A Decision-based Model for Optimizing the Deployment of Cloud-hosted Application Components to Guarantee Multitenancy Isolation

Laud Ochei, Andrei Petrovski
Robert Gordon University, United Kingdom

Abstract

Tenants associated with a cloud-hosted application share components and resources in order to reduce resource consumption and running cost. However, if one of the tenants experiences a high workload, the performance and access privileges of other tenants may be affected, especially if the application does not scale-up when the workload of one of the tenents increases suddenly. This problem may be more serious when some components have a higher or varying degrees of isolation among them. This paper presents a decision-based model composed of: (i) optimization model, (ii) open multiclass queuing network (QN) model, and (iii) metaheuristic algorithm to provide optimal solutions for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation. Our experiments show that the optimal solutions obtained from our model had low variability and percent deviation when compared to the optimal solutions. This paper additionally provides areas of application of our optimization model as well as challenges and recommendations for deploying components associated with varying degrees of isolation.

1. Introduction

Architecting the deployment of components of a multitenant cloud-hosted service demands consideration concerning the type of components to deploy, the number of components to share and the underlying cloud resources that will support the execution of the components. [1] This is because there are different or varying degrees of multitenancy isolation that can be implemeted. For instance, the degree of isolation for components that offer a critical functionality would be higher than components that require minor re-configuration before deployment [2].

A low degree of isolation promotes the sharing of components and resources between tenants leading to reduced resource consumption and running cost but there are performance and security challenges if one of the components experiences a high workload. A high degree of isolation guarantees low performance and security interference but with the challenges of high resource consumption and running cost since tenants do not share resources [2]. Therefore the challenge for an architect is to resolve the conflicting trade-off between a high degree of isolation (with the challenge of high resource consumption and running cost) versus a low degree of isolation (with the challenge of performance interference).

Motivated by this problem, this paper provides a model for providing the optimal solutions for deploying components of a cloud-hosted application in a way that will guarantee multitenancy isolation. The approach involves building an optimization model, mapping it to a Multichoice Multidimensional Knapsack Problem(MMKP) and after that solving it utilizing a metaheuristic. We assess our approach by contrasting the optimal solutions obtained and the solutions form an exhaustive search of the entire solutions space of a small problem instance. This paper addresses the following research question: "***How can we optimize the deployment of components of a cloud-hosted application to guarantee multitenancy isolation***". The required degree of isolation between tenants can be quaranteed while at the time manage the underlying resources efficiently by optimizing the deployment of compoennts of a cloud-hosted application.

This article is an extension of the previous work by Ochei et al. [20]. The main contributions of this paper are:

1. A decision-based model composed of: (i) a mathematical optimization model to provide optimal solutions for deploying components of a cloud-hosted application to guarantee multitenancy isolation, (ii) mathematical equations based on an open multiclass queuing network model for calculating the both the average number of requests accessing each resource supporting a component and the whole component, (iii) architecture and algorithm for solving the decision-based model (DbM).

2. Metaheuristic solutions in different variants based on simulated annealing for solving the optimization model.

3. Recommendations and best-practice guidelines for deploying components of a cloud-hosted application for guaranteeing the required degree of multitenancy isolation.

4. Application areas where our optimization model can be applied to the deployment components of a

cloud-hosted service for guaranteeing the required degree of multitenancy isolation.

The rest of the paper is organized as follows - Section II discusses the challenge of providing near-optimal solutions for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation. In Section III, we present the optimization model. The next section presents the metaheuristic solution. Section V discusses the open multiclass queuing model. Section VI is the evaluation and experimental setup. Section VII is the discussion of results, while Section VIII discusses the application areas for the model. Section IX concludes the paper with future work.

2. Optimizing the Deployment of Components of Cloud-hosted Application with Guarantee for Multitenancy Isolation

This section discusses multitenancy isolation, the conflicting trades-offs in achieving optimal deployment based on the required degree of multitenancy isolation and related work on optimal allocation of cloud resources.

2.1. Multitenancy Isolation and Trade-offs for Achieving Varying Degrees of Isolation

Multitenancy is an important cloud computing property where a single instance of an application is provided to multiple tenants [21] [22], and so would have to be isolated against each other whenever there are workload changes. Just as multiple tenants can be isolated, multiple components being accessed by a tenant can also be isolated.

We define "***Multitenancy isolation***" as a way of ensuring that the required performance, stored data volume and access privileges of one component does not affect other components of a cloud-hosted application being accessed by tenants [3].

When a component of a cloud-hosted application receives a high workload and there is little or no possibility of a significant influence on other tenants, we say that there is a high degree of isolation and vice versa. The varying degrees of multitenancy isolation can be captured in three main cloud deployment patterns:

(i) dedicated component, where components cannot be shared, although a component can be associated with either one tenant/resource or group of tenants/resources;

(ii) tenant-isolated component, where components can be shared by a tenant or resource instance and their isolation is guaranteed; and

(iii) shared component, where components can be shared with a tenant or resource instance and are unaware of other components [1].

Assuming that there is a requirement for a high degree of isolation between components, then components have to be duplicated for each tenant which leads to high resource consumption and running cost. A low degree of isolation may also be required, in which case, it might reduce resource consumption, and running cost, but there is a possibility of interference when workload changes and the application does not scale well [3] [1]. Therefore, the challenge is how to determine near-optimal solutions that address these trade-offs in the presence of conflicting alternatives.

2.2. Related Work on Optimal Deployment and allocation of Cloud Resources

There are several research works on optimal allocation resources on the cloud. However, there is little or no work that focuses on providing optimal solutions for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation. The authors in [4] and [5] used a multitenant SaaS model to minimize the cost of cloud infrastructure. In [6], the authors focused on minimizing the resource consumption for SaaS providers and improving the execution time. In this work, an evolutionary algorithm, rather than heuristics were used. The authors in [7] developed a heuristic for capacity planning based on a utility model for the SaaS. This utility model mainly focused on the business aspects related to offering a SaaS application with the aim of increasing the profit.

In [8], the authors described how the optimal configuration of a virtual server can be determined, for example, the amount of memory to host an application through a set of tests. Fehling et al [9], considered how to evaluate optimal distribution of application components among virtual servers. A closely related work to ours, is that of Aldhalaan and Menasce [10], where the authors used a heuristic search technique based on hill climbing to minimize the SaaS cloud provider's cost of using VMs from an IaaS with response time SLAs constraints.

The above research works are mostly focused on minimizing the cost of using the cloud infrastructure resources. These works do not use metaheuristics to provide optimal solutions in a way that guarantees the required degree of multitenancy isolation. In addition, optimization models in previous works considered a single objective, where for example, the authors in [10] were minimizing the cost of using VMs. In our model, we consider a bi-objective case (i.e., maximizing the required degree of multitenancy isolation and the number of requests allowed to

access a component) and then use a modern metaheuristic based on simulated annealing to solve the model.

3. Problem Formalization and Notation

In this section, we will formalize the problem and then describe how it is mapped to a multichoice multidimensional knapsack problem (MMKP).

3.1. Description of the Problem

Let us assume that there are multiple components of the same tenant on the same underlying cloud infrastructure. A tenant in this context may represent a team or department of a software development company, whose responsibility is to build or maintain a cloud-hosted application and their supporting processes with various components. The components which are of different types and sizes are required to design (or integrate with) a cloud-hosted application for deployment in a multitenant fashion. The components may also be categorized into different groups based on type (e.g., storage components, processing components, communication components, user interface components, etc.), purpose or size or some other feature. Different groups may have components with varying degrees of isolation, which means that some components can provide the same functionality, and hence can be shared with other tenants while other components are exclusively dedicated for 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. If one of the components of the cloud-hosted application experiences a very high load, then how can an architect select components for optimal deployment in response to workload changes in a way that: (i) maximizes the degree of isolation between components by ensuring that they behave as if they were components of different tenants and, thus, are isolated against each other; and (ii) maximizes the number of requests that can access each component.

3.2 Mapping the Problem to a Multichoice Multidimensional Knapsack problem (MMKP)

The above optimal component deployment problem can be mapped to a 0-1 multichoice multidimensional knapsack problem (MMKP). An MMKP is a variant of the Knapsack problem which has been shown to be a member of the NP-hard class of problems. Our problem is formally defined as follows:

Definition 1 (Optimal Component Deployment Problem): Suppose there are N groups of components (C_1,...,C_N) with each having a_i ($1 \leq i \leq N$) 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}; (iii) the average number of requests that can be allowed to access the component Q_{ij} and (iv) resources required to support the component, $r_{ij} = r_{ij}^1, r_{ij}^2, ..., r_{ij}^n$. The total amount of available resources in the cloud required to support all the application components is $R = (R^1, R^2, ..., R^n)$.

The goal is to select one component from each group for deployment to the cloud in such a way that if one of the components experiences high load, then the:

(a) degree of isolation of the other components is maximized

(b) the number of requests allowed to access the component (and the application as a whole) is maximized, without having the total resources used exceeding the available resources.

As demonstrated in Definition 1, there are two objectives in the problem. We use an aggregation function to transform the multiobjective problem into a single objective problem by combining the two objective functions, (i.e., g1=degree of isolation, and g2=number of requests) into a single objective function (i.e., g=optimal function) in a linear way. As our optimal function is linear, we used an *priori single weight* strategy which consists in defining the weight vector to be selected according to the preferences of the decision maker [11].

Therefore, we re-state the goal as follows: to provide a near-optimal solution for deployment to the cloud in such a

way that meets the system requirements and also provides the best value for the optimal function, G.

Definition 2 (Optimal Function): The required multitenancy isolation optimization problem faced by a cloud architect for deploying components of a cloud-hosted application due to workload changes can be expressed as:

$$\sum_{i=1}^{N} \sum_{j \in C_i}^{n} g_{ij} . a_{ij}$$

Subject to

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

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

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

where (i) a_{ij} is set to 1 if component j is selected from group C_i and 0 otherwise; (ii) g_{ij} is defined by a weighted sum of parameters including the degree of isolation, average number of requests allowed to access the component, and the penalty for solutions that violates the constraints.

$$g_{ij} = (w1 \times I_{ij}) + (w2 \times Q_{ij}) - (w3 \times P_{ij}) \quad (2)$$

The weight values assigned to w1, w2, and w3 are 100,1 and 0.1 respectively. The weights are assigned in such a way as to give preference to the required degree of isolation. The penalty, P_{ij}, for the component for exceeding its resource limit is given as:

$$P_{ij} = \sum_{i=0}^{n} max\left\{0, \left(\frac{R^k - R^k_{max}}{R^k_{max}}\right)\right\}^2 \quad (3)$$

The degree of isolation, I_{ij}, for each component (that is, $\boldsymbol{g}$), is set to either 1, 2, or 3 to shared components, tenant-isolated components, and dedicated components, respectively. The notation given as: $r_{ij} = r^1_{ij}, r^2_{ij}, \ldots, r^n_{ij}$ is the resource consumption of each application component j from group C_i. The total consumption of all resources r^{α}_{ij} of all application components must be less than the total amount of resources available in the cloud infrastructure $R = R^{\alpha}$,(α = 1,...,m).

We assume that the service demands at the CPU, RAM, Disk I/O, and Bandwidth that supports each component are known and/or can be easily measured by either the SaaS provider or the SaaS customer. This assumption allows the calculation of the number of requests, Q_{ij} that can be allowed to access each component by solving an open multiclass QN model [12]. The open multiclass network will be elaborated more in the next section.

4. Queuing Network (QN) Model

Queueing network modelling is an approach to computer system modelling in which the computer system is represented as a network of queues which is evaluated analytically. A network of queues is a collection of service centers, which represent system resources, and customers, which represent users or transactions [12]. Service centers refers to the resources that resources that support the components such as CPU, RAM and disk space and bandwidth.

Assumptions: We make the following assumptions about a component:
(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 behaviours whose arrival rate is independent of the system state.
(iii) the service demands at the CPU, RAM, Disk, and Bandwidth that supports each component are known or can be easily measured by either the SaaS provider or the SaaS customer.
(iv) the resources supporting each component is enough to handle the magnitude of new incoming requests as the workload changes. This ensures that there are no overloads when all components are functional.

The above assumptions allow us to use an open multiclass queuing network (QN) model to determine the average number of requests that can be allowed to access the component while meeting the required degree of isolation and system requirements. In an open multiclass QN, the workload intensity is specified by the request arrival rate. This arrival rate usually does not depend on the system state, that is, it does not depend on the number of other tenants in the system [12].

Definition 4 (Open Multiclass Queuing Network Model): Given N number of classes in a model, where each class c is an open class with arrival rate λ_c. The vector of arrival rates is denoted by $\vec{\lambda} = (\lambda_1, \lambda_2, \ldots \lambda_N)$. The utilization of each component of class c at centre k is given by:

$$U_{c.k}(\vec{\lambda}) = \lambda_c D_{c.k} \quad (4)$$

In solving the QN model, we assume that a component represents a single open class system with four service centers (i.e., the resources that supports the component - CPU, RAM, disk capacity and bandwidth). The average number of requests at a particular service center (e.g., CPU) for a particular component is:

$$Q_{c,k}(\vec{\lambda}) = \frac{U_{c.k}(\vec{\lambda})}{1 - \sum_{i=1}^{N} U_{i.k}(\vec{\lambda})} \quad (5)$$

Therefore, to get the average number of requests that would access this component, we would add together the queue length of all requests that visit all the service centers (i.e., the resources that support the components - CPU, RAM, disk capacity and bandwidth).

$$Q_c(\vec{\lambda}) = \sum_{i=0}^{n} Q_{c,k}\ \vec{\lambda} \quad (6)$$

5. Metaheuristic Search

The optimization problem described in the previous section is an NP-hard problem which has been known to have a feasible search space that grows in an exponential way [13]. The number of feasible solutions for our optimal component deployment problem is given by the following equation:

$$\left\{\binom{n}{r}\right\}^{N} \quad (7)$$

Equation 4 above represents the number of ways we can select one or more (r) components from each a group (made of up n components) out of several (N) groups of components to design and (integrate) into a cloud-hosted application when the workload sent to the component changes. Thus, in response to workload changes, the number of ways of selecting one component (i.e., r=1) each from 20 groups (i.e., N=20) containing 10 items in each group (i.e., n=10) will result in approximately 10.24 x 10^{12} possible solutions. For a large cloud-hosted service and depending on the number of times and frequency with which the workload changes, the number of possible solutions could grow to a much larger magnitude.

Therefore, an efficient metaheuristic is needed to find a near-optimal solution to the optimization problem, which must be provided to the SaaS customer (or a cloud deployment architect) in almost real-time. We present two variants of a hybrid simulated annealing algorithm: (i) *SAGreedy*, uses greedy principles along with simulated annealing algorithm; (ii) *SARandom*, uses randomly generated solutions along with simulated annealing algorithm. Either of two variants of the simulated annealing algorithm can be used to obtain near-optimal solutions for deploying a component. In addition, we also developed an algorithm to perform an exhaustive search of the entire solution space for a small problem. The algorithms for *SA(Greedy* are presented as Algorithm 1. The other variant SA(Random) requires only a slight modification as will be explained in the next section. A high-level description of the algorithm is presented below:

Algorithm 1 SA(Greedy) Algorithm

```
1:  SA (Greedy) (mmkpFile, N)
2:  Randomly generate N solutions
3:  Set initial temperature to T0 to st. dev. of all
    optimal solutions
4:  Create greedySoln a^1 with optimal value g(a^1)
5:  optimalSoln = g(a^1)
6:  bestSoln = g(a^1)
7:  for I = 1, N do
8:  Create neighbor 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 = a^2
12: else
13:     if random[0,1) < exp(-(g(a^2) – g(a^1))/T) then
14:         a^2 = bestSoln
15:     end if
16:     end if
17:     T_{i+1} = 0.9 * T_i
18: end for
19: optimalSoln = bestSoln
20: Return (optimalSoln)
```

5.1. The *SAGreedy* for Near-optimal Solution

This algorithm combines simulation annealing and greedy algorithm to find a near-optimal solution to our optimization problem which has been modeled 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, the average number of requests, and components resource consumption). We use a simple linear cooling schedule, where $T_{i+1} = 0.9Ti$. Our strategy for setting the initial temperature T_0 is to randomly generate a number of optimal solutions equal in value to the number of groups (n) in the problem instance multiplied by the number of iterations (N) set for the experiment, before running the simulated annealing aspect of the algorithm.

When the problem instance and/or the number of iterations is small, the number of generated optimal solutions might be set to the number of groups (n) in the problem instance multiplied by the number of iterations (N) set for the experiment. After that, we set the initial temperature T_0 to the standard deviation of all the randomly generated optimal solutions (line 2-3). It then creates the greedy solution as an initial solution (line 4). This greedy solution is taken as the best solution at this point. The simulated annealing process improves the greedy solution, and provides the near-optimal solution for deploying components to the cloud.

A simple dry run of the algorithm for the instance C(4,5,4) is as follows: assuming that the number of iterations set for the experiment is 100, we randomly generate 400 (i.e., 4 groups x 100 iterations) optimal solutions and then find the standard deviation of all the solutions. Assuming this value is 50.56, we set T_0 to 50. Also, assuming the algorithm constructs an initial greedy solution with $g(a^1) = 2940.12$, and then a current random solution with $g(a^2) = 2956.55$. The solution a^2 will replace a^1 with probability, P =exp(-16.43/50)=0.72, because $g(a^2) > g(a^1)$. In lines 14 to 16, we generate a random number rand between 0 and 1, and if rand <0.72, a^2 replaces a^1 and we continue with a^2. Otherwise, we continue with a^1. At this point, we reduce the temperature T which now becomes $T_1 = 45$ (line 17). We continue iterations until we reach N, the number iterations set for the

algorithm to run, and so the search converges with a high probability to the near-optimal solution.

5.2. The SA(Random) for Optimal Solutions

The second variant of the metaheuristic, SA(Random), randomly generates a solution and then passes it to the simulated annealing process to become the initial solution. That is, in line 4, instead of constructing a greedy solution, we simply generate a random solution. The algorithm provides an optimal solution that represents a selection of components with the highest total isolation value and the highest number of requests that can be allowed to access the components. The optimal solution would be provided each time there is a change in workload.

6. Developing a Decision-based Model

A decision-based model is simply model that helps a software architect to make decisions about software engineering problems that may be rapidly changing (e.g., due to workload changes) and which may not easily be specified in advance [23].

Our decision-based model emphasizes access to and manipulation of a combination of several models such as an optimization model, queuing network model and metaheuristics.

In previous section, we describe the optimization model (based on MMKP) and queuing network (QN) model (based on open multiclass QN), and the metaheuristic solution. The following section describes how these components can be combined to form a decision-based model for providing optimal solutions for deploying components of a cloud-hosted application.

The decision-based model can be transformed into a cloud-based application and provided to a decision maker (e.g., software engineer or cloud architect) to access and manipulate the model data to provide optimal solutions for deploying components of a cloud-hosted application.

6.1. Architecture of the Decision-based Model

The architecture of the decision-based model is composed on five constituents:

(i) Input Interface: This module handles the task of sending inputs to the decision-based model. These inpiut include the system workload (i.e., the arrival rate of requests (λ)) and the id MMKP instance. The MMKP instance contains the configuration of the components.

(ii) Information Repository: This module is responsible for storing information and files required to run the decision support module. These include the MMKP file (which contains the component configuration), the workload file (which contains service demands and arrival rate of request of each component, and the updated MMKP file (which contains the updated details of MMKP when the workload changes).

(iii) Queuing Network Model: This module is used to solve the open multiclass queuing network model based on the arrival rate of the requests to each component and the id of the required (MMKP) problem instance. The module produces number of requests allowed to access each component and the total number of requests allowed to access the whole cloud-hosted service can be computed and sent to the output interface.

(iv) Optimization Module: This module is responsible for running the metaheuristic on the updated MMKP instance to provide optimal solutions for deploying components of the cloud-hosted application in a way that guarantees the required degree of multitenancy isolation.

(v) Output Interface: The output interface shows all information required for optimal deployment of components of a cloud-hosted service to guarantee multitenancy isolation. Such information includes but are not limited to the optimal solutions, the number of requests sent to each component and the total number sent to whole the system.

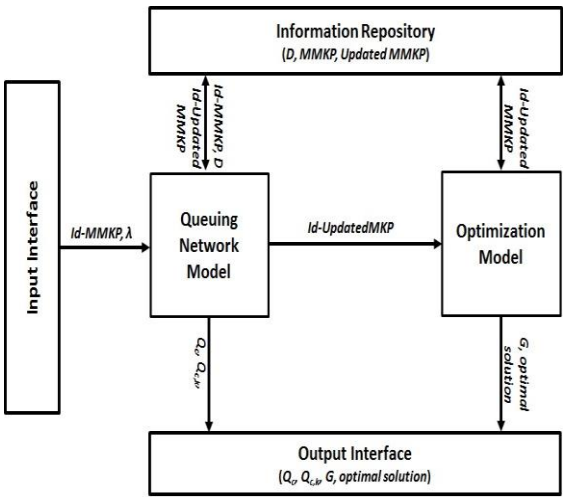

Figure 3. Architecture of the Decision-based model for Multitenant Isolation

6.2. Decision-based Model Algorithm for Optimal Deployment of Components

The following algorithm can be used to support the decision-based algorithm so that components of a cloud-hosted application can be deployed in a way

that guarantees multitenancy isolation. The algorithm runs as follows:

Algorithm 2 Decision-based Model (DbM) Algorithm	
1:	DbM (**workloadFile, mmkpFile**)
2:	*optimalSoln* ← null
3:	Receive workload from SaaSusers
4:	Load workloadFile, mmkPfile; populate global variables
5:	repeat
6:	/*Solve QN Model*/
7:	for *i* ← 1,*NoGroups* do
8:	for *i* ← 1,GroupSize do
9:	Compute Utilization
10:	Compute No. of req.
11:	Compute Total No. of req.
12:	Store fitValue,Isol, qLen,opSoln.
13:	end for
14:	end for
15:	Update the mmkpFile with qLength
16:	/*Execute Metaheuristic*/
17:	SA(GREEDY)
18:	/*Display optimal solution for deployment*/
19:	until no more workload
20:	Return (**opSoln, fitValue, Isol, qLen**)

A high-level description of the DbM algorithm (i.e., **Algorithm 2**) 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 (line 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 (line17). 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.

7. Evaluation

In this section, we describe how we generated instance for the experiments, and the experimental setup and procedure.

7.1. Instance Generation

We randomly generated several problem instances of various sizes and densities. Two categories of instances were generated and tailored on the instances widely cited in literature: (i) OR benchmark Library [14] and other standard MMKP benchmarks, and (ii) the new irregular benchmarks used by Shojaei et al. [15]. These benchmarks are used for single objective problems. So, we modified and extended this benchmark format to conform to a multitobjective case by associating each component with two different profits values: isolation values and average number of requests [16].

The values contained in the instance were generated as follows: (i) isolation values were randomly generated in the interval [1-3]; (ii) values of component's consumption of CPU, RAM, disk and bandwidth (i.e., the weights) were generated in the interval [1-9]; (iii)resource limit for each component (i.e., knapsack capacities for CPU, RAM, disk and bandwidth) are generated by setting it to half of the maximum possible resource consumption (see equation 7).

$$c_k = \frac{1}{2} \times m \times R \qquad (8)$$

The same principle has been used to generate instances available at OR Benchmark Library, and also for instances used in [17] [18]. In this work, the number of resources/constraints in each group is four, which corresponds to the basic resources required for a component to be deployed to the cloud. The notation for each instance is: C(n,r,m), which stands for number of groups, number of components in each group and number of resources.

7.2. Experimental Setup and Procedure

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 1 shows the parameters used for the experiments.

Table 1. Parameters used in the experiments

Parameters	Value
Isolation Value	[1,2,3]
No. of Requests	[0,10]
Resource consumption	[0,10]
No. of Iterations	N=100 (except Table 4)
No. of Random Changes	5
Temperature	T_0 = st.dev of N randomly generated solns.
Linear Cooling Schedule	$T_{i+1}=0.9Ti$

We tested the algorithm with instances of varying sizes and densities. We could not perform an exhaustive search on large instances. This was principally because of the low memory of the machine used. Due to this limitation, we first used

the MMKP instance, C(4,5,4) as a benchmark for evaluation and comparison of the algorithms.

8. Results

In this section, we discuss the results of the study.

8.1. Comparison of the Obtained Solutions with the Optimal Solutions

We first compared the results obtained from the SA(Greedy) and SA(Random) algorithms with the optimal solutions obtained by running an algorithm that performs an exhaustive search of the entire solution space for a small problem instance (i.e., C(4,5,4)). The results are summarized in Table 2 and Table 3. In Table 2, the first column shows the instance id used. The second, third and fourth columns show the optimal function variables as (FV/IV/RV), which stand for the value of optimal function, isolation value, and number of allowed requests, for Optimal, SA(Random) and SA(Greedy) algorithm, respectively. In Table 3, the first and second columns shows a fraction of the optimal values for SA(Random) and SA(Greedy) algorithm, respectively. The last two columns show the absolute percentage difference for SA(Random) and SA(Greedy) algorithm. The absolute difference which indicates the quality of the solution, is measured as:

$$\frac{|f(s)-f(s^*)|}{f(s^*)} \quad (8)$$

where s is the obtained solution and s* is the optimal solution obtained from the exhaustive search [11].

As shown, the SA(Greedy) and SA(Random) produced very similar results. For SA(Random) the solutions obtained are nearly 100% close to the optimal solution in most cases, and above 83% in others, and less than 66% only in one case. For the SA(Greedy) the solutions obtained is nearly 100% close to the optimal solution in most cases, and above 83% in others. Overall the SA(Greedy) performed slightly better than SA(Random) in terms of the percent deviation from the optimal solution.

8.2. Comparison of the Obtained Solutions to a Target Solution

We could not solve instances larger than C(4,5,4) due to limitations in the hardware requirements (i.e., CPU and RAM) of the machine used for the experiments. As a result of this, we compared the results to a reference/target solution [11]. The target solution for percent deviation and performance rate is set to (n x max(I) x w1) and ((n x max(I) x w1) + (0.5 x (n x max(Q) x w2))), respectively. So for instance C(150,20,4), the target solution for computing percent deviation is 45000.

Tables 3, 4, and 5 show the average behaviour of the solutions: (i) on a large variety of different instances using the same parameters; (ii) over different runs on the same instance (with varying number of optimal evaluation) and (iii) over different runs on the same instance, respectively. We measured the robustness of the solutions in terms of its behaviour on different types of instances using the same parameters. As shown in Table III, the robustness of the solutions is strong in terms of the average deviation behaviour of the solutions for both the SA(Greedy) and SA(Random) as indicated by their low variability.

The average percent deviation and standard deviation (of the percent deviations), for SA(Greedy) is slightly higher than that of SA(Random). This is due to the large absolute difference between the some of the obtained solutions and the reference solution. For example, the percent deviations of SA(Greedy) for the instances, C(100,20,4) and C150,20,4) are much higher than that of SA(Random). The results shows that SA(Random) performs significantly better than SA(Greedy) on small instances up to C(80,20,4).

In Table 4, we compared the quality of solutions with the number of optimal function evaluations. The quality of solutions is generally good when both algorithms are tested on large instances. Again, the standard deviation for SA(Greedy) is significantly lower than that of SA(Random) and percent deviations are also very stable. Table 5 also shows that SA(Greedy) is more robust than SA(Random) as indicated by the average optimal values and the low variability of the solutions. We computed the performance rate (PR) that the reference solution has been attained as a function of the number of optimal evaluations. The PR of SA(Greedy) is slightly higher than that of SA(Random).

In Figure 1, we show the relationship between the quality of solutions in terms of the optimal values (i.e., fitness value) and the number of optimal function evaluations. The diagram shows that SA(Greedy) slightly benefited from the initial greedy solution more than SA(Random) when the number of optimal function evaluations is small. However, the quality of solutions for both algorithms improves as the number of iterations increases. The optimal solution stabilizes after 100 optimal evaluations and thereafter fails to improve significantly.

Figure 3 shows the relationship between the quality of solutions in terms of the percent deviation and the number of function evaluations. As expected, the SA(Random) had lower percent deviation than SA(Greedy) for most of the cases especially when the number of function evaluations is small. This may be due to the small number of function evaluations used for the experiments. However, the

percent deviation for SA(Greedy) was more stable even though they were higher than that of SA(Greedy).

Table 2. Comparison of SA(Greedy) and SA(Random) with the optimal solution

Inst -id	Optimal (FV/IV/RV)	SA(R) (FV/IV/RV)	SA(Greedy) (FV/IV/RV)
I1	1213.93/12/24	1218/12/18	1218/12/18
I2	1213.97/12/14	1208.99/12/9	1109/11/9
I3	1222.99/12/23	1120/11/20	1120/11/20
I4	1119.98/11/20	1023/10/23	1019/10/19
I5	1219.99/12/20	1017/10/17	1017/10/17
I6	1229.92/12/30	1020.96/10/21	1022.99/10/23
I7	1224.90/12/25	1018/10/18	1018/10/18
I8	1228.96/12/29	822/8/22	1224.99/12/29
I9	1021.97/10/22	912/9/12	912/9/12
I10	1236/12/36	1236/12/36	1236/12/36
Std			

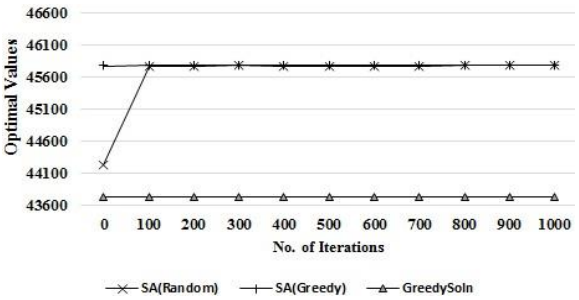

Figure 2. Relationship between Optimal values and Function Evaluations

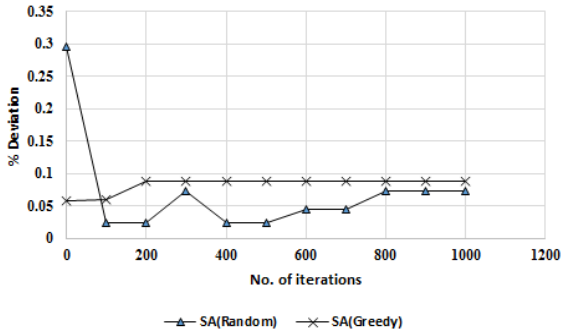

Figure 3. Relationship between Percent deviation and Function evaluations

Table 3. Computation of a fraction of the obtained solutions with the optimal solution

Inst-id	SA-R/Opt	SA-G/Opt	%D (SA-R)	%D (SA-G)
I1	0.995	0.995	0.48	0.48
I2	0.996	0.914	0.41	8.65
I3	0.916	0.916	8.42	8.42
I4	0.913	0.910	8.66	9.02
I5	0.834	0.834	16.64	16.64
I6	0.830	0.832	16.99	16.82
I7	0.831	0.831	16.89	16.89
I8	0.669	0.997	33.11	0.32
I9	0.892	0.892	10.76	10.76
I10	1	1	0	0
ST.DEV	0.097	0.064	9.73	6.43

9. Discussion

In this section, we discuss the implication of the results and present some recommendations for deploying components of a cloud-hosted application with guarantees for multitenancy isolation.

9.1. Quality of the Solutions

The quality of solution was measured in terms of the percent deviation from either the optimal solution or the reference solution. As shown in Tables II, III and IV, solutions obtained from SA(Greedy) show a low percent deviation. The results also showed that SA(Greedy) performed well for large instances. However, in Table III, it can be clearly seen that SA(Random) outperformed SA(Greedy) on small instances, that is, from C(5,20,4) to C(80,20,4).

9.2. Computational Effort and Performance Rate of Solutions

Due to limitations in the hardware of the machine used for the experiments, we did not consider the computational effort of the metaheuristics. However, we have still computed the performance rate of the algorithm as a function of the number of optimal evaluations. The number of optimal evaluations is generally regarded as an indicator for computational effort that is independent of the computer system [11]. As shown in Table V, the performance rate of SA(Greedy) was slightly better than that of SA(Random) considering the number of reference solutions attained.

Table 4. Comparison of the Quality of Solution with Instance size

Inst (n=20)	SA-R	SA-G	SA-R	SA-G
C(5)	1528/15/28	1517/15/17	1.87	1.13
C(10)	3058/30/58	3057/30/57	1.93	1.9
C(20)	6082/60/82	6002/59/102	1.37	0.03
C(40)	12182/120/182	12182/120/182	1.52	1.52
C(60)	18317/180/317	18317/180/317	1.76	1.76
C(80)	24388/240/388	24380/240/380	1.62	1.58
C(100)	30252/298/452	30505/300/505	0.84	1.68
C(150)	45648/449/748	45793/450/793	1.44	1.76
Avg. %D			1.54	1.42
STD of Avg. %D			0.33	0.57

Table 5. Comparing Solution Quality with no. of Optimal Evaluations

No. of Iteratn	SA(R)	SA(G)	%D (SA-R)	%D (SA-G)
1	44242.1	45776.8	3.30	0.06
100	45760.7	45776.9	0.02	0.06
200	45760.7	45790	0.02	0.09
300	45783	45790	0.07	0.09
400	45760.7	45790	0.02	0.09
500	45760.7	45790	0.02	0.09
600	45770.1	45790	0.04	0.09
700	45770.1	45790	0.04	0.09
800	45783	45790	0.07	0.09
900	45783	45790	0.07	0.09
1000	45783	45790	0.07	0.09
Worst	44242.1	45776.8	0.02	0.06
Best	45783	45790	3.30	0.09
Average	45632.5	45787.6	0.34	0.08
STD	439.77	5.07	0.93	0.01

9.3. Robustness of the solutions

The SA(Greedy) algorithm showed low variability especially when tested with large instances, thus, indicating that it provides solutions that are more robust than that of SA(Random). This makes SA(Greedy) suitable for use in a dynamic real-time environment where the workload changes frequently. It can be seen in Figure 2 that the SA(Random) was very unstable for a small number of evaluations, although it eventually converged at the optimal solution after 1000 evaluations. The SA(Greedy) on the other hand improved steadily as the number of evaluations increased.

9.4. Required Degree of Isolation

The optimization model we have presented assumes that each component (or group of components) to be deployed is associated with a particular degree of isolation. This was achieved by mapping the problem to a multichoice multidimensional knapsack problem where each component is associated with two profit values: isolation value and the number of requests to access each component. This allows us to monitor each component independently and to respond to the specific demands of each component. In a case where there are limitations in either cost, time, effort in tagging each component, then a separate algorithm may be required to perform this operation dynamically. In our previous work [19], we developed an algorithm that can dynamically learn the properties of existing components in a repository and then use this information to associate each component with the required degree of isolation.

Table 6. Robustness of solutions over different runs on the same instance

Runs	SA(R)	SA(G)	%D SA-R	%D SA-G
1	45767	45658	1.70	1.46
2	45793	45744	1.76	1.65
3	45663	45793	1.47	1.76
4	45460	45658	1.02	1.46
5	45567	45658	1.26	1.46
6	45562	45793	1.25	1.76
7	45569	45658	1.26	1.46
8	45567	45658	1.26	1.46
9	45682	45658	1.52	1.46
10	45663	45767	1.47	1.70
Worst	45460	45658	1.02	1.46
Best	45793	45793	1.76	1.76
Avg	45629	45705	1.40	1.57
STD	97.67	58.40	0.22	0.13
Perf. Rate	2.0E-04	3.0E-04		

10. Application Areas for Utilizing the Optimisation Model

Here various areas through which the optimisation model, for the purpose of a cloud-hosted application aiming to guarantee the necessary levels of multitenancy isolation, can be applied to the

deployment components, are discussed in greater depth.

10.1. Optimal Allocation in a Resource Constrained Environment

The optimisation model is effective in optimising resource allocation, particularly in environments where resources are more limited or regular workload changes are experienced. Optimisation can be obtained by integrating the model into a load balancer/manager. It is common for cloud providers to use auto-scaling programs (e.g., Amazon Auto Scaling) to manage scaling applications deployed on their cloud infrastructure and are usually influenced by pre-defined scaling regulations. However, such programs are not capable of guaranteeing the isolation of tenants (or components) for deployment by the application or service. The responsibility of implementing such functionality to benefit cloud deployment for the individual lies with the customer.

Where frequent and substantial changes to workload are experienced, it is highly likely that performance interference is also experienced in the same environment. Environments like this are ideal for our model to operate capable of identifying optimal configuration for component deployment whilst maximising the total number of permitted component access requests and at the same time maximising amongst tenants and/or components the degree of isolation.

10.2. Monitoring Runtime Information of Components

Another area for effective use of our model is monitoring component runtime information. Many suppliers can provide information monitoring on network availability and component deployment of cloud infrastructures guided by pre-defined configuration rules. However, it is not possible at application level for information to assure efficient component operation or to guarantee the necessary levels of multitenancy isolation. The customer is responsible for obtaining and analyzing these values and also for altering the configuration to determine an optimal configuration that can guarantee the necessary levels of multitenancy isolation.

The optimisation model defined in this study can be utilized as a basic web service-based application capable of monitoring the service and responding to changes such as automatically changing rules influenced by learnt information and previous experience, for example, user input when a defined allocation of resources is exceeded. In addition, it is possible for this application to be either separately deployed or fully integrated into various cloud-based services to monitor health statuses.

10.3. Managing the Provisioning and Decommissioning of Components

The provisioning and decommissioning of components and/or functionality provided to customers through cloud suppliers is done so through the configuration of pre-defined rules. For example, a pre-defined rule can ascertain that when the defined threshold of a resource system average utilization (e.g., RAM, disk space) is exceeded, then a particular component is deployed. When extracting and making available component runtime information, important decisions can be made relating to the provisioning of required components and decommissioning of unused or no longer required components.

11. Conclusion and Future work

In this paper, we have presented a decision-based model composed of an optimization model (based on a multichoice multidimensional knapsack problem-MMKP), an open multiclass queuing network (QN) model and a metaheuristic algorithm to provide optimal solutions for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation. This decision-based model contributes to literature on multitenancy isolation and optimization of component deployment for cloud-hosted applications. We first formulated an optimization problem that captures the problem of implementing the required degree of isolation between components, mapped it to a multichoice multidimensional knapsack problem so that it can be easily solved using a metaheuristic. Thereafter, we wrapped all the constituents including the optimization model, QN model, and metaheuristic using an algorithm we developed to support the decision-based model.

The study revealed that SA(Greedy) is more stable and consistent, especially when used on large problem instances, while SA(Random) performs well on small instances. In terms of robustness, the SA(Greedy) showed low variability and so would produce better solutions in an environment where workload the changes frequently. The SA(random) seems to be more sensitive to small deviations (i.e. produces unstable results) on input instances than SA(Greedy) especially when tested on large instances. SA(Random)

We plan to challenge our MMKP problem instances with different types of metaheuristics (e.g., genetic algorithm, estimated distribution algorithm) and with combinations of different metaheuristics (e.g., simulated annealing combined with genetic algorithm) with a view to selecting suitable metaheuristics to produce the optimal solutions for different cloud deployment scenarios. In future, we will develop a real-time cloud-based decision

support system that can be used to design (or integrate into) an elastic load balancer to monitor runtime information about individual components and then provide near-optimal solutions for deploying components of a cloud-hosted application in response to workload changes. Such information could be very helpful for a cloud architect or SaaS customer in making decisions about how and when to provision required components and decommission unused components.

12. Acknowledgment

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

13. References

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

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

[3] L. C. Ochei, J. Bass, and A. Petrovski, "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, 2016.

[4] F. Shaikh and D. Patil, "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, 2014, pp. 1–4.

[5] D. Westermann and C. Momm, "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, 2010, p. 3.

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

[7] D. Candeia, R. A. Santos, and R. Lopes, "Business-driven long-term capacity planning for saas applications," *IEEE Transactions on Cloud Computing*, vol. 3, no. 3, pp. 290– 303, 2015.

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

[9] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar, "Moving applications to the cloud: an approach based on application model enrichment," *International Journal of Cooperative Information Systems*, vol. 20, no. 03, pp. 307–356, 2011.

[10] A. Aldhalaan and D. A. Menascé, "Near-optimal allocation of vms from iaas providers by saas providers," in *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*. IEEE, 2015, pp. 228–231.

[11] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[12] D. Menasce, V. Almeida, and D. Lawrence, *Performance by design: capacity planning by example*. Prentice Hall, 2004.

[13] F. Rothlauf, *Design of modern heuristics: principles and application*. Springer Science & Business Media, 2011.

[14] J. E. Beasley, "Or-library: distributing test problems by electronic mail," *Journal of the operational research society*, vol. 41, no. 11, pp. 1069–1072, 1990.

[15] Z. Eckart and L. Marco. Test problems and test data for multiobjective optimizers. Computer Engineering (TIK) ETH Zurich. [Online]. Available: http://www.tik.ee.ethz.ch/sop/.../testProblemSuite/

[16] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[17] R. Parra-Hernandez and N. J. Dimopoulos, "A new heuristic for solving the multichoice multidimensional knapsack problem," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 35, no. 5, pp. 708–717, 2005.

[18] N. Cherfi and M. Hifi, "A column generation method for the multiple-choice multi-dimensional knapsack problem," *Computational Optimization and Applications*, vol. 46, no. 1, pp. 51–73, 2010.

[19] L. C. Ochei, A. Petrovski, and J. Bass, "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)*, 2016.

[20] L. C. Ochei, A. Petrovski, and J. Bass, "Optimizing the Deployment of Cloud-hosted Application Components for Guaranteeing Multitenancy Isolation," 2016 International Conference on Information Society (i-Society 2016).

[21] Pearson, S. (2013). Privacy, security and trust in cloud computing, *Privacy and Security for Cloud Computing*, Springer, pp. 3–42.

[22] Krebs, R., Momm, C. & Kounev, S. (2014). Metrics and techniques for quantifying performance isolation in cloud environments, Science of Computer Programming 90: 116–134.

[23] Richardson, I., Casey, V., Mccaffery, F., Burton, J., & Beecham, S. (2012). A process framework for global software engineering teams. *Information and Software Technology*, *54*(11), 1175-1191.

