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

TITLE: Evolutionary computation for optimal component deployment with multitenancy isolation in cloud-hosted applications.

YEAR: 2018

Publisher citation: OCHEI, L.C., PETROVSKI, A. and BASS, J.M. 2018. Evolutionary computation for optimal component deployment with multitenancy isolation in cloud-hosted applications. In Proceedings of Institute of Electrical and Electronics Engineers (IEEE) Systems Man and Cybernetics Society (SMC) international innovations in intelligent systems and applications conference 2018 (INISTA 2018), 3-5 July 2018, Thessaloniki, Greece. New York: IEEE [online], article ID 8466315. Available from: https://doi.org/10.1109/INISTA.2018.8466315

OpenAIR citation: OCHEI, L.C., PETROVSKI, A. and BASS, J.M. 2018. Evolutionary computation for optimal component deployment with multitenancy isolation in cloud-hosted applications. In Proceedings of Institute of Electrical and Electronics Engineers (IEEE) Systems Man and Cybernetics Society (SMC) international innovations in intelligent systems and applications conference 2018 (INISTA 2018), 3-5 July 2018, Thessaloniki, Greece. New York: IEEE [online], article ID 8466315. Held on OpenAIR [online]. Available from: https://openair.rgu.ac.uk/

Evolutionary Computation for Optimal Component Deployment with Multitenancy Isolation in Cloud-hosted Applications

Laud Charles Ochei, Andrei Petrovski

School of Computing Science and Digital Media
Robert Gordon University
Aberdeen, United Kingdom
Emails: {l.c.ochei,a.petrovski}@rgu.ac.uk

Julian M. Bass

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

***Abstract*—A multitenant cloud-application that is designed to use several components needs to implement the required degree of isolation between the components when the workload changes. The highest degree of isolation results in high resource consumption and running cost per component. A low degree of isolation allows sharing of resources, but leads to degradation in performance and to increased security vulnerability. This paper presents a simulation-based approach operating on computational metaheuristics that search for optimal ways of deploying components of a cloud-hosted application to guarantee multitenancy isolation When the workload changes, an open multiclass Queuing Network model is used to determine the average number of component access requests, followed by a metaheuristic search for the optimal deployment solutions of the components in question. The simulation-based evaluation of optimization performance showed that the solutions obtained were very close to the target solution. Various recommendations and best practice guidelines for deploying components in a way that guarantees the required degree of isolation are also provided.**

***Index Terms*—Evolutionary computation, cloud-hosted services, simulation-based optimization, metaheuristics, deployment patterns, multitenancy isolation.**

I. Introduction

Software architectures encompass one or many structures and components that affect the way systems achieve functional and non-functional requirements. These structures are based on software elements, relations between them, and properties of both [1]. Reasoning on optimal software deployment patterns for multitenant applications in cloud environments with dynamic resource provisioning is a non-trivial problem necessitating the consideration of several challenges.

One of the challenges relates to the implementation of multitenancy, i.e. how to ensure that there is isolation between multiple components of a cloud-hosted application, when one of the components experiences high load. This challenge is hereafter referred to as *multitenancy isolation* [2] [3]. A high degree of isolation can be achieved by deploying an application component exclusively for one tenant. This would ensure that there is little or no performance interference between the components when workload changes. However, because components are not shared, there might be multiple copies of the component for each tenant, which leads to high resource consumption and running cost.

Therefore, in order to optimize the deployment of components, the software architect has to satisfy two objectives: maximize the degree of isolation between components and at the same time maximize the number of requests that can be allowed to access the component. This is a multi-objective optimisation problem that involves run-time optimisation of several resources used, which means new solving techniques need to be applied that are capable of taking into consideration the behaviour of cloud-hosted applications at run-time [4].

Motivated by the stated problem, this paper presents a simulation-based evolutionary optimisation approach utilizing metaheuristics that can be used to provide sufficiently near-optimal solutions for deploying components of a cloud-hosted application in a way that guarantees multitenancy isolation and allowing as many requests as possible to access them. We implemented our approach by first applying an open multiclass Queuing Network (QN) model to determine the number of requests allowed to access a component. This information is used to update a multiobjective optimization model (derived by mapping our problem to a Multichoice Multidimensional Knapsack Problem (MMKP)). Thereafter, a metaheuristic based on simulated annealing is used to find near-optimal solutions for component deployment.

The proposed approach is evaluated by comparing the solutions obtained through evolutionary optimisation with the optimal results obtained from an exhaustive search of the entire solution space for a small problem. This first step would verify the viability of the proposed approach. Then, the evolutionary optimisation metaheuristics will be tuned to figure out the best way of deploying components of a cloud-hosted application that guarantees multitenancy isolation. Thus, the main contributions of this paper are:

1. Creating a novel simulation-based approach, *optimalSoln*, that combines: (i) an open multiclass QN model; and (ii) an optimization task, to provide a near-optimal solution for deploying components of a cloud-hosted application with guarantees for multitenancy isolation.
2. Developing three variants of a metaheuristic which are based on a *simulated annealing* algorithm for solving the optimization task. These variants were extensively evaluated and compared.
3. Presenting recommendations and best practice guidelines based on the experience gained from extensive evaluation and comparison of the algorithms.

This paper is a revised and expanded version of our previous work [5]. In this current study, we have included mathematical equations for the optimization model, the open multiclass queuing network(QN) model and a new algorithm that combines the optimization model and the QN model to provide optimal deployment solutions for guaranteeing multitenancy isolation. The experiments have been expanded by increasing the sizes and dimensions of the problem instances and the number of runs and iterations.

To the best of our knowledge, this study is the first to present an evolutionary computation approach that combines a simulation-based model with metaheuristic techniques to provide a near-optimal solution for deploying components of a cloud-hosted application necessitating multitenancy isolation. The rest of the paper is organized as follows: Section II formalises the optimization problem, presents the optimalSoln algorithm that drives the proposed approach, and explains the metaheuristics used. Section III is devoted to the evaluation of the proposed evolutionary computation approach, while Section IV draws conclusions and speculates possible future work.

II. Problem Formalization and Notation

The recent advances in evolutionary simulation-based optimization have allowed for several different fields of science to benefit from computational simulation in experimental studies. In [6], the authors developed a hybrid evolutionary algorithm for task scheduling and data assignment to process data-intensive workflows within a cloud computing environment. Their main concern was to optimize a transfer of a large amount of data from one virtual machine to another. The usage of Genetic Algorithms to match services and cloud resources was advocated in [7]. Various resource provision algorithms in cloud computing are surveyed by [8], some of which - metaheuristic-based and greedy strategies - are applied in this work.

This section formalises the problem to be addressed, describes how it is mapped to a Multichoice Multidimensional Knapsack Problem (MMKP), and presents the open multiclass queuing network model used for simulation-based optimization.

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

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 (see Figure 1), the problem of providing optimal solutions that guarantee multitenancy isolation 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 [9]. 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 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} (equivalent to $D_{c,k}$ in the QN model); (iv) the average number of requests that can be allowed to access the component Q_{ij} (equivalent to $Q_{c,k}$ in the QN model); 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^{α} $(\alpha = 1,..., m)$.

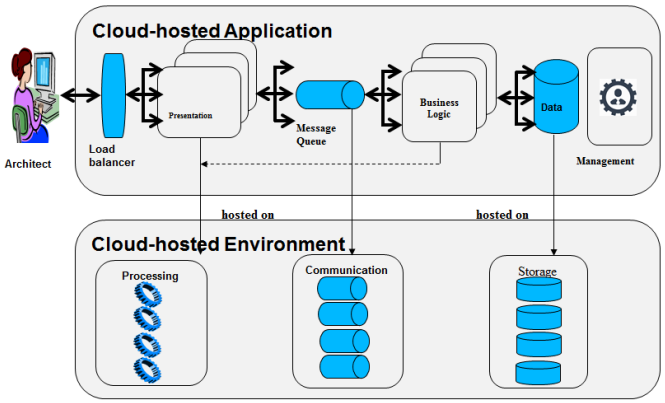

Fig. 1. System Model of a Cloud-hosted Service with multiple groups of components

An aggregation method is used to transform the multiobjective problem into a single objective problem by a linear combination of two objectives, (i.e., maximizing the degree of isolation (g_1) and the number of requests (g_2)) into a single maximization function ($\mathcal{G}$). The particular aggregation strategy used is the *priori single weight* strategy which consists of defining the weight vector to be selected based on domain knowledge and the preferences of the decision maker [10]. This approach has been widely used in literature for various metaheuristics, such as for instance, genetic algorithm and simulated annealing [11].

Therefore, the goal is re-stated as follows: to provide an 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, $\mathcal{G}$. $\mathcal{G}$ is defined by a weighted sum of single objectives including the degree of isolation and the average number of requests allowed to access the component. A penalty measure is applied for solutions that violate the existing constraints.

Definition 2: Optimal Function - Given an isolation value of a component $\mathcal{I}$, and the average number of request $\mathcal{Q}$ that can be allowed to access the component:

$$g_{ij} = (w_1 \times I_{ij}) + (w_2 \times Q_{ij}) - (w_3 \times P_{ij}) \quad (1)$$

The penalty, $\mathcal{P}$, for violating constraints of a component is:

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

where w_1, w_2, w_3 are the weights for isolation value (w_1=100), number of requests(w_2=1) and penalty(w_3=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 the 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. The penalty function, P_{ij}, is subtracted from the optimal function to avoid excluding all infeasible solutions from the search space. The expression $R_j - R_j^{max}$ in the penalty function shows the degree of constraint violation, which is divided by the resource limit and squared to make the penalty heavier for violating any constraint.

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

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

subject to

$$\sum_{i=1}^{N} \sum_{j \in C_i} r_{ij}^{\alpha}.a_{ij} \leq R^{\alpha} (\alpha = 1, 2, ..., m) \quad (3)$$

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

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

where a_{ij} is set to 1 if component j is selected from group C_i and 0 otherwise. 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^{α} (α = 1,..., m).

To calculate the number of requests, Q_{ij} that can be allowed to access the component, an open multiclass QN model has to be applied [12] 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). Section III describes how the average number of requests allowed to access each component is computed. 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 [13].

B. Open Multiclass Queuing Network Model

Definition 3: (Open Multiclass Queuing Network Model): Assume there are N 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 $\overrightarrow{\lambda} \equiv (\lambda_1, \lambda_2, ... \lambda_N)$. The utilization of each component of class *c* at centre *k* is given by:

$$U_{c,k}(\overrightarrow{\lambda}) = \lambda_c D_{c,k} \quad (4)$$

In our QN model, it is assumed that a component represents a single open class system with four service centres (i.e., the resources that support the component CPU, RAM, Disk capacity and Bandwidth). The average number of requests at a particular service centre (e.g., CPU) for a particular component is:

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

Therefore, to obtain the average number of requests that would access this component, the queue length of all requests that visit all the service centres (i.e., the resources that support the components - CPU, RAM, Disk capacity and Bandwidth) are added together.

$$Q_c(\overrightarrow{\lambda}) = \sum_{k=1}^{K} Q_{c,k}(\overrightarrow{\lambda}) \quad (6)$$

C. OptimalSoln Algorithm and Metaheuristic Search

The *optimalSoln* algorithm, presented as Algorithm 1, is used to find a new optimal solution for deploying components with the highest degree of isolation and the highest number of supported requests every time there is a change in the workload on the cloud-hosted service. 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 a 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).

The optimization problem described in Section II-A is an NP-hard problem which has been known to have a feasible

Algorithm 1 optimalSoln Algorithm

```
1: optimalSoln (workloadFile, mmkpFile)
2: opSoln ← 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 ← 1, NoGroups do
8:         for i ← 1,GroupSize do
9:             Calculate Utilization /*see Equation 4*/
10:            Calculate No. of req. /*see Equation 5*/
11:            Calculate Total No. of req. /*see Equation 6*/
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)
```

state space that grows in a combinatorial way [13]. An efficient heuristic is needed to find an optimal solution to the optimization problem, which must be solved by our simulation-based approach, and provided to the SaaS customer (or a cloud deployment architect) in almost real-time. Algorithm 1 shows the optimalSoln combined with SA(Greedy), a variant of simulated annealing algorithm, to find an optimal solution to the stated optimization problem (line 17 of Algorithm).

We developed four variants of a metaheuristic solution as summarised below [5]:

(i) *SA(Greedy)*: This 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, populates the global variables, and then a greedy solution is created as an initial solution. The simulated annealing process improves the greedy solution, and provides the optimal solution for deploying components to the cloud.

(ii) *SA(Random)*: In the SA(Random) variant, a random solution is generated (instead of constructing a greedy solution) and passed to the simulated annealing process to become the initial solution. The two variants based on simulated annealing algorithm (i.e, SA(Greedy) and SA(random)) can be converted to a local search based on the hill-climbing algorithm by setting the initial temperature to zero (i.e., T=0).

(iii) *HC(Greedy)*: In *HC(Greedy)*, a greedy solution is constructed first and then used as the starting point in the search.

(iv) *HC(Random)*: The *HC(Random)* uses a randomly generated solution as the initial solution to run the algorithm.

III. Evaluation and Results

The dataset used for conducting our experiments with the optimization model were based on a simulation testbed.

TABLE I
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	$T_{i+1} = T_0 + (A - T_0)$

We randomly generated problem instances of different sizes and densities. The instances generated were tailored on the instances widely cited in literature (e.g., the OR benchmark Library [14]. The benchmark format was transformed to a multiobjective case by associating each component with two different profit values: isolation values and the average number of requests [15].

The specification of the machine used for experiments is summarised as follows: SAMSUNG Laptop installed with Intel(R) CORE(TM) i7-3630QM running Windows 8.1 operating system at 2.40GHZ, with 8GB memory and 1TB swap space on the hard disk. The experimental parameters are shown in Table 2. The workload associated with each instance is used to run the algorithm as shown in the table.

In our experiments, we test the applicability and effect of the different variants of the metaheuristics in driving the optimalSoln algorithm. The performance evaluation will be presented in terms of the quality of solution, robustness and computational effort of the optimalSoln algorithm when combined with any of the four different variants of metaheuristics:(i) HC(Random) - Hill climbing with a random solution as the initial start; (ii) HC(Greedy) - Hill climbing with a greedy solution as a starting point; (iii) SA(Random) - Simulated Annealing with a random start; and (iv) SA(Greedy) - Simulated Annealing starting with a greedy solution.

The solutions obtained from running the *optimalSoln* algorithm was compared the optimal solutions obtained by running the *optimalSoln* algorithm with the exhaustive search of a small problem size. The machine used to run the algorithm could not cope with large instances due to limitations in the hardware specification of the machine (i.e., CPU and RAM). As a result of this, we challenged the metaheuristic with small instances (i.e., C(4,5,4)). It was observed that the optimal solutions produced for all workloads were the same when tested on all the four variants of the metaheuristic.

As it were not possible to obtain optimal solutions with large instances (e.g., C(500,20,4)), the results of running the metaheuristic were compared to a target solution as proposed by [10]. In this study, the target solution represents a requirement defined by the decision maker based on domain knowledge

and the quality of the solutions to obtain. This is expressed as:

$$TargetSoln = ((n \times max(I) \times w1) + e) \quad (7)$$

where e is expressed as $0.05 \times (n \times max(Q) \times w2))$, n is the number of groups, *max(I)* is the maximum isolation value, *max(Q)* is the maximum possible number of requests (calculated using the upper limit of the arrival rate), and w_1 and w_2 are the weights assigned to *I* and *Q* respectively. This equation, when used to compute the target solution of C(4,5,4) with arrival rate of 2.7 requests/sec, gives 1219.2, which is very close to the optimal solution. The rest of the experiment were conducted with an arrival rate of 3.9 requests per second.

The simulation was executed for 1000000 functions evaluations so that reliable results would be produced. It is therefore expected that the success rate would be nearly 100% based on the corresponding performance rate as the optimal solution would have converged. As a result of this, the evaluation of the optimization model was expanded to cover scenarios where there is: (i) limitation in available resources or a requirement to optimise resources that are available while providing optimal solutions; and (ii) limitation in the time required to provide optimal solutions, for example, when the algorithm can be run for limited number of iterations (e.g., 1000 iterations).

1) Measuring the Quality of Solutions: The quality of the solutions was measured in terms of the percent deviation from the target solution [10]. As shown in Table II, the standard deviation (STD) 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 instance, the percent deviation for C(500,5,4) is 3.5 when the number of components is 5. But when the number of components is increased to 20, the percent deviation reduces to 1.49. This means that the quality of solutions is dependent on the number of components per group. The more choices of a particular type of component are available, the better the possibility of achieving attaining an optimal configuration. This is especially essential for large open-source projects that are either designed to utilize a large number of components within the cloud-hosted service or can be integrated with several components residing in other locations.

2) Measuring the Robustness of the Solutions: In measuring robustness, we checked how sensitive the solutions are to small deviations in the size of the instances (i.e., from small instances of C(10,20,4) to large instances of C(1000, 20,4)). The lower the number of function evaluations it takes to attain the target solutions, the better the robustness. Having carefully analysed the number of function evaluations (FE) it takes to reach the target solution, it was observed that the function evaluations for SA(Random) and SA(Greedy) was larger than that of HC(Random) and HC(Greedy), especially for large instances. For example, as shown in Table III, it takes 13,169 for SA(Random) to attain the target solution, whereas HC(Greedy) required only 1,986 evaluations. For small instances above C(70,20,4), Table III shows that SA(Greedy) was slightly better than other variants.

This means that the metaheuristics based on hill-climbing were more stable and robust than the other variants based on simulated annealing, especially for large instances; at the same time, the metaheuristics based on simulated annealing showed better stability for small instances.

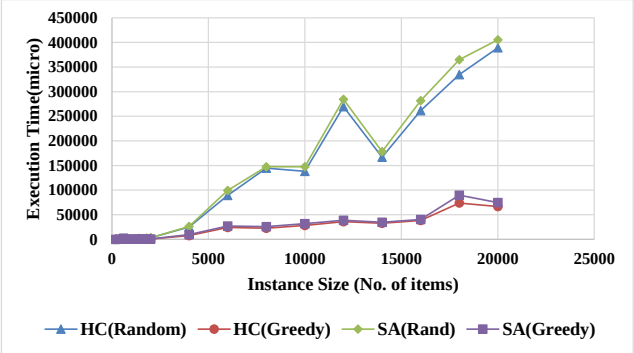

Fig. 2. Computational Effort for a large instance size - C(500,20,4)

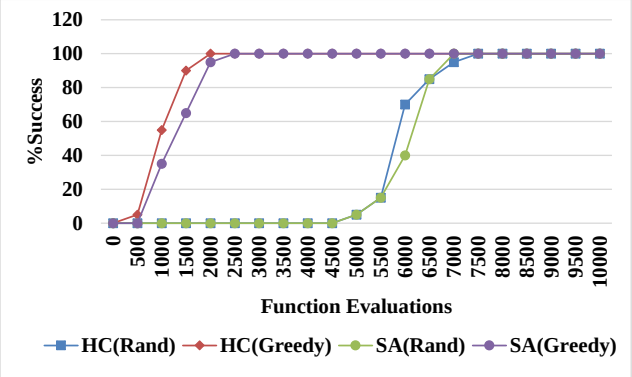

Fig. 3. Run Length Distribution for a large Instance size - C(500,20,4)

3) Measuring the Computational Effort: The computational effort was measured in terms of the estimated execution time required by each variant of the metaheuristics to reach the target solution for different instance sizes. It appears from the experimental studies conducted that the time to compute the initial greedy solutions does not significantly affect the overall execution times for HC(Greedy) and HC(Random). As Figure 2 illustrates, the average number of function evaluations required by the metaheuristics that starts with greedy solutions (i.e., HC(Greedy) and SA(Greedy)) is substantially smaller than those that start with random solutions. Therefore, the variants of the metaheuristics that start with the greedy solution use less computational effort irrespective of the metaheuristic used.

The results of the study can be summarized as follows:
(i) The percent deviation for all metaheuristic variants was nearly the same, although the percent deviation of variants based on greedy solutions was smaller and more stable when applied on large instances;

TABLE II
AVERAGE PERFORMANCE ON DIFFERENT INSTANCE SIZES(M=5; M=20)

Instance Size	HC (rn)	HC (gr)	SA (rn)	SA (gr)	Gr	Instance Size	HC (rn)	HC (gr)	SA (rn)	SA (gr)	Gr
C(10,5,4)	7.64	7.64	7.64	7.64	10.94	C(10,20,4)	1.38	1.38	1.38	1.38	0.17
C(20,5,4)	0.7	0.7	0.7	0.7	9.39	C(20,20,4)	1.98	1.98	1.98	1.98	4.75
C(30,5,4)	1.44	1.44	1.44	1.44	9.35	C(30,20,4)	0.09	0.09	0.09	0.09	6.83
C(40,5,4)	1.34	1.34	1.34	1.34	4.32	C(40,20,4)	1.39	1.39	1.39	1.39	2.18
C(50,5,4)	3.38	3.38	3.38	3.38	11.41	C(50,20,4)	1.4	1.4	1.4	1.4	3.56
C(60,5,4)	1.99	1.99	1.99	1.99	8.11	C(60,20,4)	2.04	2.04	2.04	2.04	2.46
C(70,5,4)	0.96	0.96	0.96	0.96	4.58	C(70,20,4)	1.03	1.03	1.03	1.03	3.68
C(80,5,4)	4.08	4.08	4.08	4.08	8.29	C(80,20,4)	1.36	1.36	1.36	1.36	3.93
C(90,5,4)	1.62	1.62	1.62	1.62	5.28	C(90,20,4)	2.01	2.01	2.01	2.01	4.47
C(100,5,4)	5.03	5.03	5.03	5.03	9.87	C(100,20,4)	2.11	2.11	2.11	2.11	4.09
C(200,5,4)	3.79	3.79	3.79	3.79	8.04	C(200,20,4)	1.48	1.48	1.48	1.48	4.37
C(300,5,4)	5.22	5.22	5.22	5.22	10.7	C(300,20,4)	1.13	1.13	1.13	1.13	3.61
C(400,5,4)	3.7	3.7	3.7	3.7	8.92	C(400,20,4)	1.29	1.29	1.28	1.28	4.19
C(500,5,4)	3.53	3.53	3.53	3.53	8.03	C(500,20,4)	1.49	1.49	1.48	1.48	4.53
C(600,5,4)	3.36	3.36	3.36	3.36	7.7	C(600,20,4)	1.25	1.25	1.25	1.25	4.99
C(700,5,4)	3.78	3.78	3.78	3.78	8.49	C(700,20,4)	1.25	1.25	1.24	1.24	4.7
C(800,5,4)	3.84	3.84	3.84	3.84	8.91	C(800,20,4)	1.43	1.43	1.43	1.43	3.82
C(900,5,4)	3.44	3.44	3.44	3.44	8.05	C(900,20,4)	1.11	1.11	1.11	1.11	4.39
C(1000,5,4)	4.28	4.28	4.28	4.28	8.99	C(1000,20,4)	1.17	1.17	1.16	1.16	4.1
AVG	3.32	3.32	3.32	3.32	8.39	**AVG**	1.39	1.39	1.39	1.39	3.94
STD	1.66	1.66	1.66	1.66	1.89	**STD**	0.44	0.44	0.45	0.45	1.29

TABLE III
FUNCTION EVALUATIONS TO ATTAIN TARGET SOLUTION

Instance	HC(rn)	HC(gr)	SA(rn)	SA(gr)
C(10,20,4)	88	0	97	0
C(20,20,4)	220	102	204	93
C(30,20,4)	613	616	504	2620
C(40,20,4)	361	0	455	0
C(50,20,4)	558	145	459	140
C(60,20,4)	522	0	550	0
C(70,20,4)	884	236	490	262
C(80,20,4)	899	74	940	74
C(90,20,4)	865	103	979	105
C(100,20,4)	1022	0	1019	0
C(200,20,4)	2331	611	2449	816
C(300,20,4)	3679	923	4090	1046
C(400,20,4)	4874	689	4968	788
C(500,20,4)	5763	1055	6154	1217
C(600,20,4)	7416	892	7826	979
C(700,20,4)	8764	1510	9355	1628
C(800,20,4)	8771	1140	9448	1198
C(900,20,4)	11330	2324	12353	2865
C(1000,20,4)	12642	1986	13169	2238

(ii) Metaheuristics which start with greedy solutions reach the 100% success rate considerably faster (see Figure 3) and utilized less execution time than those which started with random solutions.
(iii) There was no significant effect on the robustness and quality of the solutions produced when applied to small instances. For large instance sizes, the variants of the metaheuristics that start with a greedy solution needed fewer function evaluations to finish the search.
(iv) The percent deviation of instances that had more components per group was smaller, thus, leading to a better chance of producing high quality solutions.

Therefore, the discussion of experimental results can be summarized as follows: The benefit of our simulation-based optimization approach is in providing the monitoring, evaluation, adjustment and deployment of cloud-hosted service components (especially for large-scale projects) that guarantees multitenancy isolation when the workload changes. The quality and robustness of optimal solutions produced for largescale cloud-hosted services can be significantly improved when the proposed approached is executed with metaheuristics whose search starts with a greedy solution (compared to random solutions). The solutions from hill-climbing metaheuristics were more stable and robust than that of simulated annealing, particularly for large instances. However, when there is a limitation in terms of time and resources, simulated annealing can produce more robust and stable solutions.

The metaheuristics that start with greedy solutions are more scalable and require fewer function evaluations with reach the target solution compared to those that start randomly. Due to the possibility of encountering components having several interdependencies with other components or services when working on large open-source projects, it is advisable to limit the number of component choices per group, or, better still, to use a combination of local search with greedy strategies.

IV. CONCLUSION AND FUTURE WORK

This paper presents the implementation of a simulation-based approach for providing optimal ways of deploying components designed to use (or be integrated with) a cloud-hosted service, guaranteeing at the same time multitenancy isolation. The main aim of this approach is to address the issue

of multitenancy isolation, whilst optimising the deployment of components that run cloud-hosted services.

The suggested approach works as follows: when a request arrives indicating that there are workload changes, the developed system uses an open multiclass QN model to determine the average number of requests that can access each component, updates the component configuration file with this information, and then applies a metaheuristic to find optimal solutions for deploying components that have the highest degree of isolation, while at the same time maximizing the possible number of component access requests.

The simulation-based study revealed that the *optimalSoln* algorithm (i.e., the main algorithm that drives the model) when combined with metaheuristics that starts with an initial greedy solution (e.g., SA(Greedy)), produces solutions that are robust and of better quality when compared with the metaheuristic that starts with random solutions (e.g., SA(Random)). This suggests that the optimal deployment solutions obtained from randomly generated initial solutions are more sensitive to workload changes than those starting from initial greedy solutions. For large projects, starting the metaheuristics with a greedy solution can boost the general performance. Also, for large instances, when there are time (e.g., real-time and dynamic) and resource constraints, the simulated annealing metaheuristic produces solutions that are more robust and stable compared to those of hill-climbing.

The approach presented in this paper assumes that the resources supporting each component are sufficient to handle all incoming requests. In the event that this condition cannot be guaranteed, we suggest using an elastic queue to control incoming requests. Another option could be to implement some form of admission control mechanism, for example, restricting the number of requests that are handled concurrently by each component, to avoid *overloads* or any degradation in the component's performance. The findings of this study are mostly applicable to components of cloud-hosted applications developed and deployed using a multitenant architecture. The approach and the associated algorithms that have been presented are relevant to cloud-hosted services at the application level, and so are implemented almost at runtime.

We plan to develop other metaheuritics for use with our simulation-based approach to handle larger problem instances. For example, we can integrate other types of metaheuristics into the optimalSoln algorithm or combine simple and more advanced metaheuristics. In addition to the work mentioned in Section II, several researchers have developed metaheuristics that combine genetic algorithm with simulated annealing. For example, the authors in [16] came up with a GA-SA hybrid algorithm for optimization of wideband antenna matching networks, which can potentially be applied in the domain of resource allocation in cloud computing.

In future, we plan to continue this research and develop a Decision Support System (DSS) based on the proposed approach to semi-automatically suggest near-optimal solutions for deploying components of a cloud-hosted application, which guarantees multitenancy isolation. Some work has already been done in this area [1], but without addressing the issue of multitenancy isolation. We intend to base the DSS on the simulation model described in this paper for investigating and predicting how components and/or cloud-hosted services will react to workload changes at runtime. In particular, we are thinking of designing a rule-based system to specify how a new set of components can be selected for deployment when the average utilization of either certain components or of the whole system exceeds a defined threshold. The same method can be applied for the case of the request arrival rate exceeding a specified limit. Such decisions can be of benefit and help in long-term investments in resource provision and in estimating the running cost of components and cloud services.

ACKNOWLEDGMENT

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

REFERENCES

[1] N. Tankovic, T. Grbac, and M. Zagar, "Elaco: A framework for optimizing software application topology in the cloud environment," *Expert Systems With Applications*, vol. 90, pp. 62–86, 2017.

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

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

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

[5] L. Ochei, A. Petrovski, and J. Bass, "Optimizing the deployment of cloud-hosted application components for guaranteeing multitenancy isolation." IEEE Conference Publications, 2016, pp. 77 – 83, 2016 International Conference on Information Society (i-Society 2016).

[6] L. Teylo, U. De Paula, Y. Frota, D. De Oliveira, and L. Drummond, "A hybrid evolutionary algorithm for task scheduling and data assignment of data-intensive scientific workflows on clouds," *Future Generation Computer Systems*, vol. 76, pp. 1–17, 2017.

[7] G. Anastasi, E. Carlini, M. Coppola, and P. Dazzi, "Qos-aware genetic cloud brokering," *Future Generation Computer Systems*, vol. 75, pp. 1–13, 2017.

[8] J. Zhang, H. Huang, and X. Wang, "Resource provision algorithms in cloud computing: A survey," *Journal of Networks and Computer Applications*, vol. 64, pp. 23–42, 2016.

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

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

[11] E. K. Karasakal and M. Köksalan, "A simulated annealing approach to bicriteria scheduling problems on a single machine," *Journal of Heuristics*, vol. 6, no. 3, pp. 311–327, 2000.

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

[13] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 6, 2007.

[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] 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.

[16] A. Chen, T. Jiang, Z. Chen, and Y. Zhang, "A genetic and simulated annealing combined algorithm for optimization of wideband antenna matching networks," *International Journal of Antennas and Propagation*, vol. 2012, 2012.