.78/0.27	291762.85/0.17

Table 6 Optimal values and standard deviation of different instances(m=20)

Instance	Target Soln	SA(Rand)	SA(Greedy)
C(10,20,4)	3048	3090.18/0.0	3090.18/0.0
C(20,20,4)	6096	6216.57/2.1E-4	6216.57/2.11
C(30,20,4)	9144	9151.83/0.00	9151.83/0.00
C(40,20,4)	12192	12361.50/0.00	12361.50/0.00
C(50,20,4)	15240	15452.76/0.01	15452.76/0.01
C(60,20,4)	18288	18661.62/0.01	18661.62/0.01
C(70,20,4)	21336	21555.85/0.03	21555.85/0.03
C(80,20,4)	24384	24715.80/0.01	24715.77/0.06
C(90,20,4)	27432	27982.69/0.03	27982.68/0.03
C(100,20,4)	30480	31124.28/0.03	31124.28/0.03
C(200,20,4)	60960	61861.11/0.17	61861.10/0.16
C(300,20,4)	91440	92473.23/0.28	92473.13/0.40
C(400,20,4)	121920	123486.36/0.52	123486.44/0.42
C(500,20,4)	152400	154662.53/0.70	154662.7/0.60
C(600,20,4)	182880	185158.51/0.67	185158.34/0.71
C(700,20,4)	213360	216010.27/1.26	216010.57/0.95
C(800,20,4)	243840	247325.61/1.28	247325.92/1.18
C(900,20,4)	27432	277354.00/1.46	277353.75/1.86
C(1000,20,4)	304800	308344.05/2.00	308344.64/1.67

Table 7. Computational Effort (in sec) of different instances

Instance Size	AVG(gr)	STD(gr)	AVG(rn)	STD(rn)	AVG(fe)	STD(fe)	SA(rn)	SA(gr)
C(10,20,4)	30.78	3.5	0.23	0.67	0.46	0.42	44.85	30.78
C(20,20,4)	66.91	4.05	0.55	0.14	0.54	1.59	110.71	117.13
C(30,20,4)	117.81	2.48	0.78	0.15	0.75	0.16	378.78	2082.81
C(40,20,4)	170.51	3.45	0.11	1.06	0.93	0.08	423.26	170.51
C(50,20,4)	223.66	4.25	0	0.02	1.05	0.05	481.95	370.66
C(60,20,4)	277.6	12.55	0.28	0.07	1.67	0.26	918.78	277.6
C(70,20,4)	328.99	6.55	0.65	0.11	1.89	0.08	926.75	824.17
C(80,20,4)	386.56	5.96	0.78	1.95	1.62	0.21	1523.58	506.44
C(90,20,4)	435.65	8.76	0.86	0.11	2.14	0.08	2095.92	660.35
C(100,20,4)	508.17	30.24	1.01	0.03	2.94	0.17	2996.87	508.17
C(200,20,4)	1007.66	21.9	2.24	0.08	10.63	0.24	26035.11	9681.74
C(300,20,4)	1536.93	71.97	3.2	1.95	24.28	0.22	99308.4	26933.81
C(400,20,4)	2163.23	69.04	4.96	0.06	29.7	0.2	147554.56	25566.83
C(500,20,4)	2638.34	34.8	5.98	0.08	23.99	0.97	147640.44	31834.17
C(600,20,4)	3246.27	60.23	7.03	0.07	36.36	0.23	284560.39	38842.71
C(700,20,4)	3799.79	77.58	8.34	0.22	19.04	0.16	178127.54	34796.91
C(800,20,4)	4417.39	114.31	10.17	0.11	29.82	0.18	281749.53	40141.75
C(900,20,4)	5004.77	112.35	11.01	0.09	29.54	3.06	364918.63	89636.87
C(1000,20,4)	5592.63	87.27	12.06	0.16	30.78	3.09	405353.88	74478.27

faults, administration etc. Other work related to multitenancy isolation can be seen in (Krebs et al., 2013) (Krebs and Loesch, 2014).

None of the above work considers implementing multitenancy in a way that guarantees the required degree of isolation between tenants. In our previous work (Ochei, Bass and Petrovski, 2015*b*) and (Ochei, Petrovski and Bass, 2015), we described an architecture for implementing multitenancy isolation as well as an algorithmic framework for evaluating empirically the required degree of isolation between tenants accessing a cloud-hosted GSD tool. This paper extends our previous work (Ochei, Bass and Petrovski, 2015*b*) and (Ochei, Petrovski and Bass, 2015) by describing the algorithm in more detail and also presents an additional algorithm that can be used to determine the isolation level of an application component or functionality in almost real-time. In addition, we also present supporting algorithms for providing optimal solutions for deploying components of a cloud-hosted service.

Similar to our proposed approach, most cloud offerings (Amazon's Auto Scaling and EC2 (Amazon, 2016), and Microsoft Azure's Web Role (Microsoft, 2016)) also implement techniques that are able to intercept a user request, inspect it, and then decide what level of isolation is required. This is typically what production systems do across the overall application logic, for example, when providing subscriptions with different levels of isolation at different price tiers. However, while carrying out these provisioning and decommissioning operations, most cloud providers do not guarantee the availability and multitenancy isolation of specific components/individual IT resources (for example, a particular virtual server or disk storage), but only for the offering as a whole (for example, starting new virtual servers). Our algorithm can address this problem by initially tagging each component and thereafter identifying which isolation level is suitable for deploying a component based on the metadata of existing components. This will allow the component and the application to run efficiently and also help in optimizing the deployment of the cloud application.

Research work on optimal deployment and allocation of cloud resources on the cloud are quite significant (Yusoh and Tang, 2012; Shaikh and Patil, 2014; Westermann and Momm, 2010; Candeia et al., 2015; Abbott and Fisher, 2009). Most of this research focuses on minimising the cost of using the cloud infrastructure resources (Yusoh and Tang, 2012; Westermann and Momm, 2010). Previous work does not use metaheuristic to provide optimal solutions for deploying components of a cloud-hosted service in a way that guarantees the required degree of multitenancy isolation. Most of the research concerning optimization of cloud resources does not use heuristic at al, although a few use simple heuristics. For example, the authors in (Aldhalaan and Menascé, 2015*a*,*b*) used a heuristic based on hill climbing for minimising the cost of a SaaS cloud providers with response time SLAs constraints. Our work, unlike others, focuses on providing an optimal solution for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation.

8 Recommendations

In Table 8, we provide recommended patterns for achieving multitenancy based on the paired t-test and the plots of the EMMC. This will also help in optimizing the performance and resource consumption of tenants. Table 9 provides different options for applying the

Table 8 Recommended Patterns for achieving Multitenancy Isolation

Isolation Aspects	**Factors**	**Shared**	**Tenant -isolated**	**Dedicated**
Performance and Security	Response time		✓	✓
	Error%	✓	✓	
	Throughput		✓	✓
Resource Utilization	CPU		✓	✓
	Memory	✓		✓
	Disk I/O	✓	✓	✓
	System Load	×	-	-

Table 9 Implementing Multitenancy Isolation in Different Deployment Conditions with regard to Application Stack Level

Multitenancy Patterns	Application layer	Platform layer	IaaS layer
Shared Component	Multiple tenants share a single instance of bug database. Degree of isolation would be low	Tenants can use the bug database installed on a shared database platform	Set up bug database multiple times on a shared server
Tenant-Isolated Component	Tenants can share the bug database, and introduce a tenant-id field to tables to differentiate different tenants	Bug database are deployed on *virtual spaces* specially created on the database platform; virtual spaces isolates DB against DB of other tenants	Set up bug database schemas or databases for each tenant using different hypervisors
Dedicated Component	Associate each tenant with one bug database instance or a certain number of instances of the bug database	Install multiple bug databases on a hypervisor for different tenants. Instantiate same implementation for shared component exclusively for one tenant	Each tenant uses the bug database installed on its own instance of virtual hardware/servers they use; the number of tenants served may be limited

COMITRE approach to implement multitenancy isolation in different layers of the cloud stack for each multitenancy pattern.

The COMITRE approach we have put forward is to be implemented dynamically at runtime because we are dealing with components in most cases rather than the whole virtual server or application hosted on the server. This makes it easy for the algorithm to collect information for adjusting the behaviour of the component and/or the application (e..g, provisioning/decommissioning specific components hosted on virtual servers, increasing/decreasing the amount of storage, enabling/disabling networking connectivity etc.) in line with the required degree of isolation.

With respect to providing optimal solutions for deploying components of a cloud-hosted service, we make the following recommendations:
(1) There is a greater chance of obtaining better quality solutions when there are more components to choose from in the component repository.
(2) The result seems to suggest that starting a metaheuristic with an initial set of solutions (e.g., greedy solution) can significantly boost the optimal solutions obtained in order to guarantee multitenancy isolation.
(3) When there is limitation in terms of time and available resources, then a variant of metaheuristic based on simulated annealing would produce more stable and robust solutions.
(4) The variant of the metaheuristic that starts with a greedy solution would be more suitable for handling large projects that may have a significant number of interdependent components.

9 Limitations of the Study

This study focused on demonstrating the applicability of the algorithm for achieving the required degree of isolation between tenants based on the three multitenancy patterns. Due to the small size of the cloud infrastructure used for the experiments (i.e., Ubuntu Enterprise Cloud), we simulated the scenario with a large instant request to compensate for the large volume of requests that would have been required to create a significant effect on the tenants.

The algorithm requires an initial contribution from the software architect in the sense that the component has to be tagged to differentiate the varying degrees of isolation. This is at least necessary to populate the component repository and generate initial metadata for the algorithm. Subsequently, the tagging of each component is done dynamically by relying on the metadata of existing components.

With respect to the optimization model, it is assumed that the resources supporting each component are enough to handle incoming requests. If this condition cannot be guaranteed, we recommend using an elastic queue to control incoming requests. Another option could be to implement some form of admission control mechanism, for example, limiting the number of requests that are handled concurrently by each component, to avoid *overloads* or any degradation in the component's performance.

This research assumed a small-medium size search space (i.e., in the order of 10^{12} states). Although this is not the kind of magnitude expected when solving optimization problems (e.g., cost-minimization of resources) in large data centres, this research chose simulated annealing because they are well suited for problems with a small-to-medium size search space, and also make our approach scalable under the experimental conditions (Rothlauf, 2011). It would be interesting to challenge our approach with other heuristics such as constraints programming, genetic algorithm and estimation distribution algorithm.

The findings of this study are limited to cloud-hosted services developed and deployed using a multitenant architecture, and so should not be generalized to components developed for all types of commercial applications that may also be deployed to the cloud. Although the three multitenancy patterns were implemented by modifying Bugzilla deployed to cloud, the same approach (i.e., COMITRE) can also be used to implement multitenancy isolation in other cloud-hosted software tools (e.g., Hudson, Subversion).

The study assumed a private cloud that receives a small-to-medium size number of requests. Therefore, it is important to carefully vary the number of requests that would cope with the size of the cloud infrastructure used especially when implementing the approach and associated algorithms in a production environment. The framework and the associated algorithms that we have presented in this work apply to cloud-hosted applications at the application level, and so are implemented almost at runtime.

10 Conclusion and Future Work

In this paper, we have described a framework for achieving the required degree of multitenancy isolation for a cloud-hosted service. The two constituents of the framework have been described: (i) applying COMITRE to implement varying degrees of multitenancy isolation using a bug tracking system (i.e., Bugzilla) and (ii) an optimization model for providing optimal solutions for deploying components of a cloud-hosted service to guarantee multitenancy isolation.

This study contributes to literature on achieving the required degree of multitenancy isolation for cloud-hosted services, and in particular, cloud-hosted Global Software Development tools. The study demonstrated the applicability of the framework in a case study involving multiple tenants submitting bugs at varying frequency to the bug database. This study also presented an optimization model together with a metaheuristic solution for providing optimal solutions for deploying components of a cloud-hosted service while guaranteeing multitenancy isolation.

We plan to apply the algorithm to different cloud deployment scenarios such as when locking is enabled in both SQL and NoSQL databases used in distributed bug tracking environments. In the future, we plan to develop a simulation model (i.e., a simulator) or a rule-based model which is based on computer programs that emulate the different aspects of a system as well as their static structure. The simulator will allow us to capture and analyse different architectural components of a cloud-hosted service in situations where requirements are often difficult and complex to interpret and could change suddenly due to workload interference.

Acknowledgement

This study was supported by the Tertiary Education Trust Fund, Nigeria and Robert Gordon University, Aberdeen, United Kingdom.

References

Abbott, M. L. and Fisher, M. T. (2009), *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*, Pearson Education.

Akbar, M. M., Rahman, M. S., Kaykobad, M., Manning, E. G. and Shoja, G. C. (2006), 'Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls', *Computers & operations research* **33**(5), 1259–1273.

Aldhalaan, A. and Menascé, D. A. (2015*a*), Near-optimal allocation of vms form iaas providers by saas providers, Technical report, George Mason University.

Aldhalaan, A. and Menascé, D. A. (2015*b*), 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.

Amazon (2016), 'Amazon ec2 instance types'. [Online: accessed in September 12, 2016 from https://aws.amazon.com/ec2/instance-types/].

Bahill, A. T. and Madni, A. M. (2017), Discovering system requirements, *in* 'Tradeoff Decisions in System Design', Springer, pp. 373–457.

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

Beasley, J. E. (1990), 'Or-library- distributing test problems by electronic mail', *Journal of the operational research society* **41**(11), 1069–1072.

Bugzilla (2016), 'The bugzilla guide'. [Online: accessed in November, 2015 from http://www.bugzilla.org/docs/.

Candeia, D., Santos, R. A. and Lopes, R. (2015), 'Business-driven long-term capacity planning for saas applications', *IEEE Transactions on Cloud Computing* **3**(3), 290–303.

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

Cherfi, N. and Hifi, M. (2010), 'A column generation method for the multiple-choice multi-dimensional knapsack problem', *Computational Optimization and Applications* **46**(1), 51–73.

Chipperfield, A. J., Whidborne, J. F. and Fleming, P. J. (1999), Evolutionary algorithms and simulated annealing for mcdm, *in* 'Multicriteria Decision Making', Springer, pp. 501–532.

Eckart, Z. and Marco, L. (n.d.), 'Test problems and test data for multiobjective optimizers'. **URL:** *http://www.tik.ee.ethz.ch/sop/.../testProblemSuite/*

Fehling, C., Leymann, F., Retter, R., Schupeck, W. and Arbitter, P. (2014), *Cloud Computing Patterns*, Springer.

Garg, S. K., Versteeg, S. and Buyya, R. (2012), 'A framework for ranking of cloud computing services', *Future Generation Computer Systems* .

Grady, J. O. (1995), *System engineering planning and enterprise identity*, Vol. 7, CRC Press.

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

Hifi, M., Michrafy, M. and Sbihi, A. (2004), 'Heuristic algorithms for the multiple-choice multidimensional knapsack problem', *Journal of the Operational Research Society* **55**(12), 1323–1332.

Huang, X. (2003), 'A polynomial-time algorithm for solving np-hard problems in practice', *ACM SIGACT News* **34**(1), 101–108.

Karasakal, E. K. and Köksalan, M. (2000), 'A simulated annealing approach to bicriteria scheduling problems on a single machine', *Journal of Heuristics* **6**(3), 311–327.

Kellerer, H., Pferschy, U. and Pisinger, D. (2004), *Introduction to NP-Completeness of knapsack problems*, Springer.

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P. et al. (1983), 'Optimization by simulated annealing', *science* **220**(4598), 671–680.

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

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

Legriel, J., Le Guernic, C., Cotton, S. and Maler, O. (2010), Approximating the pareto front of multi-criteria optimization problems, *in* 'Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 69–83.

Martello, S. and Toth, P. (1990), *Knapsack problems: algorithms and computer implementations*, John Wiley & Sons, Inc.

Martens, A., Ardagna, D., Koziolek, H., Mirandola, R. and Reussner, R. (2010), A hybrid approach for multi-attribute qos optimisation in component based software systems, *in* 'Research into Practice–Reality and Gaps', Springer, pp. 84–101.

Menasce, D., Almeida, V. and Lawrence, D. (2004), *Performance by design: capacity planning by example*, Prentice Hall.

Microsoft (2016), 'Introducing microsoft azure'. [Online: accessed in September 13, 2016 from https://azure.microsoft.com/].

Moser, M. and O'Brien, T. (2016), 'The hudson book'. Online: accessed in November, 2015 from http://www.eclipse.org/hudson/the-hudson-book/book-hudson.pdf.

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

Ochei, L. C., Bass, J. and Petrovski, A. (2015*b*), 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.

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

Ochei, L. C., Petrovski, A. and 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)'.

Parra-Hernandez, R. and Dimopoulos, N. J. (2005), 'A new heuristic for solving the multichoice multidimensional knapsack problem', *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* **35**(5), 708–717.

Rothlauf, F. (2011), *Design of modern heuristics: principles and application*, Springer Science & Business Media.

Shaikh, F. and Patil, D. (2014), Multi-tenant e-commerce based on saas model to minimize it cost, *in* 'Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on', IEEE, pp. 1–4.

Talbi, E.-G. (2009), *Metaheuristics: from design to implementation*, Vol. 74, John Wiley & Sons.

Vlissides, J., Helm, R., Johnson, R. and Gamma, E. (1995), 'Design patterns: Elements of reusable object-oriented software', *Addison-Wesley* **49**, 120.

Westermann, D. and Momm, C. (2010), Using software performance curves for dependable and cost-efficient service hosting, *in* 'Proceedings of the 2nd International Workshop on the Quality of Service-Oriented Software Systems', ACM, p. 3.

Yu, T., Zhang, Y. and Lin, K.-J. (2007), 'Efficient algorithms for web services selection with end-to-end qos constraints', *ACM Transactions on the Web (TWEB)* **1**(1), 6.

Yusoh, Z. I. M. and Tang, M. (2012), Composite saas placement and resource optimization in cloud computing using evolutionary algorithms, *in* 'Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on', IEEE, pp. 590–597.

Zitzler, E. and Thiele, L. (1999), 'Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach', *IEEE transactions on Evolutionary Computation* **3**(4), 257–271.

Zoraghi, N., Najafi, A. A. and Akhavan Niaki, S. T. (2012), 'An integrated model of project scheduling and material ordering: a hybrid simulated annealing and genetic algorithm', *Journal of Optimization in Industrial Engineering* **5**(10), 19–27.