Ochei *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2019) 8:1
https://doi.org/10.1186/s13677-018-0124-5

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Optimal deployment of components of cloud-hosted application for guaranteeing multitenancy isolation

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

Abstract

One of the challenges of deploying multitenant cloud-hosted services that are designed to use (or be integrated with) several components is how to implement the required degree of isolation between the components when there is a change in the workload. Achieving the highest degree of isolation implies deploying a component exclusively for one tenant; which leads to high resource consumption and running cost per component. A low degree of isolation allows sharing of resources which could possibly reduce cost, but with known limitations of performance and security interference. This paper presents a model-based algorithm together with four variants of a metaheuristic that can be used with it, to provide near-optimal solutions for deploying components of a cloud-hosted application in a way that guarantees multitenancy isolation. When the workload changes, the model-based algorithm solves an open multiclass QN model to determine the average number of requests that can access the components and then uses a metaheuristic to provide near-optimal solutions for deploying the components. Performance evaluation showed that the obtained solutions had low variability and percent deviation when compared to the reference/optimal solution. We also provide recommendations and best practice guidelines for deploying components in a way that guarantees the required degree of isolation.

Keywords: Cloud-hosted application, Multitenancy, Degree of isolation, Queuing network, Metaheuristic, Component, Cloud deployment, Optimization, Decision support system

Introduction

Multitenancy is an essential cloud computing property. Multitenancy is a software architecture where one instance of a cloud offering is used to serve multiple tenants and/or components [1, 2]. One of the challenges of implementing multitenancy is how to ensure that there is isolation between multiple components of a cloud-hosted application when one of the components experiences high load [1, 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 (e.g., in a case where there are strict laws and regulations preventing them from being shared), it implies duplicating the components for each tenant, which leads to high resource consumption and running cost. Overall, this will limit the number of requests allowed to access the components. It may also be that a low degree of isolation is required for a component, for example, to allow sharing of the component's functionality, data, and resources. This would reduce resource consumption and running cost, but the performance of other components may possibly be affected when one of the components experiences a change in workload.

Therefore, in order to optimize the deployment of components, the architect has to satisfy two objectives: maximize the degree of isolation between components and at the same time maximize resource sharing (especially the number of requests that can be allowed to access the component). In other words, we have to resolve the trade-off between a lower degree of isolation versus the possible influence that may occur between

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

components or a high degree of isolation versus the challenge of high resource consumption and running cost of the component. This is a decision-making problem that requires an optimal decision to be taken in the presence of a trade-off between two or more conflicting objectives [4, 5].

Motivated by this problem, this paper presents a model-based algorithm together with a metaheuristic technique 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, while at the same time allowing as many requests as possible to access the components. We implemented our model-based algorithm by first solving 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.

We evaluated our approach by comparing the solutions obtained from our approach with the optimal results obtained from an exhaustive search of the entire solution space for a small problem. The research question addressed in this paper is: **"How can we provide near-optimal solutions for deploying components of a cloud-hosted application in a way that guarantees multitenancy isolation."**. To the best of our knowledge, this study is the first to present a model-based algorithm, an optimization model and a metaheuristic solution to provide a near-optimal solution for deploying components of a cloud-hosted application in a way that guarantees multitenancy isolation.

Most related work looks at this problem from the perspective of the cloud provider (i.e., SaaS, PaaS or IaaS). Examples include the autoscaling algorithms offered by IaaS providers like *Amazon* and optimisation models proposed for use by SaaS providers such as *Salesforce.com* [6]). However, this paper looks at the problem from the perspective of the tenant who owns software components and is responsible for configuring them to design and deploy its own cloud-hosted application on a shared cloud platform whose provider does not control these components.

This paper extends and expands on the previous work conducted by Ochei et al. [7]. We summarise the additions to the previous work as follows. Firstly, we extend our previous work by presenting mathematical equations both for the optimization model and the open multiclass Queuing network models. For the open multiclass model, we provide mathematical equations for calculating the average number of requests that can be allowed to access the components and the entire cloud-hosted system. Secondly, we present a new algorithm that integrates the optimization model and QN model to provide optimal solutions for deploying components of a cloud-hosted service for guaranteeing multitenancy isolation. Thirdly, the experiments presented in this paper have been expanded and strengthened by (i) increasing the size and dimension of the datasets; (ii) increasing the number of runs (and iterations for each run) required for each experimental scenario; (iii) incorporating a rigorous analysis of the results using a two-way ANOVA model. Fourthly, a new methodology section has been added to explain how a modelling and simulation technique is used in the study and how it fits into the overall research goal of providing a framework to architect the optimal deployment of components of a cloud-hosted service in order to guarantee the required degree of multitenancy isolation. Fifthly, the paper also includes more related work, examples, explanations and areas where our model can be applied.

The main contributions of this paper are:

1. Creating a novel model-based algorithm, *optimalDep*, that combines (i) an open multiclass QN model; and (ii) an optimization model, to provide a near-optimal solution for deploying components of a cloud-hosted application with guarantees for multitenancy isolation.

2. Presenting a novel system architecture, *optimalArch*, for transforming the model-based algorithm into a decision support system for architecting the deployment of components of cloud-hosted services for guaranteeing multitenancy isolation.

3. Developing four variants of a metaheuristic which are based on a *simulated annealing* algorithm and *hill climbing* for solving the optimization model. The four variants were extensively evaluated and compared.

4. Presenting recommendations and best practice guidelines based on the experience gathered from extensive evaluation and comparison of the algorithms.

The rest of the paper is organized as follows: "Near-optimal deployment of components for guaranteeing multitenancy isolation" section first presents a motivating example and then highlights the challenges pointed to by the motivating example. It also discusses related work on multitenancy isolation and optimal deployment of cloud-hosted services. "Methodology" section presents the methodology used in the study. "Problem formalization and notation" section formalizes the optimization problem. "Open multiclass queuing network model" section discusses the open multiclass queuing network model, while "Metaheuristic search" section discusses the metaheuristic. "Model-based algorithm for optimal deployment of components" section presents a simulation-based algorithm, *optimalDep*, together with a system architecture that can be used to transform the algorithm into a decision support system. "Evaluation" section presents the evaluation methodology of the study, "Results" section presents the results, while "Discussion" section discusses the results. Recommendations and limitations of the study are detailed in "Limitations of the study" section. "Conclusion and future work" section concludes the paper with future work.

Near-optimal deployment of components for guaranteeing multitenancy isolation

As applications and services are increasingly being deployed to the cloud to be used by multiple tenants/users, there is a need to isolate tenants, processes, and components, and thus implement multitenancy. Multitenancy architectures are typically used for deploying components of cloud-hosted services for multiple tenants/users. This is based on the assumption that tenants share resources as much as possible which leads to a reduction in resource consumption and running cost per tenant. Overall this makes it feasible to target a wider audience as more resources would be made available [1]. Before continuing our discussion, let us consider the following scenario which captures a similar thought process, and serves to elaborate more on our motivation.

Motivating example

Let us assume that there are multiple components of the same tenant on the same underlying cloud infrastructure. A component is simply an encapsulation of functionality or resource that is shared between multiple tenants (e.g., message queue, database). A tenant in this context represents 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. These components which are of different types and sizes are required to integrate with or designed to use a cloud-hosted application for deployment in a multitenant fashion. The components may also be categorised 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 (see Fig. 1). 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 to some tenants or group of tenants. Each application component requires a certain amount of resources of the underlying cloud infrastructure to support the number of requests it receives. In the next section, we will highlight several significant problems pointed to by the motivating scenario, which will be addressed subsequently in this paper.

Challenge of implementing the required degree of multitenancy isolation on the cloud

There are several design questions and significant problems pointed to by the motivating scenario, which tenants face with respect to the sharing of the components with other tenants.

1. Firstly, the motivating scenario highlights a key design question when implementing multitenancy which is how to ensure that the performance demands and resource consumption of one tenant does not adversely affect another tenant. When a cloud customer is not guaranteed a certain service level, possibly as a result of the activities of other tenants, then customers may not be willing to access the cloud-hosted application. Instead, the customer may decide to move to a different cloud provider which may even be at a higher cost in order to get a better service. For a cloud provider, this may mean losing customers and revenue, while for the cloud customer it may mean a waste of time and resources.
2. Secondly, the motivating scenario points to the fact that software architects need to be able to control the required degree of isolation between tenants sharing components of a cloud-hosted application. This is an important issue in a cloud environment since varying degrees of tenant isolation are possible, depending on the type of component being shared, the process supported by the component and the location of the component on the cloud application stack (i.e., application level, platform level, or infrastructure level). At the very basic degree of multitenancy, tenants would be able to share application components as much as possible which translates to increased utilisation of underlying resources. However, while some application components may benefit from a low degree of isolation between tenants, other components may need a higher degree of isolation because the component may either be too critical or not shareable due to certain laws and regulation. For example, there is growing evidence that many cloud providers are unwilling to set data centres in mainland Europe because of tighter legal requirements that disallow the processing of data outside Europe [8, 9]. This requirement will traverse down to the IaaS level, and customers must take this into consideration if intending to host applications outsourced to such cloud providers [1].
3. Thirdly, the motivating scenario highlights that depending on the required degree of isolation, there are fundamental trade-offs that would have to be taken into consideration when deploying components of a cloud-hosted service. Many cloud providers (e.g., Amazon and Microsoft) do not guarantee isolation

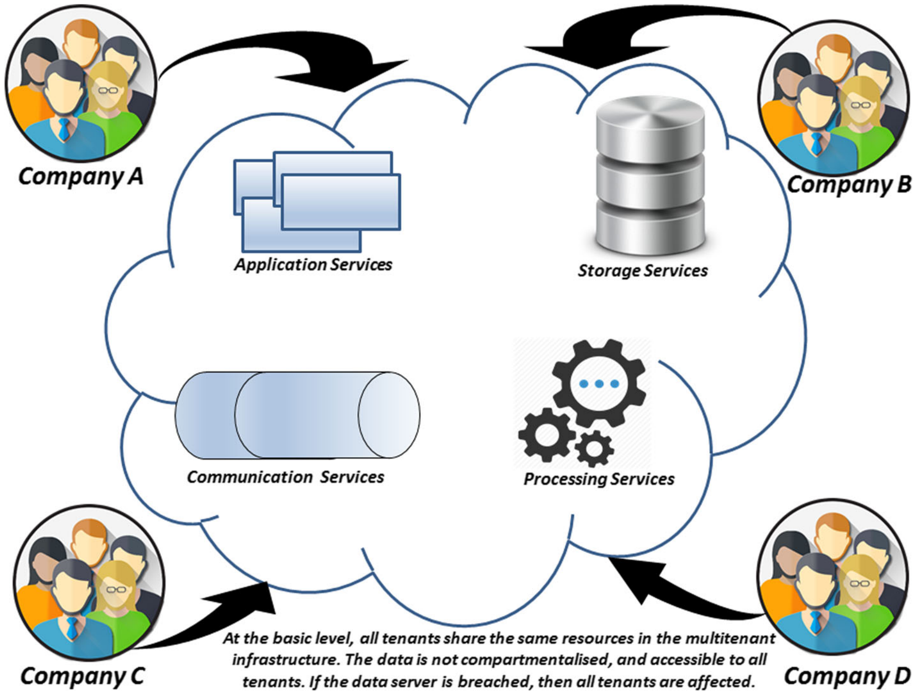

Company A
Company B
Company C
Company D
Application Services
Storage Services
Communication Services
Processing Services
At the basic level, all tenants share the same resources in the multitenant infrastructure. The data is not compartmentalised, and accessible to all tenants. If the data server is breached, then all tenants are affected.

Fig. 1 Deploying Components of a Cloud-hosted Service in a Multitenant Infrastructure

and availability for a single component (e.g., disk) and but only for the whole system. This re-enforces the need to automate the monitoring and management of components of cloud-hosted services to guarantee multitenancy isolation.

In order to optimise the deployment of components, the architect has to resolve the trade-off between a lower degree of isolation versus the possible influence that may occur between components or a high degree of isolation versus the challenge of high resource consumption and the running cost of the component. This is a decision-making problem that requires an optimal solution to be determined in the presence of a trade-off between two or more conflicting objectives [4, 5]. In order to resolve this trade-off, we can model the problem as a multi-objective optimization problem. Many multi-objective optimization problems result in a trade-off situation that involves losing some quality of one objective function in return for gaining quality in some of the other objective functions [4, 5, 10].

Examples of real scenarios with components of a cloud-hosted service

There are several real-life scenarios where our approach can be applied. A typical scenario is a multitenant cloud-hosted streaming system deployed using Apache Kafka-base subsystem integrated with Cassandra backend for streaming data for the forthcoming Olympics Games to cater and produce live data (e.g., past games, competitions schedule/draws for upcoming games or match analysis from favourites sports analysts) for various websites. The system allows tenants (i.e., Producers) to generate data to persist their data in real-time in an Apache Kafka Topic (regarded as components). Any of these Topics can then be read by any number of tenants (i.e., Consumers) who have subscribed for the data in real-time. Furthermore, as this particular Kafka component is reusable for additional websites, other tenants can register their components as listeners and use the Cassandra for accessing the archived data [11].

Streaming applications usually require varying degrees of isolation. This is because a large number of components may be published in various locations on the cloud system, conflicts can occur with other previously installed components or when components are removed [12]. Therefore, the approach presented in this paper can be used to provide an optimal deployment solution that guarantees the required degree of isolation for components shared between different tenants by intercepting all published components and subscribed components and re-distributing them to alternate brokers(servers).

Another interesting scenario is that of a tenant developing automated build verification and testing infrastructure using a Jenkins-based continuous integration system to support the development of a large-scale software project. For example, the continuous integration system can be configured to poll one or more directories and starts a build if certain changes are detected within those directories [13, 14].

In such a large project, it is expected that multiple builds will interact with multiple components to create several dependencies and supported behaviour with each other thereby making builds difficult and complex. Complex and difficult builds are those that are composed of a vast number of modular components including different frameworks, components developed by different teams or vendors, and open source libraries [13].

In this type of situation, it is essential that components designed to use (or integrate with) such a cloud-hosted service are made shareable to other tenants as well provided the initiating tenant gives its consent. This also makes it easy to implement the required degree of isolation between tenants.

Related work on multitenancy isolation and optimal deployment of cloud-hosted services

Cloud computing refers to both the applications delivered as a service over the Internet and the hardware and systems software in the data centers that provide those services [15]. The cloud could either be a *public cloud* (that is, cloud is provided in a prepaid manner to the general public), *private cloud* (that is, internal IT infrastructure of an organization is inaccessible to the general public), or a *hybrid cloud* (that is, the computing ability of the private cloud is boosted by the public cloud).

There are three basic cloud service models: *(i) Software as a Service (SaaS):* cloud providers can install, operate and access their application software using a web browser, thus eliminating the need for customers to run and install the application on their own computers (e.g., Salesforce delivering customer relationship management software); *(ii) Platform as a Service (PaaS):* cloud providers deliver cloud platforms which represent an environment for application developers to create and deploy their applications. (e.g., Google App Engine); and *(iii) Infrastructure as a Service (IaaS):* cloud providers offer physical (computers, storage) and virtualized computer resources (e.g., Amazon EC2, and Azure Services Platform).

Related work on multitenancy isolation

One of the challenges of implementing multitenancy on the cloud is how to enable the required degree of isolation between multiple components of a cloud-hosted application (or tenants accessing a cloud-hosted application). We refer to this as multitenancy isolation. In the context of this paper, 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 being accessed by multiple tenants when the workload changes*" [16, 17].

The varying degrees of multitenancy isolation can be captured in three main cloud deployment patterns: (i) shared component (i.e., tenants share the same resource instance, and are unaware of other tenants); (ii) tenant-isolated component (i.e., tenants share the same resource and their isolation is guaranteed); and (iii) dedicated component (i.e., tenants do not share resources, though each tenant is associated with one instance or a certain number of instances of the resource) [1]. The shared component represents the lowest degree of isolation between tenants while the dedicated component represents the highest. The degree of isolation between tenants accessing a tenant-isolated component would be in the middle. There are several approaches to implementing multitenancy that have been widely discussed in the literature. Multitenancy can be implemented at different layers of the cloud stack: the application layer, the middleware layer, and data layer. For example, in [18], the author discusses several approaches for implementing multitenancy in the application tier and data tier.

Multi-tenancy can also be realised at the PaaS level so that service providers can offer multiple tenants customizable versions of the same version for consumption by their users. The authors in [19] discussed how to implement multitenancy at the PaaS (or middle tier) of an application/cloud stack. In this work, the requirements for multitenancy in an Enterprise Service Bus (ESB) solutions, a key component in service-oriented architecture (SOA), were identified and discussed as part of the PaaS model. An implementation-agnostic ESB architecture was proposed whereby multitenancy can be integrated independently of the implementation into the ESB.

In [20], several approaches for implementing multitenancy are discussed and more importantly suggest that customization is the solution to addressing the hidden constraints on multitenancy such as complexities, security, scalability, and flexibility. The author in [21] presents a qualitative discussion of different approaches to implementing multi-tenant SaaS offerings, while the author in [22] discusses the advantages and disadvantages of multitenancy in SaaS offerings. They both agree that a plugin is the solution to true multitenancy and that most of the available options for implementing multitenancy to some extent require a re-engineering of the cloud service.

Several work of literature acknowledges that there could be varying degrees of isolation between tenants. In [23], three approaches to managing multi-tenant data are discussed. Chong et al. state that the distinction between the shared data and isolated data is more of a continuum, where many variations are possible between the two extremes. The authors in [24] explore key implementation patterns of data tier multi-tenancy based on different aspects of isolation such as security, customization, and scalability. For example, under the resource tier design pattern, the authors identified the following patterns: (i) totally isolated (dedicate database pattern); (ii) partially shared (Dedicate table/schema pattern); and (iii) totally shared (Share table/schema pattern). These patterns are similar to the shared component, tenant-isolated component and dedicated component patterns at the data tier, respectively [1]. The author [25] describes three forms of database consolidation which offers differing degrees of inter-tenant isolation as follows: (i) multiple application schemas consolidated in a single database, multiple databases hosted on a single platform; and (iii) a combination of both.

The authors [26] describe how the services (or components) in a service-oriented SaaS application can be deployed using different multi-tenancy patterns and how the chosen patterns influence the customizability, multi-tenant awareness, and scalability of the application. These patterns are referred to as a single instance, single configurable instance and multiple instances. Although this work describes how individual services of a SaaS application can be deployed with different degrees of customizability, we believe that these concepts are similar to different degrees of isolation between tenants.

The three main aspects of multitenancy isolation are performance, stored data volume, and access privileges. For example, in performance isolation, other tenants should not be affected by the workload created by one of the tenants. Guo et al. evaluated different isolation capabilities related to authentication, information protection, faults, administration, etc. [27]. Bauer and Adams discuss how to use virtualization to ensure that the failure of one tenant's instance does not cascade to other tenant instances [3].

In the work of Walraven et al., the authors implemented a middleware framework for enforcing performance isolation [28]. They used a multitenant implementation of a hotel booking application deployed on top of a cluster for illustration. Krebs et al. implemented a multitenancy performance benchmark for web application based on the TCP-W benchmark where the authors evaluated the maximum throughput and the number of tenants that can be served by a platform [29]. Another related work is [30] where the authors defined a feature-based cloud resource management model where applications are composed of feature instances using a service-oriented architecture. The approach produced a higher degree of multitenancy for the scenarios considered when the relationships between the features are taken into account using the application-oriented placement.

Related work on optimal deployment of cloud-hosted services

Research work on optimal deployment and allocation of cloud resources on the cloud are quite significant. However, there has been little or no work on providing an optimal solution for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation. In [31], the authors used an evolutionary algorithm to minimize resource consumption for SaaS providers and improve execution time. The authors in [32, 33] used a multitenant SaaS model to minimize the cost of cloud infrastructure. Heuristics were not used in this work. The authors in [34] developed a heuristic for capacity planning that is based on a utility model for the SaaS. This utility model mainly considers the business aspects related to offering a SaaS application with the aim of increasing profit.

In [35], 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. [36], considered how to evaluate the optimal distribution of application components among virtual servers. A closely related work to ours is that of Aldhalaan and Menasce [6], 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 SLA constraints.

Related work on multitenancy isolation has largely focused on isolation at the data tier [37]. The main aspect of isolation is usually performance isolation. For example, the authors in [38] mainly focus on performance isolation in a multitenant application in the cloud. The varying degrees of multitenancy isolation based on multitenancy patterns and the different aspects of isolation are described in [39].

Most work on optimal deployment and allocation of cloud resources on the cloud focuses on minimising the cost of using the cloud infrastructure resources [31]. Previous work concerning optimization of cloud resources does not use heuristic at all, although a few use simple heuristics. For example, the authors in [6, 40] used a heuristic based on hill climbing for minimising the cost of SaaS cloud providers with response time SLAs constraints. This study, unlike others, focuses on providing an optimal solution for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation.

Methodology

In this section, we will first present the methodology (i.e., modelling and simulation technique) used in this paper and thereafter discuss how it fits into the overall research goal of providing a framework to architect the optimal deployment of components of a cloud-hosted service in order to guarantee the required degree of multitenancy isolation.

Modelling and simulation for optimal deployment of cloud-hosted services

A model represents the key characteristics, behaviours, and functions of a selected physical or abstract system or process. A simulation is an imitation of the operation of a real-world process or system [41]. The architectures (or architectural patterns) used to deploy services to the cloud are of great importance to software architects, because they determine whether or not the system's required quality attributes (e.g., performance) will be exhibited [42]. Cloud patterns represent a well-defined format for describing a suitable solution to a cloud-related problem, for example, describing the cloud and its properties, and how to deploy and use various cloud offerings [1].

In a cloud environment, it is very difficult (if not impossible) to do a purely top-down architectural design of a large and complex cloud-hosted system: there are too many considerations (e.g., deployment parameters and deployment scenarios) to weigh at once and it is too hard to predict all of the relevant technological barriers. Requirements may change in dramatic ways or a key assumption may not be met. For example, there are cases where a cloud providers API did not work as specified or where an API exposing a critical function was simply missing or inaccessible.

Therefore the focus of this study is to model a real-life system (i.e., degrees of isolation between components of a cloud-hosted service) so that it can be studied to see how the system works. By changing variables in the simulation, we can turn several architectural parameters into values that can be easily measured to resolve conflicting trade-offs and, thus, implement a system that guarantees the required degree of isolation between tenants. Specifically, this study models a real cloud-hosted service that has multiple components that can be deployed to multiple tenants in a way that would guarantee the required degree of isolation between tenants. Thereafter, we performed simulations experiments on the model by assuming a large-scale cloud-hosted service with different variables and cloud deployment scenarios.

As shown in Fig. 2, the modelling and simulation study is carried in an iterative fashion whereby the system to study becomes the optimised system which then becomes the real cloud-hosted service under study and the cycle repeats. The study starts by using the information gained from case studies of real cloud-hosted software tools and then a synthesis of findings of the case studies (using cross-case analysis technique) to formulate and develop a model. The next step is to develop a simulation program by relying on the model-based algorithm. Simulation experiments were conducted which produces results for analysis. Finally, the conclusions and recommendations were made to provide guidelines for architecting the development of cloud-services for guaranteeing multitenancy isolation.

It is important to note that this study describes a component which is fed with data obtained from realistic experiments conducted with real cloud-hosted software development tools. We previously conducted separate case studies to empirically evaluate the degree of tenant

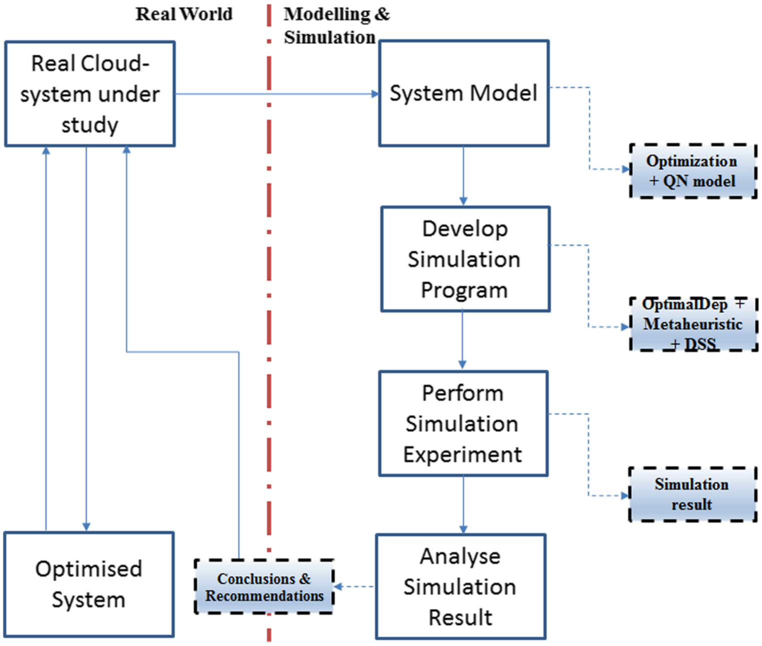

Real World
Modelling & Simulation
Real Cloud-system under study
System Model
Optimization + QN model
Develop Simulation Program
OptimalDep + Metaheuristic + DSS
Perform Simulation Experiment
Simulation result
Optimised System
Conclusions & Recommendations
Analyse Simulation Result

Fig. 2 Modelling and Simulation technique used for the study

isolation in three cloud-hosted software development process tools: continuous integration (with Hudson), version control (with File SCM Plugin and Subversion), and bug/issue tracking (with Bugzilla) [16, 17, 43].

Overall methodology to architect the deployment of cloud-hosted services to guarantee multitenancy isolation

The overall research process for architecting the deployment of cloud-hosted services for guaranteeing multitenancy isolation is shown in Fig. 3. It starts from the exploratory study on cloud multitenancy patterns and selection of the cloud-hosted Global Software Development tools and processes up to modelling and simulation.

At first, we carried out an empirical study to find out the type of software processes and the supporting tools used in Global Software development projects and also explore the different cloud deployment patterns for deploying these tools and support services to the cloud. Thereafter, we conducted separate case studies to empirically evaluate the degree of tenant isolation in three cloud-hosted software development tools (and associated processes): continuous integration (with Hudson), version control (with File SCM Plugin and Subversion), and bug/issue tracking (with Bugzilla). The case studies allowed us to investigate a contemporary software engineering phenomenon (that is, the effect of varying degrees on tenant isolation on cloud-hosted services) within its real-life context using real-world Global Software Development (GSD) tools deployed in a cloud environment [44].

Based on the information derived from the cases studies, we used modeling and simulation methodology to first create a model that represents a cloud-hosted service being accessed by multiple tenants that require varying degrees of isolation. Thereafter, we performed simulation experiments on the model by assuming a large-scale project size with different variables and cloud deployment scenario.

The validation approach used follows widely accepted guidelines for evaluating experimentally a metaheuristic and comparing metaheuristics in a rigorous way. The guidelines entail selecting the performance measures and indicators to compute: solution quality, computation effort, and robustness. After the experimental results are obtained for different indicators, methods from statistical analysis (ANOVA) is used to conduct the performance assessment of algorithms/metaheuristics [45].

Problem formalization and notation

This section formalises the problem and then describes how it is mapped to a Multichoice Multidimensional Knapsack Problem (MMKP).

System model and description of the problem

The motivating example presented in "Motivating example" section describes the problem faced by a software architect when selecting optimal components of a cloud-hosted application for deployment. Assuming that one of the components of the cloud-hosted application experiences a very high load, 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 from each other; and (ii) maximizes the number of requests allowed to access the component (and the

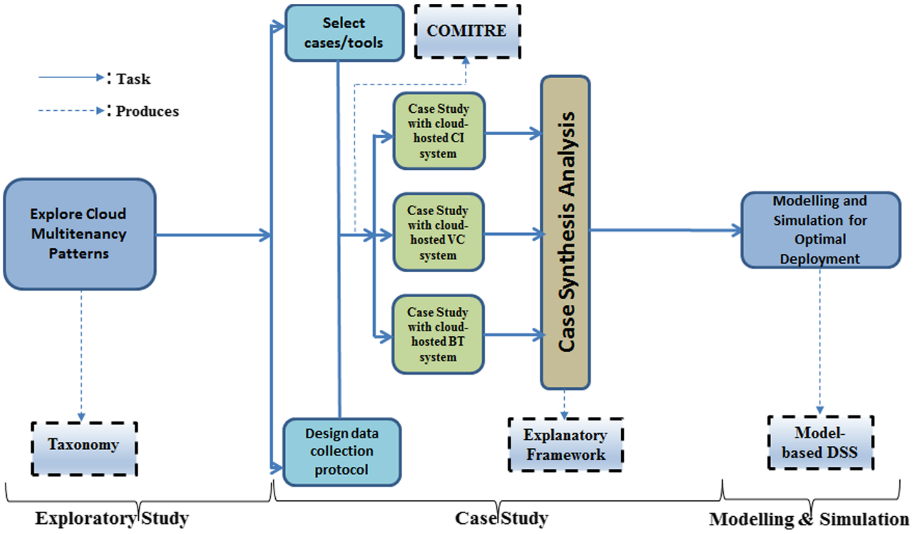

Task
Produces
Explore Cloud Multitenancy Patterns
Taxonomy
Select cases/tools
COMITRE
Case Study with cloud-hosted CI system
Case Study with cloud-hosted VC system
Case Study with cloud-hosted BT system
Design data collection protocol
Case Synthesis Analysis
Explanatory Framework
Modelling and Simulation for Optimal Deployment
Model-based DSS
Exploratory Study
Case Study
Modelling & Simulation

Fig. 3 Overall methodology for architecting the deployment of cloud-hosted services for guaranteeing multitenancy isolation

application as a whole) without having the total resources used to exceed the available resources (see Fig. 4).

System notations and assumptions

The following notations (Table 1) and assumptions are used in this study.

1. *Component of a cloud-hosted service:* A component of a cloud-hosted service is an encapsulation of functionality or resource that is shared between multiple tenants. An application component could be a communication component (e.g., message queue), data handling component (e.g., databases), or processing component (e.g., load balancer), hardware (e.g., virtual server) or microservice (e.g., Amazon Web Services Lambda which is a responsive cloud service used to inspect actions within an application and responds to events and also consumes resources only when an event actually occurs). Tenants interact differently with the components depending on the level in which the component is located in cloud application stack (i.e., SaaS, PaaS or IaaS layer). Each component is associated with six parameters: the isolation value, the number of requests allowed to access the component, and resource consumption for CPU, RAM, Disk and Network bandwidth.
2. *Tenant and Multi-tenant:* This study extends the notion of a tenant and multitenant from a single user/customer to a team, the department of a software company or software company, whose responsibility is to build or maintain a cloud-hosted application and their supporting processes with various components. This study assumes a Business-to-Consumer (B2C) multitenancy model rather than the Business-to-Business (B2B) model. In B2B cloud services, a tenant is likely to be an account with several individual users. In contrast, for B2C cloud services, a tenant is a single user or a group of users(e.g., a team of developers).
3. *Component Group:* A component group is a collection of components (e.g., database components, virtual servers) with a common functionality or purpose but with different configurations and hence different resource consumption requirements. A typical example of a component group can be seen in a Kafka-based application[1], where consumers label themselves with a consumer group name so that each record published to a topic is delivered to one consumer instance within each subscribing consumer group.
4. *A cloud-hosted service/application:* A cloud-hosted service/application is made up of different interacting micro-services where each micro-service is regarded as a component. These components are used to integrate with or designed to use a cloud-hosted service to serve multiple users.
5. *Utilizing the Model-based Algorithm for Decision Support Guidance:* This study presents a model-based algorithm for providing near-optimal solution to deploy components of a cloud-hosted application with guarantees for multitenancy isolation. With the system architecture provided in "Model-based algorithm for optimal deployment of components" section, this model can be transformed into a decision support system and utilized for some decision support guidance to help developers and

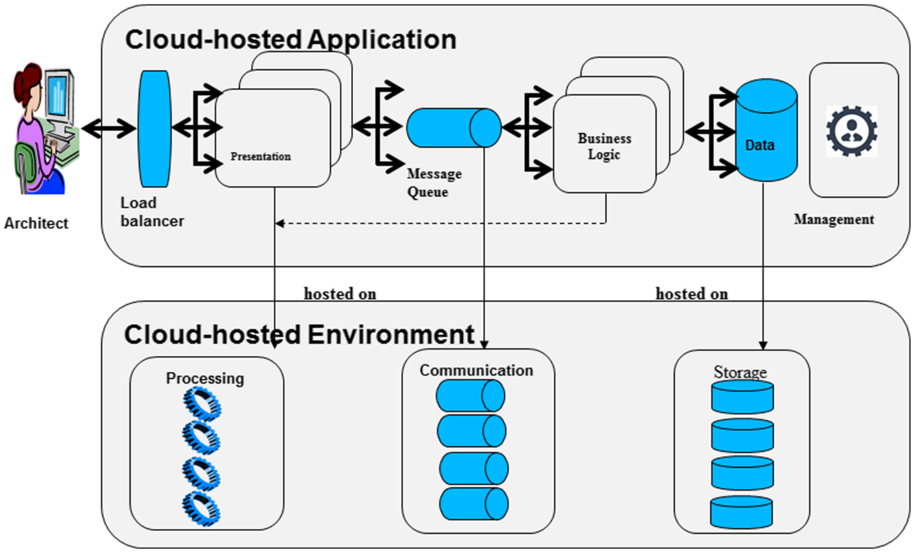


Fig. 4 System Model of a Cloud-hosted Service with multiple groups of components

Table 1 Notations and mapping of multitenancy problem to QN model and MMKP

Notation	Multitenancy isolation problem	MMKP	QN model
N	Total number of groups of components	Total number of groups of objects	Total number of classes
I	Total number of items in a group	Total number of objects in a group	-
m (K is used for QN)	Total number of resources	Total number of resources	Total number of service centers
α (k is used for QN)	Index for resource supporting a component	Index for resource supporting a component	Index for service centers
i (c is used for QN)	Index value for the Group	Index value for the Group	Index value for the Class
j	Index value for the component	Index value for the object	-
a_{ij}	A component which is associated with isolation value, number of requests, cpu, ram, disk and bandwidth size	Objects in a group	-
c	Group of component $(c_1, ..., c_N)$	Group of objects	Class
r_{ij}^{α}	Resource consumption of each component	Resources required by the object in the knapsack	Service centres in the system (cpu, ram, disk, bandwidth)
R	Limit of each resource supporting each component (R(=1,m))	Resources available in the knapsack (knapsack capacity)	System/Component capacity
I_{ij}	Isolation value for a component.Used to compute G.	-	-
Q_{ij}	The number of requests allowed to access a component. Used to compute G	-	The queue length of class c at centre k,
$D_{c,k}$	The service demands at the cpu,ram,disk, and bandwidth	-	Service demand of class c at k service centres (cpu, ram, disk, bandwidth)
λ_{ij}	Workload on the component (arrival rate of request to the component/system)	-	Workload on the component (arrival rate of request to the component/system)
g_{ij}	Optimal value for one component in a group	Profit of one object in MMKP	-
$\mathcal{G}$	Optimal function of the solution	Profit of the solution in MMKP	-

cloud deployment architects to determine the appropriate degree of resource sharing and performance isolation between tenants.

6. *Provision of Middleware Support by cloud provider:* There is need for a middleware support, for example, middleware API provided by the cloud provider for discovering common components and making these components accessible to other tenants. For example, the cloud provider can either provide an API to tenants to directly retrieve the service demands of the resources supporting the components or provide access to such an API for the service demands to be easily measured. This information is required to compute the maximum number of requests that can be allowed to access the components.

   Furthermore, both cloud provider and tenants can share the cost reduction benefits of multitenant components. Where tenants give their consent that resource sharing is possible then cloud providers can lower their hosting costs assuming the tenant is willing to provide component details.

7. *Constraints to support sharing of components:* The components have to meet certain constraints in order to support sharing with other independent tenants. Specifically, the components have to adhere to the principles of reusable component software applied by the tenants. For example, one of the requirements from tenants would be that the service demands of the resources supporting the components can be easily measured [46].

Optimal function of the problem

As stated in the "System model and description of the problem" section, there are two objectives in the problem. An aggregation method is used 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 request) into a single objective function (i.e., $\mathcal{G}$=optimal function) in a linear way. The particular aggregation strategy used is the *priori single weight* strategy which consists of defining the weight vector to be selected based on a combination of the decision makers' preferences and domain knowledge of the problem [45]. This approach has been widely used in literature for metaheuristic such as genetic algorithm and simulated annealing [47, 48].

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 parameters including the degree of isolation, the average number of requests allowed to access the component, and the penalty for solutions that violate the constraints.

Definition 1 (Optimal Function): *Given an isolation value of a component I, and the average number of request Q, that can be allowed to access the component, then optimal function $\mathcal{G}$ is defined as:*

$$g_{ij} = (w1 \times I_{ij}) + (w2 \times Q_{ij}) - (w3 \times P_{ij}) \tag{1}$$

The penalty, P, for violating constraints of a component of the cloud-hosted service is:

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

where w1,w2,w3 are the weights for the isolation value, number of requests and penalty. The values assigned to w1, w2 and w3 are: w1=100; w2=1 and w3=0.1. The weights are chosen based on problem-specific knowledge so that more importance or preference is given to the isolation value and number of requests which are parameters to be maximised in our model. The degree of isolation, I_{ij}, for each component, is set to either 1, 2, or 3 for *shared component, tenant-isolated component and dedicated component*, respectively. 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. This expression is divided by the resource limit and squared to make the penalty heavier for violating any constraint.

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, the problem of providing optimal solutions that guarantee multitenancy isolation can be mapped to a 0-1 multichoice multidimensional knapsack problem (MMKP) [49, 50]. An MMKP is a variant of the Knapsack problem which has been shown to be a member of the NP-hard class of problems [51]. Our problem is formally defined as follows:

Definition 2 (Optimal Component Deployment Problem): *Suppose there are N groups of components $(c_1, ..., c_N)$ with each having l_i $(1 \leq i \leq l)$ 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 as shown in the "Open multiclass queuing network model" section); (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 as shown in the "Open multiclass queuing network model" section) and (v) m resources which are required to support the component, $r_{ij}^{\alpha} = r_{ij}^1, r_{ij}^2, ..., r_{ij}^m$. The total amount of available resources in the cloud required to support all the application components is $R = R^{\alpha}$ $(\alpha = 1, ..., m)$. The objective of an MMKP is to pick exactly one component from each group for a maximum total value of the collected items, subject to m resource constraints of the knapsack* [52, 53]. *Concerning our problem, the goal is to deploy components of a cloud-hosted service by selecting one component from each group to meet the resource constraints of the system and maximise the optimal function $\mathcal{G}$. There are unique features in our problem that lend to solving it using an MMKP and an open multiclass problem. For example, the resources supporting each component are mapped to the resources required by the object in MMKP and are also mapped to the service centres of each class in the open multiclass QN.*

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

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

subject to

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

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

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

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 $R = R^{\alpha}$ $(\alpha = 1, ..., m)$.

To calculate the number of requests, Q_{ij} that can be allowed to access the component, an open multiclass QN model has to be solved [41] 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). "Open multiclass queuing network model" section describes how the average number of requests allowed to access each component is computed.

Open multiclass queuing network 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 centres, which represent system resources, and customers, which represent users or transactions [41]. Fig. 5 shows an example of an open QN model with two service centres (i.e., CPU and disk).

Assumptions: This study makes the following assumptions about a component:

(i) requests sent to a component have significantly different behaviours whose arrival rate is independent of the system state.

(ii) the service demands at the CPU, RAM, Disk, and Bandwidth that support each component are known or can be easily measured by either the SaaS provider or the SaaS customer.

(iii) the resources supporting each component are enough to handle the magnitude of new incoming requests as the workload changes. This ensures that there is no overload when all components are functional.

(iv) the consumption of resources supporting the components reflects the running cost of the cloud-hosted service. Therefore the cost implications are managed in the same way regardless of the deployment model used - SaaS, PaaS, or IaaS. For example, in IaaS this would include the servers together with the resources and components running on the servers.

The above assumptions allow us to use an open multiclass queueing 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 [41]. The open multiclass queuing network model is a widely used technique to optimize the deployment of services in cloud computing environments [54, 55]. This study is a further contribution to literature by applying an open multiclass queuing model to provide optimal solutions for deploying cloud services to guarantee the required degree of multitenancy isolation.

Definition 3 (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*

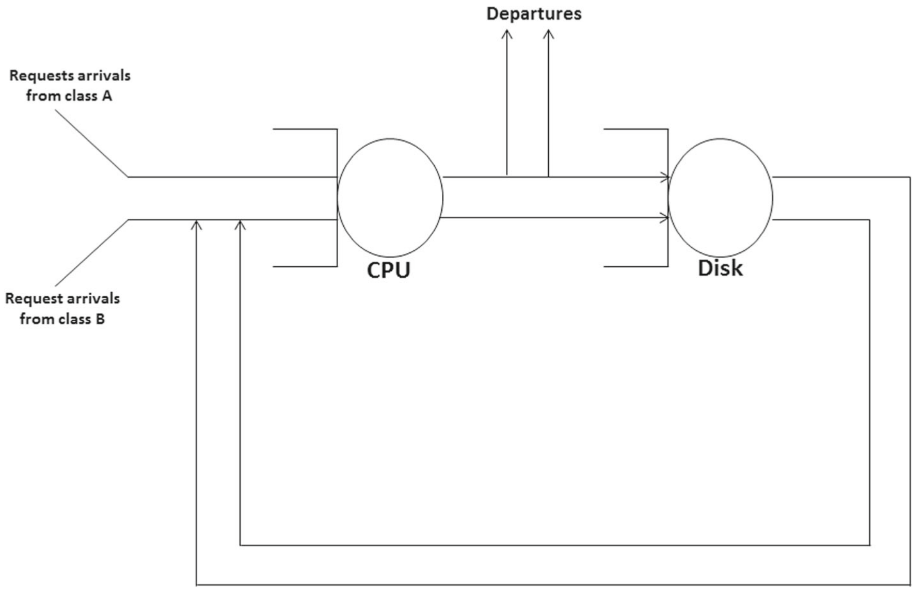


Fig. 5 Open multiclass queuing network model

of arrival rates is denoted by $\overrightarrow{\lambda} \equiv (\lambda_1, \lambda_2, \ldots \lambda_N)$*. The utilization of each component of class c at centre k is given by:*

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

In solving the 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:

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

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.

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

The consumption of the resources supporting the components are interpreted as follows: CPU usage is interpreted as the computing speed of the hardware device (i.e., in terms of clock speed). CPU is measured as Megahertz (MHz), Gigahertz (GHz) or as the percentage of CPU time spent. Memory usage is the amount of used memory, usually measured in kilobytes (e.g., 20 kb). Disk space is the total amount of bytes that a disk drive can hold, usually measured in kilobytes (KB), megabytes (MB), gigabytes (GB), or terabytes (TB). Bandwidth is the amount of data that can be transmitted in a fixed amount of time and measured in bits per second (bps). The term service demand refers to the total amount of time required to use a resource that supports a component (e.g., CPU and RAM), and is measured in time units (e.g., seconds). For example, we can specify that users requests arrive at a component of a cloud-hosted system at a rate of one every five seconds and that such request(s) requires an average of 0.25 seconds of service at the CPU.

Metaheuristic search

The optimisation problem described in "System Model and Description of the Problem" section is an NP-hard problem which has been known to have a feasible state space that grows in a combinatorial way [52]. The number of feasible states for our optimal component deployment problem is given by the following expression:

$$\left\{\binom{l}{j}\right\}^{N} \tag{7}$$

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

Therefore, an efficient heuristic is needed to find an optimal solution to the optimisation problem, which must be solved by the model-based algorithm and provided to the SaaS customer (or a cloud deployment architect) in almost real-time. The section that follows presents four variants of metaheuristic solutions; two are based on hill climbing (i.e., HC(Random and HC(Greedy)), and the other two are based on simulated annealing (i.e., (SA(Random) and SA(Greedy)). The justification for deciding to base the variants of the metaheuristic on hill climbing and simulated annealing is that Hill climbing represents a family of improvement heuristic, while Simulated annealing represents a family of modern heuristic. The difference between improvement heuristic and modern heuristic are summarised as follows [56]:

(i) Usually, modern heuristics are considered as problem-independent, whereas improvement heuristics are explicitly problem-specific and exploit problem structure. This means that modern heuristic can be applied to a wide range of different problems with little or no modification while improvement heuristic is demanding to design and use as it requires knowledge and exploitation of problem-specific properties.

(ii) Improvement heuristic starts with a complete solution and iteratively tries to improve the solution, while modern heuristic use during search both intensification (exploitation) and diversification (exploration) phases.

(iii) In contrast to modern heuristic where improvement steps alternate with diversification steps, which usually lead to solutions with a worse objective value, improvement heuristic uses no explicit diversification steps.

Any of the four variants of the metaheuristic solution can be utilised with *OptimalDep* (see line 17 of Algorithm 1). Also, an algorithm is developed to perform an exhaustive search of the entire solution space for a small problem. The algorithms for *optimalDep* and *SA(Greedy)* are presented as Algorithms 1 and 2, respectively. A high-level description of these algorithms is provided below:

The *SA(Greedy)* for optimal Solution: 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

Algorithm 1 optimalDep Algorithm

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

Algorithm 2 SA(Greedy) Algorithm

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

loads the MMKP problem instance and then populates the global variables (i.e., arrays of varying dimensions that store the values of isolation, and the average number of requests, and component resource consumptions). A simple cooling schedule is used which is expressed as:

$$T_{i+1} = T_0 + (A - T_0) \tag{8}$$

In the above cooling schedule, the variable A is computed as follows:

$$\begin{aligned} A &= \frac{(NoItrn)^1 \times \beta}{(NoItrn)^1} \\ \beta &= 1 - \left(\frac{1}{NoItrn}\right) \end{aligned} \tag{9}$$

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

A simple dry run of the algorithm for the problem instance[2] C(20,20,4) is as follows: 20 optimal solutions are randomly generated and then the standard deviation of all the solutions is computed. Assuming this value is 5.26, the T_0 is set to 5. At the first iteration, $g\left(a^2\right) = 151634.9773$ and $g\left(a^1\right) = 151535.7984$ and the current temperature then becomes 4.999995. At the next iteration, the current temperature is expected to reduce further (see equation 8 and 9). After five iterations, the algorithm constructs an initial/first solution with $g\left(a^1\right) = 151732.4362$, a current/second random solution with $g\left(a^2\right) = 151733.9821$ and with a current temperature of 4.999975. The solution a^2 will replace a^1 with probability, $P = \exp(-(1.5459)/4.999975)=0.7340$, because $g\left(a^2\right) > g\left(a^1\right)$. In lines 13 to 15, a random number between 0 and 1 (i.e., rand = 0.0968) is generated, and since *rand* $<$ *0.7340*, a^2 replaces a^1 and the algorithm continues with a^2. Otherwise, the algorithm continues with a^1. At the next iteration, the temperature T is reduced which now becomes $T_6 = 4.99997$ (line 17). The iteration continues until N (i.e., the number of iterations set for the algorithm to run) is reached, and so the search converges with a high probability to the optimal solution.

SA(Random): This variant of the metaheuristic requires only a slight modification. The 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, a random solution is simply generated. It is important to note that the two variants which are based on the 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) so that the simulated annealing is forced to systematically explore the neighbourhood around the current solution and ensure that the search returns a local minimum.

***HC(Random) and HC(Greedy)*:** The *HC(Random)* use a randomly generated solution as the initial solution to run the hill climbing algorithm, while the *HC(Greedy)*, uses a greedy solution as the initial solution to run the hill climbing algorithm. From an implementation standpoint, this translates to leaving outlines 12-15 (i.e., the else part of the if statement) of Algorithm 2.

Model-based algorithm for optimal deployment of components

In this section, we describe the *optimalDep* algorithm that combines the optimization model, QN model, and metaheuristic to provide optimal solutions for deploying components of a cloud-hosted service that would guarantee multitenancy isolation. Thereafter, we present an architecture together with a theoretical description of how the different components of the algorithm work together, thus allowing it to be transformed into a decision support system to support the deployment of components to the cloud for guaranteeing multitenancy isolation.

OptimalDep: an algorithm for optimal deployment of components

This section describes the OptimalDep algorithm and also show how the open multiclass QN model and the heuristic search fits into the model-based algorithm.

Description of optimalDep algorithm

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

Note that the algorithms described in this paper are different from the autoscaling algorithms offered by IaaS providers like *Amazon* and existing optimisation models proposed for use by SaaS providers such as *Salesforce.com* [6]. Saas providers may be able to monitor and estimate to a certain degree the performance and resources utilisation of application components integrated within applications running on VMs that they have rented out to SaaS customers. However, SaaS providers do not know the required degree of isolation of each application component (e.g., components that offer critical functionality), the number of available components to be deployed, and the number and capacities of resources required to support each component. In some cases, it may also be necessary to associate a particular user/request to certain components or group of components to guarantee the required degree of isolation. These details are only available to SaaS customers (e.g., a cloud deployment architect) since they own the components and are also responsible for deploying the components to the cloud.

OptimalDep algorithm example

The following example (Table 2) shows the different solutions evaluated by *optimalDep* combined with the SA(Greedy) algorithm to find an optimal solution to the optimization problem. Every time there is a change in the workload, the optimalDep algorithm finds a new optimal solution for deploying components with the highest degree of isolation and the highest number of supported requests.

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

When users requests arrives indicating a change in workload (i.e., in our case, this means an arrival rate between 0 to 3.7 req/min), the QN model Eqs. 4, 5, and 6 are solved to find the average number of requests that can access the components. The ninth row shows the updated problem instance with the current number of requests (i.e., 5.2) that can access the components in each group. The updated problem instance is then solved with the metaheuristic and the state with the highest optimal function value is returned. Solution 1 (in row twelve) shows the

Table 2 An example of optimal component deployment

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

optimal value of 515.6 for selecting a solution that deploys the first component from all the groups. This solution results in an optimal value of 515.6 (isolation value=500; and number of request=15.60). Note that no component can be selected for deployment and hence no changes can be effected in the cloud environment until the search is over and a better solution is found.

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

Architecture for transforming *optimalDep* algorithm to a model-based decision support system

The model-based simulation algorithm (i.e., *optimalDep*) presented in this paper can be transformed into a model-based decision support system (DSS) by relying on the architecture, *optimalArch*, which is presented in Fig. 6. Model-based DSS will utilize data and parameters provided by users to assist decision makers (e.g., software architects, cloud developers) in analysing different cloud deployment scenarios in for deploying components of cloud-hosted to guarantee multitenancy isolation. The decision support system can be implemented in different ways, for example, as a web application or a desktop application. It can also be deployed on the cloud as a cloud-hosted service or embedded into other applications running on the cloud or distributed environment. The architecture is composed of five main modules as described below:

Input interface

This module is used to send input to the model-based algorithm. In our case, the main input is the workload of the system, which is represented as the arrival rate of requests (λ) and the id of the (MMKP) problem instances, which represents the different configurations of components integrated with or designed to use a cloud-hosted service.

Information repository

This module stores MMKP instances (which contains information about component configuration) and the service demands of resources supporting the components in the MMKP instance). The service demand of the component together with the arrival rate of request to the component is used to solve the QN model to obtain the average number of requests that can be allowed to access

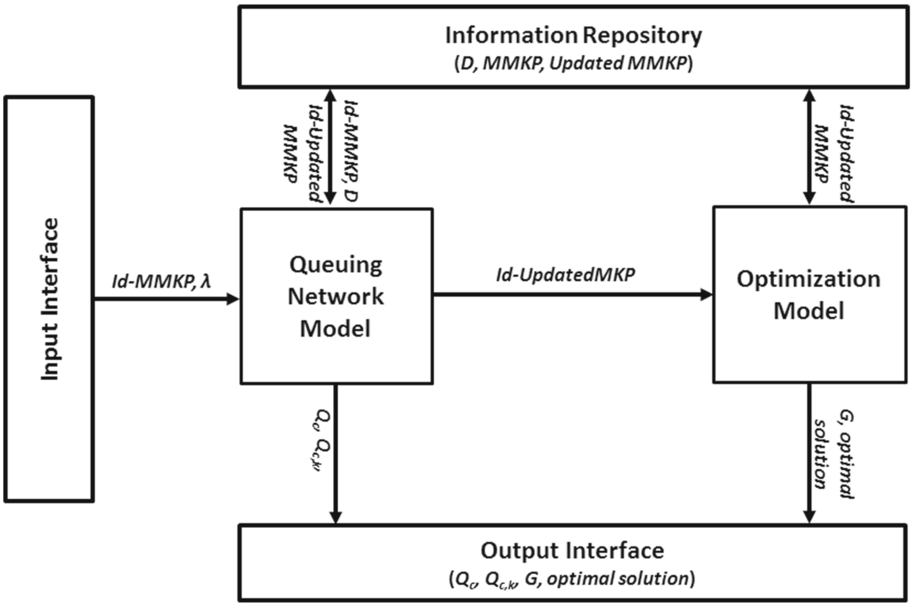


Fig. 6 *optimalArch*: architecture for implementing a decision support system based on the model-based algorithm

the component. From the implementation standpoint, the repository stores three types of file: (i) MMKP instances, which contain component configuration, (ii) workload file, which contains service demands and arrival rate for each component, and (iii) updated MMKP instance, which contains updated details on the MMKP each time there are changes in workload. Note that there may be multiple workload files associated with/generated for a single MMKP instance.

Queuing network model

The input to this module is the arrival rate of the requests to each component and the id of the required (MMKP) problem instance. When there is a change in workload based on the arrival rate of requests, the id of the MMKP instance is used to retrieve the service demand of resources that support the components whose configuration are in the MMMKP instance file. This information (i.e., arrival rate and service demands) is used to calculate the number of requests, Q, allowed to access the component. The new/current value of Q is then used to update the MMKP instance to reflect the current change in workload regarding the number of requests that can be allowed to access the component. This updated MMKP instance is returned to the repository, and the id of this problem instance is passed to the optimisation module. The number of requests allowed to access each component and the total number of requests allowed to access the whole/entire cloud service can be computed and sent to the output interface.

Optimization module

The input to the optimisation module is the id of the updated MMKP instance. In the optimisation module, the metaheuristic is invoked to search for and provide optimal solutions. The optimal value (i.e., fitness/objective function) of the obtained solution together with the optimal solution for deploying components of the cloud-hosted service is evaluated. This information (i.e., optimal value and the optimal solution) is sent to the output interface for use in architecting the deployment of components of a cloud-hosted service to guarantee the required degree of multitenancy isolation.

Output interface

The output interface will display several details associated with the optimal deployment of components of a cloud-hosted service for guaranteeing multitenancy isolation when there are workload changes. These include the optimal function, optimal solution, the number of requests accessing each component and the total number of requests accessing the whole cloud service.

Applicability of the model-based algorithm in a typical cloud deployment scenario

Our proposed model-based algorithm has several applications in the real cloud computing environment. Some example scenarios where our work can be applied are presented below:

(i) Optimal Allocation in a Resource-constrained Environment: In a resource-constrained environment, users are always looking for options to optimise the

consumption of resources while guaranteeing the required degree of isolation between tenants. Our model-based algorithm can be transformed into a decision support system to achieve this by first setting a limit on the resources (i.e., CPU, RAM, Disk and Bandwidth) that are used to support each component. Thereafter, the model-based algorithm can be used to provide optimal solutions that represent the required degree of isolation (i.e., either the highest, average or lowest degree) and the maximum number of requests that can access each component based on the available resources.

(ii) Monitoring Runtime Information of Components: Another application of our model-based algorithm is that it can be used as a cloud deployment pattern or integrated into other cloud patterns like an elastic load balancer, and an elastic manager to monitor runtime information about individual components. Examples of information that could be monitored include the number of requests that can concurrently access the application components and the feasibility of the limits/capacities set for the resources supporting each component to achieve the required degree of isolation.

Even though many cloud providers offer a significant amount of rule-based scaling or load balancing functionality (e.g., Amazon's Auto Scaling [3] and Microsoft Azure Traffic Manager [4]), our decision support system can be customised to monitor and adjust the configuration of components that were created as part of the original scaling rules, and thus provide optimal solutions that guarantee the required degree of multitenancy isolation. This is especially important when there are frequent workload changes and different or varying user behaviours.

(iii) Controlling the Provisioning and Decommissioning of Components: When runtime information of components is available, they can be used to make important decisions concerning scaling, provisioning of required components and decommissioning of unused components. For example, when the required degree of components is known, this information can be used to adjust the number of component instances to reflect the current workload experienced by the application. Decommissioning components that would not impact negatively on the performance of other components and the application could lead to significant cost savings for users.

Our model-based algorithm can be customised to provide information for decommissioning of failed components or components that are not working properly to achieve the required degree of isolation. Although many providers offer monitoring information, for example, information about network availability and utilisation of components deployed on their cloud infrastructure, it is the responsibility of the customer to extract, deduce and interpret these values and then provide information regarding the availability of components.

Evaluation

In this section, we describe how we generated problem instances used for the experiment, and the experimental setup and procedure.

Dataset and instance generation

This section discusses the generation and composition of the dataset used for the experiments. Also discussed is the applicability of the generated instances to real-life cloud deployment scenarios.

Dataset

The dataset used for simulation experiments on the optimisation model was based on a simulation testbed. There are two datasets used in this study: the MMKP instance file and the workload file.

(a) *MMKP Instance file:* Due to the unique nature of our problem, the multichoice multidimensional knapsack (MMKP) instances used in the experiments were randomly generated and not based on a publicly available dataset of MMKP instance. However, the instance was generated based on the standard approach widely used in literature [57, 58]. The format of the MMKP instance is shown in Fig. 7. The description of the MMKP instance format is summarised below:

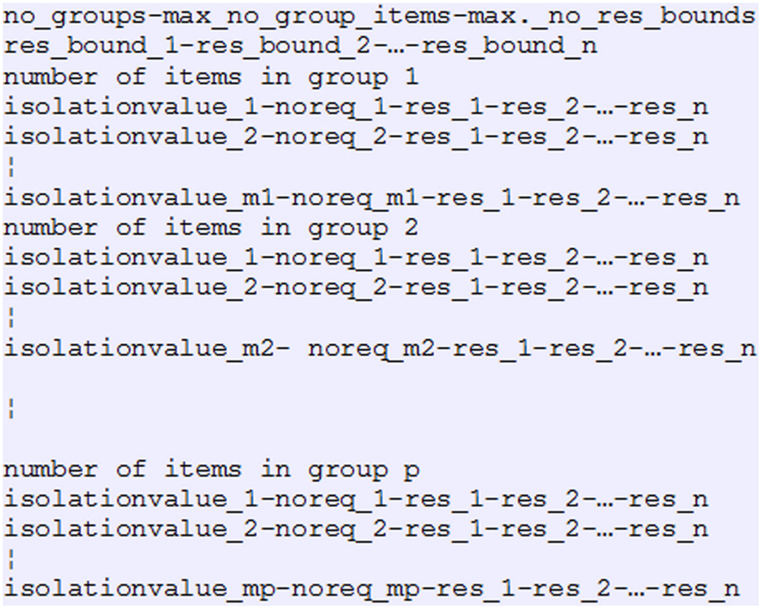
(i) the first row contains three values - the number of component groups in the MMKP instance, the maximum number of components in each group and the maximum number of resources supporting each component. The maximum number of resources supporting each component is four (i.e., represented by CPU, RAM, Disk space and Bandwidth) and remains the same for all instance types.

(ii) the second row contains four values which represent the limit of the resource supporting each component.

(iii) the third column contains the number of components for the first group.

(iv) the rows that follow contain six properties associated with each component of the group. These properties are the isolation value of the component, the number of requests allowed to access the component, and the resource consumption for CPU, RAM, Disk space and Bandwidth which support the component. So assuming the column contains the value 20, it means that the first group contains 20 components. Row four to row twenty-three contains the properties associated with each of the twenty components of the group.

(v) after the row that contains the properties of the last item of group one, then the number of items for group two follows. The format for the remaining groups follows the same pattern. The next section explains how these



```
no_groups-max_no_group_items-max._no_res_bounds
res_bound_1-res_bound_2-…-res_bound_n
number of items in group 1
isolationvalue_1-noreq_1-res_1-res_2-…-res_n
isolationvalue_2-noreq_2-res_1-res_2-…-res_n
⋮
isolationvalue_m1-noreq_m1-res_1-res_2-…-res_n
number of items in group 2
isolationvalue_1-noreq_1-res_1-res_2-…-res_n
isolationvalue_2-noreq_2-res_1-res_2-…-res_n
⋮
isolationvalue_m2- noreq_m2-res_1-res_2-…-res_n

⋮

number of items in group p
isolationvalue_1-noreq_1-res_1-res_2-…-res_n
isolationvalue_2-noreq_2-res_1-res_2-…-res_n
⋮
isolationvalue_mp-noreq_mp-res_1-res_2-…-res_n
```

Fig. 7 Format of the MMKP instance

values (e.g., the resource capacities and consumption) were generated.

(b) *Workload file:* workload file contains the values that are used to simulate the workload offered to the system. The key values it contains are the arrival rate of requests and the service demands of each resource supporting the components. The format of the MMKP instance shown in Fig. 7 can be used to explained workload file as follows:

(i) the first, second, and third row is the same as in the MMKP instance. (ii) the only difference is in the composition of the properties that are associated with each component. For the workload file, there are five properties: the arrival rate of the requests to the component and the service demands each for CPU, RAM, Disk space and Bandwidth that support the component. The next section explains how the arrival rate and service demands were generated.

Instance generation

Several problem instances of various sizes and densities were randomly generated. After that, these instances were solved using each variant of the metaheuristic. Two categories of instances were generated and tailored on the instances widely cited in literature: (i) OR benchmark Library [59] and other standard MMKP benchmarks, and (ii) the new irregular benchmarks used by Shojaei et al. [60]. These benchmarks are usually used for single objective problems. This benchmark format was modified and extended to conform to a multiobjective case by associating each component with two different profit values: isolation values and the average number of requests [61].

(i) Defining an Instance Generating Function: To generate the values associated with components in each class i, the values were first bound with two parameters: v_i^{min} and v_i^{max}, and then a uniform generating function was applied to draw values uniformly and randomly within this interval. The uniform generating function is given as:

$$p_{ij} = \mathcal{U}\left(v_i^{min}, v_i^{max}\right) \tag{10}$$

(ii) Generating Isolation, Number of Requests and Resource Consumption: For isolation, the values were randomly generated in the interval [1-3]. The value for the average number of requests supported by each item was initially set to zero (0) for all items. This value is updated in the problem instance by solving the QN model each time the workload changes. This updated instance is then solved by the metaheuristic to obtain optimal solutions for deploying components to the cloud. The values of a component's consumption of CPU, ram, disk capacity, and bandwidth (i.e., the weights) were generated in the interval [1-9].

(ii) Generating Resource Capacities: Values for capacities of component resources (i.e., knapsack capacities for CPU, ram, disk and bandwidth) are generated by setting it to half of the maximum possible resource consumption.

$$c_k = \frac{1}{2} \times m \times R \tag{11}$$

The same principle has been used to generate instances available at OR Benchmark Library, and also for instances used in [57, 58].

(iv) Generating Workload and Service Demands: For workload, the values were randomly generated following a Poisson distribution (with mean=3) in the interval [1, 5]. Values for service demand were in the interval [0.05,0.25]. In this work, the number of resources in each group is four, which corresponds to the basic resources (i.e., CPU, RAM, disk, network bandwidth) required for a component to be deployed to the cloud.

(v) Notation for defining an MMKP Instance: The notation used to define a problem instance is given as: C(number of groups, number of component per group, number of resource types supporting each component). So for example, C(4,5,4) means that the problem instance has 4 groups of components, 5 components per group and supported by 4 resource types -CPU, memory, disk space and bandwidth.

(vi) Sample Dataset files: We have included in the "Additional files" section, four sample files that represents the dataset used in the experiment to improve reproduceability (Additional files 1, 2, 3, and 4). The files included are - a large problem instance file (i.e., C(500, 20,4), a service demand file associated with C(500,20,4), an updated instance file associated with C(500,20,4), and a workload file associated with C(500,20,4).

Applicability of the generated instances to real-life cloud deployment scenario

The MMKP problem instances represent a repository of components that can be deployed to design (or integrate with) a cloud-hosted service. A component could be a database, a database table, a message queue, VM or even a Docker container. It is also important to note that although the weight values (i.e., the resource consumption of the components) generated in the MMKP instance may appear to be in the same interval, in reality, these values could be normalised (or transformed) to represent different resources units of the components.

As an illustration, one of Amazon's EC2 instance types, named *"'compute optimized (c4.xlarge model)"'*, has the following specification: 4 vCPU, 8 GiB of memory, EBS-optimized only storage (which is similar to an IOPS provisioned on an Amazon Elastic Block store volume) and 750 Mbps of dedicated EBS bandwidth [62]. An Amazon EBS can be created with Provisioned IOPS SSD(io1) volumes up to 16 TiB in size. So assuming the weights of a component on a generated MMKP instance are given as [4, 8, 8, 8], this specification could easily be transformed to the actual specification of the above named Amazon EC2 instance using this normalisation format: [CPU, RAM, DISK/2, BANDWIDTH/100]. This means that this particular component is supported with 4 virtual CPUs, 8GB of memory, 8 TB of disk space and 8 Mbps of bandwidth. Another approach suggested by Han et al. [63], is to include the dimension index k as a parameter of the generating function so that the weight for a dimension k can be chosen in a range that depends on k for the uniform generating function.

The randomness of the values generated is consistent with the unstable and unpredictable nature of cloud-hosted systems. This unpredictability is due to increased complexity and dynamicity of interactions between applications and workload sharing the cloud infrastructure [15, 64, 65]. As a result, it is difficult to provide QoS guarantees and also set service level agreements for both users and providers of cloud-hosted applications [10]. In addition, the randomness of the values and solutions is a salient feature of evolutionary algorithms [56].

Experimental setup and procedure

The problem/MMKP instances used for our experiments were generated as described in the "Dataset and instance generation" sub-section (i.e., under the "Evaluation" section). The instance generating program and the algorithms were written using Java programming with Netbeans IDE 7.3.1. All experiments have been carried out on the same computation platform, which is a Windows 8.1 running on a SAMSUNG Laptop with an Intel(R) CORE(TM) i7-3630QM at 2.40GHZ, with 8GB memory and 1TB swap space on the hard disk. Table 3 shows the parameters used for the experiments. Each instance is tested with a workload associated with it. The exhaustive search algorithm was incapable of solving large instances. This was because of the low memory of the used machine. And so a small MMKP instance, C(4,5,4) was used for the evaluation and comparison of the algorithms. This MMKP instance consists of 4 groups of components, 5 components per group and supported by 4 resource types.

Aim of the experiment: The aim of the experiment is to evaluate the performance (i.e., regarding obtained solution quality, robustness, and computational effort) of the different variants of the metaheuristic when integrated into the model-based algorithm (i.e., optimalDep).

Evaluation metrics and statistical analysis

The model-based algorithm is novel in the sense that it combines a QN model and metaheuristics to find optimal solutions for component deployment while guaranteeing the required degree of multitenancy isolation. Thus, there are no existing approaches that can be used to make a direct comparison with our novel model-based algorithm. Because of this, the solutions obtained from our approach were compared with the optimal solutions obtained from an exhaustive search of a small problem instance. Thereafter, the obtained solutions were also compared with the

Table 3 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]
Metaheuristic	
No. of iterations	1000000
Population size	1000
No. of runs	20
Temperature	T_0 = st. dev of N randomly generated solutions (N=no. of groups)
Cooling schedule	$T_{i+1} = T_0 + (A - T_0)$

target solution obtained from different problem instances of varying sizes and densities. The performance indicators considered are:

(1) Quality of Solution: The quality of solutions obtained was measured in terms of the percent deviation of the obtained solution to the target/reference solution. This is given as:

$$\frac{|f(s) - f(s*)|}{f(s*)} \tag{12}$$

where s is the obtained solution and s* is the reference solution obtained from the exhaustive search [45].

(2) Robustness: Robustness was measured in terms of the variability of the solutions over different iterations of the metaheuristic on the same instance; the lower the variability, the better the robustness [45]. The standard deviation was used as a measure of this variability.

(3) Computational Effort: The computation effort required to produce the solutions was measured in terms of the average execution time of the metaheuristic. The execution time for the SA(Greedy) and HC(Greedy) is computed as:

$$ExecTime = GreedyTime + (FEvalTime * NoFEval) \tag{13}$$

where *ExecTime* means the total time to run the metaheuristic, *GreedTime* is the time to produce the initial greedy solution, *FEvalTime* is the time to evaluate a randomly generated solution, and *NoFEval* is the number of function evaluations to reach the target solution. For SA(Random) and HC(Random), the *GreedyTime* is replaced with *RandomTime*, which is the time to produce an initial random solution. The *NoFEval* represents the average number of function evaluations over 20 runs for each instance size.

In addition to the above metrics, we also computed the success rate and performance rate of producing the solutions from the different variants of the metaheuristics. The success rate was measured as the number of iterations to reach the target solution over the total number of runs or trials. The percent success (i.e., success %) is the percentage number of iterations to reach the target solution over the total number of runs (i.e.,20 runs). The success rate is given as:

$$\frac{\text{number of iterations to reach the target solution}}{\text{total number of runs}} \tag{14}$$

The performance rate of our approach when compared to the optimal solution takes into account the computational effort by considering the total number of iterations for each run [45]. This is given as:

$$\frac{\text{number of iterations to reach the target solution}}{\text{total no. of iterations} \times \text{total no. of runs}} \tag{15}$$

Furthermore, graphs and statistical plots were used to analyse the interaction between the different performance indicators. For example, a graph of run-time length distribution (RLD) was plotted to analyse the convergence behaviour of the metaheuristic on the number of function evaluations. RLD indicates the probability of reaching a pre-specified objective function value over a specified number of functional evaluations [66, 67]. The probability value (success rate) is the ratio between the number of runs to find a solution of a certain quality and the total number of runs. RLD is usually used when time is measured with any architecture-independent parameter, such as the number of evaluations or generations [68, 69].

It is important to note that there were limitations in the computational power of the machine used for the experiments and so the overall computation time required by the optimalDep algorithm to produce the optimal solutions was not considered. To address this challenge, the execution time of the metaheuristic was measured based on the average number of function evaluations which is independent of the computer system. In addition to this, the simulation experiments were performed with very large MMKP problem instances.

Statistical analysis was used to conduct a performance assessment of optimalDep algorithm (i.e., the main supporting algorithm for the model-based algorithm) when combined with the different variants of the metaheuristic solution. The two-way ANOVA was adopted to understand if there is an interaction between the two independent variables (i.e., type of instance size and variants of metaheuristic) on the dependent variables (i.e., percent deviation, standard deviation and execution time). The statistical test focused on three performance indicators: quality of solutions, robustness and computational effort required to produce the solutions.

Results

The optimalDep algorithm requires a metaheuristic (see line 17 in Algorithm 3) to provide optimal values from the MMKP instance. The rest of the algorithm requires computation of the queuing network model equations. Therefore, it will test the applicability and the effect of the different variants of the metaheuristic in driving the optimalDep algorithm.

The performance evaluation will be presented in terms of the quality of solution, robustness and the computational effort of the optimalDep algorithm when combined with any of the four different variants of metaheuristics solution:(i) HC(Random) - Hill climbing with a random solution as the initial solution; (ii) HC(Greedy) - Hill climbing with a greedy solution as the initial solution; (iii) SA(Random) - Simulated Annealing with a random solution as the initial solution; and (iv) SA(Greedy) - Simulated Annealing with a greedy solution as the initial solution.

Comparison of solutions obtained from optimalDep algorithm with the optimal solution

The approach presented in this paper is novel in that it combines a QN model and metaheuristics to find optimal solutions for component deployment for guaranteeing the required degree of multitenancy isolation. Therefore, there are no existing approaches that can be used to make a direct comparison with our approach. Because of this, the solutions obtained from the optimalDep algorithm (when running either with HC(Random), HC(Greedy), SA(Random), SA(Greedy) are compared with the optimal solutions obtained by running the OptimalDep algorithm with the exhaustive search of a small problem size. The quality of the optimal solutions was measured in terms of the percent deviation from the optimal solution. The instance used is C(4,5,4) because it was small enough to cope with the requirements of the machine. The C(4,5,4) instance consists of four groups of components, 5 components per group and 4 resource types supporting each component. The workload (i.e., the arrival rate) for each component was randomly generated between 0.0 and 4 requests per seconds.

The results are summarised in Table 4. Each row of the first column shows a different workload with an arrival rate ranging from 2.7-3.9. The second column shows the optimal function variables as (OP/IV/RV), which stand for the value of the optimal function, isolation value, and the number of allowed requests, for the optimal solution. The third, fourth, fifth and sixth columns show the optimal function variables as (OP/FEval) for hill climbing(random), hill climbing(greedy), simulated annealing(random) and simulated annealing(greedy), respectively, which stand for the value of the optimal function and the number of function evaluation to attain the optimal solution.

As shown in Table 4, all the four variants of the metaheuristic produced results that were the same as the optimal solution for all workloads. This means that the four variants of the metaheuristic attained a 100% success rate and 0% percent deviation. The similarity seen in the results may be due to the small size of the instance. This small size was chosen to cope with the machine used for the experiments which could not solve problem instances larger than C(4,5,4) due to limitations in its hardware requirements (i.e., CPU and RAM).

In Fig. 8, the Run Length Distribution (RLD) of the instance is shown based on the arrival rate of 3.9 request per seconds for only 20 iterations since the target solution is attained after about 20 iterations due to the small size of the instance used. This plot shows the performance of the metaheuristic in a scenario where there is limited regarding the time and amount of resources required to execute the model-based algorithm before attaining an optimal value. It is observed that HC(Greedy) and SA(Greedy) reach a 100% success rate and a corresponding performance rate after the first iteration. However, the other variants that start with a random solution

Table 4 Comparing HC(Rand), HC(Greedy), SA(Rand), SA(Greedy) with optimal solution

Workload(λ)	Optimal	HC(Rand)	HC(Greedy)	SA(Rand)	SA(Greedy)
2.7	1220.8/12/20.8	1220.8/41	1220.8/0	1220.8/41	1220.8/0
2.9	1225.69/12/25.69	1225.69/38	1225.69/0	1225.69/51	1225.69/0
3.1	1232.38/12/32.38	1232.38/56	1232.38/0	1232.38/60	1232.38/0
3.3	1242.14/12/42.14	1242.14/52	1242.14/0	1242.14/38	1242.14/0
3.5	1257.99/12/57.99	1257.99/38	1257.99/0	1257.99/41	1257.99/0
3.7	1289.77/12/89.77	1289.77/32	1289.77/0	1289.77/32	1289.77/0
3.9	1415.09/12/215.09	1415.09/17	1415.09/0	1415.09/18	1415.09/0

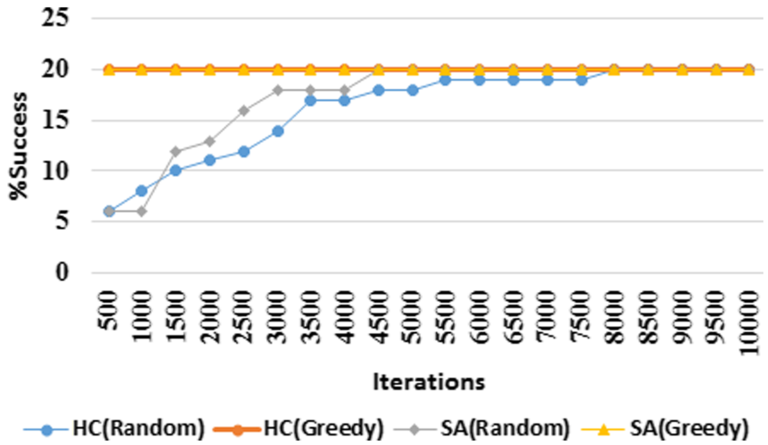


Fig. 8 Run Length Distribution for Small Instance (C(4-5-4))

(i.e., HC(Random) and SA(Random)) attain 100% success after 9 and 15 iterations, respectively. This means that for small instances there may be not much difference between the Hill climbing and the simulated annealing when the initial solution starts with a greedy solution.

Comparison of solutions obtained from optimalDep algorithm with the target solution

As an optimal solution could not be obtained with large instances (e.g., C(500,20,4)), the results were compared to a target solution as proposed by [45] in a situation where there are limitations in the computation machine on which the metaheuristic is executed or it is infeasible to obtain the optimal solution. The MMKP instance C(500,20,4) consists of 500 groups of components, 20 components per group and 4 resource types supporting each component. In our case, the target solution represents a requirement defined by a decision maker on the quality of the solutions to obtain. This is expressed as:

$$TargetSoln = ((n \times max(I) \times w1) + e) \tag{16}$$

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 based on the upper limit of the arrival rate), *w1* is the weight assigned to I and *w2* is the weight assigned to the Q. This equation when used to compute the target solution of C(4,5,4) with arrival rate of 2.7 req/sec gives 1219.2 which is very close to the optimal solution as shown in Table 4. The computed target solution for all instance sizes ranging from C(10,5,4), C(10,20,4) up to C(1000,5,4), C(1000,20,4) and the optimal values obtained for each instance of the four different variants of the metaheuristic are shown in Tables 5 and 6. The rest of the experiment was conducted with an arrival rate of 3.9 requests per second. The last column of Tables 5 and 6 show the initial greedy solutions for all instance sizes. The standard deviation was not computed for the greedy solution, but for the optimal solution. The greedy solution is the initial solution which is later improved by the metaheuristic to produce the optimal solution.

It should be noted that the simulation ran for 1000000 function evaluations in order to be able to attain the best possible solution for the algorithm. Therefore, the success rate would be expected to be nearly 100%, with the corresponding performance rate, since the optimal solution would have converged. Because of this, the study extends the evaluation to cover scenarios where there is limitation regarding the resources and time required to provide optimal solutions, for example, when the algorithm can run for only 1000 iterations (see Table 8 and "Robustness of the solutions" section).

Quality of solutions

The quality of the solutions was measured in terms of the percent deviation from the target solution. As shown in Table 7, the percent deviation for all the variants of the metaheuristic was the same. It was noticed that the percent deviation of solutions is lower when the number of components per group is high. For example, the percent deviation for C(500,5,4) is 3.5 when the number of components per group is 5 and the percent deviation of C(500,20,4) is 1.49 when the number of components per group is increased to 20. The problem instance C(500,5,4) consists of 500 groups of components, 5 components per group, and 4 resource types and the instance C(500,20,4) consists of 500 groups of components, 20 components per group and 4 resource types. This means that the quality of solutions is a function of the number of components per group. The more choices of a particular type of component there are, the

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

Instance	Target solution	HC(Rand)	HC(Greedy)	SA(Rand)	SA(Greedy)	Greedy
C(10,5,4)	3048	2815.09/0.0	2815.09/0.0	2815.09/0.0	2815.09/0.0	2714.43
C(20,5,4)	6096	6053.26/1.5E-4	6053.26/1.5E-4	6053.26/1.5E-4	6053.26/1.50	5523.81
C(30,5,4)	9144	9012.30/0.0	9012.30/0.0	9012.30/0.0	9012.30/0.0	8289.1
C(40,5,4)	12192	12028.67/0.0	12028.67/0.0	12028.67/0.0	12028.67/0.0	11665.49
C(50,5,4)	15240	14725.40/0.0	14725.40/0.0	14725.40/0.0	14725.40/0.0	13501.17
C(60,5,4)	18288	17923.88/0.0	17923.88/0.0	17923.88/0.0	17923.88/0.0	16805.41
C(70,5,4)	21336	21130.89/5.5E-4	21130.89/5.5E-4	21130.88/7.3E-4	21130.89/7.3E-4	20359.45
C(80,5,4)	24384	23389.81/0.0	23389.81/0.0	23389.81/0.00	23389.81/0.00	22361.58
C(90,5,4)	27432	26987.22/0.0	26987.22/0.0	26987.22/0.0	26987.22/3.5E-4	25983.6
C(100,5,4)	30480	28945.60/0.0	28945.60/0.0	28945.60/0.00	28945.60/0.00	27472.12
C(200,5,4)	60960	58647.49/0.0	58647.49/0.0	58647.47/0.01	58647.47/0.01	56055.95
C(300,5,4)	91440	86662.80/0.003	86662.80/0.00	86662.77/0.02	86662.77/0.02	81659.39
C(400,5,4)	121920	117405.24/0.0	117405.24/0.0	117405.15/0.04	117405.14/0.05	111049.71
C(500,5,4)	152400	147023.93/0.0	147023.93/0.00	147023.73/0.09	147023.77/0.07	140156.27
C(600,5,4)	182880	176735.26/0.00	176735.26/0.0	176734.98/0.10	176734.94/0.10	168795.78
C(700,5,4)	213360	205301.82/0.00	205301.82/0.00	205301.49/0.12	205301.44/0.14	195237.57
C(800,5,4)	243840	234472.96/0.0	234472.96/0.00	234472.51/0.16	234472.44/0.16	222105.9
C(900,5,4)	27432	264883.40/0.00	264883.40/0.00	264882.74/0.20	264882.83/0.18	252231.84
C(1000,5,4)	304800	291763.61/0.0	291763.61/0.0	291762.78/0.27	291762.85/0.17	277411.4

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

Instance	Target Solution	HC(Rand)	HC(Greedy)	SA(Rand)	SA(Greedy)	Greedy
C(10,20,4)	3048	3090.18015/0.0	3090.18/0.0	3090.18/0.0	3090.18/0.0	3042.95
C(20,20,4)	6096	6216.57/2.11E-4	6216.57/2.11E-4	6216.57/2.1E-4	6216.57/2.11	5806.68
C(30,20,4)	9144	9151.83/0.0	9151.83/0.0	9151.83/0.00	9151.83/0.00	8519.55
C(40,20,4)	12192	12361.51/0.0	12361.51/0.0	12361.50/0.00	12361.50/0.00	11925.67
C(50,20,4)	15240	15452.77/0.0	15452.77/0.0	15452.76/0.01	15452.76/0.01	14697.84
C(60,20,4)	18288	18661.63/2.4E-4	18661.63/2.4E-4	18661.62/0.01	18661.62/0.01	17837.44
C(70,20,4)	21336	21555.88/ 4.9E-4	21555.88/4.9E-4	21555.85/0.03	21555.85/0.03	20550.67
C(80,20,4)	24384	24715.83/0.0	24715.83/0.0	24715.80/0.01	24715.77/0.06	23426.28
C(90,20,4)	27432	27982.72/9.8E-4	27982.72/9.8-E4	27982.69/0.03	27982.68/0.03	26206.78
C(100,20,4)	30480	31124.34/5.98	31124.34/5.98	31124.28/0.03	31124.28/0.03	29233.1
C(200,20,4)	60960	61861.47/0.0	61861.47/0.0	61861.11/0.17	61861.10/0.16	58297.87
C(300,20,4)	91440	92474.27/0.0	92474.27/0.0	92473.23/0.28	92473.13/0.40	88139.89
C(400,20,4)	121920	123488.32/0.00	123488.32/0.00	123486.36/0.52	123486.44/0.42	116808.95
C(500,20,4)	152400	154665.71/0.0	154665.71/0.0	154662.53/0.70	154662.7/0.60	145493.58
C(600,20,4)	182880	185163.64/0.00	185163.64/0.00	185158.51/0.67	185158.34/0.71	173758.37
C(700,20,4)	213360	216017.65/0.0	216017.65/0.0	216010.27/1.26	216010.57/0.95	203323.86
C(800,20,4)	243840	247335.56/0.0	247335.56/0.0	247325.61/1.28	247325.92/1.18	234522.64
C(900,20,4)	27432	277366.77/0.0	277366.77/0.0	277354.00/1.46	277353.75/1.86	262264
C(1000,20,4)	304800	308359.13/0.00	308359.13/0.00	308344.05/2.00	308344.64/1.67	292307.23

Table 7 Percent deviation from the optimal solution on different instance sizes(l=5; l=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

better the chance of obtaining an optimal configuration. This is particularly important for large open-source projects that are either designed to use a large number of components within the cloud-hosted service or be integrated with several components residing in other locations.

Table 8 shows the obtained solutions and percent deviation (%dev) of obtained solutions to the target solutions for a large instance size (i.e., C(500,20,4)). Columns 2 to 3 shows the obtained solutions while columns 6 to 9 shows the percent deviation of the different metaheuristics. It was observed that the percent deviation for SA(Greedy) and HC(Greedy) was better than the other variants. For example, the percent deviation for SA(Greedy) was less than 0.96 in most cases and was much more controlled and stable than the other variants. Therefore, for large problem instances, while HC(Greedy) may produce the best optimal solutions, the SA(Greedy) will still produce more stable solutions than other variants.

In Fig. 9, the quality of solutions is shown for the first 10000 iterations. This represents a scenario where there is a limitation in time or resources to do an exhaustive search of the entire problem size. The two variants that started with the greedy solution as the initial solution (i.e., HC(Greedy) and SA(Greedy) benefited significantly from the greedy solution than the other two variants. For example, it will take up to 7500 function evaluations (which translates to more time and resources) for the SA(Random) and HC(Random) to attain an optimal value of at least 153000. That same optimal value would have been reached by HC(Greedy) after about 2500 iterations.

Robustness of the solutions

Robustness was measured in terms of the variability of the solutions over different iterations of each variant of the metaheuristic on the same instance. The standard deviation was used as a measure of this variability. Measuring the variability or otherwise (i.e., stability) of the optimal solutions is important in cloud environments where there are varying workload changes. The standard deviation of the optimal functions was computed for different numbers of iterations of the same instance. Tables 5 and 6 show the standard deviation for all instance sizes in the variable, Opt/Std, which stands for optimal value and standard deviation. As shown in Tables 5 and 6, the standard deviation of the instance c(1000,5,4) for SA(Random) and SA(Greedy) are 0.27

Table 8 Obtained solution (ob.sln) and Percent deviation(%dev) of obtained soln to the target soln - C(500,20,4)

ITRN	HC(rn)-ob.soln	HC(gr)-ob.soln	SA(rn)-ob.soln	SA(gr)-ob.soln	HC(rn)-%dev	SA(gr)-%dev	SA(gr)-%dev	SA(gr)-%dev
0	102493.35	151639.99	102553.45	151635.01	32.75	0.50	32.71	0.50
500	120066.77	152030.99	120232.32	152011.96	21.22	0.24	21.11	0.25
1000	130650.02	152453.57	130639.45	152296.21	14.27	0.04	14.28	0.07
1500	137610.24	152720.79	137510.18	152469.75	9.70	0.21	9.77	0.05
2000	142329.22	152932.51	141946.25	152670.75	6.61	0.35	6.86	0.19
2500	145443.36	153086.08	145022.77	152815.44	4.56	0.45	4.84	0.27
3000	147491.14	153262.74	147231.31	152991.31	3.22	0.57	3.39	0.39
3500	148966.65	153406.7	148858.75	153136.21	2.25	0.66	2.32	0.48
4000	150116.54	153533.35	150050.23	153246.17	1.50	0.74	1.54	0.56
4500	151066.03	153643.3	150837.07	153329.15	0.88	0.82	1.03	0.61
5000	151679.1	153726.85	151530.88	153420.63	0.47	0.87	0.57	0.67
5500	152093.62	153822.14	152003.19	153486.9	0.20	0.93	0.26	0.71
6000	152468.36	153888.66	152361.67	153329.15	0.04	0.98	0.03	0.61
6500	152752.03	153937.59	152683.1	153620.12	0.23	1.01	0.19	0.80
7000	152982.12	153986.83	152947.19	153673.2	0.38	1.04	0.36	0.84
7500	153151.13	153986.83	153111.49	153734.31	0.49	1.04	0.47	0.88
8000	153355.56	154102.75	153258.93	153756.01	0.63	1.12	0.56	0.89
8500	153512.38	154129.54	153392.12	153799.05	0.73	1.13	0.65	0.92
9000	153623.17	154162.96	153492.14	153829.5	0.80	1.16	0.72	0.94
9500	153752.4	154198.51	153570.9	153855	0.89	1.18	0.77	0.95
10000	153752.4	154226.25	153669.08	153870.42	0.89	1.20	0.83	0.96
Min	102493.35	151639.99	102553.45	151635.01	32.75	0.50	32.71	0.50
Max	153752.4	154226.25	153669.08	153870.42	0.89	1.20	0.83	0.96
Avg	145683.60	153470.43	145566.78	153189.35	4.41	0.70	4.48	0.52
Std	12877.15	725.97	12822.48	635.43	8.45	0.48	8.41	0.42

and 0.17, respectively. It was observed that the standard deviation for SA(Random) and SA(Greedy) was higher than that of HC(Random) HC(Greedy) in most of the cases. This means that metaheuristic based on hill climbing were more stable and robust than the other variants based on simulated annealing, especially for large instances.

Table 8 shows that although the minimum, maximum, average values of solutions produced by the HC(Greedy) when applied to a large instance (i.e., C(500-

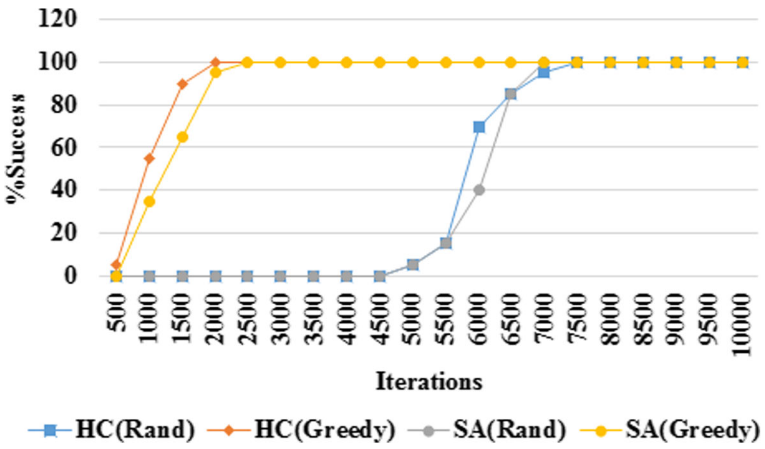


Fig. 9 Quality of Solution for a large instance size(C(500-20-4))

2-4)) for the first 10,000 function evaluations are better than the other variants of the metaheuristic, the standard deviation values for HC(Greedy) were slightly higher than that of SA(Greedy). As expected, the standard deviation for HC(Random) was greater than that of SA(Random) and all other variants. This means that for large instances when there is a limitation in terms of time and available resources, the variants of metaheuristic that start with an initial greedy solution, especially when used with simulated annealing (i.e., SA(Greedy) produce solutions that are more robust and stable.

Computational effort of the metaheuristic

The computational effort was measured in terms of success rate, performance rate and average execution time required to produce a solution. Table 9 presents the success rate and performance rate for C(500-20-4) after running the algorithms for 10,000 iterations. It was observed that the variants of the metaheuristics that start with the initial greedy solution performed better. For example, the HC(Greedy) requires 2000 function evaluations to attain a 100% success rate whereas HC(Random) requires 5000 function evaluations.

Figure 10 shows the run length distribution of a large instance (i.e., C(500-20-4)) for all the variants of our metaheuristic. As expected, the variants that start with the initial greedy solution have a better %success than the other variants. This confirms our earlier conclusion that in a real-time environment when there are fewer resources, HC(Greedy) will provide better results than the other variants.

Table 10 shows the number of function evaluations reached for each run before attaining the target solution. The number of function evaluations is not the same as the number of iterations. Function evaluation refers to a call of the function that is being optimized (i.e., the optimal function - G). That is, it means a call to the function for evaluating each individual solution in the population. An iteration is a step in a looped optimized process. In this study the population size is set to 1000, the maximum number is set to 1000000 (see Table 3). Therefore, the

Table 9 Success Rate(s.rt) and Performance Rate (p.rt) based on Target Solution (C(500-20-4))

ITRN	HC(rn)-s.rt	HC(gr)-s.rt	SA(rn)-s.rt	SA(gr)-s.rt	HC(rn)-p.rt	HC(gr)-p.rt	SA(rn)-p.rt	SA(gr)-p.rt
0	0	0	0	0	0	0	0	0
500	0	5	0	0	0	0.01	0	0
1000	0	55	0	35	0	0.06	0	0.04
1500	0	90	0	65	0	0.06	0	0.04
2000	0	100	0	95	0	0.05	0	0.05
2500	0	100	0	100	0	0.04	0	0.04
3000	0	100	0	100	0	0.03	0	0.03
3500	0	100	0	100	0	0.03	0	0.03
4000	0	100	0	100	0	0.03	0	0.03
4500	0	100	0	100	0	0.02	0	0.02
5000	5	100	5	100	0	0.02	0	0.02
5500	15	100	15	100	0	0.02	0	0.02
6000	70	100	40	100	0.01	0.02	0.01	0.02
6500	85	100	85	100	0.01	0.02	0.01	0.02
7000	95	100	100	100	0.01	0.01	0.01	0.01
7500	100	100	100	100	0.01	0.01	0.01	0.01
8000	100	100	100	100	0.01	0.01	0.01	0.01
8500	100	100	100	100	0.01	0.01	0.01	0.01
9000	100	100	100	100	0.01	0.01	0.01	0.01
9500	100	100	100	100	0.01	0.01	0.01	0.01
10000	100	100	100	100	0.01	0.01	0.01	0.01
Min	0	0	0	0	0	0	0	0
Max	100	100	100	100	0.01	0.06	0.01	0.05
Avg	41.43	88.10	40.24	85.48	0	0.02	0	0.02
Std	46.47	29.42	46.33	31.66	0	0.02	0	0.01

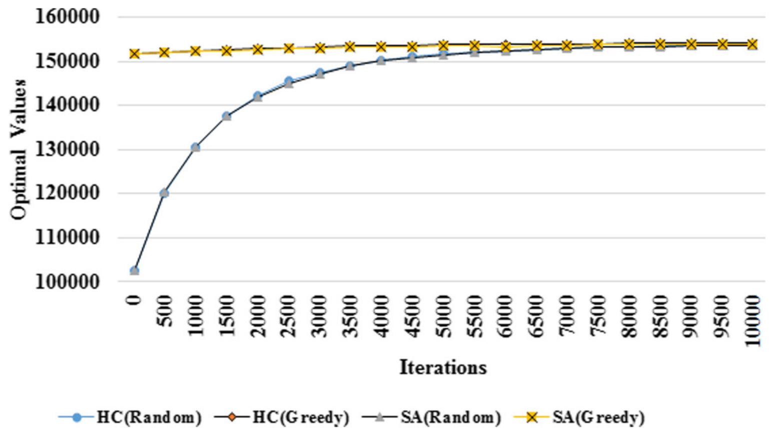


Fig. 10 Run Length Distribution for a Large Instance(C(500,20,4))

maximum number of function evaluations is 1000000000. The results show that it takes the variants of the metaheuristic that start with the greedy solution a far less number of function evaluations to reach the target solutions. For example, the number of function evaluations for HC(Random) in most of the cases are between 3 to 8 times more than that of HC(Greedy).

In addition, the execution time required by each variant of the metaheuristic to reach the target solution for different instance sizes was also computed. First, the actual times to produce both the initial greedy solution and a randomly generated solution for each instance was computed and logged separately. Thereafter, these times were used to calculate the execution time using the formula given in Eq.13. Table 11 summarises the results. Each row of the first column shows a different problem/instance size ranging from C(10,20,4) to C(1000, 20, 4). The second, fourth, and sixth columns show the mean execution times for obtaining a greedy solution, random solution and optimal value from a randomly generated solution, respectively. The third column, fifth, and seventh column show the standard deviation of the mean execution times

Table 10 Function Evaluations to attain the target solution (*N*=1000000)

Instance	HC(Random)	HC(Greedy)	SA(Rand)	SA(Greedy)
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

Table 11 Computational Effort of different instant sizes

Instance size	AVG (gr)	STD (gr)	AVG (rn)	STD (rn)	AVG (fe)	STD (fe)	HC (rn)	HC (gr)	SA (rn)	SA (gr)
C(10,20,4)	30.78	3.5	0.23	0.67	0.46	0.42	40.71	30.78	44.85	30.78
C(20,20,4)	66.91	4.05	0.55	0.14	0.54	1.59	119.35	121.99	110.71	117.13
C(30,20,4)	117.81	2.48	0.78	0.15	0.75	0.16	460.53	579.81	378.78	2082.81
C(40,20,4)	170.51	3.45	0.11	1.06	0.93	0.08	335.84	170.51	423.26	170.51
C(50,20,4)	223.66	4.25	0	0.02	1.05	0.05	585.9	375.91	481.95	370.66
C(60,20,4)	277.6	12.55	0.28	0.07	1.67	0.26	872.02	277.6	918.78	277.6
C(70,20,4)	328.99	6.55	0.65	0.11	1.89	0.08	1671.41	775.03	926.75	824.17
C(80,20,4)	386.56	5.96	0.78	1.95	1.62	0.21	1457.16	506.44	1523.58	506.44
C(90,20,4)	435.65	8.76	0.86	0.11	2.14	0.08	1851.96	656.07	2095.92	660.35
C(100,20,4)	508.17	30.24	1.01	0.03	2.94	0.17	3005.69	508.17	2996.87	508.17
C(200,20,4)	1007.66	21.9	2.24	0.08	10.63	0.24	24780.77	7502.59	26035.11	9681.74
C(300,20,4)	1536.93	71.97	3.2	1.95	24.28	0.22	89329.32	23947.37	99308.4	26933.81
C(400,20,4)	2163.23	69.04	4.96	0.06	29.7	0.2	144762.76	22626.53	147554.56	25566.83
C(500,20,4)	2638.34	34.8	5.98	0.08	23.99	0.97	138260.35	27947.79	147640.44	31834.17
C(600,20,4)	3246.27	60.23	7.03	0.07	36.36	0.23	269652.79	35679.39	284560.39	38842.71
C(700,20,4)	3799.79	77.58	8.34	0.22	19.04	0.16	166874.9	32550.19	178127.54	34796.91
C(800,20,4)	4417.39	114.31	10.17	0.11	29.82	0.18	261561.39	38412.19	281749.53	40141.75
C(900,20,4)	5004.77	112.35	11.01	0.09	29.54	3.06	334699.21	73655.73	364918.63	89636.87
C(1000,20,4)	5592.63	87.27	12.06	0.16	30.78	3.09	389132.82	66721.71	405353.88	74478.27

for obtaining a greedy solution, random solution and optimal value, respectively. Columns eight, nine, ten, and eleven show the execution times for reaching the target solution for each of the variants of the metaheuristic.

As expected, Table 11 shows that the average execution times for producing the initial greedy solution is larger than that of the random solution. Surprisingly, as illustrated in Fig. 11, the time to compute the initial greedy solutions seems not to affect the overall execution times for HC(Greedy) as it is even less than that of HC(Random). Table 11 shows that the execution time required to produce an initial greedy solution is 400 times in most of the cases more than that of random solutions. However, because the average number of function evaluations required by the metaheuristic that starts with greedy solutions (i.e., HC(Greedy) and SA(Greedy)) is far less than those that start with random solutions. Thus, the overall execution time of HC(Greedy) and SA(Greedy) is still less than that of HC(Random) and SA(Random). Therefore, the variants of the metaheuristic that start with the greedy solution used less computational effort regardless of whether or not it is used with Hill climbing or Simulated annealing.

The results of the study can be summarised as follows:

(i) Percent deviation for all variants was nearly the same especially when a limited number of iterations are possible. For example, in as shown in Table 8 for large instances, percent deviation of variants based on greedy solutions was smaller and more stable.

(ii) Standard deviation of solutions from simulated annealing was higher than that of hill climbing. However, for a limited number of function evaluations (i.e., less than 10,000 in our experiments), the standard deviation of simulated annealing was lower than that of hill climbing.

(iii) Metaheuristics that started with greedy solutions attained a 100% success rate much faster and used less execution time than those that started with random solutions.

(iv) Small instance size had no significant effect on robustness and quality of solutions. However, as with large instance sizes, the variants of the metaheuristics that start with a greedy solution required fewer function evaluations to reach the target solution.

(v) Instances with more components per group had less percent deviation, hence a higher chance of producing better quality solutions

The implication of the results is as follows: The benefit of our model-based algorithm is in monitoring, evaluating, adjusting and deploying components of cloud-hosted service (especially for large-scale projects) for guaranteeing multitenancy isolation when there are workload changes. For large-scale cloud-hosted services, running the model-based algorithm with a metaheuristic whose initial solution starts with a greedy solution (compared to random solutions) can significantly boost the quality and robustness of the solutions produced. Solutions from hill climbing were more stable and robust than that of simulated annealing, especially in large instances. However, when there is a limitation in terms of time and resources, simulated annealing will produce more robust and stable solutions for large instances compared to hill climbing. Metaheuristics that started with greedy solutions were more scalable and require fewer function evaluations to reach the target solution when compared to metaheuristics that start with random solutions.

Statistical analysis of results

This section presents a performance assessment of the metaheuristic using the two-way ANOVA model. The primary purpose of a two-way ANOVA is to understand whether there is an interaction between the two independent variables on the dependent variable [70]. The variables of interest are (i) the obtained solutions (for testing quality of solution); (ii) percent deviation of the obtained solution to the target solution(for testing robustness and variability); and (iii) execution time based on the number of functional evaluations required to reach a target solution (for testing computational effort). There are two factors being studied: (i) type of instance, which is classified into two levels - small instances and large instances, and (ii) variant of metaheuristic, which is classified into four levels - HC(Random), HC(Greedy), SA(Random) and SA(Greedy). The computational aspect involves computing F-statistic and p-value ($\alpha = 0.005$) for the hypothesis. This study assumes typical conditions of normality, independence, and equality of variance [45, 71].

In the design, the *type of instance* and the *variant of metaheuristic* has two and four levels, respectively. In all there are $2 \times 4 = 8$ groups. The version of the two-way ANOVA used is the one with more than one observation per cell, but the number of observations in each cell is equal. In our case, each group had six observations making it a total of 46 cells. This version is useful for determining if the *type of instance* and the *variant of metaheuristic* are independent of each other (or if there is interaction); they are independent if the effect of instance size on percent deviation (and standard deviation, success rate, execution time) remains the same, irrespective of whether the variant of metaheuristic used is taken into consideration. Additionally, if there is interaction, then a follow-up analysis is done to determine whether there are any *"simple mains effect"* and what these effects are. Simple mains effect for our problem involves determining the mean difference in percent deviation/standard deviation/success rate/execution time between the type of instance for each variant of the metaheuristic, as well as between variants of the metaheuristic for each type of instance.

The null hypothesis to be tested is:

- H_0: The two factors (i.e., type of instance and variant of metaheuristic) are independent, or that an interaction effect is not present.
- H_1: The two factors (i.e., type of instance and variant of metaheuristic) are not independent, or that an interaction effect is present

The hypothesis is tested for a large instance - C(500, 20, 4), which has 500 groups, 20 components per group and supported by 4 resource types. The data used for the test are as follows: obtained solutions were tested with the set of results from Table 8 (columns 2, 3, 4 and 5), the percent deviation was tested with sets of results from Table 8 (columns 6, 7, 8 and 9) and the computational effort was tested with results from Table 11 (columns 8, 9,10 and 11). Six instance types were selected for statistical analysis each to represent a small instance type and large instance type as follows: (i) small instances ranging from instance C(10,20,4) to C(60,20,4), and (ii) large instances ranging from instance C(500,20,4) to C(1000,20,4).

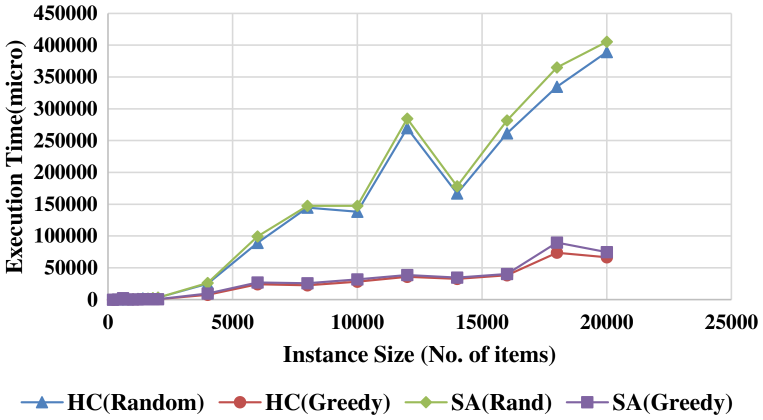


Fig. 11 Computational Effort for a large instance size(C(500-20-4))

The results of the statistical analysis are summarised below:

(i) Quality of Solutions: There was no statistically significant interaction between the effects of instance sizes and different variants of metaheuristic on percent deviation of the obtained solution to the target solution, (F(3, 40) = 0.000, p=1.000)). This means that type of instance and variant of metaheuristic are independent of each other. In other words, the effect of a variant of metaheuristic on quality of solutions (i.e., regarding percent deviation from the target solution) remains the same irrespective of whether the type of instance used is taken into consideration. This result is expected because each variant of the metaheuristic is run for 1000000 function evaluations, which ensured that the search converges to an optimal solution. (see Figs. 12 and 13).

(ii) Robustness: There was a statistically significant interaction between the effects of instance size and variants of the metaheuristic on standard deviation, F (3, 40) = 0.033, p = 0.010. Simple mains effect shows that there was no difference in standard deviation when the different variants of the metaheuristic were applied to small instance sizes. However, there was a significant difference in standard deviation when the different variants of the metaheuristic were applied to large instances. Specifically, out of the six possible combinations, the results shows that there was a significant difference between the following metaheuristics: HC(Random) and SA(Random)(p=0.09), HC(Random) and SA(Greedy) (p=0.013), HC(Greedy) and SA(Random)(p=0.09); and also HC(Random) and SA(Greedy)(p=0.013). There was no difference between HC(Random) and HC(Greedy)(p=1.000) and SA(Random) and SA(Greedy)(p=0.882). This means that for large instance sizes, there is no difference in robustness if hill climbing is started either with the random or greedy solution. The same holds for simulated annealing.(see Figs. 14 and 15).

(iii) Computation Effort: There was a statistically significant interaction between the effects of instance size and variants of the metaheuristic on the execution time (based on the number of function evaluations), F (3, 40) = 19.114, p = 0.000. Simple mains effect shows that there was no difference in execution time when the different variants of the metaheuristic were applied to small instance sizes. However, there was a significant difference in execution time when the different variants of the metaheuristic were applied to large instances. Specifically, out of the six possible combinations, the results show that there was a significant difference between the following metaheuristic - HC (Random) and HC (Greedy) (p=0.000), HC (Random) and SA (Greedy) (p=0.000), HC (Greedy) and SA (Random) (p=0.000) and SA (Greedy) and SA (Greedy) (p=0.000). There was no difference between HC (Random) and SA (Random) (p=0.561) and HC (Greedy) and S A(Greedy) (p=0.843). This means that for large instance sizes, there is no difference in execution time if a random solution is

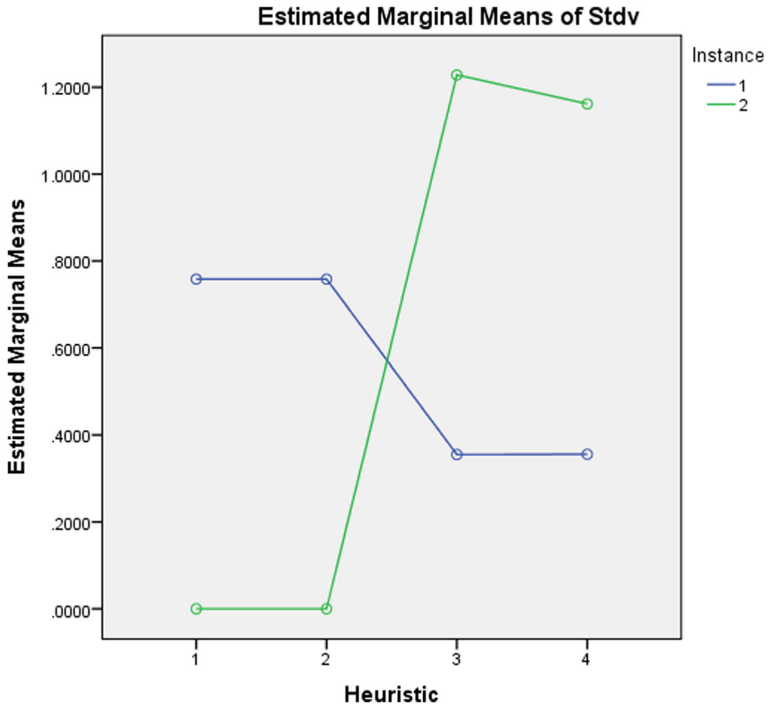


Fig. 12 Quality of solution- 2

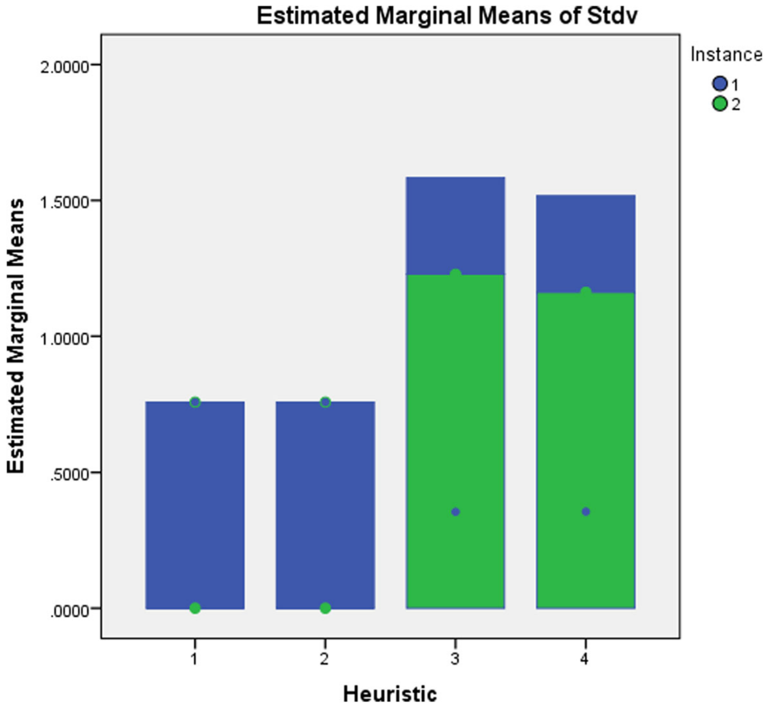


Fig. 13 Robustness of solution - 1

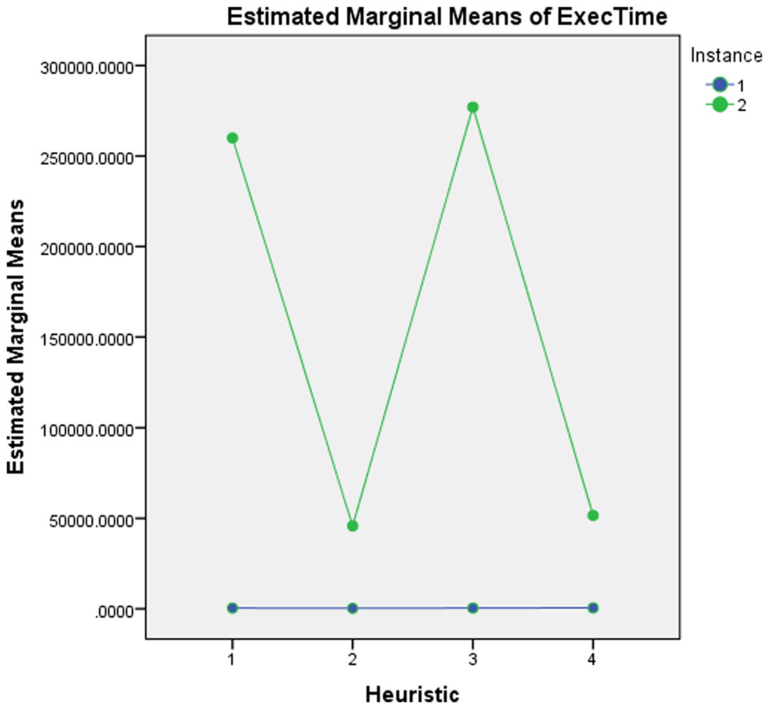


Fig. 14 Robustness of solution - 2

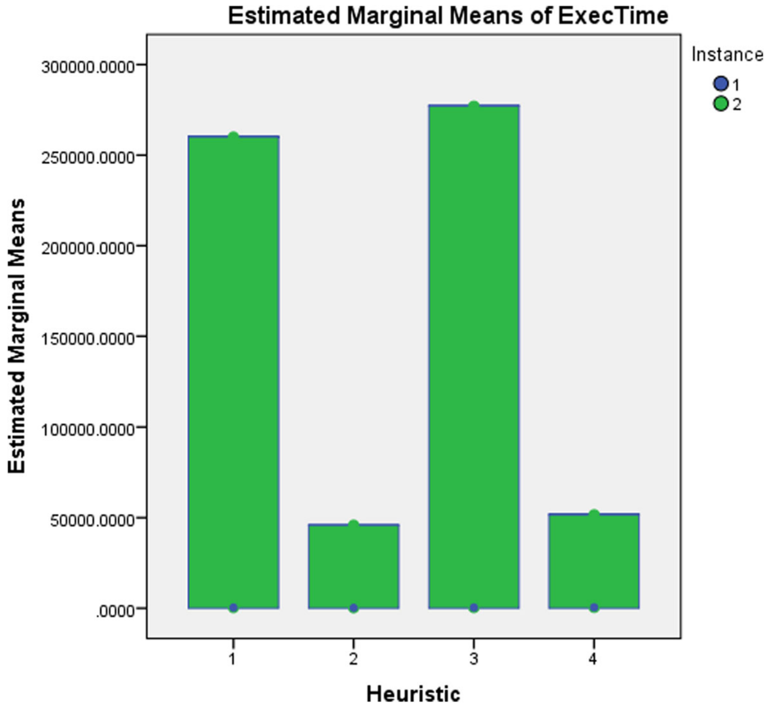


Fig. 15 Computational effort - 1

used to start either hill climbing or simulated annealing. The same holds for the greedy solution. Therefore, the difference is in terms of the initial starting solutions, and not in terms of the variants of the metaheuristic used as was the case with robustness (see Fig. 16).

Discussion

In this section, the results of the study on modelling and simulation are discussed.

(1) Quality of the solutions: The model-based algorithm can be used to obtain high-quality solutions with any of the four variants of metaheuristic when dealing with small instances. The model-based algorithm would perform well both on small problem instances and large problem instances when started with an initial greedy solution (i.e., HC(Greedy) and SA(Greedy). Using a greedy solution and other forms of improvement heuristics to construct an initial solution for the metaheuristic has been shown in several research works to improve the quality of solutions. Many variants of metaheuristic often use initial solutions generated randomly.

The results show that the percent deviation of solutions from instances with five components in a group was higher than the percent deviation from instances with twenty components in a group. This seems to suggest that there may be a greater chance of obtaining better quality solutions when there are more components in a group (i.e., more deployment configurations to choose from). Our approach is well suited for this type of scenario in the sense that it allows us to use during search both intensification (exploitation) and diversification (exploration). A good balance of both will usually improve the performance of the metaheuristic and hence the quality of solutions [56].

Another important lesson from this study is that starting the metaheuristic with an initial set of solutions (e.g., the greedy solutions as used in our approach) can significantly improve the quality of optimal solutions for guaranteeing the required degree of multitenancy isolation. This corresponds with the conclusions from [72] of which the author developed a prototype meta-heuristic load balancer to allocate services on a cloud system without overloading the nodes and maintaining the system stability with minimum cost. The author recommends that better results can be achieved if a solutions pool is initially created from an already pre-computed set.

(2) Robustness of the solutions: The results show that optimalDep algorithm when used with HC (Greedy) and HC (Random) were more robust and stable in small problem instances. However, it was discovered that the problem instances with more components (i.e., m=20) were less robust because the standard deviation was much higher. This means that a cloud-hosted service with several components per group may have a higher chance of producing solutions that are of better quality, but with low robustness or stability.

This could have an adverse impact on cloud-hosted services that may have several interdependencies with other components or cloud-hosted services. Therefore, when working on large open-source projects, it is advisable to limit the number of component choice per group or better still use a combination of local search with greedy principles. This can also help to improve robustness and avoid unstable solutions in environments where the workload is expected to change very frequently. Several research work have made reference to such unstable environments where there are frequent workload changes [1], unpredictable and aggressive workloads [28, 73].

The result shows that variants of metaheuristic based on hill climbing were more stable and robust than simulated annealing. However, when there is a limitation in terms of time and available resources, then simulated annealing would produce stable and robust solutions. Also, in a situation where the workload changes frequently, then hill climbing would be more suitable, but when time and resources are limited, then simulated annealing would be more appropriate.

(4) Computational Effort: The result of the experiments show that the scalability of the solutions and the computational effort required to attain an optimal solution depend in part on the instance size and the type of metaheuristic used. The results of the experiment show that variants of the metaheuristic that start with an initial greedy solution (i.e., HC(Greedy) and SA(Greedy)) were more scalable and they also attained the target solution much faster (i.e., with a fewer number of function evaluations), especially for large instance sizes. Variants of metaheuristic that start with random solutions are suitable either for small problem instances or when there is a need to produce optimal solutions frequently and quickly from large problem instances. Therefore, if frequent provisioning and decommissioning of components characterise a cloud deployment scenario, then the OptimalDep algorithm should be run with either HC(Greedy) or SA(Greedy). As with previous works, our results show that metaheuristics which start with greedy solutions as the initial solution will require less computational effort to provide optimal solutions for deployment [72].

Our approach assumes that the initial solution is computed first before running the metaheuristic, and so it is expected that the time and effort required to calculate the greedy solution will be more than that of a random solution which would have a negative impact on the variants of metaheuristic that start with the greedy solution. However, the results show that the high execution time required to produce a greedy solution was not enough to counter the small number of function evaluations required by metaheuristic that start with greedy solutions to attain the target solutions. Therefore, our model-based algorithm when supported with metaheuristic that starts with greedy solutions would be suitable for handling large-scale cloud deployment projects

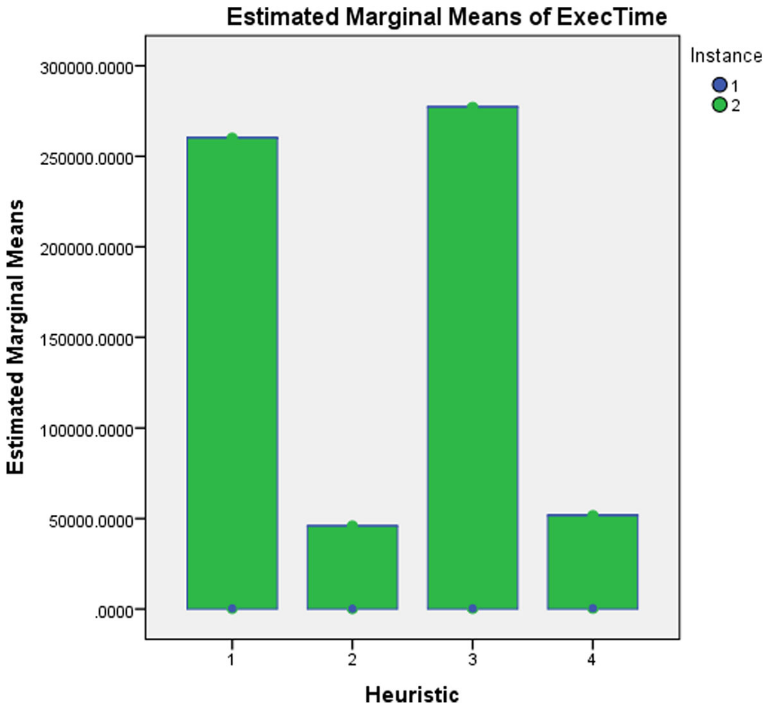


Fig. 16 Computational effort - 2

that may have a significant number of interdependent components.

There are several situations where there is a need to reduce the computational effort required to produce an optimal solution. For example, many customers and cloud providers would be interested in being able to provision and decommission resources so that tenants can access servers and other IT resources more quickly and efficiently while guaranteeing the required degree of multitenancy isolation. Another situation is when there is a need to ensure that a cloud service is failure resistant to guarantee the availability of specific/individual components. Existing approaches do not often guarantee the availability and isolation of individual components but only for a whole cloud service [1]. Our model-based DSS addresses this challenge by first tagging each component and then using a suitable metaheuristic to provide optimal solutions for deploying components of a cloud-hosted service with less computational effort.

Limitations of the study

This study assumed a special case of multitenancy isolation where multiple components of the same tenant, behave as if they were components of different tenants and, thus, are isolated from each other. This is different from the common scenario of multitenancy isolation where there are multiple tenants accessing a component or a cloud-hosted application, and behaving as if they were different tenants [1]. A tenant in this context is a cloud customer (e.g., a single user, a team, department or even a software development company) whose responsibility is to create cloud-hosted applications on top of a cloud platform (such as PaaS offered by Salesforce and Heroku) from several components that it owns and controls.

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

This study assumes that the cost implication of implementing our approach is reflected in the resource consumption of the component; the higher the consumption the costlier the running cost. For example, in IaaS, multitenancy isolation also includes avoiding servers with tenants that need to use the same kinds of resources. Therefore, if running cost is a major concern (e.g., for a small to medium size cloud-hosted service), then the software architect can use the model to select optimal components for deployment that has low resource consumption while guaranteeing the required degree of isolation.

This study did not use a real-world cloud application because our focus was on simulation based on a model. In our previous work, we conducted three case studies using real-world cloud-hosted Global Software Development tools to evaluate the effect of varying degrees of multitenancy isolation on the performance and resource of consumption of components [16, 17, 43]. The study established the basis for developing a model for optimizing the deployment components in a way that guarantees multitenancy isolation.

The dataset (i.e., MMKP instances) used for the simulation experiments on the model-based algorithm was generated randomly following a standard approach used for similar problems. As we could not challenge our exhaustive search algorithm with large instances to produce optimal solutions for comparison with the obtained solutions due to the limitation of the machine used for the experiments, a target solution was computed and used for this purpose. Also, we could not measure the overall computation time of the OptimalDep algorithm (i.e., the main algorithm supporting the DSS) due to the limitation in the hardware(e.g., processor) of the machine used. Therefore we used the number of function evaluations which is a performance indicator that is independent of the computer system for measuring the computational effort required by the metaheuristic solutions to produce the optimal solutions.

The findings of this study should not be generalized to components developed for all types of applications. We focused on cloud-hosted applications(e.g., SaaS applications) developed and deployed using a multitenant architecture. The approach and the associated algorithms that we have presented in this work apply to cloud-hosted applications at the application level, and so are implemented almost at runtime.

Conclusion and future work

This paper presents the implementation of the model-based algorithm for providing optimal solutions for deploying components designed to use (or be integrated with) a cloud-hosted application in a way that guarantees multitenancy isolation, to contribute to the literature on multitenancy isolation and optimising the deployment of components of cloud-hosted services.

The model-based algorithm works as follows: when users requests arrive indicating a change in workload, the DSS solves 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 uses a metaheuristic to find an optimal solution for deploying components with the highest degree of isolation together with the maximum possible number of requests that can be allowed to access the component.

The study revealed that the model-based algorithm, optimalDep, when combined with a metaheuristic 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 seems to suggest that the obtained solutions produced from randomly generated solutions are more sensitive to workload changes than obtained solutions from greedy solutions. For large projects, starting the metaheuristic with an initial solution with a greedy solution can boost the model-based DSS. Also, for large instances, when there are limitations regarding time (e.g., real-time and dynamic environments) and resources (e.g., resource-constrained environment) then simulated annealing produces solutions that are more robust and stable when compared to hill climbing.

We plan to develop other metaheuristics for use with our model-based system to handle large problem instances. For example, we can integrate other types of metaheuristics into the OptimalDep algorithm (i.e., the main algorithm driving the model-based algorithm) or combining simple heuristics with more advanced metaheuristics. Several research works have developed algorithms that combine a genetic algorithm (i.e., a population-based algorithm) with simulated annealing for solving various optimization problems [74, 75]. For example, the authors in [75] have developed the GA-SA-combined algorithm, an algorithm that combines genetic algorithm with simulated annealing for optimization of wideband antenna matching networks.

In the future, we plan to transform the model-based algorithm into a decision support system (DSS) for studying tenant isolation on large-scale cloud-hosted systems designed with multitenant architecture. The DSS can also be used to investigate and predict how components and/or cloud-hosted services will react to workload changes at runtime, based on the required degree of isolation (e.g., shared component or dedicated component). For example, we can design rule-based algorithms to specify that a new set of components be selected for deployment either once an average utilization of components or the whole system exceeds a defined threshold or once the arrival rate of requests exceeds a defined threshold. This decision can help in long-term investments regarding resource consumption and the running cost of components and cloud services.

Endnotes

[1] Apache Kafka is an open-source distributed stream-processing platform for handling real-time data feeds. It allows systems that generate data (called Producers) to persist their data in real-time in an Apache Kafka Topic. Any topic can then be read by any number of systems who need that data in real-time (called Consumers)

[2] The problem instance is defined as: C(number of groups, number of component per group, number of resource types supporting each component). So for example, C(20,20,4) means that the problem instance has 20 groups of components, 20 components per group and supported by 4 resources-CPU, memory, disk space and bandwidth.

[3] Available at https://aws.amazon.com/autoscaling/

[4] Available at https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-overview

Additional files

Additional file 1: Sample file for a large problem instance-C(500,20,4)). (TXT 178 kb)

Additional file 2: Sample file for service demand associated with C(500,20,4). (TXT 810 kb)

Additional file 3: Sample file of an updated instance associated with C(500,20,4). (TXT 340 kb)

Additional file 4: Sample workload file associated with C(500,20,4). (TXT 859 kb)

Abbreviations

DSS: Decision support system; MMKP: Multichoice multidimensional knapsack problem; QN: Queuing network; In addition, Table 1 shows the notations and mapping of the multitenancy problem to QN Model and MMKP

Acknowledgements

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

Funding

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

Availability of data and materials

Not applicable.

Authors' contributions

LCO is the main author of this research paper. AP supervised and reviewed the optimization modelling and the associated experiments. JB contributed to the literature review and general organization of the paper. All authors read and approved the final manuscript.

Authors' information

1. *Dr. Laud Charles Ochei* holds PhD from Robert Gordon University, Aberdeen, United Kingdom. His research interests are in software engineering, distributed systems, cloud computing, and the Internet of things. He has published several research papers in International Conferences and Journals.
2. *Dr. Andrei Petrovski* is a Reader in Computational Systems at Robert Gordon University, Aberdeen, United Kingdom. His primary research interests lie in the field of Computational Intelligence (CI) - particularly, in the application of CI heuristics (such as Genetic Algorithms and Particle Swarm Optimisation) to single- and multi-objective optimisation problems. Andrei has an interest in computer-assisted measurements, virtual instrumentation, and sensor networks.

3. *Dr. Julian Bass* is a Senior Lecturer at the University of Salford, UK. His research interests are in software development for large-scale systems focusing on multi-national teams and using modern lean and agile methods. He also has interests in deployment architectures used in cloud-hosted software services and leading KTP with Add Latent Ltd to develop and deploy cloud-hosted asset management applications for their major clients in the energy and utility sectors.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details

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

Received: 25 January 2018 Accepted: 28 November 2018
Published online: 24 January 2019

References

1. Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P (2014) Cloud Computing Patterns. Springer, London
2. Roche K, Douglas J, Beginning google app engine for java (2009) Beginning Java Google App Engine. Apress, New York City. https://doi.org/10.1007/978-1-4302-2554-6
3. Bauer E, Adams R (2012) Reliability and availability of cloud computing. Wiley, New Jersey
4. Martens A, Ardagna D, Koziolek H, Mirandola R, Reussner R (2010) A hybrid approach for multi-attribute qos optimisation in component based software systems. In: Research into Practice–Reality and Gaps. Springer, Berlin. pp 84–101. https://doi.org/10.1007/978-3-642-13821-8_8
5. Legriel J, Le Guernic C, Cotton S, Maler O (2010) Approximating the pareto front of multi-criteria optimization problems. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin. pp 69–83. https://doi.org/10.1007/978-3-642-12002-2_6
6. Aldhalaan A, Menascé DA (2015) Near-optimal allocation of vms from iaas providers by saas providers. In: Cloud and Autonomic Computing (ICCAC), 2015 International Conference on. IEEE, Boston. pp 228–231. https://doi.org/10.1109/ICCAC.2015.16
7. Ochei L, Petrovski A, Bass J (2016) Optimizing the deployment of cloud-hosted application components for guaranteeing multitenancy isolation. In: IEEE Conference Publications. pp 77–83. 2016 International Conference on Information Society (i-Society 2016)
8. Hon K, Millard C (2017) Eu data protection law and the cloud. International Association of Privacy Professionals. [Online: accessed in February, 2017 from https://iapp.org/resources/article/]
9. Google (2017) Google cloud platform and the eu data protection directive. Google Inc. [Online: accessed in February, 2017 from https://cloud.google.com/security/compliance/eu-data-protection/]
10. Garg SK, Versteeg S, Buyya R (2012) A framework for ranking of cloud computing services. In: Future Generation Computer Systems
11. Kreps J (2016) Introducing kafka streams: Stream processing made simple. Confluent, Inc, California, USA. Online: accessed in November, 2018 from https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/
12. Kafka (2016) Introduction: Apache kafka is a distributed streaming platform. what exactly does that mean? Confluent, Inc., California, USA. Online: accessed in November, 2018 from https://kafka.apache.org/documentation/
13. Manfred Moser M, O'Brien T (2011) The hudson book. Oracle, Inc., USA. Online: accessed in November, 2018 from http://www.eclipse.org/hudson/the-hudson-book/book-hudson.pdf
14. Hudson (2018) Files found trigger. [Online: accessed in January, 2018 from https://plugins.jenkins.io/files-found-trigger]
15. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58. [Online]. Available http://doi.acm.org/10.1145/1721654.1721672
16. Ochei LC, Bass J, Petrovski (a) A (2015) Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools. In: 2015 International Conference on Cloud and Autonomic Computing (ICCAC). IEEE, Boston. pp 101–112. https://doi.org/10.1109/ICCAC.2015.17
17. Ochei LC, Bass J, Petrovski (b) A (2016) Implementing the required degree of multitenancy isolation: A case study of cloud-hosted bug tracking system. In: 13th IEEE International Conference on Services Computing (SCC 2016). IEEE, San Francisco. https://doi.org/10.1109/SCC.2016.56
18. Mehta A (2017) Successful strategies for a multi-tenant architecture. Developer.com(http://www.devx.com/) http://www.devx.com/architect/Article/47662. Accessed Jan 2017
19. Strauch S, Andrikopoulos V, Leymann F, Muhler D (2012) Esb mt: Enabling multi-tenancy in enterprise service buses. In: Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on. IEEE, Taipei. pp 456–463. https://doi.org/10.1109/CloudCom.2012.6427524
20. Khan MF, Mirza AU, et al. (2012) An approach towards customized multi-tenancy. Int J Mod Educ Comput Sci 4(9):39
21. Momm C, Krebs R (2011) A qualitative discussion of different approaches for implementing multi-tenant saas offerings. Softw Eng (Workshops) 11:139–150
22. Aiken L (2017) Why multi-tenancy is key to successful and sustainable software-as-a-service (saas). Cloudbook Journal. [Online: accessed in February 2017 from http://www.cloudbook.net/resources/stories/]
23. Chong F, Carraro G (2006) Architecture strategies for catching the long tail. technical report, microsoft. [Online: accessed in February 2015 from https://msdn.microsoft.com/en-us/library/aa479069.aspx]
24. Wang ZH, Guo CJ, Gao B, Sun W, Zhang Z, An WH (2008) A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In: E-Business Engineering, 2008. ICEBE'08. IEEE International Conference on. IEEE, Xi'an. pp 94–101. https://doi.org/10.1109/ICEBE.2008.60
25. Vengurlekar N (2012) Isolation in private database clouds. Oracle Corporation. Online: Accessed in March, 2015 from http://www.oracle.com/technetwork/database/database-cloud/
26. Mietzner R, Unger T, Titze R, Leymann F (2009) Combining different multi-tenancy patterns in service-oriented applications. In: Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International. IEEE. pp 131–140. https://doi.org/10.1109/EDOC.2009.13
27. Guo CJ, Sun W, Huang Y, Wang ZH, Gao B (2007) A framework for native multi-tenancy application development and management. In: E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on E-Commerce Technology. IEEE, Tokyo. pp 551–558. https://doi.org/10.1109/CEC-EEE.2007.4
28. Walraven S, Monheim T, Truyen E, Joosen W (2012) Towards performance isolation in multi-tenant saas applications. In: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing. ACM, Limassol. p 6. https://doi.org/10.1109/UCC.2015.27
29. Krebs R, Wert A, Kounev S (2013) Multi-tenancy performance benchmark for web application platforms. In: Web Engineering. Springer, Berlin. pp 424–438
30. Moens H, Truyen E, Walraven S, Joosen W, Dhoedt B, De Turck F (2014) Cost-effective feature placement of customizable multi-tenant applications in the cloud. J Netw Syst Manag 22(4):517–558
31. Yusoh ZIM, Tang M (2012) Composite saas placement and resource optimization in cloud computing using evolutionary algorithms. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. IEEE, Honolulu. pp 590–597. https://doi.org/10.1109/CLOUD.2012.61
32. Shaikh F, Patil D (2014) Multi-tenant e-commerce based on saas model to minimize it cost. In: Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on. IEEE, Unnao. pp 1–4. https://doi.org/10.1109/ICAETR.2014.7012861
33. Westermann D, Momm C (2010) Using software performance curves for dependable and cost-efficient service hosting. In: Proceedings of the 2nd International Workshop on the Quality of Service-Oriented Software Systems. ACM, Oslo. p 3. https://doi.org/10.1145/1858263.1858267
34. Candeia D, Santos RA, Lopes R (2015) Business-driven long-term capacity planning for saas applications. IEEE Trans Cloud Comput 3(3):290–303
35. Abbott ML, Fisher MT (2009) The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education, New Jersey. https://doi.org/10.1145/1858263.1858267

36. Leymann F, Fehling C, Mietzner R, Nowak A, Dustdar S (2011) Moving applications to the cloud: an approach based on application model enrichment. Int J Coop Inf Syst 20(03):307–356
37. Vanhove T, Vandensteen J, Van Seghbroeck G, Wauters T, De Turck F (2014) Kameleo: Design of a new platform-as-a-service for flexible data management. In: Network Operations and Management Symposium (NOMS), 2014 IEEE. IEEE, Krakow. pp 1–4. https://doi.org/10.1109/NOMS.2014.6838331
38. Krebs R (2015) Performance isolation in multi-tenant applications. Ph.D. dissertation. PhD thesis (Karlsruhe Institute of Technology). https://se.informatik.uni-wuerzburg.de/fileadmin/10030200/user_upload/dissKIT_BW.PDF
39. Krebs R, Loesch M (2014) Comparison of request admission based performance isolation approaches in multi-tenant saas applications. ACM, Montreal. https://doi.org/10.1145/2405178.2405184
40. Aldhalaan A, Menascé DA (2015) Near-optimal allocation of vms form iaas providers by saas providers. George Mason University, Fairfax
41. Menasce D, Almeida V, Lawrence D (2004) Performance by design: capacity planning by example. Prentice Hall, Upper Saddle River
42. Bass L, Clements P, Kazman R (2013) Software Architecture in Practice, 3/E. Pearson Education, United States
43. Ochei LC, Petrovski A, Bass J (2015) Evaluating degrees of isolation between tenants enabled by multitenancy patterns for cloud-hosted version control systems (vcs). Int J Intell Comput Res 6(3):601–612
44. Cruzes DS, Dybå T (2011) Research synthesis in software engineering: A tertiary study. Inf Softw Technol 53(5):440–455
45. Talbi E-G (2009) Metaheuristics: from design to implementation. Wiley, New Jersey
46. Szyperski C (2007) Component Software: Beyond Object-Oriented Programming, second edition ed. Pearson Education Limited, London WC2E 9AW
47. Chipperfield AJ, Whidborne JF, Fleming PJ (1999) Evolutionary algorithms and simulated annealing for mcdm. In: Multicriteria Decision Making. Springer, New York. pp 501–532
48. Karasakal EK, Köksalan M (2000) A simulated annealing approach to bicriteria scheduling problems on a single machine. J Heuristics 6(3):311–327
49. Martello S, Toth P (1987) Algorithms for knapsack problems. North-Holland Math Stud 132:213–257
50. Kellerer H, Pferschy U, Pisinger D (2004) Introduction to NP-Completeness of knapsack problems. Springer, Berlin. https://doi.org/10.1007/978-3-540-24777-7_16
51. Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. Wiley, New York. https://doi.org/10.1109/NOMS.2014.6838331
52. Yu T, Zhang Y, Lin K-J (2007) Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans Web (TWEB) 1(1):6
53. Akbar MM, Rahman MS, Kaykobad M, Manning EG, Shoja GC (2006) Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. Comput Oper Res 33(5):1259–1273
54. Vondra T, Šedivỳ J (2017) Cloud autoscaling simulation based on queueing network model. Simul Model Pract Theory 70:83–100
55. Dubois DJ, Casale G (2016) Optispot: minimizing application deployment cost using spot cloud resources. Clust Comput 19(2):893–909
56. Rothlauf F (2011) Design of modern heuristics: principles and application. Springer, Berlin. https://doi.org/10.1007/978-3-540-72962-4
57. Parra-Hernandez R, Dimopoulos NJ (2005) A new heuristic for solving the multichoice multidimensional knapsack problem. IEEE Trans Syst Man Cybern Syst Hum 35(5):708–717
58. Cherfi N, Hifi M (2010) A column generation method for the multiple-choice multi-dimensional knapsack problem. Comput Optim Appl 46(1):51–73
59. Beasley JE (1990) Or-library: distributing test problems by electronic mail. J Oper Res Soc 41(11):1069–1072
60. Eckart Z, Marco L Test problems and test data for multiobjective optimizers. Computer Engineering (TIK) ETH Zurich. [Online]. Available: http://www.tik.ee.ethz.ch/sop/…/testProblemSuite/. Retrieved in December, 2018
61. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Trans Evol Comput 3(4):257–271
62. Amazon (2016) Amazon ec2 instance types. Amazon Web Services, Inc. [Online: accessed in September 12, 2016 from https://aws.amazon.com/ec2/instance-types/]
63. Han B, Leblet J, Simon G (2010) Hard multidimensional multiple choice knapsack problems, an empirical study. Comput Oper Res 37(1):172–181
64. Hauck M, Huber M, Klems M, Kounev S, Müller-Quade J, Pretschner A, Reussner R, Tai S (2010) Challenges and opportunities of cloud computing – trade-off decisions in cloud computing architecture,. Karlsruhe Institute of Technology(KIT), Germany. Technical Report. Vol. 2010-19, Tech. Rep
65. Schad J, Dittrich J, Quiané-Ruiz J (2010) Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proc VLDB Endowment 3(1–2):460–471
66. Banati H, Bajaj M (2013) Performance analysis of firefly algorithm for data clustering. Int J Swarm Intell 1(1):19–35
67. Hoos HH, Stützle T (2004) Stochastic local search: Foundations & applications. Elsevier, New York
68. Barrero DF, Muñoz P, Camacho D, R-Moreno MD (2015) On the statistical distribution of the expected run-time in population-based search algorithms. Soft Comput 19(10):2717–2734
69. Hoos H, Stutzle T (1998) Characterizing the run-time behavior of stochastic local search. In: AAAI-99 - Proceedings of the Sixteenth National Conference on Artificial Intelligence and The Eleventh Annual Conference on Innovative Applications of Artificial Intelligence. AAAI Press
70. Laerd.com (2017) Two-way anova in spss statistics. In: Lund Research Ltd. [Online: accessed in February, 2017 from https://statistics.laerd.com/spss-tutorials/]
71. Cohen P (1995) Empirical methods for artificial intelligencemit press. MIT Press, Cambridge
72. Sliwko L, Getov V (2015) A meta-heuristic load balancer for cloud computing systems. In: Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual, vol. 3. IEEE, Taichung. pp 121–126. https://doi.org/10.1109/COMPSAC.2015.223
73. Doddavula SK, Agrawal I, Saxena V (2013) Cloud computing solution patterns: Infrastructural solutions. In: Cloud Computing: Methods and Practical Approaches. Springer, London. pp 197–219
74. Gan G-n, Huang T-l, Gao S (2010) Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In: Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on. IEEE, Taichung. pp 60–63. https://doi.org/10.1109/ICISS.2010.5655013
75. Chen A, Jiang T, Chen Z, Zhang Y (2012) A genetic and simulated annealing combined algorithm for optimization of wideband antenna matching networks. Int J Antennas Propag 2012:1–6. Article ID 251624, https://doi.org/10.1155/2012/251624