AUTHOR: OCHEI, L.C.

TITLE: Architecting the deployment of cloud-hosted services for guaranteeing multitenancy isolation.

YEAR: 2017

This work was submitted to- and approved by Robert Gordon University in partial fulfilment of the following degree:
Doctor of Philosophy, School of Computing Science and Digital Media.

Architecting the Deployment of Cloud-hosted Services for Guaranteeing Multitenancy Isolation

Laud Charles Ochei

A thesis submitted in partial fulfilment
of the requirements of
Robert Gordon University
for the degree of Doctor of Philosophy

May 2017

Abstract

In recent years, software tools used for Global Software Development (GSD) processes (e.g., continuous integration, version control and bug tracking) are increasingly being deployed in the cloud to serve multiple users. Multitenancy is an important architectural property in cloud computing in which a single instance of an application is used to serve multiple users. There are two key challenges of implementing multitenancy: (i) ensuring isolation either between multiple tenants accessing the service or components designed (or integrated) with the service; and (ii) resolving trade-offs between varying degrees of isolation between tenants or components.

The aim of this thesis is to investigate how to architect the deployment of cloud-hosted service while guaranteeing the required degree of multitenancy isolation. Existing approaches for architecting the deployment of cloud-hosted services to serve multiple users have paid little attention to evaluating the effect of the varying degrees of multitenancy isolation on the required performance, resource consumption and access privilege of tenants (or components). Approaches for isolating tenants (or components) are usually implemented at lower layers of the cloud stack and often apply to the entire system and not to individual tenants (or components).

This thesis adopts a multimethod research strategy to providing a set of novel approaches for addressing these problems. Firstly, a taxonomy of deployment patterns and a general process, CLIP (CLoud-based Identification process for deployment Patterns) was developed for guiding architects in selecting applicable cloud deployment patterns (together with the supporting technologies) using the taxonomy for deploying services to the cloud. Secondly, an approach named COMITRE (COmponent-based approach to Multitenancy Isolation Through request RE-routing) was developed together with supporting algorithms and then applied to three case studies to empirically evaluate the varying degrees of isolation between tenants enabled by multitenancy patterns for three different cloud-hosted GSD processes, namely-continuous integration, version control, and bug tracking. After that, a synthesis of findings from the three case studies was carried out to provide an explanatory framework and new insights about varying degrees of multitenancy isolation. Thirdly, a model-based decision support system together with four variants of a metaheuristic solution was developed for solving the model to provide an optimal solution for deploying components of a cloud-hosted application with guarantees for multitenancy isolation.

By creating and applying the taxonomy, it was learnt that most deployment patterns are related and can be implemented by combining with others, for example, in hybrid deployment scenarios to integrate data residing in multiple clouds. It has been argued that the shared component is better for reducing resource consumption while the dedicated component is better in avoiding performance interference. However, as the experimental results show, there are certain GSD processes where that might not necessarily be so, for example, in version control, where additional copies of the files are created in the repository, thus consuming more disk space. Over time, performance begins to degrade as more time is spent searching across many files on the disk. Extensive performance evaluation of the model-based decision support system showed that the optimal solutions obtained had low variability and percent deviation, and were produced with low computational effort when compared to a given target solution.

Keywords: Cloud-hosted Services, Cloud Pattern, Application Component, Global Software Development(GSD) tools, Multitenancy, Degree of Isolation, Metaheuristic, Continuous Integration, Version control, Bug tracking.

Acknowledgments

My family has been very supportive. Thanks so much to Nkem, my darling wife, and our twin boys- David and Daniel. I would also like to thank my Mum, Dad, Aunties and Uncles, Grandma, in-laws, other family members and friends, for their support and prayers.

I would like to thank my supervisors, Julian Bass and Andrei Petrovski, especially for the patience and understanding they showed me when I was struggling in the initial stages of the research. I would like to sincerely thank Julian Bass for agreeing to be part of my supervisory team despite leaving RGU in August 2015. I would like to thank Olivier for the comments I received during the modelling and simulation phase of the research. Many thanks to the School of Computing, IT staff: Colin Beagrie and Tommy Taylor for the technical support in setting up and managing the private cloud used for the experiment. I would also like to thank the School of Computing for approving funds for me to attend and present papers at research conferences in Nice (France), Cambridge(USA) and London (England).

I also wish to thank my network of friends Anthony Etuk, Sadiq Sani, Aminu, Amina, Mariam, Mayowa and Blessing and Ikechukwu. Thanks so much for sharing your experiences with me; that made the research process less stressful. Finally, I would like to thank Sylvester Ofogba, Stanley Okosodo, Gbenga Goker, Rhibetnan Yaktal, for their encouragement; and members of the Fountain of Love (FOL), RCCG, Aberdeen, for being ever present with love and cheer.

Lastly, I am grateful to God Almighty, for His love and care in my life.

This research was sponsored by the Tertiary Education Trust Fund (TETFUND) and the University of Port Harcourt (UNIPORT), Nigeria. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies of the Nigerian or UK Governments. TETFUND, UNIPORT, the Nigerian and U.K. Governments are authorised to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation.

Declarations

I declare that I am the sole author of this thesis and that all verbatim extracts contained in the thesis have been identified as such and all sources of information have been specifically acknowledged in the bibliography. Parts of the work presented in this thesis have appeared in the following publications:

- L. C. Ochei, J. M. Bass, and A. Petrovski, A novel taxonomy of deployment patterns for cloud-hosted applications: A case study of global software development (gsd) tools and processes, International Journal On Advances in Software., vol. volume 8, numbers 3 and 4, pp. 420434, 2015. **(Chapter 4)**

- L. C. Ochei, J. M. Bass, and A. Petrovski, Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools, in 2015 International Conference on Cloud and Autonomic Computing (ICCAC). IEEE, 2015, pp. 101112. **(Chapter 5 and 6)**

- L. C. Ochei, A. Petrovski, and J. Bass, Evaluating degrees of isolation between tenants enabled by multitenancy patterns for cloud-hosted version control systems (vcs), International Journal of Intelligent Computing Research, vol. 6, Issue 3, pp. 601 612, 2015. **(Chapter 5 and 6)**

- L. C. Ochei, A. Petrovski, and J. Bass, An approach for achieving the required degree of multitenancy isolation for components of a cloud-hosted application, in 4th International IBM Cloud Academy Conference (ICACON 2016), 2016. **(Chapter 5 and 6)**

- L. C. Ochei, J. Bass, and A. Petrovski, Implementing the required degree of multitenancy isolation: A case study of cloud-hosted bug tracking system, in 13th IEEE International Conference on Services Computing (SCC 2016). IEEE, 2016.**(Chapter 5 and 6)**

- L. C. Ochei, A. Petrovski, and J. Bass, Optimizing the Deployment of Cloud-hosted Application Components for Guaranteeing Multitenancy Isolation, 2016 International Conference on Information Society (i-Society 2016). ***Best Paper Award***. **(Chapter 7)**

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The advent of the web has led to a significant shift in the way business software is organised. Business applications are no longer designed with a monolithic architecture, where single programs running on a single computer or computer clusters does everything (e.g., data input and output, data processing, error handling, and the user interface). In the past, communications were local and often within an organisation, but now software is highly distributed, sometimes across the world. Business applications are usually designed from extensive reuse of components and programs instead of being programmed from scratch (Sommerville 2011, Stephens 2015).

It has been proposed in the last few years that business applications will not usually run on local computers but run on a "cloud computing environment" that is accessed over the internet. Cloud computing is a "new computing paradigm, whereby shared resources such as infrastructure, hardware platform, and software applications are provided to users on-demand over the Internet (cloud) as services" (Khazaei, Misic & Misic 2012). In a cloud computing environment, software is owned and managed by a software provider, rather than the organisation using the software. Users do not buy software but pay according to how much software is used or are given free access in return for watching adverts that are displayed on their screen (Goth 2008).

For software users, the key benefit of deploying services (or software) to the cloud is that the cost associated with managing the services or software is transferred to the cloud provider (e.g., Google and Amazon). The cloud provider is responsible for installing the software, fixing bugs,

upgrading the software, and dealing with changes to the operating system platform, and ensuring that hardware capacity can meet demand. The cost of managing the software license is zero since there would be no need to license software for several computers owned by the same person. Again, if software is only used occasionally, the pay-per-use model may be cheaper than buying an application (Sommerville 2011).

Assuming the services or functionality delivered through the SaaS model (i.e., hosted centrally and licensed on a subscription basis) is implemented using a service-oriented architecture (SOA) technology, then it becomes possible for applications to use service APIs to access the functionality of other applications, so they can be integrated into more complex systems. We refer to these services as *"cloud-hosted service"*. A cloud-hosted service refers to any resource or functionality hosted in a cloud computing environment. Examples of such services include business software such as CRM, software development tools such as Hudson, office applications such as Google docs, web-based email, photo sharing, etc.

When cloud-hosted services are delivered to multiple users, there is a need for implementing multitenancy (Fehling, Leymann, Retter, Schupeck & Arbitter 2014). Multitenancy is an essential cloud computing architectural property where a single instance of an application is used to serve multiple users. One of the challenges of implementing multitenancy isolation is how to ensure that when there are workload changes, the performance and resource consumption of one of the tenants does not affect other tenants (hereafter referred to as *multitenncy isolation*). The fact that a tenant may require different or varying degrees of isolation makes the task of achieving the required degree of multitenancy isolation even more challenging due to the existence of conflicting trade-offs (Fehling et al. 2014, Sommerville 2011). A high degree of isolation (e.g., a component offering critical functionality) is important for avoiding performance interference, but leads to high resource consumption and running cost while a low degree of isolation (e.g., a component that requires minimal reconfiguration) promotes resource sharing but is more prone to performance interference when workload changes.

This thesis investigates how to architect the deployment of cloud-hosted services in a way that guarantees the varying degrees of multitenancy isolation for tenants (or components) associated with a cloud-hosted service. The type of cloud-hosted service used for illustration is within the domain of software engineering, and in particular software tools used to support Global Software development practices (e.g., Hudson used for continuous integration).

1.2 Problem Context

In recent years, software tools used for Global Software Development (GSD) processes such as continuous integration (CI), version control (VC) and bug tracking (BT), are increasingly being deployed on the cloud (Chauhan & Babar 2012, Buyya, Broberg & Goscinski 2011). For example, large companies like Apple and Oracle are using software tools like Hudson to set up deployments and automate the management of cloud-based infrastructure (Moser & O'Brien 2016). The CI systems used by Saleforce.com (a major cloud provider), runs 150000 + test in parallel across many servers and if it fails it automatically opens a bug report for software architects and developers responsible for that *checkin* (Hansma 2012). It is becoming common practice for distributed enterprises to hire cloud deployment architects or "application deployers" to deploy and manage cloud-hosted GSD tools (Badger, Grance, Patt-Corner & Voas 2012).

These software tools are moving to the cloud in response to the widespread adoption of Global Software Development practices and collaboration tools that support geographically distributed enterprises software projects (Lanubile, Ebert, Prikladnicki & Vizcaíno 2010). In global software development, there are not only software developers, but also many stakeholders such as database administrators, test analysts, project managers, etc. Therefore, there is a need to have software tools that support collaboration and integration among members of the team involved in the software development project (Aspray, Mayadas, Vardi et al. 2006, Herbsleb 2007, Larman & Vodde 2010). This trend will continue because the cloud offers a flexible and scalable platform for hosting a broad range of software services including, APIs and developments tools (Armbrust, Fox, Griffith, Joseph, Katz, Konwinski, Lee, Patterson, Rabkin, Stoica & Zaharia 2010, Buyya et al. 2011, Chauhan & Babar 2012, Bass, Clements & Kazman 2013). Figure 1.1 illustrates the concept of Global Software Development where a groups of developers located in different parts of the world are working together to develop a single software product.

The architectures (or architectural patterns) used to deploy these tools 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 (Junuzovic & Dewan 2006, Bass et al. 2013, Stol, Avgeriou & Babar 2011). Architectural and design patterns have long been used to provide known solutions to a number of common problems facing a distributed system (Bass et al. 2013, Vlissides, Helm, Johnson & Gamma 1995). Collections of patterns mined from projects across

Figure 1.1: Global Software Development. A group of developers located across the world are working together to deliver a single software product.

several industries, already exist for capturing real-life solutions and proven practices. (Hohpe & Woolf 2004, Hanmer 2013).

In a cloud computing environment, cloud patterns represent a well-defined format for describing a suitable solution to a cloud-related problem. For example, collections of cloud patterns exist for describing the cloud and its properties, and how to deploy and use various cloud offerings (Fehling et al. 2014, Homer, Sharp, Brader, Narumoto & Swanson 2014). However, there is little or no research into applying these patterns to describe the cloud-specific properties of applications in the software engineering domain (e.g., collaboration tools for GSD, hereafter referred to as GSD tools) and the trade-offs to consider during cloud deployment. This makes it very challenging to know the deployment patterns (together with the technologies) required for deploying GSD tools to the cloud to support specific software development processes (e.g., continuous integration (CI) of code files with Hudson). Figure 1.2 shows examples of software tools used for Global Software development.

As these software tools are 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 is an important cloud computing property where a single instance of an application is provided to multiple tenants (or components), and so would have to be isolated from each other when-

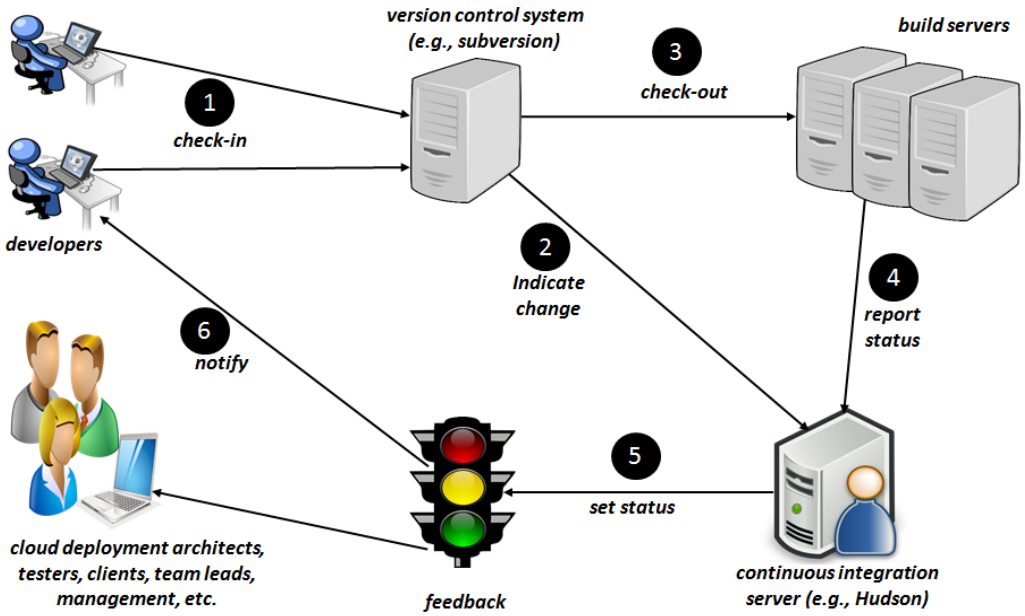

Figure 1.2: Software tools used for Global Software Development

ever there are workload changes (Moens, Truyen, Walraven, Joosen, Dhoedt & De Turck 2014). Therefore, implementing multitenancy implies ensuring that the performance and resources utilisation of one tenant do not affect other tenants when there are workload changes. (Fehling et al. 2014, Bauer & Adams 2012). Furthermore, it is important to note that there are different or varying degrees of isolation. For example, the degree of isolation for a component used for critical functions would be higher than that of a component that requires minor configuration.

If there is a requirement for a high degree of isolation between components, then components must be duplicated or created for each tenant. This duplication leads to high resource consumption and running cost. A low degree of isolation may also be required, in which case, it might reduce resource consumption and running cost since there is sharing of resources, but there is a possibility of interference when workload changes and the application does not scale well (Fehling et al. 2014, Ochei, Bass & Petrovski 2016) . Therefore, the challenge is how to determine optimal solutions that address these trade-offs in the presence of conflicting alternatives (Martens, Ardagna, Koziolek, Mirandola & Reussner 2010, Legriel, Le Guernic, Cotton & Maler 2010).

Multitenancy isolation has been tackled mostly at the data tier level (Chong, Carraro & Wolter 2017, Schneider & Uhle 2013, Zeng 2016), and the main aspects of isolation have been the performance isolation of tenants or components (Kurmus, Gupta, Pletka, Cachin & Haas 2011, Herbst, Krebs, Oikonomou, Kousiouris, Evangelinou, Iosup & Kounev 2016, Krebs 2015). For example,

it is common for companies to install the database software multiple times on a shared server or by using a hypervisor or use hardware virtualization to share resources. This approach is not feasible at the application level, especially when there is need to modify an existing application to compensate for availability and performance challenges.

This thesis focuses on developing approaches and models for architecting the deployment of components of a cloud-hosted service at the application level. In particular, our focus is to enable architects to deploy components of a cloud-hosted service in a way that guarantees the required degree of multitenancy isolation when there are workload changes. The rest of this chapter is organised as follows: Section 1.3 discusses the research motivation including a motivating example and problem statement. The aim and objectives of the research are presented in Section 1.4. Section 1.5 discusses the contributions of the thesis. Section 1.6 outlines the structure of the thesis.

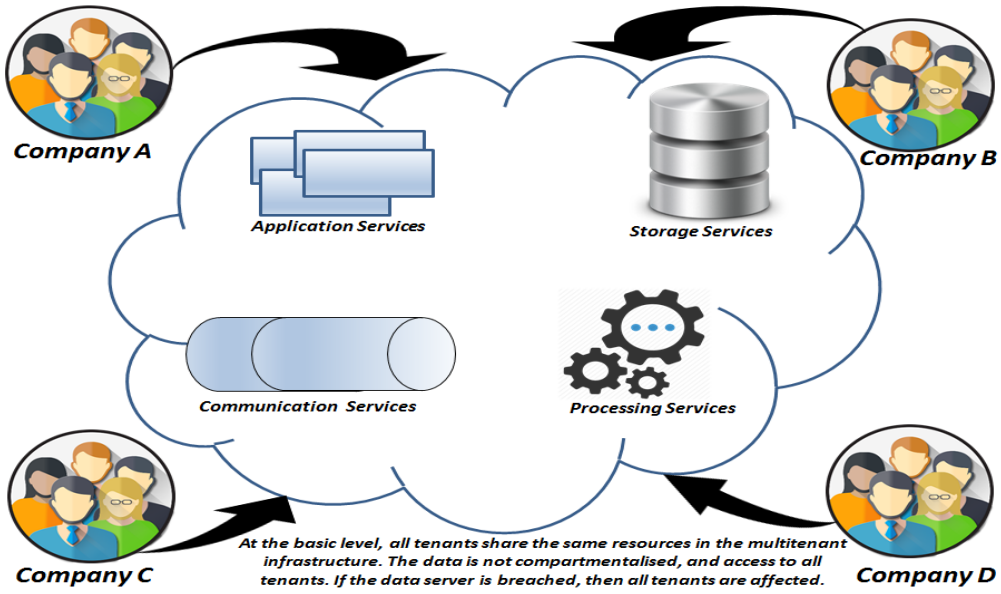

Figure 1.3: Deploying Components of a Cloud-hosted Service in a Multitenant Infrastructure

1.3 Research Motivation

Typical approaches for deploying components of cloud-hosted services for multiple tenants/users rely on multitenancy architectures. 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 (Fehling et al. 2014). Before continuing our discussion, let us consider the following scenario which captures a similar thought process, and serves to elaborate more on our motivation.

Let us assume that there are multiple components of a cloud service hosted on the same or different cloud infrastructure. These components which are of various types and sizes are required to design (or integrate with) a cloud-hosted service and their supporting processes for deployment to multiple tenants. Tenants, in this case, may be multiple users, departments of a company or different companies as shown in Figure 1.3. The laws and regulations of the company make it liable to share and archive data generated from the component (e.g., builds of source code) and keep it accessible for auditing purposes. However, access to some components or some aspects of the archived data will be provided solely to particular groups of tenants for security reasons. The question is: in a resource-constrained environment, how can we architect the optimal deployment of components of this cloud-hosted service in a way that guarantees the required degree of isolation between other tenants when the workload of one of the tenants (or components) experiences a high workload.

The above hypothetical scenario highlights several significant problems as summarised below:

1. The motivating scenario points to the fact that it would not be possible to use one cloud pattern to deploy the service to the cloud due to the different requirements of the service including accessibility of the service to a wider audience and a combined assurance for security and privacy. For instance, the architect would require a combination of several deployment patterns together with supporting technologies for archiving components of the cloud-hosted service (i.e., in a hybrid fashion) to integrate components located in a different cloud environment to form one cloud solution. Moreover, if communication is required internally to exchange messages between application components, then a message-oriented middleware technology would also be needed. The challenge, however, is that there are no existing classifications or frameworks that can be referenced to select suitable patterns together with the supporting technologies.

2. From the motivating scenario, it is clear that some of the tenants would require a higher or different degree of isolation than others. 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 com-

ponents 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 (Hon & Millard 2017, Google 2017). 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 (Fehling et al. 2014). The challenge, therefore, for a cloud deployment architect is that there are no case studies to understand and evaluate the effect of the required degree of isolation on the performance, systems resources and access privileges at different levels of a cloud-hosted service when opting for one (or combinations) of a particular degree of isolation between tenants.

3. Another important point highlighted in the motivating scenario is 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. For example, 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 strict laws and regulations are 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. A low degree of isolation would 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 be affected when one of the components experiences a change in workload.

   Many cloud providers (e.g., Amazon and Microsoft) do not guarantee isolation and availability for a single component (e.g., disk) and but only for the whole system (Fehling et al. 2014). This re-enforces the need to automate the monitoring and management of components of cloud-hosted services to guarantee multitenancy isolation. Therefore, 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 decision to be taken in the presence of a trade-off between two or more conflicting objectives (Martens et al. 2010) (Legriel et al. 2010).

The main research question this thesis addresses is: ***"How can we architect the deployment of cloud-hosted services for guaranteeing multitenancy isolation?"***. This research question is further divided into three sub-questions as follows:

1. How can we create and use a taxonomy for selecting appropriate deployment patterns together with the supporting technologies for deploying services to the cloud?

2. How can we evaluate the varying degrees of isolation between tenants enabled by multitenancy patterns for cloud-hosted services?

3. How can we optimise the deployment of components of a cloud-hosted service to guarantee multitenancy isolation?

1.4 Research Aim and Objectives

The aim of this research is to provide a framework for architecting the optimal deployment of components of a cloud-hosted service in order to guarantee the required degree of multitenancy isolation. The specific objectives of the research are:

1. To create a taxonomy and demonstrate its practicality for selecting applicable deployment patterns together with the supporting technologies for cloud deployment of GSD tools.

2. To develop an approach for implementing the required degree of multitenancy isolation and demonstrate its practicality using different cloud-hosted GSD processes.

3. To conduct three case studies that apply the approach developed in 2 to empirically evaluate the varying degrees of isolation between tenants enabled by multitenancy patterns for three different cloud-hosted GSD processes, namely-continuous integration, version control, and bug tracking.

4. To synthesise the findings of the three case studies (conducted in 3) to provide an explanatory framework and new insights on implementing the varying degrees of multitenancy isolation.

5. To develop a model-based *decision support system* (DSS) for providing optimal solutions for deploying components of a cloud-hosted service for guaranteeing multitenancy isolation.

6. To develop and evaluate a metaheuristic solution for solving the model (developed in 5).

1.5 Contributions of the Thesis

As discussed earlier, deploying services to the cloud to serve multiple users requires implementing multitenancy so that the required performance and resource consumption of other tenants are not affected if one of the tenants experiences high load. However, achieving multitenancy isolation is challenging due to the demand for different or varying degrees of isolation between tenants. When implementing multitenancy, the usual assumptions are: (i) the shared component promotes resource sharing, but is prone to performance interference and so guarantees a low degree of isolation, (ii) the dedicated component guarantees a high degree of isolation, but with limitations of high resource consumption and running cost, and reduction in the number of tenants allowed to access the cloud-hosted service.

This thesis argues that by employing a set of approaches including using taxonomy to select suitable deployment patterns, evaluating the varying degrees of multitenancy isolation required by tenants, and simulation based on an optimization model, we can architect the deployment of cloud-hosted services for guaranteeing multitenancy isolation by maximising both the required degree of isolation for tenants and the number of requests allowed to access a cloud-hosted service.

The key contributions of this thesis are:

1. A novel taxonomy of deployment patterns and a general process, CLIP (CLoud-based Identification process for deployment Patterns) has been developed for guiding architects in selecting applicable cloud deployment patterns (together with the supporting technologies) using the taxonomy for deploying services/application to the cloud.

2. A novel approach, COMITRE (Component-based approach to multitenancy isolation through Request Re-routing) has been developed together with supporting algorithms for implementing the varying degrees of multitenancy isolation for cloud-hosted services.

3. The practicality of the COMITRE approach was demonstrated by applying it to three case studies that empirically evaluated the varying degrees of isolation between tenants enabled by multitenancy patterns for three different Global Software Development processes: continuous integration, version control, and bug tracking.

4. A synthesis of findings from the three case studies was carried out to provide an explanatory framework and new insights on the effect of the required degree of multitenancy isolation on the optimal deployment of components of cloud-hosted services under different cloud deployment conditions.

5. A novel model-based *decision support system*(DSS), OptimalDep, has been developed to provide optimal solutions for deploying components of a cloud-hosted service for guaranteeing multitenancy isolation. The model-based decision support system combines an open multiclass Queuing Network Model and multiobjective optimisation model (based on Multichoice Multidimensional Knapsack Problem MMKP).

6. Four variants of a metaheuristic solution: HC(Random), HC(Greedy), SA(Random) and SA(Greedy) have been developed for solving the optimisation model integrated into the decision support model (in 5). The first two variants are based on the Hill climbing algorithm while the last two variants are based on Simulated annealing algorithm. These metaheuristic solutions are required in a cloud environment to sample a sets of solutions for guaranteeing multitenancy isolation which are often too large to be completely sampled or are required in a dynamic and real-time situations (e.g., timely provisioning of components due to frequent workload changes).

These contributions can be illustrated through a layered architecture in Figure 1.4, which shows how the different approaches presented in this thesis work together to support the task of architecting the deployment of components of a cloud-hosted service for guaranteeing multitenancy isolation. Layer one represents our novel taxonomy and a general process, CLIP, for guiding software architects in selecting applicable cloud deployment patterns (together with the support-

ing technologies) for deploying GSD tools. CLIP has been applied to a motivating deployment problem involving the cloud deployment of a GSD tool to serve multiple users in such a way that guarantees isolation among different users. Layer two represents our approach, COMITRE, for implementing and empirically evaluating the required degree of multitenancy isolation between tenants enabled by three multitenancy patterns in three case studies involving different GSD processes. The three GSD processes (continuous integration, version control and bug tracking) and the three (cloud) multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) were identified in layer one.

In layer three, a new whole out of parts (i.e., by synthesising the findings of the three primary case studies) has been made to provide a novel explanatory framework and new insights into the effect of multitenancy isolation on the three different GSD processes. In layer four, new insights acquired from the case study synthesis are used to develop a model-based decision support system (DSS) for providing near-optimal solutions for deploying components of a cloud-hosted service for guaranteeing the required degree of multitenancy isolation.

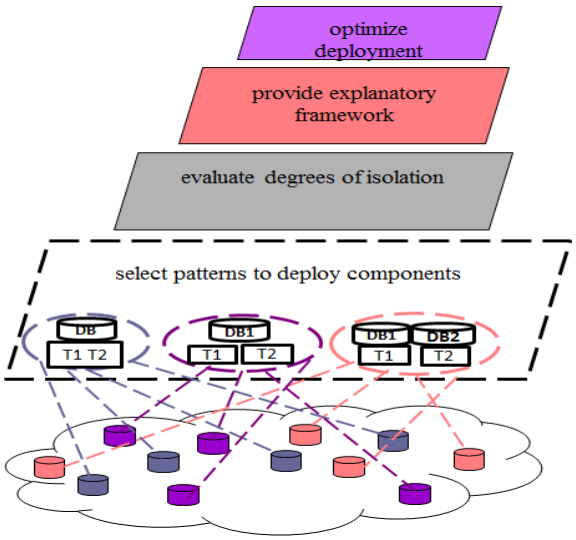

Figure 1.4: A layered architecture for architecting the deployment of cloud-services for guaranteeing multitenancy isolation

1.6 Thesis Structure

This thesis is divided into nine chapters. The remaining chapters are organised as follows:

Chapter 2 Literature Review: Chapter 2 provides an overview of basic concepts used in the

thesis, including relevant work related to Global Software Development (GSD), cloud-hosted GDS tools, cloud deployment patterns, multitenancy isolation, and optimal allocation of resources in a cloud environment.

Chapter 3 Methods: In Chapter 3, the methodology of the research which is the multimethod research method, is discussed. The multimethod research combines two or more research methods (which are conducted separately following the usual procedure of each method) to form one single research process. The key research methods used in this thesis are described: exploratory study, case study (i.e., made up of the three case studies), case study synthesis and simulation based on a model.

Chapter 4 Taxonomy of Cloud Deployment Patterns for Cloud-hosted Services: Chapter 4 discusses the development and use of a taxonomy for selecting applicable deployment patterns together with the supporting technologies for deploying GSD tools. A general process for guiding architects in selecting appropriate deployment patterns using the taxonomy is described.

Chapter 5 Case Studies of Evaluating Degrees of Multitenancy Isolation: Chapter 5 presents three case studies that empirically evaluated the effect of varying degrees of isolation on the required performance, and the resource consumption of tenants. The case studies were based on three different cloud-hosted Global Software Development (GSD) processes: continuous integration, version control, and bug tracking.

Chapter 6 Synthesis of Case Studies of Evaluating Degrees of Multitenancy Isolation: Chapter 6 discusses the synthesis of findings of the three case studies to provide an explanatory framework and new insights on multitenancy isolation. This framework provides information on the (i) commonalities and differences observed in the case studies, and (ii) trade-offs to consider when implementing the required degree of multitenancy isolation.

Chapter 7 Optimal Deployment of Components for Guaranteeing Multitenancy Isolation: Chapter 7 presents a model-based decision support model which combines a queuing network model and an optimisation model to provide optimal solutions for deploying components of a cloud-hosted service for guaranteeing multitenancy isolation. Also, four variants of a metaheuristic solution are presented for solving the model integrated into the decision support system.

Chapter 8 Discussion: Chapter eight discusses the implications of the results and how the different aspects of the work contribute to solving the problem addressed in this thesis. The different areas where our work can be applied to support the deployment of components of a cloud-hosted

service in a way that guarantees multitenancy isolation have been presented.

Chapter 9 Conclusion and Future Work: Chapter 9 revisits the contributions of the thesis and then discusses the scope and limitations of the study. A reflection on the PhD has been presented by highlighting the challenges and lessons learned. Finally, this chapter presents the future work and after that concludes the thesis.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents an overview of the basic concepts and existing literature related to the task of architecting the deployment of components of a cloud-hosted service for guaranteeing multi-tenancy isolation. Our literature review will be divided into four main sections as follows. The first section gives an overview of Global Software Development (GSD), software processes and supporting tools that have been found to have the most impact on Global Software Development, and the deployment of software tools to the cloud to support GSD. Section two discusses existing taxonomies and classifications of architectural patterns and cloud deployment patterns for deploying cloud services and software tools to the cloud. Section three gives an overview of multi-tenancy isolation and the challenges of achieving the required degree of isolation between tenants (or components) when workload changes. Section four will discuss related work on the optimal deployment of components of cloud-hosted services for guaranteeing multitenancy isolation.

2.2 Cloud-hosted GSD Processes and Supporting Tools

The section will first introduce the concept of Cloud computing, Global Software Development and thereafter discusses the key software processes and their role in supporting Global Software Development practices.

2.2.1 Cloud Computing

Simply put, cloud computing is the delivery of software and functionality as services over the internet by service providers. According to the National Institute of Standards and Technology, it is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (Mell & Grance 2011). (Buyya et al. 2011) defined cloud computing as a "parallel and distributed computing system consisting of a collection of inter-connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the service provider and consumers."

According to (Armbrust et al. 2010), "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 provides those services." The cloud entails the data center hardware and the software. 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).

Although there are so many definitions that have been given for the term cloud computing, there is common agreement on the basic characteristics of a cloud computing environment. These include (Buyya et al. 2011) - pay-per-use, elastic capacity and the illusion of infinite, self-service interface, and resources that are abstracted or virtualized.

There are three basic cloud service models:

(i) Software as a Service (SaaS): In the SaaS model, cloud providers can install, operate and access their application software using a web browser. An example of a SaaS provider is Salesforce.com, which utilizes the SaaS model to provide Customer Relationship Management (CRM) applications located on their server to customers. This eliminates the need for customers to run and install the application on their own computers.

(ii) Platform as a Service (PaaS): In the PaaS model, cloud providers deliver cloud platforms which represent an environment for application developers to create and deploy their applications. A notable example of PaaS is the Google App Engine, which provides an environment for creating and deploying web-based applications written in specific programming languages.
(iii) Infrastructure as a Service (IaaS): In the IaaS model, cloud providers offer physical (computers, storage) and virtualized computer resources. Examples of IaaS providers include: Amazon EC2, and Azure Services Platform.

Cloud computing provides a dependable and scalable platform for delivering services and software either as a SaaS, PaaS or IaaS to multiple users. Most services such as business software (e.g., CRM) and software development tools (e.g., Hudson) are deployed to the cloud using the SaaS delivery model. The SaaS model is very attractive to both customers and providers because the Web browser can be utilised as a universal client. For the customers, the service is flexible and easy to use. For the cloud provider, the service can be easily delivered and improved since a single version of the service/software is at a centralised location. Because of the ability and desire to change or upgrade services provided using the SaaS model, the Agile software development process is mostly used in the software development life-cycle (Fox, Patterson & Joseph 2013, Sommerville 2011).

2.2.2 Global Software Development

In recent times, Global Software Development has emerged as the dominant methodology used is developing software for geographically distributed enterprises. The number of large-scale geographically distributed enterprise software development projects involving governments and large multi-national companies is on the increase (Aspray et al. 2006, Herbsleb 2007, Larman & Vodde 2010).

Definition 2.1: **Global Software Development.** GSD is defined by Lanubile (Lanubile 2009) as the splitting of the development of the same software product or service among globally distributed sites. Global Software Development involves several partners or sites of a company working together to reach a common goal, often to make a product (in this case, software) (Lanubile 2009, Pesola, Tanner, Eskeli, Parviainen & Bendas 2011).

In geographically distributed enterprise software development, there are not only software developers, but many stakeholders such as database administrators, test analysts, project managers,

etc. Therefore, there is a need to have software tools that support collaboration and integration among members of the team involved in the software development project. As long as a software project includes more than one person, there has to be some form of collaboration (Babar & Zahedi 2012, Pesola et al. 2011, Bass 2014, Herbsleb & Mockus 2003).

2.2.3 GSD Tools and Supporting Processes

Cloud-hosted software services play an important role in Global Software Development (GSD) practices. There are different types of essential software processes used to support GSD. Four examples of widely used Global Software Development processes are discussed below (Portillo-Rodriguez, Vizcaino, Ebert & Piattini 2010):
(1) Continuous Integration (CI): CI is a development practice that requires developers to integrate the source code into a shared repository several times. Each check-in is then verified by an automated build, allowing teams to detect problems early (Fowler 2017). Hudson is a widely used GSD tool used for continuous integration, which is written in Java for deployment in a cross-platform environment. Hudson is hosted partly as an Eclipse Foundation project and partly as a Java.NET project. It has a comprehensive set of plugins, making it easy to integrate with other software tools. Organisations such as Apple and Oracle use Hudson for setting up production deployments and automating the management of cloud-based infrastructure (Moser & O'Brien 2016).

(2) Version Control: Version control is the process of tracking incremental versions of files and, in some cases, directories over time, so that specific versions can be recalled later (Collins-Sussman, Fitzpatrick & Pilato 2004). In Global software development, version control systems are being relied upon as a communication medium for developers in a software development team. For example, viewing past revisions and changesets is a valuable tool to see how a project has evolved and for reviewing teammates code (Herbsleb 2007).

In Global Software Development, cloud-hosted Version Control Systems are used to ensure that changes happening across different environments (some of which may be static data centres) are properly monitored and controlled across various layers and environments of an application software (Krishna & Jayakrishnan 2013).

There are two main categories of version control systems: **centralized** (e.g., Subversion) and **distributed** (e.g., Git and Mercury). This thesis focuses on the centralized version control system,

which works in a client and server relationship. That is, the repository is located in one place and provides access to many clients. It can be likened to a scenario where an FTP client connects to an FTP server. All changes and commits by users are sent and received from the central repository. A widely used GSD tool for version control is Subversion (Collins-Sussman et al. 2004). Subversion implements a centralised repository architecture whereby a single central server hosts all project metadata. This facilitates distributed file sharing (Lanubile et al. 2010).

(3) Issue/Bug Tracking: Bug tracking (or issue tracking) is the process of keeping track of reported software bugs or issues in software development projects. Examples of widely used error and bug tracking tools are JIRA (Atlassian.com 2016), ITracker, Rational ClearQuest, and TrackStudio. This thesis focuses on Bugzilla, a web-based general-purpose bug tracker and testing tool, originally developed and used for the Mozilla project (Bugzilla 2016).

Bug tracking, as used in this thesis, also includes issues and enhancements to an application and is not only restricted to error-related data such as stack traces and log files. However, we do not include task registry, which is more related to the function of a project management system (Serrano & Ciordia 2005a).

The main component of a bug tracking system is the database that stores bugs and attachments, which require isolation. Attachments are usually added to complement the process of submitting a bug. Developers are usually encouraged to use attachments instead of comments especially for large chunks of ASCII data, such as trace, debugging output files, or log files (Bugzilla 2016). These attachments have to be isolated as bugs can be assigned to different teams members for resolution.

(4) Agile Management: The development of cloud-hosted services is not usually driven by user requirements, but by the service providers assumptions about what users need. The software, therefore, needs to be able to evolve quickly after the providers get feedback from users on their requirements. Agile development with incremental delivery is, therefore, a commonly used approach for software that is to be delivered as a service (Fox et al. 2013).

Agile methodologies are increasingly being used in Global Software Development projects. Agile architects propose initial architecture and run with that until its technical requirements become too difficult or complicated, at which point they need to refactor (Bass et al. 2013). Agile

management has to be adapted to cope with large projects. Sommerville suggests some critical adaptations that can be introduced into agile management such as continuous integration and cross-team communication mechanisms (Sommerville 2011). One of the essential tools used for managing agile practices is VersionOne (Versionone.com 2017a, VersionOne.com 2017b).

2.2.4 Deployment of Software Tools to the Cloud

Software tools used for Global Software Development projects are increasingly being moved to the cloud (Chauhan & Babar 2012). This is in response to the widespread adoption of Global Software Development practices and collaboration tools that support geographically distributed enterprises software projects (Lanubile et al. 2010). This trend will continue because the cloud offers a flexible and scalable platform for hosting a broad range of software services including, APIs and developments tools (Buyya et al. 2011, Chauhan & Babar 2012).

Definition 2.2: **Cloud-hosted GSD Tool.** "Cloud-hosted GSD tool" refers to collaboration tools used to support GSD processes in a cloud environment. The standards adopted are (i) NIST Definition of Cloud Computing to define properties of cloud-hosted GSD tools (Liu, Tong, Mao, Bohn, Messina, Badger & Leaf 2011); and (ii) ISO/IEC 12207 standard as a frame of reference for defining the scope of a GSD tool (Singh 1996). Portillo et al. (Portillo-Rodriguez et al. 2010) identified three groups of GSD tools for supporting ISO/IEC 12207 processes:

(i) Tools to support Project Processes- These tools are used to support the management of the overall activities of the project. Examples of these processes include project planning, assessment and control of the various processes involved in the project. Several GSD tools fit into this group. For instance, JIRA and Bugzilla are software tools widely used in large software development projects for issue and bug tracking.

(ii) Tools to support Implementation Processes such as requirements analysis and integration process. For example, Hudson is a widely used tool for continuously integrating different source code builds and components into a single unit (Moser & O'Brien 2016, Wiest 2017).

(iii) Tools for Support Processes - Software tools that fall into this group are used to support documentation management processes and configuration management processes involved in the software development project. For example, Subversion is a software tool used to track how the different versions of a software evolve over time.

These GSD tools which are also referred to as Collaboration tools for GSD (Portillo-Rodriguez et al. 2010), are increasingly being deployed to the cloud for Global Software Development by large distributed enterprises. The work of Portillo et al. (Portillo-Rodriguez et al. 2010) presents the requirements and features of GSD tools and also categorises various software tools used for collaboration and coordination in Global Software Development.

2.3 Architectures for Cloud-hosted Applications

The previous section established that collaboration tools used to support Global Software Development (GSD) processes are increasingly being deployed on the cloud (Ochei, Bass & Petrovski 2015b, Buyya et al. 2011, Chauhan & Babar 2012). The architectures or cloud patterns used to deploy these tools to the cloud are of great importance to software architects because they determine whether or not the system's essential quality attributes (e.g., performance) will be exhibited (Junuzovic & Dewan 2006, Bass et al. 2013, Stol et al. 2011). The basic concepts of architectural patterns and cloud patterns, and their relevance in deploying software tools to the cloud are discussed in the sections that follow.

2.3.1 Architectural Patterns

Architectural and design patterns have long been used to provide known solutions to many common problems facing a distributed system (Bass et al. 2013, Vlissides et al. 1995). The architecture of a system/application determines whether or not its required quality attributes (e.g., performance, availability and security) will be exhibited (Junuzovic & Dewan 2006, Bass et al. 2013).

Definition 2.3: **Architectural Pattern.** Architectural patterns are compositions of architectural elements that provide packaged strategies for solving recurring problems facing a system (Bass et al. 2013). Architectural patterns can be broadly classified into three groups based on the nature of the architectural elements they use (Bass et al. 2013):

(i) Module type patterns - which show how systems are organised as a set of codes or data units in the form of classes, layers, or divisions of functionality.

(ii) Component-and-connector (C&C) type patterns - which show how the system is organised as a set of components (i.e., runtime elements used as units of computation, filters, services, clients, servers, etc.) and connectors (e.g., communication channels such as protocols, shared messages,

pipes, etc.).

(iii) Allocation patterns - which show how software elements (typically processes associated with C&C and modules) relate to non-software elements (e.g., CPUs, file system, networks, etc.) in its environment. In other words, this pattern shows how the software elements are allocated to elements in one or more external environments in which the software is executed.

2.3.2 Cloud Deployment Patterns

In the cloud computing environment, a cloud pattern represents a well-defined format for describing a suitable solution to a cloud-related problem (Fehling et al. 2014). Several cloud problems exist such as how to: (i) select a suitable type of cloud for hosting applications; (ii) select an approach for delivering a cloud service; (iii) deploy a multitenant application that guarantees the isolation of tenants. Cloud deployment architects use cloud patterns as a reference guide that documents best practice on how to design, build and deploy applications to the cloud.

Definition 2.4: **Cloud Deployment Pattern.** A "Cloud deployment pattern" is defined as a type of architectural pattern, which embodies decisions as to how elements of the cloud application will be assigned to the cloud environment where the application is executed.

Our definition of cloud deployment pattern is similar to the concept of design patterns (Vlissides et al. 1995), (architectural) deployment patterns (Bass et al. 2013), collaboration architectures (Junuzovic & Dewan 2006), cloud computing patterns (Fehling et al. 2014), cloud architecture patterns (Wilder 2012), and cloud design patterns (Homer et al. 2014). These concepts serve the same purpose in the cloud (as in many other distributed environments). For example, the generic architectural patterns- client-server, peer-to-peer, and hybrid (Bass et al. 2013) - relate to the following: (i) the 3 main collaboration architectures, i.e., centralized, replicated and hybrid (Junuzovic & Dewan 2006); and (ii) cloud deployment patterns, i.e., 2-tier, content distribution network and hybrid data (Fehling et al. 2014).

One of the key responsibilities of a cloud deployment architect is to allocate elements of the cloud application to the hardware processing (e.g., processor, files systems) and communication elements (e.g., protocols, message queues) in the cloud environment so that the required quality attributes can be achieved. Figure 2.1 shows how the elements of Hudson (a typical of GSD tool) are mapped to the elements of the cloud environment. Hudson runs on an Amazon EC2 instance

while the data it generates is regularly extracted and archived on separate cloud storage (e.g., Amazon S3).

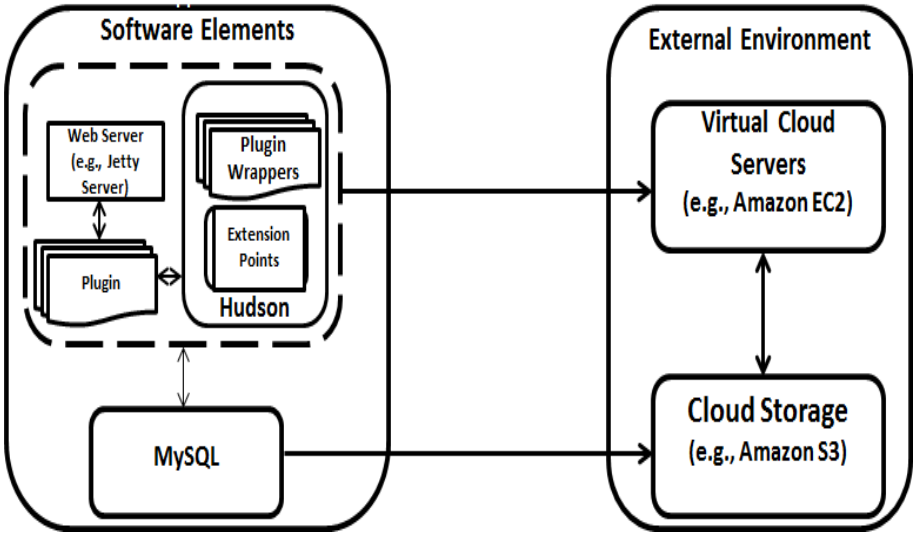

Figure 2.1: Mapping elements of a GSD tool to External Environment

2.4 Taxonomies and Classifications of Cloud Deployment Patterns

This section defines what a taxonomy is, and its relevance in software engineering. This is followed by a discussion of existing taxonomies and classifications of deployment patterns for cloud-hosted services and the shortcomings of these taxonomies. This discussion will lay the foundation for creating and using a novel taxonomy that will address these deficiencies.

2.4.1 What is a Taxonomy and its Purpose?

The IEEE Software & Systems Engineering Standards Committee defines a Taxonomy as "a scheme that partitions a body of knowledge into taxonomic units and defines the relationship among these units. It aims for classifying and understanding the body of knowledge (IEEE 1990)." As understanding in the area of cloud patterns and cloud-hosted software tools for distributed enterprise software development evolves, relevant concepts and relationships between them emerge that warrant a structured representation of these concepts. Being able to communicate that knowledge provides the prospects to advance research (Unterkalmsteiner, Feldt & Gorschek 2013).

Taxonomies and classifications facilitate the systematic structuring of complex information.

Taxonomies are mechanisms that can be used to structure, advance understanding and communicate this knowledge (Glass & Vessey 1995). According to Sjoberg (Sjoberg, Dyba & Jorgensen 2007), the development of taxonomies is crucial to documenting the theories that accumulate knowledge of software engineering. In software engineering, they are used for comparative studies involving tools and methods, for example, software evolution (Buckley, Mens, Zenger, Rashid & Kniesel 2005) and Global Software Engineering (Smite, Wohlin, Galvina & Prikladnicki 2012). The work of Glass and Vessey (Glass & Vessey 1995) and Bourque and Dupuis (Dupuis 2004) laid the foundation for developing various taxonomies for software development methods and tools in software engineering.

2.4.2 Related Work on Taxonomies and Classifications of Cloud Deployment Patterns

Several attempts have been made by researchers to create classifications of cloud patterns to build and deploy cloud-based applications. Wilder (Wilder 2012) describes eleven patterns: Horizontally Scaling Compute, Queue-Centric Workflow, Auto-Scaling, MapReduce, Database Sharding, Busy Signal, Node Failure, Colocate, Valet Key, Content Delivery Network, and Multisite Deployment. The authors then illustrate how each pattern can be used to build cloud-native applications using the Page of Photos web application and Windows Azure. Each pattern is preceded by what the authors refer to as "primers" to provide a background of why the pattern is needed. A description is provided about how each pattern is used to address specific architectural challenges that are likely to be encountered during cloud deployment. The Multisite deployment pattern is an interesting cloud pattern because it can be used to deploy a single application to more than one data center.

A collection of over 75 patterns for building and managing a cloud-native application are provided by Fehling et al. (Fehling et al. 2014). The "known uses" of the implementation of each pattern is provided with examples of cloud providers offering products that exhibit the properties described in the pattern. This helps to further give a better understanding of the core properties of each pattern. The examples of known uses of patterns under the "storage offering" category (e.g., blob storage, key-value storage) are very useful in understanding how to modify a GSD tool to access a cloud storage. For example, Amazon S3 and Google Cloud storage are products offered by Amazon and Google, respectively, for use as blob storage on the cloud. Blob storage is based on object storage architecture, and so the GSD tool should be modified to allow access using a

REST API.

Homer et al. (Homer et al. 2014) describe: (i) twenty-four patterns that are useful in developing cloud-hosted applications; (ii) two primers and eight guidance topics that provide basic information and good practice techniques for developing cloud-hosted applications; and (iii) ten sample applications that illustrate how to implement the design patterns using features of Windows Azure. The sample code (written in C#) for these sample applications is provided, thus making it easy for architects who intend to use similar cloud patterns to convert the codes to other web programming languages (e.g., Java, Python) for use in other cloud platforms.

Moyer (Moyer 2012) discusses a collection of patterns under the following categories: image (e.g., prepackaged images), architecture (e.g., adapters), data (e.g., queuing, iterator), and clustering (e.g., n-tier) and then use a simple Weblog application written using Amazon Web Services (AWS) with Python to illustrate the use of these patterns. For example, one of the architectural patterns- Adapters, is similar to "Provider Adapter" pattern described by Fehling et al. (Fehling et al. 2014), which can be used for interacting with external systems not provided by the cloud provider. The weblog application uses a custom cloud-centric framework created by the author called *Marajo*, instead of contributing extensions to existing Python frameworks (e.g., pylons). Apart from Marajo's tight integration with AWS, it may be difficult for it to be widely used by software architects since it does not offer the rich ecosystem and large public appeal which other Python-based web frameworks currently offer.

Sawant and Shah discussed patterns for handling "Big Data" on the cloud (Sawant & Shah 2013). These include patterns for big data ingestion, storage, access, discovery and visualisation. For example, it describes how the "Federation Pattern" can be used to pull together data from multiple sources and then process the data. Doddavula et al. (Mahmood 2013) present several cloud computing solution patterns for handling application and platform solutions. For instance, it discusses cloud deployment patterns for (i) handling applications with highly variable workloads in public clouds; and (ii) handling workload spikes with cloud burst.

Erl et al. (Erl & Naserpour 2014) present a catalogue of over 100 cloud design patterns for developing, maintaining and evolving cloud-hosted applications. The cloud patterns, which are divided into eight groups cover several aspects of cloud computing, such as scaling and elasticity, reliability and resilience, data management, and network security and management. For example, patterns such as shared resources, workload distribution and dynamic scalability (which are

listed under the "sharing, scaling and elasticity" category) are used for workload management and overall optimisation of the cloud environment. The major strength of Erl et al.'s catalogue of cloud patterns is in its extensive coverage of techniques for handling the security challenges of cloud-hosted applications. It describes various strategies covering areas such as hypervisor attack vectors, threat mitigation and mobile device management.

The authors in (Jamshidi, Pahl, Chinenyeze & Liu 2015) describe a catalogue of fine-grained service-based cloud architecture migration patterns that target multi-cloud settings which are specified with architectural notations. The key patterns reflect the different construction principles for cloud architecture: re-deployment, cloudification, relocation, refactoring, rebinding, replacement and modernization. These patterns are presented as migration strategies, decision making and best practices for cloud migration, and so are different from cloud patterns shown in (Wilkes 2011, Mendonca 2014, Fehling et al. 2014) and so may not be applied at runtime during the design and deployment. Other documentation of cloud deployment patterns can be found in (Strauch, Breitenbuecher, Kopp, Leymann & Unger 2012, Varia 2014b, Musser 2012, Arista.com 2014, Brandle, Grose, Young Hong, Imholz, Kaggali & Mantegazza 2014, Varia 2014a).

2.5 Implementing Multitenancy Isolation in a Cloud Computing Environment

Multitenancy is an essential cloud computing property where a single instance of a cloud offering is used to serve multiple tenants and/or components (Pearson 2013, Krebs, Momm & Kounev 2014). 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*.

2.5.1 Multitenancy Isolation

Definition 2.4: **Multitenancy isolation**. The term "Multitenancy Isolation" as used in this thesis is an approach for ensuring that the required performance, stored data volume and access privileges of one tenant does not affect other tenants accessing the component or the functionality of a shared application component. Multitenancy isolation can be captured in three main cloud multitenancy patterns (Fehling et al. 2014):

(1) Shared component: Tenants share the same resource instance, and may not be aware that it is being used by other tenants.
(2) Tenant-isolated component: Tenants share the same resource instance, but their isolation is guaranteed. This pattern allows the tenant-specific configuration of the provided functionality or resource.
(3) Dedicated component: Tenants do not share resource instance. That is, each tenant is associated with one instance (or a certain number of instances) of the resource.

Definition 2.5: **Application Component.** An Application Component is defined as an encapsulation of a 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), processing component (e.g., load balancer), or a user interface component (e.g., AJAX).

2.5.2 Related Work on Approaches for Implementing Multitenancy

There are several approaches for implementing multitenancy that have been widely discussed in the literature. Multitenancy can be implemented at different layers of the cloud stack: application layer, the middleware layer, and data layer. For example, in (Mehta 2017c, Mehta 2017a, Mehta 2017b), 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 (Strauch, Andrikopoulos, Leymann & Muhler 2012) 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 from the implementation into the ESB.

Implementing multitenancy for a cloud-hosted service particularly at the application level is very challenging and could involve rewriting the application. In agreeing with this position, Mehta states that achieving multi-tenancy can be downright hard and expensive if it is not implemented during the earliest stages of a development project (i.e., the architecture phase) (Mehta 2017b).

In (Khan, Mirza et al. 2012), 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 (Momm & Krebs 2011) presents a qualitative discussion of different approaches for implementing multi-tenant SaaS offerings, while the author in (Aiken 2017) 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.

2.5.3 Related Work on Degrees of Multitenancy Isolation

Several work of literature acknowledge that there could be varying degrees of isolation between tenants. In (Chong & Carraro 2006), 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. Three multitenancy patterns have been identified which express the degree of isolation between tenants accessing a shared component of an application (Fehling et al. 2014). These patterns are referred to as *shared component*, *tenant-isolated component* and *dedicated component*. The shared component represents the lowest degree of isolation between tenants while the dedicated component represents the highest. The degree of isolation between tenants accessing a tenant-isolated component would be in the middle.

The authors in (Wang, Guo, Gao, Sun, Zhang & An 2008) 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 (Fehling et al. 2014). The author (Vengurlekar 2012) 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 (Mietzner, Unger, Titze & Leymann 2009) describe how the services (or com-

ponents) 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. (Guo, Sun, Huang, Wang & Gao 2007). 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 (Bauer & Adams 2012).

In the work of Walraven et al., the authors implemented a middleware framework for enforcing performance isolation (Walraven, Monheim, Truyen & Joosen 2012). 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 (Krebs, Wert & Kounev 2013). Other work related to multitenancy isolation can be found in (Chong & Carraro 2006) (IEEE 2017).

At the very basic degree of multitenancy, tenants share application components as much as possible which translates to increased utilisation of underlying resources. However, while some application components may benefit from a low degree of isolation between tenants, other components may need a higher degree of isolation because the component may either be too critical or needs to be configured very specifically for individual tenants because of their unique deployment requirements. Again, tenant-specific requirements, such as laws and corporate regulations, may even further increase the degree of isolation required between tenants. The challenge, therefore, for a cloud deployment architect would be how to resolve the trade-offs between the required performance, systems resources and access privileges at different levels of an application when opting for one (or combinations) of the multitenancy patterns for cloud deployment of software tools.

The focus of our work is evaluating the degree of isolation between tenants enabled by multitenancy patterns. Specifically, we are interested in providing empirical evidence of the effect of

performance and resource utilisation on other tenants due to the high workload created by one of the tenants. In our work, we implemented multitenancy as a component integrated into an open source Global Software Development (GSD) tool. Also, our evaluation is done in a real cloud environment. The application we used for our evaluation is within the domain of software engineering, to emulate a typical software development process. Furthermore, we deployed our GSD tool to the cloud using cloud multitenancy patterns

2.6 Optimizing Components Deployment for Guaranteeing Multitenancy Isolation

This section first describes the conflicting trade-offs that exist when implementing multitenancy isolation. This is followed by a discussion of related work in optimal deployment and allocation of cloud resources.

2.6.1 Conflicting Trade-offs in Multitenancy Isolation

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

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

2.6.2 Related Work on Optimal Deployment and Allocation of Cloud Resources

Research work on optimal deployment and allocation of resources in a cloud environment are quite significant. However, there is no work undertaken on providing an optimal solution for deploying components of a cloud-hosted service in a way that guarantees the required degree of multitenancy isolation.

Fehling et al. argued that the deployment of component instances in a cloud environment can be optimized by (i) sharing the instances between tenants, especially if the components provide the same functionality to some of all the tenants, and (ii) sharing the cloud resources that the instances are hosted on so that the underlying resources are efficiently utilized, thus avoiding deployment redundancy. Regarding provisioning of application component instances, this can be achieved by limiting the number of application components deployed exclusively for one tenant (Fehling et al. 2014).

In (Yusoh & Tang 2012), the authors used an evolutionary algorithm to minimise resource consumption for SaaS providers and improve execution time. The authors in (Shaikh & Patil 2014) and (Westermann & Momm 2010) used a multitenant SaaS model to minimise the cost of cloud infrastructure. Heuristics were not used in this work. The authors in (Candeia, Santos & Lopes 2015) 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 (Abbott & Fisher 2009), the authors described how the optimal configuration of a virtual server could be determined, for example, the amount of memory to host an application through a set of tests. Fehling et al. (Leymann, Fehling, Mietzner, Nowak & Dustdar 2011), 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 (Aldhalaan & Menascé 2015b), where the authors used a simple heuristic (instead of a metaheuristic) search technique based on hill climbing to minimise the SaaS cloud provider's cost of using VMs from an IaaS with response time SLAs constraints.

2.7 Conclusions from Literature Review

This section summarises our conclusions from the literature review to show the gaps identified and how these gaps are addressed in the thesis. As stated earlier, the architectures or cloud patterns used in deploying services to the cloud are very important to architects since it determines whether or not the cloud service will be able to exhibit the required quality attributes. The literature revealed that existing classifications of cloud patterns do not arrange the individual patterns into a well-organised hierarchy or taxonomy. This is because most of the patterns tend to handle multiple architectural concerns (Wilder 2012). This makes it difficult for an architect to decide whether the implementation of the cloud can be done by modifying the cloud-application itself or the components of the cloud environment where the application is running.

Most cloud patterns in existing classifications and taxonomies were not applied to any specific application domain, such as a set of applications or a cloud-hosted GSD tools in software engineering domain. Some of these taxonomies might be less useful because they ignore some application-dependent properties such as application architecture, resource consumption and supported workload and processes that would influence their deployment to the cloud. For instance, Fehling et al. (Fehling et al. 2014) catalogued a collection of cloud patterns, but these patterns were not applied to a set application in a specific domain. In other cases, the cloud patterns were applied to simple web-based applications (e.g., Weblog application (Moyer 2012)) without considering the different application processes they support. GSD tools may have similar architectural structure but they (i) support various software development processes, and (ii) impose varying workloads on the cloud infrastructure, which would influence the choice of a deployment pattern. For example, Hudson being a compiler/build tool would consume more memory than subversion when exposed to high intensive workload.

This problem is addressed in chapter 4 by developing a taxonomy and a general process for selecting applicable deployment patterns together with the supporting technologies for deploying cloud-hosted services. This taxonomy categorises cloud deployment patterns into the two main components of an architectural deployment structure: the cloud-application (e.g., multitenancy patterns) and cloud environment (e.g., content distribution network). Moreover, the practicality of the taxonomy has been demonstrated by applying it to position a set of Global Software Development tools such as Hudson, Subversion, Bugzilla, JIRA and Versionone.

There are several research work which have established that there are varying degrees of multitenancy isolation. For example, Fehling et al. (Fehling et al. 2014) captured the degree of multitenancy in three cloud patterns: shared component, tenant-isolation component and dedicated component; and also, suggested that the degree of isolation between tenants is the main factor that can be used to distinguish between these patterns. However, the various deployment conditions which offer the required degree of isolation are not known. There is no research work that has empirically evaluated these varying degrees of isolation between tenants for applications in a particular application domain.

To address this challenge, an approach for implementing multitenancy, termed, COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing (COMITRE) has been developed. This approach is then applied to empirically evaluate the degree of isolation between tenants enabled by multitenancy patterns within the context of cloud-hosted GSD tools under different cloud deployment conditions. This study has demonstrated the practicality of the approach and provided empirical evidence of the effect of performance and resource utilisation on other tenants due to the high workload created by one of the tenants.

Case studies in software engineering often focus on a particular phenomenon in context, and it is usually not possible to investigate all aspects of a phenomenon in one case study(Cruzes & Dybå 2010, Cruzes & Dybå 2011). Therefore it is important to adopted strategies for synthesising and providing new interpretative explanations about existing case studies derived from diverse aspects of a phenomenon (Cruzes & Dybå 2010, Cruzes & Dybå 2011). To address this challenge, three case studies were conducted to extend the overall evidence base beyond a single case to empirically evaluate the effect of varying degrees of multitenancy isolation on the performance and resource consumption of tenants under different cloud deployment scenarios. After that, a synthesis of the findings of the three case studies was carried out to provide an explanatory framework and new insights for explaining the (i) commonalities and differences in the case studies, and (ii) the trade-offs to consider when implementing multitenancy isolation.

Research on multitenancy isolation has largely focused on isolation at the data tier (Chong et al. 2017, Vanhove, Vandensteen, Van Seghbroeck, Wauters & De Turck 2014, Schneider & Uhle 2013, Schiller 2015, Kurmus et al. 2011, Zeng 2016). The main aspect of isolation is usually performance isolation (Kurmus et al. 2011, Herbst et al. 2016, Krebs 2015). For example the authors in (Krebs 2015, Krebs et al. 2013) mainly focuses on performance isolation in a multi-

tenant application in the cloud. Several metrics and techniques such as the relative difference of the QoS, increased workload on the QoS of the abiding tenants, and the arithmetic mean for disruptive workloads have been described for quantifying performance isolation in multitenant cloud applications

The varying degrees of multitenancy isolation based on three multitenancy patterns and different aspects of isolation are described in (Fehling et al. 2014). Guo et al (Guo et al. 2007) evaluated different isolation capabilities for authentication, information protection, faults, administration etc. Other work related to multitenancy isolation can be seen in (Krebs et al. 2013) (Krebs & Loesch 2014). None of the related work (for example, the (Guo et al. 2007, Krebs et al. 2013, Krebs & Loesch 2014)) considers implementing multitenancy in a way that guarantees the required degree of isolation between tenants.

This thesis describes the architecture for implementing multitenancy isolation together with supporting algorithms. It also describes how to determine the isolation level of an application component or functionality in almost real-time. Similar to our proposed approach, most cloud providers also implement techniques that can intercept a user request, inspect it, and then decide what level of isolation is required. This is typically what production systems do across the overall application logic, for example, when providing subscriptions with different levels of isolation at different price tiers. However, while carrying out these provisioning and decommissioning operations, most cloud providers do not guarantee the availability and multitenancy isolation of specific components/individual IT resources (for example, disk storage), but only for the offering as a whole (for example, starting new virtual servers). Our algorithm can address this problem by initially tagging each component and after that identify which isolation level is suitable for deploying a component based on the metadata of existing components. This will allow the component and the application to run efficiently and also help in optimising the deployment of components of the cloud-hosted service.

Research work on optimal deployment and allocation of cloud resources on the cloud are quite significant. Most of this research focuses on minimising the cost of using the cloud infrastructure resources (Yusoh & Tang 2012). Previous work does not use metaheuristic to provide optimal solutions for deploying components of a cloud-hosted service in a way that guarantees the required degree of multitenancy isolation. Most of the research concerning optimization of cloud resources do not use heuristic at al, although a few use simple heuristics. For example, the authors

in (Aldhalaan & Menascé 2015a, Aldhalaan & Menascé 2015b) used a heuristic based on hill climbing for minimising the cost of a SaaS cloud providers with response time SLAs constraints. Our work, unlike others, focuses on providing an optimal solution for deploying components of a cloud-hosted application in a way that guarantees the required degree of multitenancy isolation.

This thesis addresses this problem in chapter seven by developing a model-based decision support system as a framework for providing near-optimal solutions for deploying components of a cloud-hosted application that maximises both the required degree of multitenancy isolation and the number of requests allowed to access the components. In addition, four variants of a metaheuristic solution (based on simulated annealing and hill climbing) have been developed to solve the model integrated into the decision support system.

2.8 Chapter Summary

This chapter has reviewed the relevant literature related to our research. Firstly, several studies on taxonomies and classifications of cloud patterns have been reviewed. It was discovered that many of these taxonomies were not benchmarked to existing classifications and not applied to applications in a particular domain. Secondly, related work on implementing varying degrees of multitenancy isolation was discussed. The literature review revealed that approaches for implementing multitenancy have mostly focused on the data tier, and are mostly directed towards performance isolation. Although the literature acknowledges that there are varying degrees of multitenancy isolation, their effect on performance and resource consumption on tenants have not been evaluated empirically on applications in a particular domain. Thirdly, the literature on providing optimal allocation of cloud resources was reviewed. The review has concluded that research in this area focused on minimising the cost of deploying cloud resources and does not use metaheuristic for optimisation. Furthermore, optimisation of cloud resources is not done in a way that guarantees multitenancy isolation.

These issues are addressed in subsequent chapters (i.e., in chapters four, five, six, and seven) by: (i) creating and applying a taxonomy of cloud deployment patterns to GSD tools and supporting processes; (ii) developing an approach for implementing not just multitenancy, but varying degrees of multitenancy isolation and applying the approach to three case studies (followed by a synthesis of the findings of the case studies) that empirically evaluated the varying degrees of

multitenancy isolation in GSD tools and supporting processes; and (iii) developing a model-based decision support system for providing optimal solutions for deploying components of a cloud-hosted service.

Chapter 3

Methodology

3.1 Introduction

In chapter one, it was stated that the main research question addressed in this thesis is how to architect the deployment of cloud-hosted services for guaranteeing multitenancy isolation. This question was further broken down into three sub-questions. Each sub-question addresses a different but related aspect of the research, hence the need for a combination of more than two research methods.

This chapter is organised as follows: Section 3.2 introduces the multimethod research methodology adopted for the research. After that, Section 3.3 to 3.5 discusses each research method that make up the multimethod approach. This discussion covers the selection of Global Software Development tools used for case studies, the experimental setup, procedure and metrics for evaluating the results. Section 3.6 discusses the motivation for adopting the multimethod approach and how each research method fits into the overall research process.

3.2 The Multimethod Research Approach

This research adopts the *multimethod research* method in an interlinked process. Multimethod research includes the use of more than one method of data collection or research in a study or set of related studies (Collier & Elman 2008). The basis of the method is to investigate a phenomenon using a combination of empirical research methods with the intention that the combination of techniques complement each other. Complementing may take several forms, for example, helping

to confirm research findings, one study being used to generate research hypotheses for another, or one study being used to help explain the findings of another. It is argued that such an approach can help limit the effect of threats to experimental validity, which is a particular problem in human-intensive research areas such as the social sciences and software engineering (Wood, Daly, Miller & Roper 1999).

Three main research methods are used in this thesis: the exploratory study, case study and case study synthesis and simulation based on a model. As shown in Figure 3.1, the overall research process is divided into three phases, and captures how the three research methods are linked together. The sections that follow explain the components of each research [1]. This is followed by a discussion on the motivation of using the multimethod research method and then shows how these different research methods fit into the overall research process.

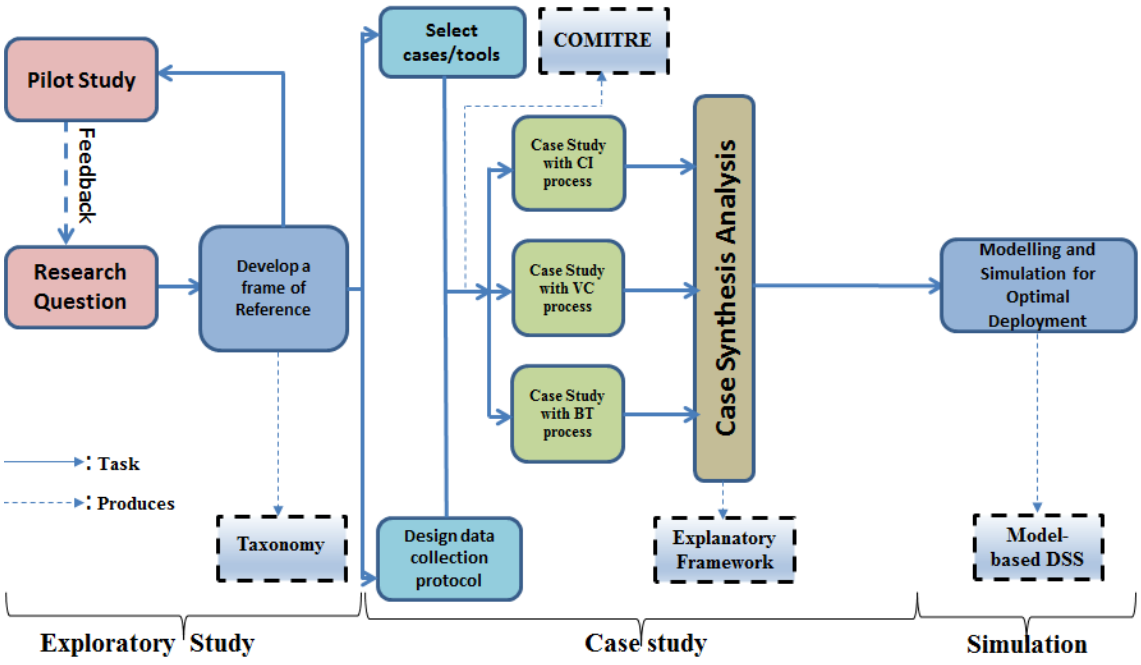

Figure 3.1: Components of the methodology adopted for the study

3.3 Phase 1: Exploratory Study

The first phase of the research applied an exploratory study. An exploratory study is used to find out what is happening, seeking new insights, and generating ideas and hypotheses for new research

[1] The symbols - CI, VC, and BT used in Figure 3.1 refers to continuous integration, version control and bug tracking, respectively.

(Runeson & Host 2009). Exploratory studies usually start with a general idea and use research as a tool to identify issues that could be the focus of future research. The first phase of the research process is composed of two steps: (i) selecting the software processes and tools used in Global Software Development; and (ii) exploring the various types of deployment patterns used to deploy services to the cloud. These steps are summarised below.

3.3.1 Selection of GSD Tools and Processes

An empirical study was conducted to find out: (1) the type of GSD tools used in large-scale distributed enterprise software development projects; and (2) what tasks they utilise the GSD tools for. The study produced a set of five GSD processes and supporting tools: JIRA for issue tracking, VersionOne for agile management, Hudson for continuous integration, Subversion for version control and Bugzilla for bug tracking (Bass 2014, Ochei, Bass & Petrovski 2015a). Details of the study are presented in chapter four.

From this dataset, three GSD processes widely used in Global Software Development were selected: continuous integration, version control and bug tracking. These GSD processes were chosen for three main reasons: (i) these processes are widely used in GSD; (ii) there are open-source tools and/or plugins that are specifically developed to support these processes; and (iii) the open-source tools that support these processes are flexible to customise and extend.

This study used the most-similar technique (that is, purposive, non-random selection procedure) to select the three cases. Random sampling is inappropriate as a selection method (Yin 2014) for case study research methodology. This is because the selection of cases is not governed by sampling logic and representativeness: rather cases are selected for being typical, critical, revelatory, or unique in some respect (Yin 2014).

Applications used in software engineering domain were chosen, and in particular, Global software development, in order to have a proper basis for comparison and evaluation. The software processes selected (i.e., continuous integration, version control and bug tracking) were not general software processes but were unique in terms of the processes they are associated with, the way tenants interact with these processes, the type of components they use to store data and the resources they consume.

It is important to note that the emphasis of this study is not on the GSD tools or plugins but on the GSD processes they support. These tools may be used for other tasks, but they are primar-

ily used for a specific software process. For example, Hudson is used primarily for continuous integration, even though it can be used to trigger other software processes such as issue tracking, software testing, and continuous deployment. Therefore, our intention was to know how these processes and the data they generate impact on multitenancy isolation with regard to performance and resource utilisation.

3.3.2 Exploring Cloud Deployment Patterns

In this step, an exploratory study on cloud deployment patterns was carried out by relying on secondary research technique which entails reviewing available literature (such as textbooks, academic journals and conference papers, white papers, technical reports etc.) on similar or related studies taken and learning from their results. This study produced a taxonomy of deployment patterns together with the supporting technologies for cloud-hosted services. The practicality of the taxonomy was demonstrated by applying it to position a set of GSD tools on the taxonomy. It is important to note that the previous step helped select the GSD processes and supporting tools used to apply against our taxonomy. Furthermore, a general process, CLIP, was created for selecting applicable deployment patterns using the taxonomy. After that, CLIP was then applied to select applicable patterns for solving a motivating cloud deployment problem.

As software tools are increasingly being deployed in the cloud to serve multiple users, there is need not just to implement multitenancy, but to also ensure proper isolation of both the tenant's data (e.g., code files) and processes (e.g., builds) associated with these tools. The next phase of the research focused on multitenancy patterns to evaluate the effect they have on the required performance, and resource consumption of tenants when there are workload changes.

3.4 Phase 2: Case Study and Case Study Synthesis

The second phase of the research was used to conduct several case studies and after that a synthesis of findings from the case studies. These procedures are discussed in the sections that follow.

3.4.1 Case Study

Case study research is an empirical study aimed at investigating contemporary phenomena in their context (Runeson & Host 2009, Yin 2014, Robson & McCartan 2016). Case studies are well suited

for software engineering research since they study contemporary phenomena in its natural context (i.e., real-world open-source GSD tools in our case) (Runeson & Host 2009).

The specific design of the case study is *multiple-case design with multiple embedded units of analysis*. This case study design represents a form of mixed method research which relies on a more holistic data collection strategy for studying the main case but then calls upon more quantitative techniques(in this case, experimentation) to collect data about the embedded unit(s) of analysis (Yin 2014). The experiments within the case study enable us to collect data for evaluating the effect of multitenancy isolation (i.e., based on different multitenancy patterns) on the performance and resource consumption of tenants under realistic cloud deployment conditions of GSD tools.

Figure 3.2 shows a component of the design for the first case study. The context of the case study is Deployment Patterns for GSD processes. Case study two and three can be captured using the same diagram by simply replacing the "'The Case"' with multi-tenancy deployment patterns for the version control system and bug tracking systems, respectively. This study did not rely on

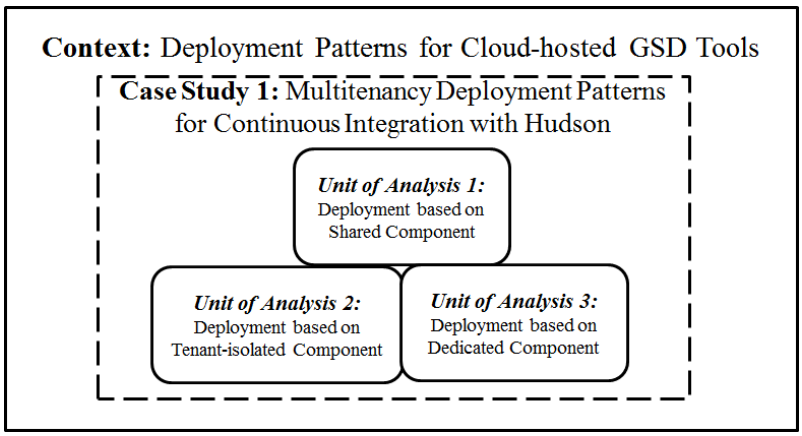

Figure 3.2: Multiple-case (embedded) design adopted for the study

any explicit theory; instead, a frame of reference was developed. Developing a research direction using theories is not well developed in software engineering (Shull & Feldmann 2008). The term "frame of reference" as used in this thesis refers to a set of basic assumptions or standards that govern the way tenants data and processes are isolated when accessing a multitenant application. In our case, the following assumptions are made: (i) that the varying degrees of multitenancy isolation are captured in three main cloud patterns: shared component, tenant-isolated, and dedicated component; (ii) shared component represents a low degree of isolation, while dedicated compo-

nent represents a high degree of isolation; and (iii) each multitenancy pattern has an impact on the required performance, stored data volume and access privileges of other tenants when one of the tenants experiences a high workload.

3.4.2 Evaluation of the Case Study

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

Experimental Design

A set of four tenants (T1, T2, T3, and T4) are configured into three groups to access an application component deployed using three different types of multitenancy patterns (i.e., shared component, tenant-isolated component, and dedicated component). Each pattern is regarded as a group in this experiment. Two different scenarios were created for all the tenants. Treatment was created by configuring T1 so that it will experience a high workload. The details of the scenarios and treatment are explained in Chapter 5. For each group, one of the four tenants (i.e., T1) is configured to experience a demanding deployment condition (e.g., large instant loads) while accessing the application component. The performance metrics (e.g., response times) and systems resource consumption (e.g., CPU) of each tenant are measured before the treatment (pre-test) and after the treatment (post-test) was introduced.

The ***aim of the experiment*** is to evaluate the degrees of isolation enabled by multitenancy patterns for cloud-hosted GSD tools.

The ***hypothesis*** of the experiment is that the performance and system's resource utilisation experienced by tenants accessing an application component deployed using each multitenancy pattern changes significantly from the pre-test to the post-test.

Based on this information, a two-way repeated measures (within-between) ANOVA was adopted as the experimental design. This experimental design is used when there are two independent variables (factors) influencing one dependent variable (Verma 2015). In our case, the first factor is the multitenancy deployment pattern, and the second factor is time. The multitenancy pattern is the between factor because our interest is in looking at the differences between the groups using different multitenancy patterns for deployment. Time is the within factor because our interest is in

measuring each group twice (pre-test and post-test). The data view of our experimental design is composed of a *Group* column that indicates which of the three groups the data belongs to, and two columns of actual data, one for the Pre-test and one for the Post-Test.

Experimental Setup

The experimental setup consists of a private cloud setup using Ubuntu Enterprise Cloud (UEC). UEC is an open-source private cloud software that comes with Eucalyptus (Johnson, Kiran, Murthy, Suseendran & Yogesh 2016). There are five basic components in UEC's architecture which are summarised below:

(i) Cloud Controller(CLC): This is the front end to the entire cloud infrastructure. It provides an interface to monitor the running instances, the availability and usage of resources in the cloud.

(ii) Walrus Storage Controller (WS3): This component provides a persistent simple storage service using REST and SOAP APIs compatible with S3 APIs.

(iii) Cluster Controller: This component communicates between cloud controllers and node controllers. In addition to this, the cloud controller manages one or more node controllers and deploys instances on them.

(iv) Storage Controller (SC): The SC provides persistent block storage (like Elastic Block storage for Amazon Web Services (AWS)) for use by the deployed instances.

(v) Node Controller (NC): The node controller runs on each node of the UEC and controls the life cycle of all instances. It queries the operating system on each node to determine, for example, the physical resources of the node, the number of cores, the size of memory, the available disk space and the state of the VM instances running on each node and then sends these details to the cluster controller.

Figure 3.3 shows a simple UEC setup with 3 physical machines- one client node and two server nodes. For our UEC setup, six physical machines were used - one head node and five sub-nodes. The experimental setup is based on the typical minimal Eucalyptus configuration where all user-facing and back-end controlling components - (Cloud Controller(CLC), Walrus Storage Controller

Table 3.1: Hardware and Network Configuration of the UEC

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

(WS3), Cluster Controller (CC), and Storage Controller (SC)) are grouped on the first machine, and the Node Controller (NC) components are installed on the second physical machine. The guidelines for installing UEC as outlined in (Pantić & Babar 2012) were followed to extend the configuration by installing the NC on all the other sub-nodes to achieve scalability. The head node was also used as the client machine since it does not have to be a dedicated machine. Installing UEC is like installing Ubuntu server; the only difference is the additional configuration screens for the UEC components. The Hardware configuration of the head node and sub-nodes is summarised in Table 3.1.

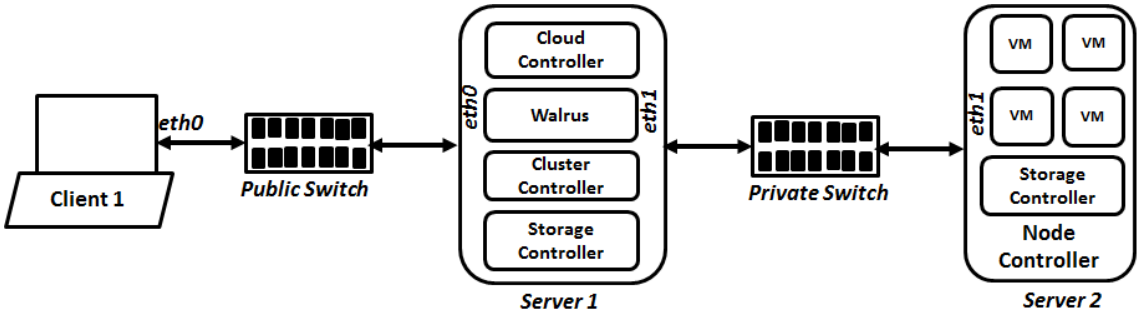

Figure 3.3: Setup of the UEC used for experiments

Experimental Procedure

A summary of the experimental procedure is outlined in Figure 3.5. The procedure outlined in Figure 3.5 captures an abstract summary which runs across all the case studies. However, it does not show how the GSD tool used for each case study was modified, and the specific details of how the processes associated the GSD tools were simulated. Chapter 5 describes the experimental procedure for each case study in more detail. In a nutshell, the GSD tool used for each case study is modified to support multitenancy isolation. This involved developing a plugin and integrating it with the GSD tool so that it can be accessed by different tenants. The GSD tool is then bundled as a VM image and uploaded to a private cloud with a typical minimal UEC configuration.

To evaluate the degree of multitenancy isolation between tenants, four tenants (referred to as tenant 1, 2, 3, and 4) were configured based on access to the functionality/component of the GSD tool that is to be served to multiple tenants. Access to this functionality is associated with a tenant identifier that is attached to every request. Based on this identifier, a tenant-specific configuration is retrieved from the tenant configuration file and used to adjust the behaviour of the GSD tool's functionality that is being accessed.

A remote client machine was used to access the GSD tool running on the instance via its IP address. Apache JMeter is used as a load balancer as well as a load generator to generate workload (i.e., requests) to the instance and monitor responses (Erinle 2013). The following system metrics were collected and analysed:

(i) CPU Usage: The %user values (i.e., the percentage of CPU time spent) reported by SAR were used to compute the CPU usage.

(ii) System load: The one-minute system load average reported by SAR was used.

(iii) Memory usage: The kbmemused (i.e., the amount of used memory in kilobytes) recorded by SAR was used.

(iv) Disk I/O: The disks input/output volume reported by SAR was recorded.

(v) Response time: The 90% latency reported by JMeter.

(vi) Throughput: The average throughput reported by JMeter was used.

(vii) Error %: The percentage of the total number of requests whose response time is unacceptably slow and above which the request is considered a failure. This is calculated statistically as the upper bound of the 95% confidence interval of the average response time of all request.

To measure the effect of tenant isolation, tenant 1 was configured in JMeter to simulate a *large instant load* by: (i) increasing the number of requests using the thread count and loop count (ii) increasing the size of the requests by attaching a large file to it; (iii) increasing the speed at which the requests are sent by reducing the ramp-up period by one-tenth, so that all the requests are sent ten times faster; and (iv) creating a heavy load burst by adding the Synchronous Timer to the Samplers in order to add delays between requests, such that a certain number of requests are fired at the same time. This treatment type is like unpredictable (i.e., sudden increase) workload (Fehling et al. 2014) and aggressive load (Walraven et al. 2012).

Each tenant request is treated as a transaction composed of all types of request simulated. For example, when using Hudson for case study one, the HTTP request triggers a build process while JDBC request logs data into a database which represents an application component that is being shared by the different tenants. The transaction controller is used to group all the samplers in order to get the total metrics (e.g., response time) for carrying out the two requests. Figure 3.4 shows the experimental setup used to configure the test plan for the different tenants in Apache JMeter. The setup values for the experiment are shown in Table 3.2. It is important to note that

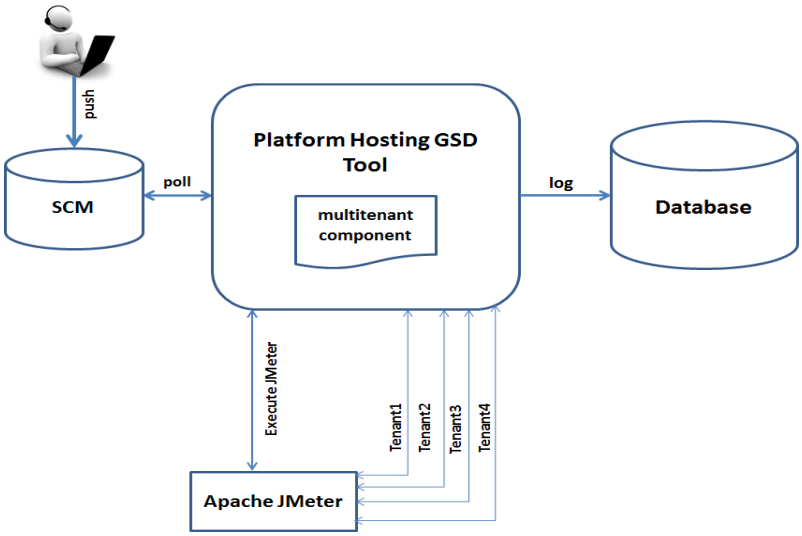

Figure 3.4: Experimental Setup

since different processes are being simulated using different GSD tools, the setup values (e.g.,

Table 3.2: Setup parameters used in the experiments

Parameters	Values for Case Study 1	Values for Case Study 2	Values for Case Study 3
No of threads	10 for tenant 1 (i.e., the tenant experiencing high load); 5 for all other tenants	2 for all tenants	5 for all tenants
Thread Loop count	2	5 for all tenants	2 for all tenants
Loop controller count	10 for HTTP requests of tenant 1, and 5 for all other tenants; 200 for JDBC requests of tenant 1, and 100 for all other tenants	4 for tenant 1, and 2 for all other tenants for each type of request (HTTP, BeanShell and FTP upload and FTP download request samplers)	20 for tenant 1 and 10 for all other tenants for the HTTP and BeanShell samplers
Ramp-up period	6 seconds for tenant 1, 60 seconds for all other tenants	6 seconds for tenant 1, 60 seconds for all other tenants	6 seconds for tenant 1, 60 seconds for all other tenants
Size of request	1MB for tenant 1, and 200KB for other tenants.	1MB for tenant 1, and 200KB for other tenants.	1MB for tenant 1, and 200KB for other tenants.

thread count, loop count and loop controller count) will vary slightly. To have a balanced basis for comparison, the workload was carefully varied to cope with the private cloud used in such a way that: (i) the number of requests sent by tenant 1 (i.e., the tenant that experiences a very high workload or aggressive load) is two times more, five times heavier, and ten times faster than the other tenants; and (ii) all other tenants regardless of the type of request being simulated send the same number of requests.

Ten iterations for each run were performed and the values reported by JMeter used as a measure for response times, throughput and error%. For system activity, the average CPU, memory, disk I/O and system load usage at a one-second interval was recorded using the SAR tool. As the experiments were concerned with determining which tenant changed significantly within each group/pattern from pre-test to post-test, the mean/average was chosen (instead of the median or mode) because it is more sensitive to ”outliers”data values at the extremes of a group. That is, the mean value always reflects the contributions of each of the data values in the groups (i.e., the three multitenancy patterns - shared component, tenant-isolated component, and dedicated component).

Approach for Statistical Analysis of the Case Study

Based on the information about the experimental design, the two-way Repeated Measures (within-between) ANOVA was adopted for the statistical analysis. There are two main advantages in adopting the two-way repeated measures (within-between) ANOVA. The first reason is that of cost advantage because it does not require many subjects since each subject would be measured twice. Four subjects (i.e., four tenants) were used in our experiments. The second reason is that this sta-

1. Prepare the Private Cloud for the Test Run
   (a) Create an Ubuntu Virtual Machine Image
   (b) Install the modified GSD tool on the image
   (c) Upload the Image to UEC
   (d) Launch the instance and SSH to the instance
2. Execute the Test Run
   (a) Start the GSD tool and view it on a browser
   (b) Start JMeter load test on the GSD tool
   (c) Start instance monitoring with SAR tool
   (d) Stop test run after all responses received
3. Collect Results
   (a) Export JMeter and SAR result to text file
   (b) Clear previous JMeter and SAR results
   (c) Reboot instances for next test run
   (d) Repeat step 2 for more runs

Figure 3.5: Experimental Procedure

tistical design eliminates the difficulty of trying to match subjects perfectly between the different conditions in all respects. By allowing the same subjects in all conditions, there is a significant reduction in the variation of scores between groups which is usually due to non-experimental factors such as random differences between the different subjects (Field 2013). Therefore, since all the tenants are used in all the conditions, the only difference between a tenant's effect for the different conditions would be due to our experimental manipulations (i.e., exposing one of the tenants to a very high load) (Howitt & Cramer 2011).

The tool used for statistical analysis is SPSS v21. The two-way (within-between) ANOVA was performed first to determine if the groups had significantly different changes from Pre-test to Post-test. After that, *planned comparisons* were carried out involving the following:
(i) a one-way ANOVA followed by Scheffe post hoc tests to determine which groups showed statistically significant changes relative to the other groups. The Dependent variable (simply called "Change"), used in the one-way ANOVA test was determined by subtracting the Pre-test from Post-test values.

(ii) a paired sample test to determine if the subjects within any particular group changed significantly from pre-test to post-test measured at a 95% confidence interval. This would give an indication as to whether or not the workload created by one of the tenants has affected the performance and resource utilisation of other tenants. The "Select Cases" feature in SPSS was used to select the three tenants (i.e., T2, T3, T4 that did not experience large instant loads) for each pattern and for each deployment scenario giving a total of 6 cases for each metrics which was measured (Sheridan & Ong 2011).

After the first two steps outlined above, the plots of estimated marginal means were analysed (see Figure 5.8 to Figure 5.27) in combination with ANOVA (plus post hoc test) and paired sample test results from SPSS output. These plots are referred to as the "Estimated Marginal Means of Change (EMMC)". Note that the word "Change" refers to the transformed variable used as the dependent variable in the one-way ANOVA. The plot of EMMC is simply a plot of the mean value for each combination of factor level.

The quasi-independent variable is nominally scaled in SPSS, and the interpolation line was changed to a bar chart to give a meaningful interpretation of the result (Sheridan & Ong 2011). Therefore, each bar chart shows how each pattern performed on the pre-test and the post-test with each bar representing a group (i.e., 1, 2, and 3 represents the shared component, tenant-isolated component and dedicated component, respectively). The bar chart also shows which groups improved significantly relative to the other groups, and the magnitude of that improvement. Therefore, in a situation where the paired sample test is the same for all groups, the plots of EMMC can be used to find out the magnitude of change for each group.

The effect of tenant 1 that experiences high load (i.e., T1) on the other three tenants (i.e., T2, T3, T4) is summarised in a tabular form for each case study. These tables can be seen in chapter five. The key used in constructing the table is as follows: YES - represents a significant change between the metrics measured from pre-test to post -test. NO - represents some level of change which cannot be regarded as significant; no significant influence on the tenants. The symbol "-" implies that the standard error of the difference is zero and hence no correlation and t-test statistics can be produced. This means that the difference between the pre-test and post-test values are nearly constant with no chance of variability.

3.4.3 Synthesizing the findings of the Case Studies

Research synthesis is used to summarise, integrate, combine and compare the findings of different studies on a specific topic or research question (Cruzes & Dybå 2010, Cruzes & Dybå 2011). Case study synthesis entails organising the relevant evidence extracted from the sources included in the case studies and then finding a way to organise the evidence. In our research, the case study synthesis involved organising and summarising key aspects of the studies using tables and charts and figures. There are several methods of conducting case study synthesis, but the most widely used ones are cross-case analysis, thematic synthesis and narrative synthesis (Cruzes, Dybå, Runeson & Höst 2015). The method chosen usually depends on the type and scope of evidence required and the preference of the researcher (Pope, Mays & Popay 2007).

This thesis adopted a hybrid approach which entails using cross-analysis complemented with narrative synthesis to synthesise the findings of the three primary case studies. Cross-case analysis was selected because it involves a highly systematic process and allows us to include diverse types of evidence. Narrative synthesis was chosen because it is very flexible and can cope with a large evidence base, made up of diverse evidence types (Cruzes et al. 2015).

The case study synthesis is based on three primary case studies which empirically evaluated the degree of isolation between tenants enabled by multitenancy patterns for Cloud-hosted GSD tools and processes under different cloud deployment conditions. Case study one involves continuous integration with Hudson, case study two involves Version Control with FileSystem SCM Plugin, and case study three involves bug tracking with Bugzilla. These three case studies were conducted and published separately ((Ochei, Bass & Petrovski 2015c, Ochei, Petrovski & Bass 2015, Ochei, Bass & Petrovski 2016)).

3.4.4 Drawing Conclusions and Discussing the Implications of the Study

The last step in this case study method involves drawing conclusions and providing an explanatory framework and new insights into how the required degree of multitenancy isolation affects the performance and resource consumption of tenants. In developing an explanatory framework, the different degrees of isolation were mapped to different software processes used in the case study to the required performance and resources consumption of tenants interacting with the processes. This step also discusses the challenges, recommendations and trade-offs to be considered in order

to achieve the required degree of isolation.

3.5 Phase 3: Modelling and Simulation

Phase three of the research was used to conduct modelling and simulation. This entails developing a **model-based** decision support (DSS) system for providing near-optimal solutions for deploying components of a cloud-hosted service in a way that guarantees multitenancy isolation when the workload of one of the tenants/components experiences a very high workload. Our DSS is a combination of an open multiclass queueing network (QN) (Menasce, Almeida & Lawrence 2004) and an optimisation model (based on a multichoice multidimensional knapsack problem (MMKP) (Martello & Toth 1987)). This phase also involves developing different variants of a metaheuristic solution for solving the optimisation model integrated into the DSS. It is important to note that the term simulation as used in this thesis means that the experiments were conducted based on the model that has been developed. This phase of the research process is covered in chapter seven of this thesis.

3.5.1 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 were 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 (Parra-Hernandez & Dimopoulos 2005, Cherfi & Hifi 2010). The justification for mapping our problem to an MMKP is explained in Section 7.2.3. The format of the MMKP instance is shown in Figure 3.6. The description of the MMKP instance format is summarised below:

(i) the first row contains three values separated by whitespace - 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 each resource support a 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 follows the number of items for group two. The format for the remaining groups follows the same pattern. The next section explains how these 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 above format of the MMKP instance can be used to explain the workload file as follows:
(i) the first, second, and third row are the same as in the MMKP instance. (ii) the only difference is in the composition of the properties that 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 CPU, RAM, Disk space and Bandwidth which 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 instance

```
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
```

Figure 3.6: Format of the MMKP Instance

were generated and tailored on the instances widely cited in literature: (i) OR benchmark Library (Beasley 1990) and other standard MMKP benchmarks (CERMSEM 2017, Hifi, Michrafy & Sbihi 2004, Khan, Li, Manning & Akbar 2002), and (ii) the new irregular benchmarks used by Shojaei et al. (Eckart & Marco n.d.). These benchmarks are usually used for single objective problems. This benchmark format was modified and extended to conform to a multiobjective case by associating each component with two different profit values: isolation values and the average number of requests (Zitzler & Thiele 1999). The format of the MMKP instance used on running the simulation experiments is shown in Figure 3.6.

(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}. After that, a uniform generating function was applied to draw values uniformly and randomly within this interval. The uniform generating function is given as:

$$p_{ij} = \mathcal{U}(v_i^{min}, v_i^{max}) \tag{3.1}$$

(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 request 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 the capacities of a component's resources (i.e., knapsack capacities for CPU, ram, disk and bandwidth) are generated by setting it to half of the maximum possible resource consumption.

$$c_k = \frac{1}{2} \times m \times R \tag{3.2}$$

The same principle has been used to generate instances available at OR Benchmark Library, and also for instances used in (Parra-Hernandez & Dimopoulos 2005, Cherfi & Hifi 2010).

(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 (CPU, ram, disk, bandwidth) required for a component to be deployed to the cloud. The notation for each instance is C(n, r, m), where n, r, and m stands for the number of groups, the number of components in each group, and the number of resources, respectively.

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

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

could be normalised (or transformed) to represent different resources units of the components.

As an illustration, one of Amazon's EC2 instance types, named *"'compute optimized (c4.xlarge model)"'*, has the following specification: 4 vCPU, 8 GiB of memory, EBS-optimized only storage (which is similar to an IOPS provisioned on an Amazon Elastic Block store volume (EBS)) and 750 Mbps of dedicated EBS bandwidth (Amazon 2016). An Amazon EBS can be created with Provisioned IOPS SSD(io1) volumes up to 16 TiB in size. So assuming the weights of a component on a generated MMKP instance is given as [4, 8, 8, 8], this specification could easily be transformed to the actual specification of the above named Amazon EC2 instance using this normalisation format: [CPU, RAM, DISK/2, BANDWIDTH/100]. This means that this particular component is supported with 4 virtual CPUs, 8GB of memory, 8 TB of disk space and 8 Mbps of bandwidth. Another approach suggested by Han et al. (Han, Leblet & Simon 2010), 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.

3.5.2 Evaluation Metric and Analysis for Simulation

The model-based decision support system is novel in the sense that it combines a Queuing Network (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 decision support system. Because of this, the solutions obtained from our approach were first compared with the optimal solutions obtained from an exhaustive search of a small problem instance. Thereafter, the obtained solutions were also compared with the 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 absolute difference of the target/reference solution. This is given as:

$$\frac{|f(s) - f(s*)|}{f(s*)} \tag{3.3}$$

where s is the obtained solution and s* is the reference solution obtained from the exhaustive search (Talbi 2009).

(2) Robustness: The robustness of the solutions was measured in terms of how sensitive the solutions are, against small deviations in the input data or other parameters; the lower the variability, the better the robustness (Talbi 2009). Standard deviation of a set of optimal solutions 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:

$$ExecTime = GreedyTime + (FEvalTime * NoFEval) \tag{3.4}$$

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.

Other important metrics computed in addition to the above metric are 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 successful runs over the total number of runs or trials. The percent success (i.e., success %) is the percentage number of runs that reached the target solution over 20 runs/trials.

$$\frac{\text{number of successful runs}}{\text{total number of runs}} \tag{3.5}$$

The performance rate of our approach when compared to the optimal solution was measured in terms of the number of successful runs that the target solution has attained over the number of runs as a function of the number of optimal function evaluations (Talbi 2009). This is given as:

$$\frac{\text{number of successful runs}}{\text{number of function evaluations} \times \text{total number of runs}} \tag{3.6}$$

Furthermore, the 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 (Banati & Bajaj 2013, Hoos & Stützle 2004). 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 unit, such as the number of evaluations or generations (Barrero, Muñoz, Camacho & R-Moreno 2015, Hoos & Stutzle 1998).

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 decision support system) when combined with the different variants of the metaheuristic solution. The two-way ANOVA was adopted to determine if there is an interaction between the two independent variables (i.e., type of instance size and variants of metaheuristic) and 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.

3.5.3 Applicability of the Experiments and Frameworks in other Cloud Environments

In this thesis, it is important to note that although the experimental procedures and the resulting framework (i.e., taxonomy, COMITRE and model-based decision support system) presented in this thesis are based on the assumption that this research is carried out in a private cloud, the frameworks are also applicable in other cloud environments such as public cloud and hybrid clouds. The two possible areas of application are (i) migrating existing service/application and workloads to a public cloud, (ii) creating a disaster recovery repository for your VM images.

(i) A service/application installed on a Virtual Image (e.g., Ubuntu VM) can be uploaded from a company's private cloud infrastructure, an on-premise virtualization or LAN infrastructure to a public cloud such as Amazon Web Services (AWS) via Amazon EC2 instances. This means that

it is possible to utilise existing investments in VMs which have been built to meet the users IT configuration, legal and security requirements. For example, a modified GSD tool can be installed on a VM image to a produce a VM-based version of the application. Thereafter, the VM-based version of the GSD tool (together with the workload needed to support it) can be migrated to the Amazon EC2 using the AWS import/export feature. This is important to preserve the GSD tool and its settings that have been configured on existing VMs, while at the same time taking advantage of running the GSD tool and its supporting workload in Amazon EC2.

(ii) Our experimental procedures and novel approaches can also be applied in a hybrid scenario. An example is in the use of a hybrid backup deployment pattern to create a backup and disaster recovery repository for the VM images installed with GSD tools. That is, the VM images can be imported from a static on-premise infrastructure to Amazon EC2 for backup and disaster recovery contingencies. The advantage of this is that in the case of an eventuality, a user can quickly launch the instances residing on AWS to preserve business continuity while at the same time exporting them to rebuild the on-premise infrastructure.

3.6 Multimethod Research: combining exploratory study, case study, case study synthesis, and simulation

As stated in the chapter introduction, this study used the multimethod research methodology. According to Bazeley, multimethod research is when different approaches or methods are used in parallel or sequence but are not integrated until inferences are being made (Bazeley 2006, Morse 2003). In our case, it means that the exploratory study, case study and the simulation were conducted according to the established research procedures of each method.

It should be noted that multimethod research is different from mixed method research. In mixed method research, quantitative and qualitative data is collected or analysed in a single study. Data may be collected concurrently or sequentially, and only the data is integrated at one or more stages in the process of the research (Teddlie & Tashakkori 2003).

3.6.1 Motivation

The main motivation for adopting the multimethod research was to achieve both realism and precision. The rationale for each research method is summarised below:

(1) Exploratory Study: The purpose of using exploratory study was to gather preliminary information to help define the problem and suggest hypotheses. As rightly pointed out by Runeson et al. (2012), our aim was to seek new insights into cloud deployment patterns, and thus generate ideas and hypotheses for the research (Runeson, Host, Rainer & Regnell 2012).

(2) Case Study: The case study method was chosen to conduct an empirical enquiry to investigate a small number of instances (i.e., the effect of varying degrees of multitenancy isolation in three case studies involving cloud-hosted GDS tools) of a contemporary software engineering phenomenon within its real-life context using real-world GSD tools deployed in a cloud environment (Runeson et al. 2012).

(3) Case Study Synthesis: The case studies synthesis was carried out to synthesise the findings from multiple case studies. This allowed us to derive the commonalities and differences found in the three case studies conducted. This synthesis also provided an explanatory framework and new insights on multitenancy isolation under different cloud deployment scenarios.

(4) Simulation based on a model: Simulation allowed us to achieve precision by validating and experimenting with the model. Also, the simulation allowed us to run experiments on the model by assuming a large-scale project size and different cloud deployment scenario.

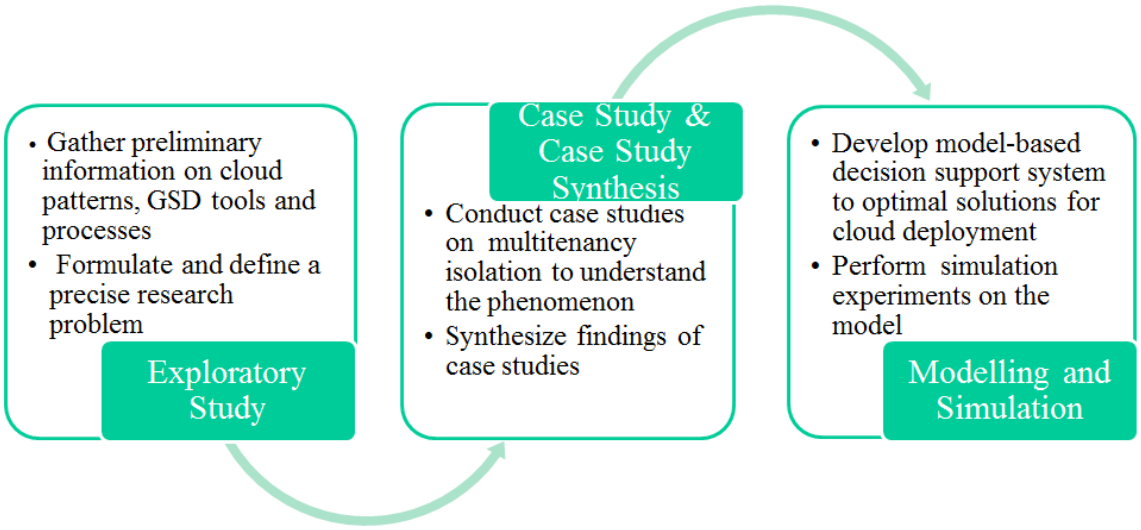

Figure 3.7: Components of the overall research process

3.6.2 How the different methods fit into the research process

The different research methods (i.e., exploratory study, case study and case study synthesis and simulation) are used in an interlinked fashion to form the essential components of the overall research process. It is interlinked because the processes and data collected in one method leads to the processes and data used in another method. Combining the different research methods has allowed us to gain synergies, harmonise weaknesses and assess the relative strengths of each method (Hellstrom & Nilsson 2006).

Figure 3.7 illustrates how the various research methods were combined to form the essential components of our thesis. This research started from an exploratory study to gather preliminary information about architectural patterns and cloud patterns to help define the research problem, suggest hypotheses, and the frame of reference for the research. The key input to the exploratory stage is data derived from secondary sources especially publications such as textbooks, journal and conference papers, software manuals, white papers and technical reports. For example, we searched the cloud patterns described in textbooks and then identified technologies used to support these technologies. This information will be used to develop the taxonomy and also position a set of GSD tools on the taxonomy. The key output of the exploratory study (which also represents the input to the case study and synthesis) are: (i) the dataset of GSD tools and supporting processes, (ii) taxonomy of cloud deployment patterns together with supporting technologies, (iii) CLIP, a general process for using the taxonomy to select suitable deployment patterns. The second stage first used the case study research to conduct several case studies for understanding the context of the phenomena, identifying and measuring relevant characteristics of the studied system. The key output of the case study and synthesis stage (which also represents the input to the modelling and simulation stage) are (i) key resources supporting the components (ii) metrics for measuring tenant isolation in terms of performance (e.g., response times) and resource consumption (CPU usage). Finally, the simulation method was used in the third stage to perform simulation experiments based on the designed model to gain further insights into the behaviour of the system. It allowed us to conduct experiments by assuming problem instances of varying sizes and densities and different cloud deployment scenarios. The output of the modelling and simulation stage are (i) the model-based decision support system which is made up of the QN model and optimisation model plus a set of metaheuristic for solving the model, and (ii) the optimal solution for deploying services or

components to the cloud.

3.7 Chapter Summary

This chapter has explained the overall methodology used in this research. The multimethod method was adopted in this research which entails combining different research methods: exploratory study, case study and case study synthesis and the simulation method. This chapter has explained the components of each method, how each method was carried out, and the metrics used for evaluation. Finally, this chapter discussed the motivation for adopting the multimethod research and how each method fits into the overall picture of the research process in an interlinked fashion.

The different research methods were chosen based on the qualities that each research method can contribute to answering our research question. The exploratory study allowed us to carry out an empirical study to find out the type of software process and the supporting tools used in Global Software development projects and also explore the different cloud deployment patterns for deploying services to the cloud. The case study provided the simulation with an in-depth understanding of the software processes selected for the research and facilitated the development of an explanatory framework for understanding how the required degree of multitenancy isolation affects the performance and resource consumption of tenants. The simulation method allowed us to achieve precision by validating and experimenting with the model in order provide an optimal solution for deploying components of a cloud-hosted service.

Chapter 4

Taxonomy of Deployment Patterns for Cloud-hosted Services

4.1 Introduction

This chapter presents a novel taxonomy together with a general process for guiding architects in selecting appropriate patterns for deploying applications/services to the cloud. The chapter also shows how the taxonomy was applied to position a set of Global Software Development tools on the taxonomy. The findings and discussion reported in this chapter have been published in (Ochei, Bass & Petrovski 2015a). The report in the paper is presented in this chapter and duly referenced.

In chapter 2 it was stated that collections of cloud patterns exist for describing the cloud, and how to deploy and use various cloud offerings (Fehling et al. 2014)(Homer et al. 2014). However, there is little or no research regarding applying these patterns to describe the cloud-specific properties of applications in the software engineering domain (e.g., collaboration tools for GSD, hereafter referred to as GSD tools) and the trade-offs to consider during cloud deployment. This makes it very challenging to know the deployment patterns (together with the technologies) required for deploying GSD tools to the cloud to support specific software development processes (e.g., continuous integration (CI) of code files with Hudson).

Motivated by the problem, this chapter explains how to create a taxonomy of deployment patterns for cloud-hosted applications to help software architects select suitable patterns for deploying GSD tools to the cloud. There are multiple taxonomies developed by researchers to categorise the

cloud computing space into various aspects such as cloud resources provisioned to customers, features of the cloud environment for research and scientific experimentation, and cloud usage scenarios (Milenkoski, Iosup, Kounev, Sachs, Rygielski, Ding, Cirne & Rosenberg 2013). This study considered cloud deployment patterns that could be used to design and deploy applications to the cloud.

The rest of the chapter is organised as follows: Section 4.1 discusses the approach used in developing the taxonomy including taxonomy development, description and validation. Section 4.2 discusses the selection of the GSD tools and processes. Section 4.3 is the application of the taxonomy including positioning a set of GSD tools on the taxonomy and applying a general process for selecting applicable patterns to a motivating cloud deployment problem. Section 4.4 presents the findings of the study while Section 4.5 discusses the findings of the study. Section 4.6 provides recommendations for using the taxonomy. Section 4.7 summarises the chapter.

4.2 Developing a Taxonomy of Cloud Deployment Pattterns

This section presents the methodology for developing and using the taxonomy, a description of the taxonomy and applying them against the taxonomy.

4.2.1 Procedure for Developing the Taxonomy

The taxonomy was developed using a modified form of the approach also used by Lilien (Lilien 2007) in his work for building a taxonomy of specialised ad hoc networks and systems for a given target application class. The approach is summarised in the following steps:

Step 1: Select the target class of Software Tool- The target class is based on the ISO/IEC 12207 taxonomy for the software life cycle process. The following class of tools are excluded: (i) tools not deployed in a cloud environment (even if they are deployed on a dedicated server to perform the same function); and (ii) general collaboration tools and development environments (e.g., MS Word, Eclipse).

Step 2: Determine the requirements for the Taxonomy- The first requirement is that the taxonomy should incorporate features that restrict it to GSD tools and Cloud Computing. In this case,

the taxonomy adopted the ISO/IEC 12207 framework (Portillo-Rodriguez et al. 2010) and NIST cloud computing definition (Mell & Grance 2011). Secondly, it should capture the components of an (architectural) deployment structure (Bass et al. 2013) - (i) software elements (i.e., GSD tool to be deployed) and (ii) external environment (i.e., cloud environment). Therefore, our proposed taxonomy is a combination of two taxonomies - Taxonomy A, which relates to the components of the cloud environment (Mell & Grance 2011), and Taxonomy B, which relates to the components of the cloud application architecture (Fehling et al. 2014).

Step 3: Determine and prioritise the set of all acceptable categories and sub-categories of the Taxonomy- The categories of the taxonomy are prioritised to reflect the structure of a cloud stack from physical infrastructure to the software process of the deployed GSD tool. The categories and sub-categories of the two taxonomies are described as follows:

(1) Application Process: the sub-categories (i.e., project processes, implementation processes and support processes) represent patterns for handling the workload imposed on the cloud infrastructure by the ISO/IEC 12207 software processes supported by GSD tools (Portillo-Rodriguez et al. 2010). For example, the unpredictable workload pattern described by (Fehling et al. 2014) can be used to handle the random and sudden increase in the workload of an application or consumption rate of the IT resources.

(2) Core cloud properties: the sub-categories (i.e., rapid elasticity, resource pooling and measured service) contain patterns used to mitigate the core cloud computing properties of the GSD tools (Fehling et al. 2014).

(3) Service Model: the sub-categories reflect cloud service models- Software as a Service (SaaS) provides software that is hosted centrally and licensed on a subscription basis; Platform as a Service (PaaS) provides a platform to allow customers to develop, run, and manage hosted services; Infrastructure as a Service (IaaS) provides virtualized computing resources over the Internet (Mell & Grance 2011).

(4) Deployment Model: the sub-categories reflect cloud deployment models - private (i.e., a cloud dedicated to a single organization), community (i.e., a specific community with common concerns such as security and jurisdiction on share cloud resources), public (i.e., cloud services delivered to multiple organizations) and hybrid (i.e., combines two or more types of cloud) (Mell & Grance 2011).

(5) Application Architecture: the sub-categories represent the architectural components that support a cloud-application such as application components (e.g., presentation, processing, and data access), multitenancy, and integration. For example, multitenancy patterns are used to deploy a multitenant application to the cloud in such a way that guarantees varying degrees of isolation of the users. The three patterns that reflect these degrees of isolation are the shared component, tenant-isolated component and dedicated component (Fehling et al. 2014).

(6) Cloud Offerings: the sub-categories reflect the major infrastructure cloud offerings that can be accessed- cloud environment, processing, storage and communication offering (Fehling et al. 2014). For example, patterns that fall under "communication patterns" are probably the best documented in this group. Examples include Priority Queue (Homer et al. 2014), Queue-centric workflow, and message-oriented middleware which are used to ensure the reliability of messages exchanged between users.

(7) Cloud Management: contains patterns used to manage both the components and processes or runtime challenges of GSD tools. The two sub-categories are - management components, which are used for managing hardware components (e.g., servers) and management processes, which are used for managing processes (e.g., database transactions) (Fehling et al. 2014). The node failure pattern described by Wilder (Wilder 2012) can be used to handle sudden hardware failures. The "Health Endpoint Monitoring" pattern (Homer et al. 2014) and the "resiliency management" pattern can be used to handle runtime failures or unexpected software failures.

(8) Composite Cloud: contains compound patterns (i.e., patterns that can be formed by combining other patterns or can be decomposed into separate components). The sub-categories are decomposition style and hybrid cloud application (Fehling et al. 2014). The patterns under the decomposition style describe how the software and hardware elements of the cloud environment are composed (or can be decomposed) into separate components. A well-known example is the two-tier (or client/server) pattern, in which each component or process on the cloud environment is either a client or a server. Another example is the multisite deployment pattern (Wilder 2012), where users form clusters around multiple data centres or are in globally distributed sites. Hybrid cloud application patterns are integrations of other patterns and environments. For example, the "hybrid development environment" pattern can be used to integrate various clouds patterns to handle different stages of software development- compilation, testing and production etc.

Step 4: Determine the space of the Taxonomy- The selected categories and their associated sub-categories define the space of the taxonomy. The taxonomy (Table 4.1) is composed of 24 sub-categories, which were systematically integrated and structured into eight high-level categories. The information that the taxonomy conveys has been arranged into four columns: deployment components, main categories, sub-categories and related patterns.

4.2.2 Description of the Taxonomy

Table 4.1 shows the taxonomy captured in one piece. The following section describes the key sections of the taxonomy.

Deployment Components of the Taxonomy: There are two sections of the taxonomy: the upper-half represents Taxonomy A, which is based on NIST Cloud Computing Definition, while the lower-half represents Taxonomy B, which is based on the components of a typical cloud application architecture. The taxonomy has twenty-four sub-categories, which are structured into eight high-level categories: four categories each for Taxonomy, A and B.

Hybrid Deployment Requirements: The thick lines (Table 4.1) show the space occupied by patterns used for hybrid deployment scenarios. There are two groups of hybrid-related patterns: one related to the cloud environment and the other related to the cloud-hosted application. For example, the hybrid cloud pattern (i.e., under "hybrid clouds" sub-category of Taxonomy A) is used to integrate different clouds into a homogenous environment while the hybrid data pattern (i.e., under "hybrid cloud applications" sub-category of Taxonomy B) is used to distribute the functionality of a data handling component among different clouds.

Examples of Related Patterns: Entries in the "Related Pattern" column show examples of patterns drawn from well-known collections of cloud patterns such as (Fehling et al. 2014, Homer et al. 2014, Wilder 2012). The cloud patterns found in these collections may have different names but they share the same underlying implementation principle. For example, the message-oriented middle-ware pattern (Fehling et al. 2014) is captured in Homer et al. (Homer et al. 2014) and Wilder (Wilder 2012) as a Queue-centric workflow pattern and competing consumers pattern,

respectively.

Table 4.1: Taxonomy of Deployment Patterns for Cloud-hosted Applications

Deployment Components	**Categories of Deployment Patterns**		**Related Patterns**
	Main Categories	*Sub-Categories*	
Cloud-hosted Environment (Taxonomy A)	Application Process	Project processes	Static workload
		Implementation processes	Continuously changing workload
		Support processes	Continuously changing workload
	Core Cloud Properties	Rapid Elasticity	Elastic platform,Autoscaling (Wilder 2012)
		Resource Pooling	Shared component, Private cloud
		Measured Service	Elastic Platform, Throttling (Homer et al. 2014)
	Cloud Service Model	Software resources	SaaS
		Platform resources	PaaS
		Infrastructure resources	IaaS
	Cloud Deployment Model	Private clouds	Private cloud
		Community clouds	Community cloud
		Public clouds	Public cloud
		Hybrid clouds	Hybrid cloud
	Composite Cloud Application	Hybrid cloud applications	Hybrid Processing, Hybrid Data,Multisite Deployment (Wilder 2012)
Cloud-hosted Application (Taxonomy B)		Decomposition style	2-tier/3-tier application, Content Delivery Network (Wilder 2012)
	Cloud Management	Management Processes	Update Transition Process, Scheduler Agent (Homer et al. 2014)
		Management Components	Elastic Manager, Provider Adapter, External Configuration Store (Homer et al. 2014)
	Cloud Offerings	Communication Offering	Virtual Networking, Message-Oriented Middleware
		Storage Offering	Block Storage, Database Sharding (Wilder 2012), Valet Key (Homer et al. 2014)
		Processing Offerings	Hypervisor, Map Reduce (Wilder 2012)
		Cloud Environment Offerings	Elastic Infrastructure, Elastic Platform, Runtime Reconfiguration (Homer et al. 2014)
	Cloud Application Architecture	Integration	Integration Provider, Restricted Data Access Component
		Multi-tenancy	Shared Component, Tenant-Isolated Component
		Application components	Stateless Component, User Interface Component

4.2.3 Validation of the Taxonomy

The taxonomy was validated in theory by adopting the approach used by Smite et al. (Smite et al. 2012) to validate his proposed taxonomy for terminologies in global software engineering. A taxonomy can be validated with respect to completeness by benchmarking against existing classifications and demonstrating its utility to classify existing knowledge (Smite et al. 2012).

Taxonomy A is benchmarked to existing classifications: the ISO/IEC 12207 taxonomy of software life cycle processes and the components of a cloud model based on NIST cloud computing definition, NIST SP 800-145. Taxonomy B is benchmarked to components of a cloud application

architecture such as cloud offering and cloud management, as proposed by Fehling et al. (Fehling et al. 2014). The collection of patterns in (Fehling et al. 2014) captures all the major components and processes required to support a typical cloud-based application, such as cloud management and integration.

The utility of the taxonomy has been demonstrated in Section 4.4.2 by (i) positioning the five selected GSD tools within the taxonomy; and (ii) applying CLIP to guide an architect in identifying applicable deployment patterns together with the supporting technologies for deploying GSD tools to the cloud. Tables 4.3 and 4.4 show that several deployment patterns (chosen from different studies such as (Fehling et al. 2014, Homer et al. 2014, Wilder 2012)) can be placed in the sub-categories of our taxonomy. Furthermore, Section 4.4.2 describes CLIP, a general process for selecting applicable patterns using the taxonomy and then demonstrate its practicality with a motivating cloud deployment problem in Section 4.4.3.

4.3 GSD Tool Selection

This section explains how the GSD tools and supporting processes used in the study were selected.

4.3.1 Research Site

An empirical study was carried to find out: (1) the type of GSD tools used in large-scale distributed enterprise software development projects; and (2) what tasks they utilise the GSD tools for. The study involved eight international companies, and interviews were conducted with 46 practitioners. The study was conducted between January 2010 and May 2012 and then updated between December 2013 and April 2014. The companies were selected from a population of large enterprises involved in both onshore and offshore software development projects. The companies had head offices in countries spread across three continents: Europe (UK), Asia (India), and North America (USA). Data collection involved document examination/reviews, site visits and interviews. Further details of the data collection and data analysis procedure used in the empirical study can be seen in Bass (Bass 2014).

4.3.2 Derived Dataset of GSD Tools

The selected set of GSD tools are: JIRA (Atlassian.com 2016), VersionOne (VersionOne.com 2017b), Hudson (Moser & O'Brien 2016), Subversion (Collins-Sussman et al. 2004) and Bugzilla (Bugzilla 2016). These tools were selected for two main reasons: (i) practitioners confirmed the use of these tools in large-scale geographically distributed enterprise software development projects (Bass 2014); (ii) the tools represent a mixture of open-source and commercial tools that support different software development processes; and are associated with stable developer communities (e.g., Mozilla Foundation) and publicly available records (e.g., developer's websites, white-papers, manuals). Table 4.2 (another view of the one in (Bass 2014)) shows the participating companies, projects and the GSD tools they used. A summary of the selected GSD tools is given below:

JIRA: JIRA is a bug tracking, issue tracking and project management software tool. JIRA products (e.g., JIRA Agile, JIRA Capture) are available as a hosted solution through Atlassian OnDemand, which is a SaaS cloud offering. JIRA is built as a web application with support for plugin/API architecture that allows developers to integrate JIRA with third-party applications such as Eclipse, IntelliJ IDEA and Subversion (Atlassian.com 2016).

Hudson: Hudson is a Continuous Integration (CI) tool, written in Java, for deployment in a cross-platform environment. Hudson is hosted partly as an Eclipse Foundation project and partly as a Java.NET project. It has a rich set of plugins making it easy to integrate with other software tools (Hudson 2016c). Organisations such as Apple and Oracle use Hudson for setting up production deployments and automating the management of cloud-based infrastructure (Moser & O'Brien 2016).

VersionOne: VersionOne is an all-in-one agile management tool built to support agile development methodologies such as Scrum, Kanban, Lean, and XP (VersionOne.com 2017b). It has features that support the handling of vast amounts of reports and globally distributed teams in complex projects covering all aspects of teams, backlog and sprint planning. VersionOne can be deployed as a SaaS (on-demand) or On-Premises (local) (Versionone.com 2017a).

Subversion: Subversion is a free, open source version control system used for managing files and directories, and the changes made to them over time (Collins-Sussman et al. 2004). Subversion

Table 4.2: Participating Companies, Software Projects, Software-specific Process and GSD tools used

Companies	Projects	Software process	GSD tool
Company A, Bangalore	Web Mail Web Calendar	Issue tracking Code integration	JIRA Hudson
Company B, Bangalore	Web Mail Web Calendar	Issue tracking Version control	JIRA Subversion
Company H, Delhi	Customer service Airline	Agile tailoring Issue tracking	VersionOne JIRA
Company D, Bangalore (Offshore Provider to Company E)	Marketing CRM	version control Error tracking	Subversion Bugzilla
Company E, London	Banking Marketing CRM	Issue tracking Agile tailoring Code Building	JIRA VersionOne Hudson

implements a centralised repository architecture whereby a single central server hosts all project metadata. This facilitates distributed file sharing (Lanubile et al. 2010).

BugZilla: Bugzilla is a Web-based general-purpose bug tracker and testing tool originally developed and used for the Mozilla project (Bugzilla 2016). Several organisations use Bugzilla as a bug tracking system for both open sources (Apache, Linux, Open Office) and proprietary projects (NASA, IBM) (Serrano & Ciordia 2005b).

4.4 Applying the Taxonomy

In this section, our focus is to demonstrate the practicality of the taxonomy in two ways: (i) positioning the selected GSD tools against the taxonomy (see Table 4.3 and Table 4.4); and (ii) presenting a process for identifying applicable deployment patterns for cloud deployment of GSD tools (see Section 4.4.3). This framework may be used for other similar GSD tools not listed in our dataset.

4.4.1 Positioning GSD Tools on the Taxonomy

The practicality of the taxonomy is demonstrated by applying it to position a selected set of GSD tools. The collection of patterns from (Fehling et al. 2014) is used as our reference point, and then complemented the process with patterns from (Homer et al. 2014, Wilder 2012).

The structure of the positioned deployment pattern, in its textual form, is specified as a string consisting of three sections-(i) Applicable deployment patterns; (ii) Technologies required to support such implementation; and (iii) Known uses of how the GSD tool (or one of its products)

implements or supports the implementation of the pattern. In a more general sense, the string can be represented as: [PATTERN; TECHNOLOGY; KNOWN USE]. When more than one pattern or technology is applicable, they are separated by commas. Each sub-category of the taxonomy represents a unique class of a reoccurring cloud deployment problem, while the applicable deployment pattern represents the solution. Table 4.3 and Table 4.4 shows how the GSD tools were positioned on the taxonomy.

4.4.2 How to Identify Applicable Deployment Patterns using the Taxonomy

Based on the experience gathered from positioning the selected GSD tools on the taxonomy, this thesis describes CLIP (CLoud-based Identification process for deployment Patterns), a general process for guiding software architects in selecting applicable cloud deployment patterns for GSD tools using the taxonomy. The development of CLIP (shown in Figure 4.1 in Business Process Model and Notation (BPMN)) was inspired by IDAPO. Stol et al. (Stol et al. 2011) used IDAPO to describe a process for identifying architectural patterns embedded in the design of open-source software tools.

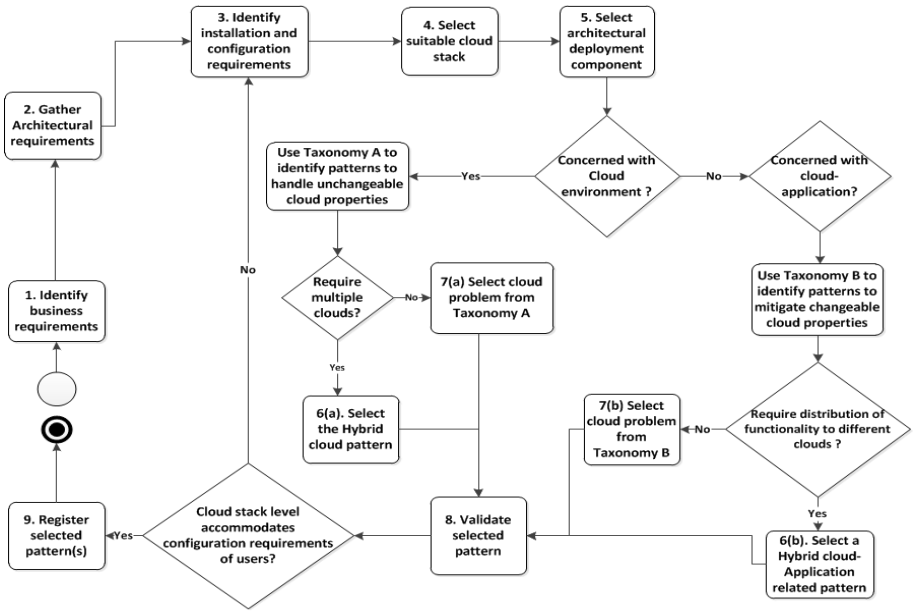

Figure 4.1: CLIP Framework for Identifying Deploying Patterns

The process of selecting the cloud deployment pattern(s) is an iterative process. The first step is to (1) **find out the main business requirements of the organization**. An example of a business requirement is fast feedback time, secure access to the shared component, and even the require-

ment of a limited resource. The next step is to (2) **gather information about the architectural structure of the GSD tool**. This study recommends the use of IDAPO, a process designed by Stol el al(Stol et al. 2011) for identifying architectural patterns in open-source software. At the end of that process, the architect would be able to identify among other things, the type of software and its domain, the technologies used, components and connectors, data source requirements (e.g., database type, data access method, file system etc.), and the default architectural pattern used in the design of the software.

After gathering information about the architectural structure of the GSD tool, the next step is to (3) **identify all the installation and configuration requirements of the GSD tool**. This information can be obtained directly from the documentation of the GSD tool or by creating a test application with the GSD tool. Based on the information gathered in the previous steps, the architect would be able to (4) **from the given cloud infrastructure, select a suitable level of the cloud application stack that will accommodate all the installation and configuration requirements of the user**. If in doubt, it would be recommended that the architect should start with the first cloud stack level, which is the application level (i.e., GSD tool together with the software process it supports).

At this stage, the architect has to (5) **choose the architectural deployment component of interest**. In the cloud (as in other distributed environments), a cloud deployment pattern targets either the cloud environment or the cloud application. If the architect is concerned with the cloud environment, then Taxonomy A should be used to select patterns for mapping business requirements to the unchangeable cloud properties, such as the location of the cloud infrastructure. However, if the architect is concerned with the cloud-hosted application, then Taxonomy B should be used to select deployment patterns for mitigating cloud properties, for example, the performance and availability of the cloud application.

The architect should then (6) **check for hybrid deployment requirements**. Usually, there are three main requirements that motivate the use of a hybrid-related cloud pattern. These include (i) elasticity where there is need to increase or decrease the availability of cloud resources; (ii) accessibility; and (iii) combined assurance of privacy, security and trust (Fehling et al. 2014). For Taxonomy A, a typical requirement would be the need for integration of multiple clouds into a homogenous environment (e.g., using the hybrid cloud pattern), while that of Taxonomy B would be the need for distribution of the functionality/components of the GSD tool among different

clouds (e.g., using the hybrid processing pattern). In either case, the respective hybrid related sub-category should be referenced to identify applicable patterns. Otherwise, the architect has to (7) **select a cloud deployment problem that corresponds to the sub-category of the chosen Taxonomy**. The cloud deployment patterns have been arranged into 8 high-level categories and 24 sub-categories that represent a recurring cloud deployment problem.

At this point, the process of selecting suitable deployment patterns involves referencing many sources of information several times. The architect can map the component/process of the GSD tool to the resources of the cloud infrastructure. It is recommended that the architect should re-visit steps 1, 2, and 3. Assuming an architect wants Hudson to communicate with other external components/applications, then a better deployment pattern of choice would be Virtual Networking (via self-service interface) to allow different users to be isolated from each other, to improve security and shield users from performance influence. However, if the communication is required internally to exchange messages between application components, then a message-oriented middleware would be the obvious choice.

After selection, the (8) **patterns have to be validated** to ensure that the chosen cloud stack level can accommodate all the installation and configuration requirements of the GSD tool. This can be done by mapping the components/process of the GSD tool identified from the previous steps to the available cloud resources. Another option would be to create a test application with the GSD tool to check if deploying to the cloud is workable. If validation fails, the architect may move one level lower in the cloud stack and repeat the process from step 4. Once confirmed, the (9) **selected pattern(s) (together with the use case that gave rise to the selection) should be registered** in a repository for later use by other architects.

4.4.3 Case Study: Selecting Patterns for Automated Build Verification Process

This section presents a simple case study of a cloud deployment problem to illustrate how to use the process described in this thesis (i.e., CLIP) given our taxonomy to guide in the selection of applicable pattern for deploying components used in an automated build verification process.

Motivating Problem: A cloud deployment architect intends to deploy a data-handling component to the cloud so that its functionality can be integrated into a cloud-hosted Continuous Integration System (e.g., Hudson). The laws and regulations of the company make it liable to archive builds

of source code once every week and keep it accessible for auditing purposes. Access to the repository containing the archived source code will be provided solely to certain groups of users. How can an architect deploy a single instance of this application to the cloud to serve multiple users, so that the performance and security of a particular user do not affect other users when there is a change in the workload?

Proposed Solution: This section will explain how to go through the steps outlined in Section 4.4.2 to select an appropriate cloud deployment pattern for handling the above cloud deployment problem.

Step 1: The key business requirements of this company are: (i) the shared repository that archives the source code cannot be shared; (ii) a single instance of this application should be deployed to the cloud to serve multiple users, and (iii) isolation among individual users should be guaranteed.

Step 2: Hudson is a web-based application, and so it can easily be modified to support a 3-tier architectural pattern. An important component of this architectural pattern is the shared repository containing the archived data.

Step 3: Information obtained from Hudson documentation suggests that Hudson needs a fast and reliable communication channel to ensure that data is archived simultaneously between different environments/clouds.

Step 4: A review of the hardware and software requirements from Hudson documents suggests that having access to the application level and middle-level of the application stack will be sufficient to provide the configuration requirements for deploying and running Hudson on the given cloud infrastructure. A self-service interface can be provided as a PaaS (e.g., Amazon's Elastic Beanstalk) for configuring the hardware and software requirements of Hudson.

Step 5: The architectural deployment component of interest is the cloud-application itself since the user has no direct access to the cloud IaaS. Therefore, the architect must select a deployment pattern that can be implemented at the application level to handle the business requirements of the company. Based on this information, the architect turns to Taxonomy B, which contain cloud patterns used to mitigate cloud properties such as performance on the application level. The fact that the architect is not attempting to integrate two cloud environments further strengthens the choice of our architectural deployment component of interest.

Step 6: After a careful review of the requirements, the architect concludes that a hybrid-related

deployment pattern is the most suitable cloud deployment pattern for addressing the requirements of the customer. It is assumed that the data archived by Hudson contains the source code that drives a critical function of an application used by the company. Any unauthorised access to it can be disastrous to the company. The hybrid backup deployment pattern seems to be the most appropriate in this circumstance. This pattern can be used to extract and archive data to the cloud environment. Fehling et al. (Fehling et al. 2014) discussed several types of hybrid -related patterns that can be used at the application level.

Step 7: As the hybrid backup pattern has been selected in the previous step, carrying out step 7 to select a deployment problem that corresponds to a particular sub-category of the taxonomy is no longer relevant. However, there are other patterns that can be selected from Taxonomy B for complementary purposes. For example, in a situation where the performance of the communication channel is an issue, the message-oriented pattern can be used to assure the reliability of messages sent from several users to access the component that is shared.

Step 8: The selected deployment pattern was validated by carefully reviewing its implementation to ensure that it can accommodate the user's configuration requirements and ultimately address the cloud deployment problem. Hudson and its supporting components are mapped to the available cloud resources.

Figure 4.2 shows the architecture of the hybrid backup that is proposed for solving the cloud deployment problem. The architecture consists of two environments: one is a static environment that hosts Hudson and the other is an elastic cloud environment where the cloud storage offering (e.g., Amazon's S3) resides. This static environment represents the company's Local Area Network (LAN) that runs Hudson. During Hudson's configuration on the "Post-build Action" section, the location of the files to archive should point to the storage offering that resides in the cloud environment.

The cloud storage (accessed via a REST API) should be configured in such a way that guarantees isolation among the different users. It is assumed that the data handling component is initially available as a shared component for all users. To ensure that the archived data is not shared by every user, the same instance of the shared component can be instantiated and deployed exclusively for a certain number of users.

From an implementation standpoint, all user id's associated with each request to Hudson are captured and those requests with exclusive access rights are then routed to the cloud storage. An

approach named "COMITRE (Cloud-based approach to Multitenancy Isolation Through Request RE-routing) is discussed in Ochei et al. (Ochei, Bass & Petrovski 2015c) for deploying a single instance of Hudson to the cloud for serving multiple users (i.e., multitenancy) in such a way that guarantees different degrees of isolation among the users.

The different degrees of isolation between users accessing an application component that is shared is captured in three multitenancy patterns: shared component, tenant-isolated component and dedicated component (Fehling et al. 2014).

Step 9: Finally, the cloud deployment scenario, the selected patterns together with the implemented architecture is documented for reference and reuse by other architects.

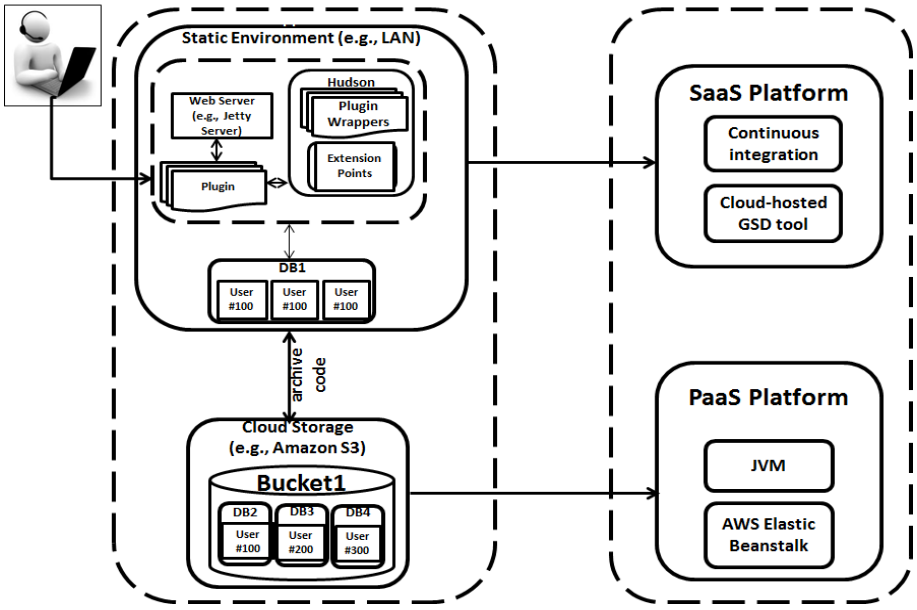

Figure 4.2: Mapping Hudson to Cloud Stack based on Hybrid Backup pattern

4.5 Findings

This section presents the findings obtained by applying the taxonomy against a selected set of GSD tools: JIRA, VersionOne, Hudson, Subversion and Bugzilla. Refer to Section 4.3.2 for details of the processes supported by these tools.

(1) Patterns Related to Cloud-Environment or Cloud-Hosted Application: The cloud deployment patterns featured in Taxonomy A (i.e., the upper part of Table 4.1) relate to the cloud environ-

ment hosting the application, while the cloud deployment patterns in Taxonomy B (i.e., the lower part of Table 4.1) relate to the cloud-hosted application itself. For example, the PaaS pattern falls within Taxonomy B and can be used to provide an execution environment to customers on the provider-supplied cloud environment. The Elastic platform pattern falls within Taxonomy A and can be used in the form of a middleware integrated into a cloud-hosted application to provide an execution environment.

(2) Hybrid-related deployment Patterns: Both taxonomies contain patterns for addressing hybrid deployment scenarios (i.e., the space demarcated by thick lines). For example, a hybrid cloud (Taxonomy A) integrates different clouds and static data centres to form a homogeneous hosting environment, while hybrid data (Taxonomy B) can be used in a scenario where data of varying sizes generated from a GSD tool resides in an elastic cloud, and the remainder of the application resides in a static environment.

(3) Patterns for Implementing Elasticity: This study observes that there are patterns that can be used by GSD tools to address rapid elasticity at all levels of the cloud stack. For example, an Elastic manager can be used at the application level to monitor the workload experienced by the GSD tool and its components (based on resource utilisation, the number of messages exchanged between the components, etc.) to determine how and when to provision or de-provision resources. Elastic platform and Elastic infrastructure can be used at the platform and infrastructure resources level, respectively.

(4) Accessing Cloud Storage: The data handling components of most GSD tools are built on block storage architectures (e.g., relational databases such Oracle and MySQL used within Hudson and Bugzilla) for storing data, which are directly accessible by the operating system. However, a vast majority of storage offerings available on the cloud are based on object storage architecture. For example, Amazon S3, Google Cloud Storage and Windows Azure Blob provide cloud storage to cloud applications according to the blob storage pattern (Fehling et al. 2014). Blob storage can be very useful for archiving large data elements (e.g., video, installers, and ISO images) arising from Hudson builds and test jobs.

(5) Positioning of GSD tools on the Taxonomy: Tables 4.3 and 4.4 show the findings obtained by positioning the cloud-hosted GSD tools on each sub-category of the taxonomy. The section that follows presents a shortlist of these findings to show how to identify applicable deployment patterns to address a wide variety of deployment problems.
(i) All the GSD tools considered in this study are based on web-based architecture. For example, Bugzilla and JIRA are designed as a web-based application, which allows for separation of the user interface, and processing layers from the database that stores details of bugs/issues being tracked.

(ii) All the GSD tools support API/Plugin architecture. For example, JIRA supports several APIs that allow it to be integrated with other GSD tools. The *Bugzilla:Web services*, a standard API for external programs to interact with Bugzilla, implies support for a stateless pattern. These APIs represent known uses of how these deployment patterns are implemented.

(iii) Virtualization is a key supporting technology used in combination with other patterns to achieve elasticity at all levels of the cloud stack, particularly in ensuring fast provisioning and de-provisioning of infrastructure resources.

(iv) The GSD tools use Web services (through a REST API in patterns such as integration provider (Fehling et al. 2014)) to hold external state information, while messaging technology (through message queues in patterns such as Queue-centric workflow (Wilder 2012) and Queue-based load leveling (Homer et al. 2014)) is used to exchange information asynchronously between GSD tools/components.

(vi) Newer commercial GSD tools (JIRA and VersionOne) are directly offered as SaaS on the public cloud. On the other hand, older open-source GSD tools (Hudson, Subversion and Bugzilla) are preferred for private cloud deployment. They are also available on the public cloud, but by third party cloud providers.

The findings are summarised as follows: Although there are a few patterns that are mutually exclusive (e.g., stateless versus stateful components, and strict versus eventual consistency

(Fehling et al. 2014)), most patterns still have to be combined with others (e.g., combining PaaS with an Elastic platform). These deployment patterns may also use similar technologies such as REST, messaging and virtualization to facilitate their implementation.

4.6 Recommendations for using the Taxonomy

This section presents a set of recommendations in the form of selection criteria in Table 4.5 to guide an architect in choosing applicable deployment patterns for deploying any GSD tool.

To further assist the architect in making a good choice, this study describes CLIP (CLoud-based Identification process for deployment Patterns), a general process for guiding architects in selecting applicable cloud deployment patterns for GSD tools using our taxonomy. The development of CLIP was inspired by IDAPO, a similar process proposed by Stol et al. for identifying architectural patterns in open source software (Stol et al. 2011).

The key for successfully using CLIP is selecting a suitable level of cloud stack that will accommodate all the configuration requirements of the GSD tool to be selected. The architect has more flexibility to implement or support the implementation of a deployment pattern when there is greater "scope of control" of the cloud stack according to either the SaaS, PaaS or IaaS service delivery model (Badger et al. 2012). For example, to implement the hybrid data pattern (Fehling et al. 2014) for deploying Hudson to an elastic cloud, the architect would require control of the infrastructure level of the cloud stack to allow for provisioning and de-provisioning of resources (e.g., storage, memory, CPU).

4.7 Chapter Summary

This chapter discussed how to create and use a taxonomy of deployment patterns for cloud-hosted applications to contribute to the literature on cloud deployment of Global Software Engineering tools. Eight categories that form the taxonomy have been described: Application process, Cloud properties, Service model, Deployment model, Application architecture, Cloud offerings, Cloud management, and Composite applications. These categories were further partitioned into 24 sub-categories, which were mapped to the components of an (architectural) deployment structure. This mapping revealed two component classes: cloud-hosted environment and cloud-hosted

Table 4.3: Positioning GSD Tools on the Proposed Taxonomy (Taxonomy A)

Category	**Sub-Category**	*JIRA*	*VersionOne*	*Hudson*	*Subversion*	*Bugzilla*
Application Process	Project processes	**Static workload, Continuously changing workload**; *SaaS*; JIRA used by small no. of users, issues tracked reduces over time(Atlassian.com 2016)	**Static workload**; *SaaS*; VersionOne is installed for a small number of users(VersionOne.com 2017b)	Process not supported	Process not supported	Process not supported
	Implementation processes	Process not supported	Process not supported	**Continuously changing workload**; *PaaS*; Hudson builds reduces gradually as project stabilizes(Moser & O'Brien 2016)	Process not supported	Process not supported
	Support processes	Process not supported	Process not supported	Process not supported	**Static workload, Continuously changing workload**;*PaaS, Hypervisor*; rate of code files checked into Subversion repository is nearly constant or reduces over time(Collins-Sussman et al. 2004)	**Continuously changing workload**; *PaaS,Hypervisor*; Errors tracked using Bugzilla reduces over time(Bugzilla 2016)
Core Cloud Properties	Rapid Elasticity	**Stateless pattern, Elastic platform**; *REST API*; JIRA is installed in cloud as SaaS(Atlassian.com 2016)	**Stateless pattern, Elastic platform**; *REST API*; VersionOne is installed in cloud as SaaS(VersionOne.com 2017b)	**Elastic infrastructure, shared component**; *hypervisor*; Hudson server is supported by hypervisor in a private cloud(Moser & O'Brien 2016)	**Elastic infrastructure, tenant-isolation component**; *hypervisor*; Subversion repository is supported by Elastic infrastructure(Collins-Sussman et al. 2004)	**Stateless pattern**; *REST API*; Bugzilla is installed in cloud as SaaS in private cloud(Bugzilla 2016)
	Resource Pooling	**Hypervisor, Public Cloud,** ; *Virtualization*; JIRA deployed on the public cloud as SaaS(Atlassian.com 2016)	**Hypervisor, Public cloud**; *Virtualization*; VersionOne deployed on public cloud as SaaS(VersionOne.com 2017b)	**Hypervisor, Tenant-isolated component**; *Virtualization*; Hudson is deployed on a hypervisor(Moser & O'Brien 2016)	**Hypervisor, Tenant-isolated component**; *Virtualization*; Subversion is deployed on a hypervisor(Atlassian.com 2016)	**Hypervisor, Public cloud**; *Virtualization*; Bugzilla deployed on the public cloud(Bugzilla 2016)
	Measured Service	**Static workload, Elastic Infrastructure,Throttling (Homer et al. 2014)**; *Virtualization*; Small number JIRA users generates a nearly constant workload(Atlassian.com 2016)	**Static workload, Elastic Infrastructure,Throttling (Homer et al. 2014)**; *Virtualization*; Small number of VersionOne users generates small workload(VersionOne.com 2017b)	**Static workload, Elastic Infrastructure,Throttling (Homer et al. 2014)**; *Virtualization*; Hudson can be supported on public cloud by elastic infrastructure(Moser & O'Brien 2016)	**Static workload, Elastic Infrastructure,Throttling (Homer et al. 2014)**; *Virtualization*; Subversion can be supported on public cloud by elastic infrastructure(Collins-Sussman et al. 2004)	**Static workload, Elastic Infrastructure,Throttling(Homer et al. 2014)**; *Virtualization*; Bugzilla can be supported on third party public cloud by elastic infrastructure(Bugzilla 2016)
Cloud Service Model	Software resources	**SaaS**; *Web Services, REST*; JIRA OnDemand(Atlassian.com 2016)	**SaaS**; *Web Services, REST*; VersionOne OnDemand(VersionOne.com 2017b)	**SaaS**; *Web Services, REST*; Hudson is offered by 3^{rd} party cloud providers like CollabNet(CollabNet n.d.)	**SaaS**; *Web Services, REST*; Subversion is offered by 3^{rd} party cloud providers like CollabNet(CollabNet n.d.)	**SaaS**; *Web Services, REST*; Bugzilla is offered by 3^{rd} party cloud providers like CollabNet(CollabNet n.d.)
	Platform resources	**PaaS**; *Elastic platform, Message Queuing*; JIRA Elastic Bamboo(Atlassian.com 2016)	**PaaS**; *Elastic platform, Message Queuing*; No known use	**PaaS**; *Elastic platform, Message Queuing*; Build Doctor and Amazon EC2 for Hudson	**PaaS**; *Elastic platform, Message Queuing*; Flow Engine powered by Jelastic for Subversion	**PaaS**; *Elastic platform, Message Queuing*; No known use
	Infrastructure resources	Not applicable	Not applicable	**IaaS**; *Hypervisor*; Hudson is a distributed execution system comprising master/slave servers(Moser & O'Brien 2016)	**IaaS**; *Hypervisor*; Subversion can be deployed on a hypervisor	Not applicable
Cloud Deployment Model	Private usage	**Private cloud**; *Hypervisor*; JIRA can be deployed on private cloud using private cloud software like OpenStack	**Private cloud**; *Hypervisor*; VersionOne On-premises(VersionOne.com 2017b)	**Private cloud**; *Hypervisor*; Hudson can be deployed on private cloud using private cloud software	**Private cloud**; *Hypervisor*; Subversion can be deployed on private cloud using private cloud software	**Private cloud**; *Hypervisor*; Bugzilla can be deployed on private cloud using private cloud software
	Community usage	**Community cloud**; *SaaS*; Bugzilla can be deployed on private cloud	**Community cloud**; *SaaS*; Bugzilla can be deployed on community cloud	**Community cloud**; *SaaS,Paas, IaaS*; Bugzilla can be deployed on community cloud	**Community cloud**; *SaaS,IaaS*; Bugzilla can be deployed on community cloud	**Community cloud**; *SaaS, PaaS*; Bugzilla can be deployed on community cloud
	Public usage	**Public cloud**; *SaaS*; JIRA OnDemand is hosted on public cloud(Atlassian.com 2016)	**Public cloud**; *SaaS*; VersionOne is hosted on public cloud(VersionOne.com 2017b)	**Public cloud**; *SaaS,PaaS,IaaS*; Hudson is hosted on public cloud(via 3^{rd} party providers)(CollabNet n.d.)	**Public cloud**; *SaaS, IaaS*; Subversion is hosted on public cloud(via 3^{rd} party providers)(CollabNet n.d.)	**Public cloud**; *SaaS, PaaS*; Bugzilla is hosted on public cloud(via 3^{rd} party providers)(CollabNet n.d.)
	Hybrid usage	**Hybrid cloud**; *SaaS*; JIRA used to track issues on multiple clouds	**Hybrid cloud**; *SaaS*; Agile projects are stored in different clouds(Collins-Sussman et al. 2004)	**Hybrid cloud**; *SaaS,PaaS, IaaS*; Hudson builds done in separate cloud	**Hybrid cloud**; *SaaS, IaaS*; Subversion repository resides in multiple clouds	**Hybrid cloud**; *SaaS, PaaS*;Bugzilla DB can be stored in different clouds

Table 4.4: Positioning GSD Tools on the Proposed Taxonomy (Taxonomy B)

Category	**Sub-Category**	*JIRA*	*VersionOne*	*Hudson*	*Subversion*	*Bugzilla*
Application Architecture	Application Components	**User interface component,Stateless**; *REST API, AJAX*; State information in JIRA thru REST API(Atlassian.com 2016)	**User-interface component,Stateless**; *jQuery AJAX, REST/Web Service*; VersionOne REST API(VersionOne.com 2017b)	**User-interface component,Stateless**; *REST API, AJAX*; Hudson Dashboard pages via REST(Moser & O'Brien 2016)	**User-interface component,Stateless**;*REST API, AJAX*; ReSTful Web Services used to interact with Subversion Repositories (Collins-Sussman et al. 2004)	**Stateless**; *Bugzilla:WebService API*; Bugzilla::WebService API(Bugzilla 2016)
	Multitenancy	**Shared component**; *Elastic Platform, Hypervisor*; JIRA login system(Atlassian.com 2016)	**Shared component**; *Hypervisor*; VersionOne supports reuseable configuration schemes(VersionOne.com 2017b)	**Shared component**; *Hypervisor*; Hudson 3.2.0 supports multi-tenancy with Job Group View and Slave isolation(Moser & O'Brien 2016)	**Tenant Isolated component**; *Hypervisor*; Global search/replace operations are shielded from corrupting subversion repository.(Collins-Sussman et al. 2004)	**Shared component**; *Hypervisor*; Different users are virtually isolated within Bugzilla DB(Bugzilla 2016)
	Cloud Integration	**Restricted Data Access component, Integration provider**; *REST API*; JIRA REST API is used to integrate JIRA with other applications(Atlassian.com 2016)	**Integration provider**; *REST, Web Services*; VersionOne OpenAgile Integrations platform, REST Data API for user stories(VersionOne.com 2017b)	**Integration provider**; *REST, Web Services*; Stapler component of Hudson's architecture uses REST(Moser & O'Brien 2016)	**Integration provider**; *REST, Web Services*; Subversion API(Collins-Sussman et al. 2004)	**Integration provider**; *REST, Web Services*; Bugzilla::WebService API(Bugzilla 2016)
Cloud Offering	Cloud environment Offering	**Elastic platform**; *PaaS*; JIRA Elastic Bamboo runs builds to create instances of remote agents in the Amazon EC2(Atlassian.com 2016)	**Integration provider**; *REST, Web Services*; Versionone's Project Management tools are used with TestComplete for automated testing environment (VersionOne.com 2017b)	**Elastic Infrastructure/Platform, Node-based Availability**; *PaaS, IaaS*; Hudson is a distributed build platform with "master/slave" configuration (Moser & O'Brien 2016)	**Elastic platform**; *PaaS*; Subversion repository can be accessed by a self-service interface hosted on a shared middleware	**Elastic Platform**; *PaaS*; Bugzilla s hosted on a middleware offered by providers(Bugzilla 2016)
	Processing Offering	**Hypervisor**; *Virtualization*; JIRA is deployed on virtualized hardware	**Hypervisor**; *Virtualization*; VersionOne can be deployed on virtualized hardware	**Hypervisor**; *Virtualization*; Hudson is deployed on virtualized hardware	**Hypervisor**; *Virtualization*; Subversion is deployed on virtualized hardware	**Hypervisor**; *Virtualization*; Bugzilla is deployed on virtualized hardware
	Storage Offering	**Block**; *Virtualization*; Elastic Bamboo can access centralized block storage thru an API integrated into an operating system running on virtual server(Atlassian.com 2016)	**Block storage**; *Virtualization*; VersionOne can access centralized block storage thru an API integrated into an operating system running on virtual server(VersionOne.com 2017b)	**Block, Blob storage**; *Virtualization*; Azure Blob service used as a repository of build artifacts created by a Hudson	**Hypervisor**; *Virtualization*; Subversion can access centralized block storage thru an API integrated into an operating system running on virtual server	**Hypervisor**; *Virtualization*; Bugzilla can access centralized block storage thru an API integrated into an operating system running on virtual server
	Communication Offering	**Message-Oriented Middleware**; *Message Queuing*; JIRA Mail Queue(Atlassian.com 2016)	**Message-Oriented Middleware**; *Message Queuing*; VersionOne's Defect Work Queues(VersionOne.com 2017b)	**Message-Oriented Middleware, Virtual networking**;*Message Queuing,Hypervisor*;Hudson's Execution System Queuing component	**Message-Oriented Middleware**;*Message Queuing*;Subversion's Repository layer(Collins-Sussman et al. 2004)	**Message-Oriented Middleware**;*Message Queuing*; Bugzilla's Mail Transfer Agent(Bugzilla 2016)
Cloud Management	Management Components	**Provider Adapter, Managed Configuration, Elastic manager**;*RPC, API*; JIRA Connect Framework(Atlassian.com 2016), JIRA Advanced configuration	**Managed Configuration**;*RPC, API*; VersionOne segregation and appl. configuration	**Elastic load balancer, watchdog**;*Elastic platform*; Hudson execution system's Load Balancer component)	**Managed Configuration**;*RPC, API*; configuration file is used to configure how/when builds are done	**Managed Configuration**;*RPC, API*; Bugzilla can use configuration file for tracking and correcting errors
	Management Processes	**Elastic management process**;*Elasticity Manager*; JIRA Elastic Bamboo, and Time Tracking feature(Atlassian.com 2016)	**Elastic management process**;*Elasticity Manager*; VersionOne's On-Demand security platform(VersionOne.com 2017b)	**Update Transition process**;*Message Queuing*; continuous integration of codes by Hudson's CI server(Moser & O'Brien 2016)	**Update Transition process**;*Message Queuing*; continuous updates of production versions of the appl. by Subversion(Collins-Sussman et al. 2004)	**Resiliency management process**;*Elasticity platform*; Bugzilla Bug monitoring/reporting feature(Bugzilla 2016)
Composite Application	Decomposition Style	**3-tier**;*stateless, processing and data access components*; JIRA is web-based application(Atlassian.com 2016)	**3-tier**;*stateless, processing and data access components*; VersionOne is a web application(VersionOne.com 2017b)	**3-tier, Content Dist. Network**;*user interface, processing, data access components, replica distr.*; Hudson is an extensible web application, code file replicated on multiple clouds(Moser & O'Brien 2016)	**3-tier**;*stateless, processing and data access components*; Subversion is a web-based application (Collins-Sussman et al. 2004)	**3-tier**;*stateless, processing and data access components*; Bugzilla is a web application(Bugzilla 2016)
	Hybrid Cloud Application	**Hybrid processing**; *processing component*; JIRA Agile used to track daily progress work(Atlassian.com 2016)	**Hybrid Development Environment**;*processing component*; VersionOne's OpenAgile Integration(VersionOne.com 2017b)	**Hybrid Data, Hybrid Development Environment**; *data access component*;Separate environment for code verification and testing	**Hybrid Data, Hybrid Backup**; *data access component,stateless*;Code files extracted for external storage	**Hybrid Processing**; *processing component*; DB resides in data center, processing done in elastic cloud

Table 4.5: Criteria for Selecting Applicable Patterns for Cloud Deployment of GSD Tools

Category	**Sub-Category**	***Selection Criteria***	***Applicable Patterns***
Application Process	Project Processes	Elasticity of the cloud environment is not required	Static workload
	Implementation Processes	Expects continuous growth or decline in workload over time	Continuously changing workload
	Support Processes	Resources required is nearly constant;continuous decline in workload	Static workload, Continuously changing workload
Core Cloud Properties	Rapid Elasticity	Explicit requirement for adding or removing cloud resources	Elastic platform, Elastic Infrastructure
	Resource Pooling	Sharing of resources on specific cloud stack level-IaaS, PaaS, SaaS	Hypervisor, Standby Pooling Process
	Measured Service	Prevent monopolization of resources	Elastic Infrastructure, Platform, Throttling/Service Metering(Homer et al. 2014)
Cloud Service Model	Software Resources	No requirement to deploy and configure GSD tool	Software as a Service
	Platform Resources	Requirement to develop and deploy GSD tool and/or components	Platform as a Service
	Infrastructure as a Service	Requires control of infrastructure resources (e.g., storage, memory) to accommodate configuration requirements of the GSD tool	Infrastructure as a Service
Cloud Deployment Model	Private Usage	Combined assurance of privacy, security and trust	Private cloud
	Community Usage	Exclusive access by a community of trusted collaborative users	Community cloud
	Public Usage	Accessible to a large group of users/developers	Public cloud
	Hybrid Usage	Integration of different clouds and static data centres to form a homogenous deployment environment	Hybrid cloud
Application Architecture	Application Components	Maintains no internal state information	User Interface component, Stateless pattern
	Multitenancy	A single instance of an application component is used to serve multiple users, depending on the required degree of tenant isolation	Shared component, tenant-isolated component, dedicated component
	Integration	Integrate GSD tool with different components residing in multiple clouds	Integration provider, Restricted Data Access component
Cloud Offering	Cloud environment	Requires a cloud environment configured to suit PaaS or IaaS offering	Elastic platform, elastic infrastructure
	Processing Offering	Requires functionality to execute workload on the cloud	Hypervisor
	Storage Offering	Requires storage of data in cloud	Block storage, relational database
	Communication Offering	(1) Require exchange of messages internally between appl. components; (2) Require communication with external components	(1) Message-oriented middleware; (2) Virtual Networking
Cloud Management	Management Components	(1) Pattern supports Asynchronous access; (2) State information is kept externally in a central storage	(1) Provider Adapter; Elastic manager; Managed Configuration
	Management Processes	(1)Application component requires continuous update; (2) Automatic detection and correction of errors	(1) Update Transition process; (2) Resiliency management process
Composite Application	Decomposition Style	Replication or decomposition of application functionality/components	(1) 3-tier; (2) Content Distribution Network
	Hybrid Cloud Application	Require the distribution of functionality and/or components of the GSD tool among different clouds	(1) Hybrid processing; (2) Hybrid Data; (3) Hybrid Backup; (4) Hybrid Development Environment

application. Cloud-hosted environment and cloud-hosted application classes capture patterns that can be used to address deployment challenges at the infrastructure level and application level, respectively.

By positioning a selected set of software tools, JIRA, VersionOne, Hudson, Subversion and Bugzilla, on the taxonomy, it is easy to identify applicable deployment patterns together with the supporting technologies for deploying cloud-hosted GSD tools. It was observed that most deployment patterns are related and can be implemented by combining with others, for example, in hybrid deployment scenarios to integrate data residing in multiple clouds.

Also, this chapter has described CLIP, a novel approach for selecting applicable cloud deployment patterns, and after that applied it to a motivating deployment problem involving the cloud deployment of a GSD tool to serve multiple users in such a way that guarantees isolation among different users. Recommendations have been provided in a tabular form, which shows the selection criteria to guide an architect in choosing applicable deployment patterns. Examples of deployment patterns derived from applying these selection criteria have been presented.

Chapter 5

Case Studies of Degrees of Multitenancy Isolation using COMITRE Approach

5.1 Introduction

This chapter presents an approach for implementing multitenancy isolation and its application to three case studies that empirically evaluates the varying degrees of multitenancy isolation for cloud-hosted GSD processes (i.e., continuous integration, version control and bug tracking). The three case studies have been published separately in (Ochei, Bass & Petrovski 2015c), (Ochei, Petrovski & Bass 2015) and (Ochei, Bass & Petrovski 2016). The report in these papers are presented and duly referenced in this chapter.

In chapter two, it was stated that as software tools used for Global Software Development (GSD) are increasingly being deployed to the cloud to serve multiple users/tenants, there is need to implement multitenancy. Multitenancy is an essential architectural practice in cloud computing that allows a single instance of a service to be used to serve multiple tenants. Since multiple users are expected to access a shared functionality or resource, therefore in addition to implementing multitenancy, there is also need to implement multitenancy isolation to ensure that the processes and data (e.g., source code, bug reports) associated with a particular tenant (or component) does not affect others. (Fehling et al. 2014) (Bauer & Adams 2012).

This chapter begins by describing an approach for implementing and evaluating the required degree of multitenancy isolation in Section 5.2. This description covers its architecture and pro-

cedure for implementation, supporting algorithms and problem scenarios for illustrating multitenancy isolation. After that, the approach is applied to empirically evaluate the varying degrees of multitenancy isolation in three case studies. The case studies are continuous integration with Hudson (Section 5.3), Version control with File System SCM plugin (Section 5.4) and Bug tracking with Bugzilla (Section 5.5). Section 5.6 summarises the chapter.

5.2 COMITRE: An Approach for Implementing Multitenancy Isolation

This section describes an approach, known as COMITRE (Component-based approach to multitenancy isolation through Request Re-routing) for implementing the required degree of multitenancy isolation and the supporting algorithms that are integrated into the GSD tools to support the implementation of the varying degrees of multitenancy isolation.

5.2.1 Architecture and Procedure for Implementation

The Component-based approach to multitenancy isolation through Request Re-routing (COMITRE) is an approach for implementing the varying degrees of multitenancy isolation for cloud-hosted services/applications. COMITRE can be seen as an abstract format that allows the implementation of multitenancy isolation in various ways. It captures the essential properties required for the successful implementation of multitenancy isolation while leaving large degrees of freedom to cloud deployment architects depending on the required degree of isolation between tenants. Furthermore, our approach can be applied at different levels of the application or cloud stack because it exploits client transactions/requests by capturing and analysing them. Figure 5.1 captures the architecture of COMITRE. The approach for implementing the varying degrees of multitenancy isolation is summarised in the following steps:

Step 1: Define the structure of the tenant request- The structure of the tenant identifier has to be clearly defined. The tenant identifier can be in various forms such as an IP address, port number, request header, or a query string attached to the request. Once the format of the tenant identifier has been chosen, this is then used to define the structure of a typical tenant request. For example, when using a load generator like Apache JMeter, the tenant identifier can be sent as a parameter along

with the request which will appear as a query string. The structure of the HTTP request looks like this: 172.19.1.2:8080/FileTrigger1/build?delay=0sec?tenant1=1. There are other attributes that can be extracted from a client request such as sessionID used in web service transactions, machine identifier of the client request (Connolly 2004).

Step 2: Configure Server to re-route tenant request to application component: The next step is to configure the web server to re-route the server request to a component of the application. This entails reconfiguring the hosts(file) (hereafter also referred to as *tenant-conf file*), an operating system file that maps hostnames to IP addresses. This reconfiguration can be done in two ways: (i) programmatically configuring the hosts (file) using programming language like Java or even in bash shell script; (ii) manually entering the tenant identifier into the hosts (file) (/etc/hosts/) so that the request of all the tenants points to the same IP address of the localhost (usually 127.0.0.1 in Ubuntu).

Step 3: Create a configuration for each multitenancy pattern: The configuration of each multi-tenancy pattern (i.e., shared component, tenant-isolated component and dedicated component) is created. A default configuration is also created to be assigned to every tenant in case a matching tenant-identifier was not found in the tenant-conf file. These configurations map to the different degrees of isolation between tenants. Assuming the component to be shared is a database, creating a configuration for the tenant-isolated component pattern simply translates to having a separate schemas/tables for each tenant within a single database.

Step 4: Tenant Identification and Resolution: This is a two-step process: (i) capture the incoming request (e.g., HTTP, FTP, JDBC request); (ii) extract the tenant-identifier (based on the format defined in step 1) from the request and use it to resolve the identity of each tenant. Assuming the request is represented in the form of a string, there are various ways to return a new string that is a substring of this string programmatically depending on the chosen programming language (e.g., the substring() in Java).

Step 5: Configure tenant-specific information: Based on this tenant-identifier and its associated information, a specific configuration is created for each tenant. The configuration includes but is

not limited to information such as tenant-identifier, tenant request, the required degree of isolation, and the application component that is to be accessed by the tenant. This component can be in any tier of the application stack- application tier, middle tier, or data tier.

Step 6: Select matching tenant configuration: From the list of tenants in the tenant-conf file, select the tenant configuration that matches the tenant-id of the user. The selected tenant configuration is returned to the tenant that made the request, otherwise the default tenant configuration is returned if a matching tenant is not found.

Step 7: Send viewable response to the user: The last step is to present the viewable response to the user. This response is the multitenant component that has been adjusted based on the tenant-specific configuration.

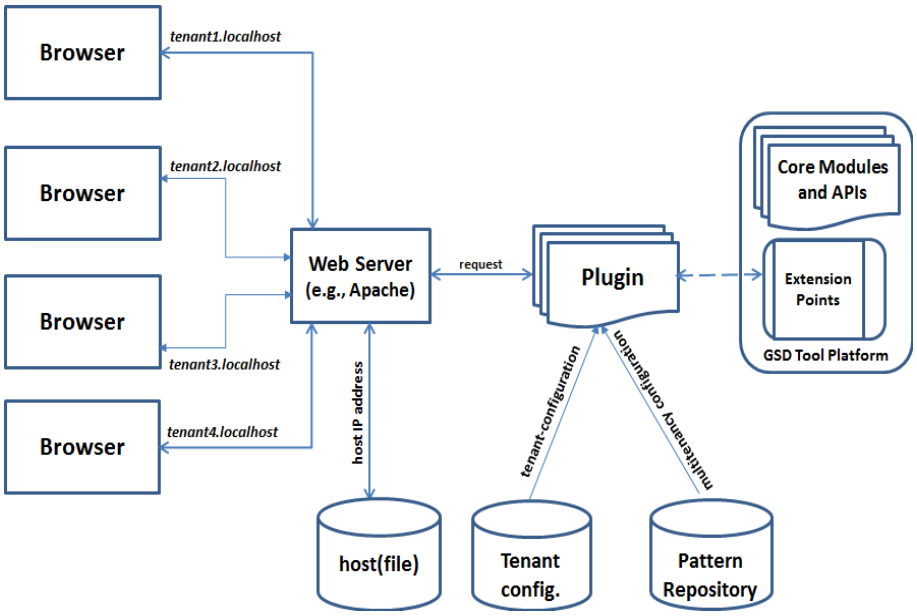

Figure 5.1: COMITRE Architecture.

5.2.2 Algorithms for Supporting COMITRE

The actual implementation of the COMITRE is anchored on shifting the task of routing a request from the server to a separate component at the application level of the cloud-hosted GSD tool. For example, this component could be a program component (e.g., Java class file) or a software component (e.g., plugin) which can be integrated into the GSD tool. Once the request is re-

routed to a component and captured, then important attributes of the request can be extracted and configured to reflect the degree of isolation required by the tenant. The logic that is implemented in the component is shown in Algorithm 1.

The input to the algorithm is the tenant request and tenant-conf file, while the output is the multitenant functionality or component that is shared among the different tenants. This input could be a text file or a database that contains among other things the tenant-identifier, the default functionality of the applications as well as the functionality that is to be exposed to the different tenants. Each tenant-specific data should be configured (either manually or programmatically) before the request for each tenant is sent to the application. Another option could be to update the "hosts" file (i.e., typically found in the "/etc/hosts" folder on Ubuntu) and add entries for the IP addresses of other tenants to point to the default IP address of the host. The algorithm begins by capturing the tenant identifier from an incoming request (e.g., HTTP, FTP, JDBC request). The tenant identifier could be a query string attached to the URL of each request or an IP address. Tenant-specific data for each tenant is selected from the configuration file and mapped to the tenant request which is then used to adjust the behaviour of the functionality or component that is being shared. If tenant configuration is not found, then the default functionality is returned.

The algorithm presented in this paper captures how to implement the required degree of isolation between tenants using the appropriate multitenancy pattern. The required isolation level of tenants is set to 1, 2, and 3 for the three different multitenancy patterns. The logic is summarised as follows: (i) if the isolation level is 1, then a tenant can access the created component regardless of where is it located; (ii) if the isolation level is 2, then the tenant has to be authenticated first and assigned a unique tenantID which is then used to adjust the behaviour of the created component; and (iii) if the isolation level is 3, then the created component is marked as not to be shared with others, and so is reserved exclusively for one tenant.

Algorithm 1 assumes that the architect specifies the required isolation level for each component. However, in a cloud environment, such decisions should be taken in almost real-time, and an algorithm is needed that can determine which isolation level is best for a component or functionality to be created. *Algorithm 2* presents an algorithm to determine the isolation level for a tenant application function based on an existing application component in a component repository. The input to the algorithm is a component repository and the shared status of the components (i.e., whether or not the component can be shared with other tenants). In line 4, if sharedStatus is false,

then isolation level is set to either 1 or 2; otherwise, control is transferred to line 14-15 to assign the isolation level 3. The first type of information is whether or not the application component is similar in functionality or configuration to existing ones. This is captured in line 5-8 of *Algorithm 2*, where similar components are searched for in the component repository. If components with similar configurations are found, then isolation level is set to 1 in line 10; otherwise, it is set to 3.

Algorithm 1 COMITRE Algorithm

```
1:  INPUT: tenantRequest, tenantConf-file, isolationLevel, tenantID
2:  OUTPUT: multApplFunctn
3:  Get tenant identifier from incoming request
4:  tenantConf ← null
5:  share ← true
6:  Select tenantData from tenantConf-file
7:  if tenantData is found then
8:      tenantConf ← tenantData
9:  end if
10: Create defaultApplFunctn
11: multApplFunctn ← defaultApplFunctn
12: if tenantConf is not null then
13:     if isolationLevel = 1 then
14:         Create tenantApplFunctn
15:     else if isolationLevel = 2 then
16:         Authenticate tenantID
17:         Create tenantApplFunctn
18:         Adjust tenantApplFunctn with tenantID
19:     else if isolationLevel = 3 then
20:         Create tenantApplFunctn
21:         share ← false
22:     end if
23:     multApplFunctn ← tenantApplFunctn
24: end if
25: return multApplFunctn
```

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

There are several research work on developing approaches for component specification and retrieval from local and global component repository, which is required in line 5 of Algorithm 2. Such approaches range from syntax-based (traditional) approaches to semantic-based approaches

Algorithm 2 IsolationLevel Algorithm

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

(Seacord, Hissam & Wallnau 1998, Braga, Mattoso & Werner 2001, Braga, Werner & Mattoso 2006). For example, the Multiple-Viewed and Interrelated Component Specification ontology model (MVICS) has been proposed and demonstrated to provide an ontology-based architecture to specify components from a range of perspectives (Li 2012). Further details about how this approach can be used to achieve the required degree of multitenancy isolation plus a case study application can be seen in (Ochei, Petrovski & Bass 2016a).

5.2.3 Validating the Implementation of Multitenancy Isolation

Our approach (i.e., COMITRE) for implementing multitenancy isolation is validated both in theory and in practice. Each application of the approach to a particular multitenancy pattern will result in a differently looking behaviour of the component that is being shared among the different tenants, but all applications of the approach will share a common set of desired properties.

Each multitenancy pattern was validated in theory by following the implementation proposed by Fehling et al (Fehling et al. 2014):

(i) A careful analysis was carried out on the sketch of the architecture proposed for the three multitenancy patterns, the description of the patterns and their behaviour after implementation.

(ii) Our implementation was systematically cross-checked against other implementations of multitenancy architectures and also examined carefully to ensure that our implementation is compliant

with how tenants access a multitenant component.

From an implementation standpoint, Fehling et al's (Fehling et al. 2014) explanation of row-based isolation (i.e., tenants with different tenant-Id's sharing the same database and table) and table-based isolation (i.e., tenants sharing the same database, but having different tables) reflects the shared component and tenant-isolated component respectively. This implementation is similar to other well-known implementations of multi-tenant (data) architecture (MSDN 2016, Wang et al. 2008).

The practicality of our approach has been demonstrated by applying it to implement varying degrees of multitenancy isolation on three GSD tools, namely, Hudson (used for continuous integration), File System Plugin (used for version control) and bugzilla (used for bug tracking). Experimental results confirm that the approach is a reasonable representation of how tenants interact with a multitenant application. Experts and researchers in the field of cloud deployment patterns and Global Software Development have confirmed that the implementation of multitenancy isolation together with the output represents the behaviour of tenants interacting with a shared functionality/component of a cloud-hosted service.

5.2.4 Scenarios for Illustrating Multitenancy Isolation

Our implementation of multitenancy isolation captures isolation both at the process level and at the data levels of a cloud-hosted application. This was achieved by introducing a process and data-handling component to the GSD tool so that the processes and data of different tenants are handled in an isolated fashion. For multitenancy isolation at the process level, the component that is being shared is a lock object, while for isolation at the data level, the component that is being shared is the database. Figure 5.2 and 5.3 captures the architecture of both implementations.

In addition to capturing isolation at the process and data levels, our implementation also captures two types of scenario which are consistent with the way tenants (e.g., developers) interact with GSD tools and supporting processes. These scenarios are summarised below:
(i) Variation in request arrival rate: This scenario represents a case where there is variation in the frequency with which code changes are committed to the source code to trigger a build process. Simulating this behaviour in JMeter simply entails adding the Gaussian Random Timer to the Samplers. Also, the Synchronous Timer was added to the Samplers and the ramp-up period was reduced by one-tenth so that all the requests are sent ten times faster. The scenario is similar to the

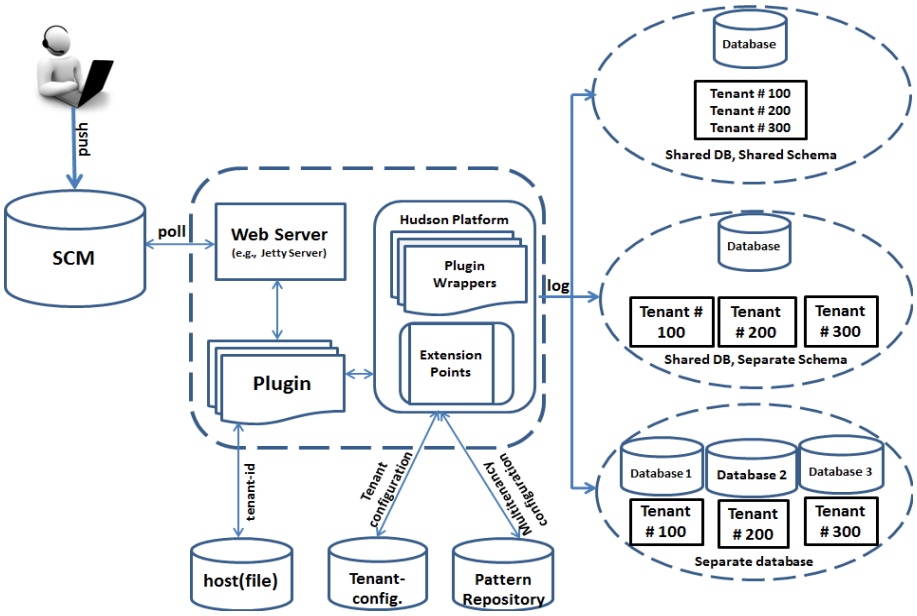

Figure 5.2: Architecture of Multitenancy Isolation at the Data Level.

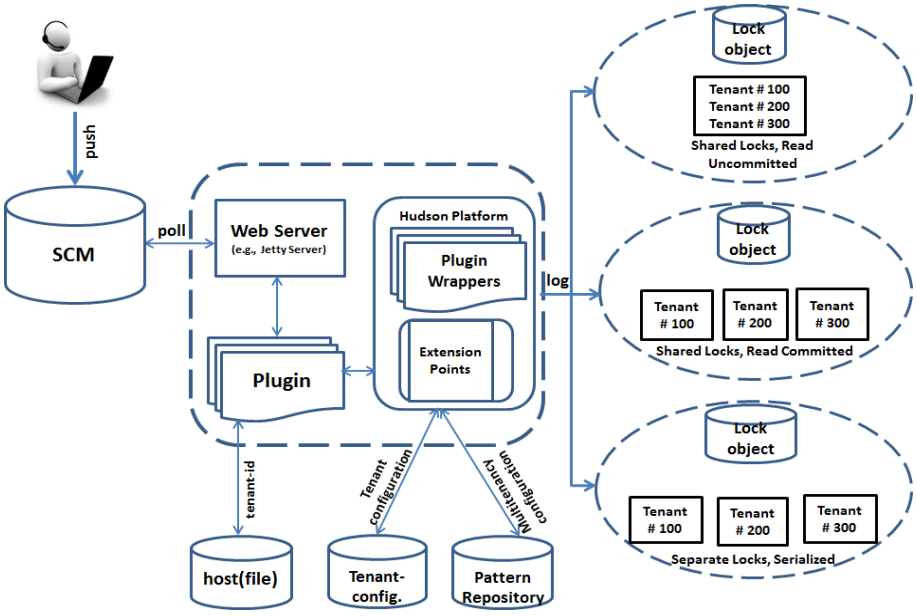

Figure 5.3: Architecture of Multitenancy Isolation at the Process Level.

unpredictable (i.e., sudden increase) workload (Fehling et al. 2014) and aggressive load (Walraven et al. 2012).

(ii) Enabling Locking on processes and data: This scenario illustrates a case where a tenant that first accesses an application component or process locks (or blocks) it from other tenants until the transaction commits. This behaviour was simulated in JMeter by setting the transaction isolation

level to TRANSACTION-SERIALIZABLE for the JDBC request.

5.3 Case Study 1 - Continuous Integration

This case study is based on continuous integration process using Hudson to show how the varying degrees of multitenancy isolation affect the performance and resource consumption of tenants.

5.3.1 Implementing Multitenancy Isolation in Hudson

Hudson is a continuous integration server, written in Java for deployment in a cross-platform environment. It has a rich set of plugins making it easy to integrate with other software tools (Hudson 2017). Large organisations such as Apple and Oracle use Hudson for setting up production deployments and automating the management of cloud-based infrastructure (Moser & O'Brien 2016). The main scenario of interest to us is capturing the isolation of a tenant's data and process during automated build verification and testing, an essential development practice when using a continuous integration system.

Multitenancy isolation was implemented by modifying Hudson using the Hudson's Files-Found-Trigger plugin, which polls one or more directories and starts a build if certain files are found within those directories (Hudson 2016a). This involved introducing a Java class into the plugin that accepts a filename as an argument. During execution, the plugin is loaded into a separate class loader to avoid conflict with Hudson's core functionality. Again, during the build process, data is logged into a database every time a change is detected in the file. To simulate locking, the concept of database isolation level is introduced. This concept is used to control the degree of locking that occurs when selecting or updating data in a database. The database component of the application was set to the highest isolation level: SERIALIZABLE level, to evaluate the impact of the lock duration when locks on data are held until the transaction completes (Oracle 2017).

5.3.2 Experimental Procedure for Case study 1

A file is pushed to a Hudson repository to trigger a build process that executes an Apache JMeter test plan configured for each tenant. Each VM instance is installed with a SAR tool (from Red Hat *sysstat* package) and a Linux du command to monitor and collect system activity information.

Every tenant executes its JMeter test plan which represents the different configurations of the multitenancy patterns.

All the tenants simultaneously send requests to Hudson. To measure the effect of tenant isolation, a tenant that experiences intense or aggressive workload is introduced. All other tenants experience the same normal load which is set to just below the maximum capacity of the system determined separately through repeated test runs. For each test run, the same number of requests are sent by all the tenants except the one that is experiencing a large intense and aggressive load. This means that the total number of requests for each run is spread over the different tenants.

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

Ten iterations were performed for each run and the values reported by JMeter were used as a measure for response times, throughput and error%. For system activity, the average CPU, memory, disk I/O and system load usage at a one-second interval was reported. The initial setup values for the experiment are presented in Table 3.2. With this setup, it means that for each run the tenant experiencing high load receives twice the number of requests received by each of the other tenants, and the requests are sent ten times faster to simulate an aggressive load.

5.3.3 Results for Case Study 1

The results of the case study are analysed based on the results of the paired sample t-test shown in Table 5.1, and supplemented with information from the plots of Estimated Marginal Means of Change(EMMC)[1]. The keys used in constructing the paired sample t-test table are explained in Section 3.2.2. Figure 5.5 to Figure 5.17 show the plots of the estimated marginal means of change[2].

(1) Response times and Error%: Table 5.1 shows that the response times and error% of tenants did not change significantly except for the dedicated component. The plot of the EMMC revealed that the magnitude of change for response times showed a much larger change for the dedicated

[1]The word ”Change in the acronym EMMC refers to the dependent variable used for paired sample t-test.
[2]The symbol CS1 used in Figure 5.4-5.17 stands for Case Study 1 under scenario 1.

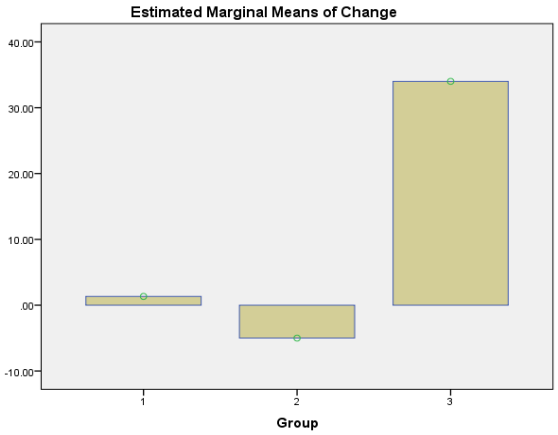

Figure 5.4: Response Time Changes [CS1]

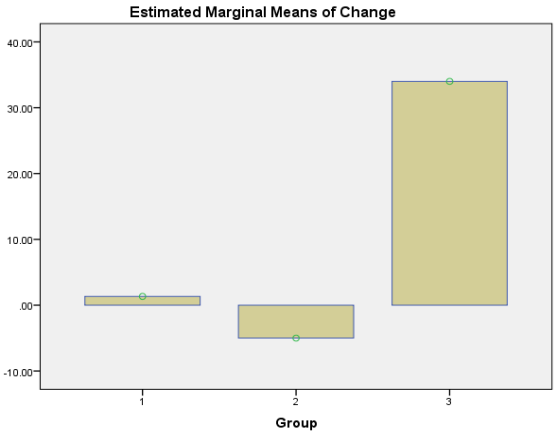

Figure 5.5: Response Time Changes [CS1]

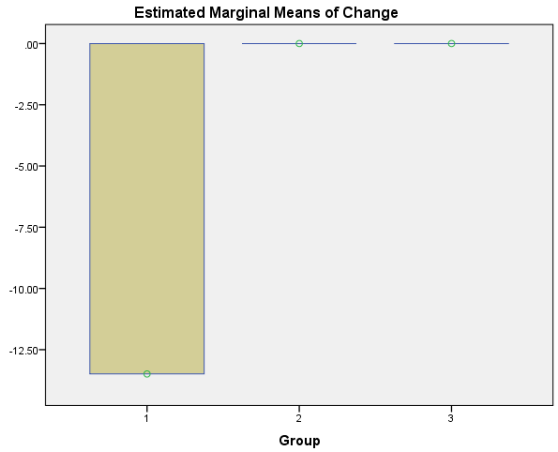

Figure 5.6: Changes in Error% [CS1]

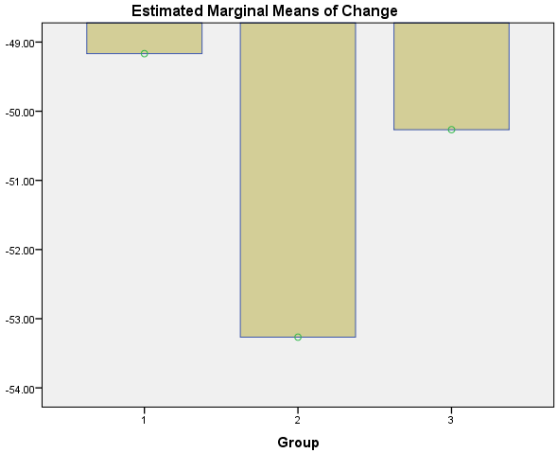

Figure 5.7: Throughput Changes [CS1]

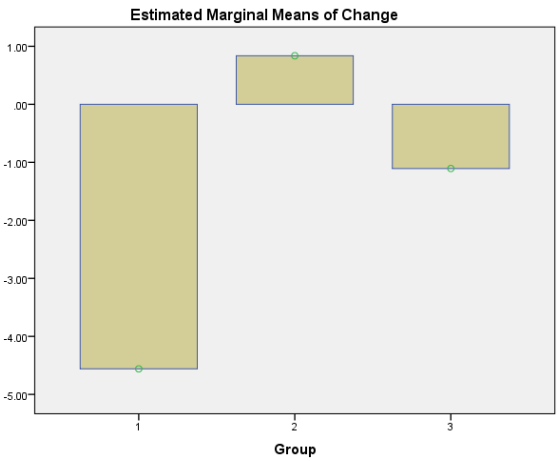

Figure 5.8: Changes in CPU [CS1]

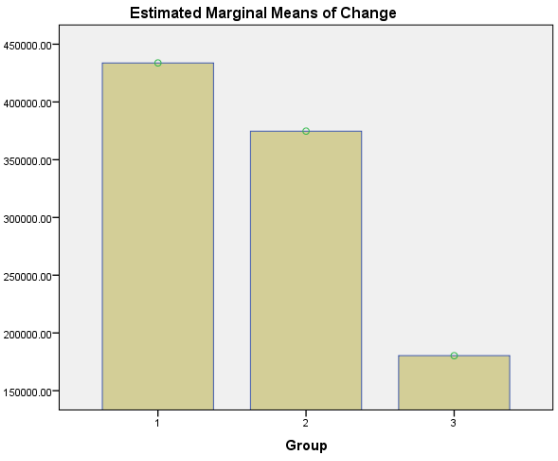

Figure 5.9: Changes in Memory [CS1]

Table 5.1: Paired Sample Test Analysis for Case Study 1

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

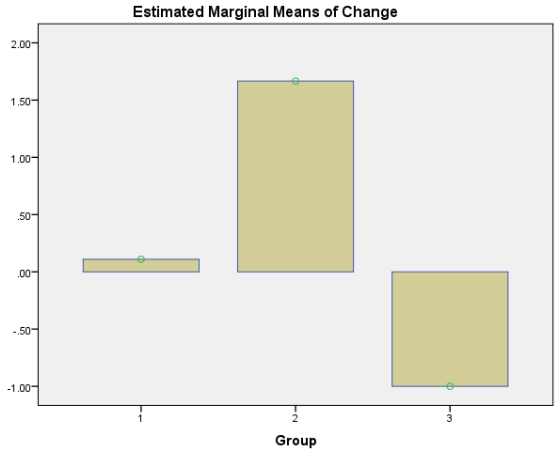

Figure 5.10: Changes in Disk I/O [CS1]

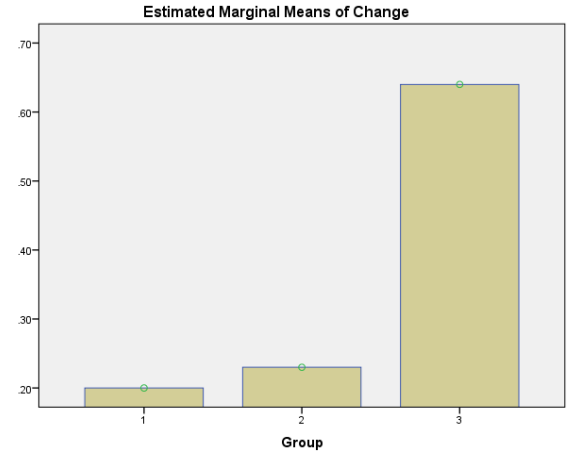

Figure 5.11: System Load [CS1]

component. This is due to the overhead incurred because of opening multiple connections to the database each time a JDBC request is made to a different database. For error%, the magnitude of change was larger for tenants deployed based on the shared component than for other patterns. A possible explanation for this is that there is resource contention since multiple connections are opened while sending requests that log all the data into the same component (i.e., database table) that is being shared. Overall, this causes delay in completion times thereby producing a negative effect on error%.

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

(3) CPU and System Load: The paired sample test showed that CPU consumption of tenants did not change significantly for most patterns except for the *shared component*. Therefore, once a reasonable CPU size (e.g., multiple CPUs or a multi-core CPU) is used, there should be no problem in performing builds. Builders are not known to consume much CPU. For example, Hudson does not consume much CPU; a build process can even be setup to run in the background without interfering with other processes (Moser & O'Brien 2016).

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

(4) Memory: The paired sample test result showed that there was a significant change in memory consumption for all three patterns. Complex and difficult builds are those that are composed of a vast number of modular components including different frameworks, components developed by different teams or vendors, and open source libraries (Electric-Cloud 2016). Compilers and builders consume a lot of memory especially if the build is difficult and complex (Moser & O'Brien 2016). In a large project, it is expected that multiple builds will interact with multiple components to create several dependencies and supported behaviour with each other thereby making builds difficult and complex.

(5) Disk I/O: Compilers and builders are known to consume disk I/O especially for I/O intensive builds (Moser & O'Brien 2016). The results show that only the shared component showed no significant change in disk I/O usage. This is understandable because multiple transactions are channelled to the same component which would either be delayed or blocked because of sharing the components. Further analysis of the plot of the EMMC confirmed that the magnitude of change for the shared component was the least, and therefore is recommended for builds that particularly involve intensive I/O activity especially when locking is enabled.

5.4 Case Study 2 - Version Control

This case study is based on a version control process using File System SCM Plugin integrated into Hudson to show how the varying degrees of multitenancy isolation affect the performance and resource consumption of tenants.

5.4.1 Implementing Multitenancy Isolation in File System SCM Plugin

The File System SCM plugin was used in the case study to illustrate the version control process because our interest was in simulating the process on a local development machine. Specifically, our aim was to point a build configuration to the locally checked out code and modified files on a shared repository residing on a private cloud. Filesystem SCM plugin can be used to simulate the file system as a source control management (SCM) system by detecting changes such as the file system's last modified date (Hudson 2016a). This plugin can be integrated into several GSD tools: continuous integration systems (e.g., Hudson), version control systems (e.g., Perforce, Git) and an error/issue tracking system (e.g., JIRA).

The File System SCM plugin was integrated into Hudson to represent a scenario where a code file is checked into a shared repository for Hudson to build. Multitenancy isolation was then implemented by modifying this plugin within Hudson. This involved introducing a Java class into the plugin that accepts a file path and the type of file(s) that should be included when checking out from the repository into Hudson workspace. During execution, the plugin is loaded into a separate class loader to avoid conflict with Hudson's core functionality.

5.4.2 Experimental Procedure for Case Study 2

A typical version control process involves a combination of continuous integration (i.e., building a code file), checkouts (i.e., file download), check-ins (i.e., file upload), and updating and synchronising files with the latest version from the repository. The experimental procedure translates into the following steps:

1. The first step is to put a new file in the repository for the first time. To achieve this, the HTTP request sampler in JMeter was used to send requests to Hudson to trigger a build. Within Hudson, the "Execute Shell" feature was used to execute a shell script. This shell script simply selects the initial contents of a MySQL database (i.e., used here to represent a shared data handling compo-

nent) and then outputs it into two separate files (referred to as *file1* and *file2*). The first file (i.e., *file1*) represents the local working copy and the second file (i.e., *file2*) represent the main copy.
2. The second step is to *checkout* the copy of the new file to the local machine. To implement this in JMeter, the FTP request sampler was used and the get (RETR) selected to download the file from the repository. In effect, this action downloads *file1* from the repository into a local machine and saves it as *file3*.
3.The third step involves making changes to the file by inserting records into the Mysql database and then outputting the latest content to the local working copy. This is simulated by using the BeanShell Sampler in JMeter to invoke a custom Java class. This Java class is specifically written to insert records into the MySQL database, and then to update *file3* with the latest content of the database.
4.The last step is to *checkin file3* back into the repository with a timestamp message ("Row added at 2015-01-01-00.00.01"). To implement this in JMeter, the FTP request sampler is used and then the put (STOR) is selected to upload the file to the repository and append the content to *file2*.

Each tenant request is treated as a transaction composed of the three types of request: HTTP request, FTP request, and File I/O operation. The JMeter Transaction controller is introduced to take the aggregate measurement of all the requests involved in the end-to-end action sequence of the scenario. The initial setup values for the experiment are presented in Table 3.2. With this setup, it means that in each run the tenant experiencing high load (i.e., tenant 1) receives twice the number of requests received by each of the other tenants, and the requests are sent ten times faster to simulate an aggressive load.

Ten iterations were performed for each run and the values reported by JMeter were used as a measure for response times, throughput and error%. The error% is computed as the percentage of the total number of requests (i.e., in the end-to-end sequence of version control process) whose response time is unacceptably slow and above which the request is considered a failure. Statistically, this translates to a response time greater than the upper bound of the 95% confidence interval of the average response time of all requests. For system activity, the average CPU, memory, disk I/O and system load usage at a one-second interval was reported.

5.4.3 Results for Case Study 2

The results of the case study are analysed based on the paired sample t-test supplemented with information from the plots of Estimated Marginal Means of Change(EMMC).

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

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

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

Table 5.2 shows that system load was nearly constant with no chance of variability, and so this means that system load did not influence any of the patterns. Therefore, with a reasonably high-speed network connection and CPU size, there should be no problem with system load when sending data across a shared repository residing in a company's LAN or VPN.

Table 5.2: Paired Sample Test Analysis for Case Study 2

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

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

5.5 Case Study 3 - Bug Tracking with Bugzilla

This case study is based on bug/issue tracking process using Bugzilla to show how the varying degrees of multitenancy isolation affect the performance and resource consumption of tenants.

5.5.1 Implementing Multitenancy Isolation in Bugzilla

Bugzilla was modified using the recommended *Bugzilla Extension* mechanism. Extensions can be used to modify either the source code or user interface of Bugzilla, which can then be distributed to other users and re-used in later versions of Bugzilla. Bugzilla maintains a list of *hooks* which represent areas in Bugzilla that an extension can hook into, thereby allowing the extension to perform any required action during that point in Bugzilla's extension (Bugzilla 2016).

For our experiments, a special extension was written and then "'hooked"' into Bugzilla using the hook named *install_before_final_checks*. This hook allows the execution of custom code before the final checks are done in *checksetup.pl*, and so the COMITRE algorithm was implemented in this hook.

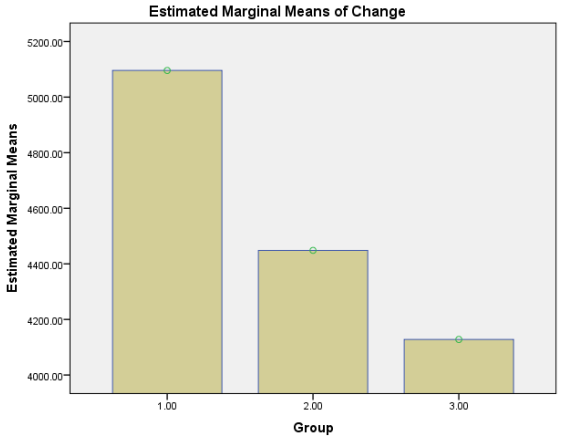

Figure 5.12: Changes in Response Times [CS2]

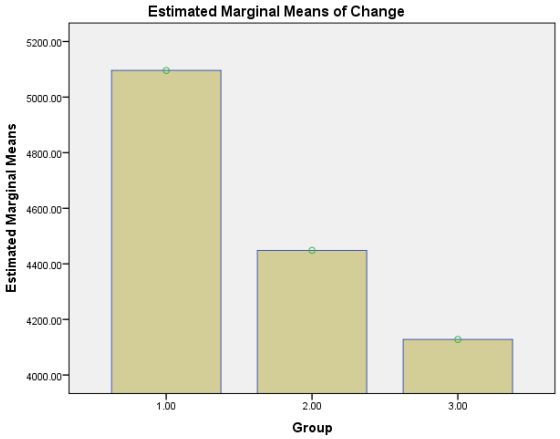

Figure 5.13: Changes in Response Times [CS2]

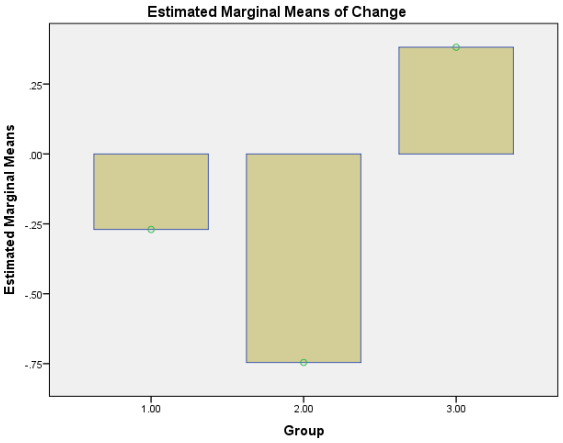

Figure 5.14: Changes in Error% [CS2]

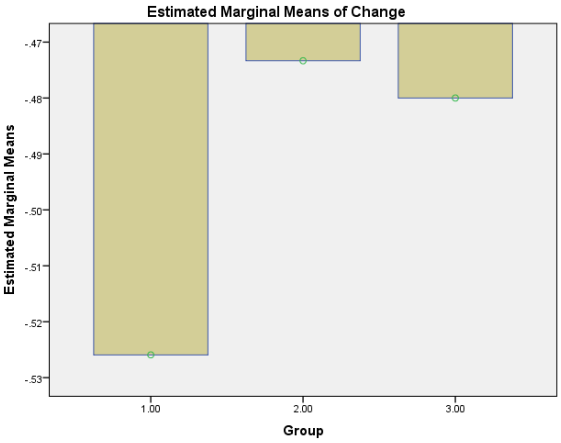

Figure 5.15: Changes in Throughput [CS2]

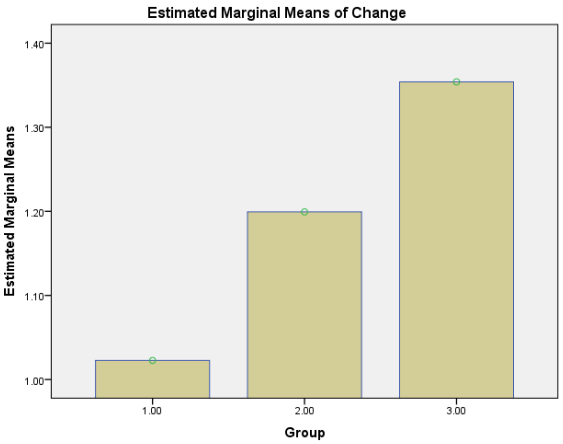

Figure 5.16: Changes in CPU [CS2]

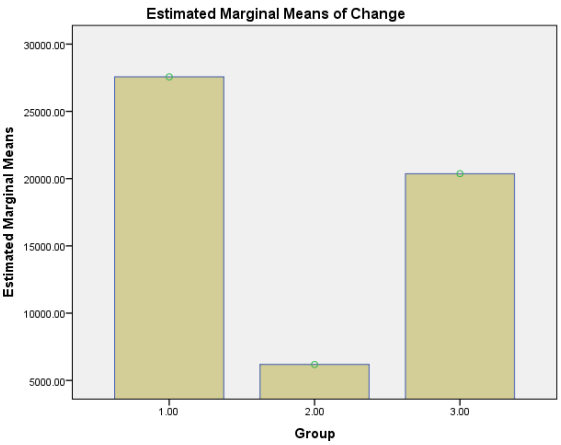

Figure 5.17: Changes in Memory [CS2]

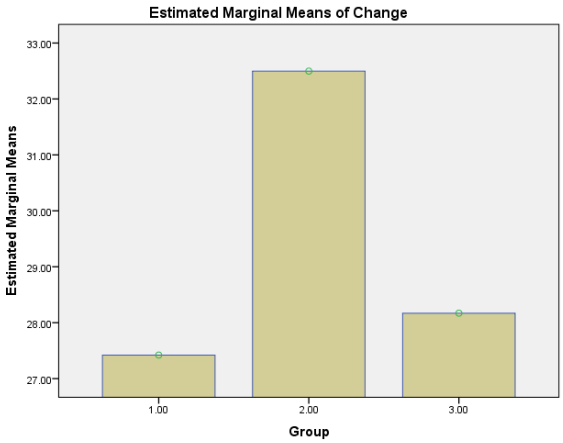

Figure 5.18: Changes in Disk I/O [CS2]

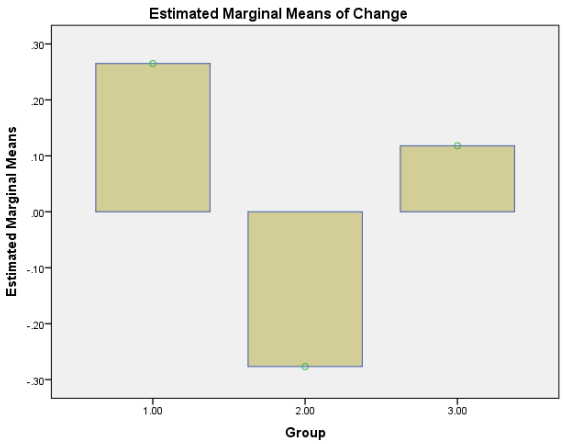

Figure 5.19: Changes in System Load [CS2]

5.5.2 Experimental Procedure for Case Study 3

The two main processes of interest to be captured in Bugzilla are: (i) creating a bug, and (ii) adding an attachment specific to a bug. Creating a simple bug with attachment in Bugzilla requires access to three main tables: bugs, attachments and attach_data. Most bug tracking systems like JIRA and Bugzilla use a database to store bugs/issues created by users during the software development process. Simulating this in Apache JMeter entails using the JMeter BeanShell sampler to invoke two separate custom Java classes that run a query that: (i) inserts multiple bugs with large attachments into the Bugzilla database concurrently; and (ii) sets the database transaction isolation level to SERIALIZABLE (i.e., the highest isolation level) during bug creation with attachment.

Our experimental procedure captures a scenario that involves variation in the frequency with which large instant bugs are submitted concurrently to a database when support for locking is enabled. Locking, in this case, is used to prevent conflicts between multiple tenants attempting to access a bug database. This type of scenario is very important in *distributed bug tracking* in which some bug trackers such as Fossil and Veracity, are either integrated with or designed to use distributed version control systems or continuous integration systems, thus allowing bugs to be generated automatically and added to the database at varying frequencies (Corbet 2009). To measure the effect of isolation between tenants, one of the tenants is configured to simulate large instance loads as explained in section 3.2.2. The experimental setting is also presented in section 3.2.2.

5.5.3 Results for Case Study 3

This section presents a summary of the experimental results for case study 3. The results for the paired sample t-test are summarised in Table 5.3, while the plots of the estimated marginal means of change are shown in Figures 5.20 - 5.27

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

(2) Throughput: The paired sample test result showed that there was no significant change in throughput for most of the patterns unlike two previous case studies. This result is similar to that of the two previous case studies where throughput was relatively stable. The implication of this is that if the component being shared is a database, then throughput should not be expected to change drastically. Based on the plot of the EMMC, the shared component would be recommended when bugs are stored in a database with locking enabled.

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

(4) Memory and Disk I/O: The paired sample test for both the memory and disk I/O showed a highly significant difference from pretest to post-test both for memory and disk I/O. For memory, the plot of the EMMC similarly showed that the dedicated component had the highest significant

Table 5.3: Paired Sample Test Analysis for Case Study 3

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

change compared to the other patterns. This is possibly due to running Bugzilla under mod_perl environment, and so using a dedicated component would not be a good option for optimising system resources. It is well known that running Bugzilla in mod_perl environment consumes a lot of RAM (Bugzilla 2016). The significant change in disk I/O consumption is due to the intense frequency of read/write activities in the database. For disk I/O consumption, having enough storage space would be required, especially if a large volume of bugs with attachments is expected. If a large number of users are expected, then applying disk space saving measures such as purging unwanted error or log files regularly could reduce disk I/O consumption and improve the chance of having a higher degree of isolation.

5.6 Chapter Summary

This chapter first presented a novel approach, COMITRE (Component-based approach to Multitenancy Isolation through Request Re-routing) for implementing multitenancy isolation on cloud-hosted services. After that, the approach is applied to implement multitenancy and also evaluate the varying degrees of isolation between tenants enabled by multitenancy patterns in three separate case studies involving GSD processes. The three case studies are - continuous integration with Hudson, version control with File System SCM Plugin and bug tracking with Bugzilla. Three multitenancy patterns (i.e., shared component, tenant-isolated component and dedicated component) were implemented by modifying the GSD tool and deploying it as a Virtual Machine (VM) instance to the UEC (Ubuntu Enterprise Cloud) private cloud.

The study to a large extent confirms that when multiple tenants are accessing a cloud-hosted service deployed based the different multitenancy patterns, the shared component provides a lower degree of isolation between tenants, while the dedicated component provides a higher degree of isolation when one of the tenants experiences a high workload. Case study 1 revealed that when code files are checked into a shared repository at a low frequency to trigger a build process,

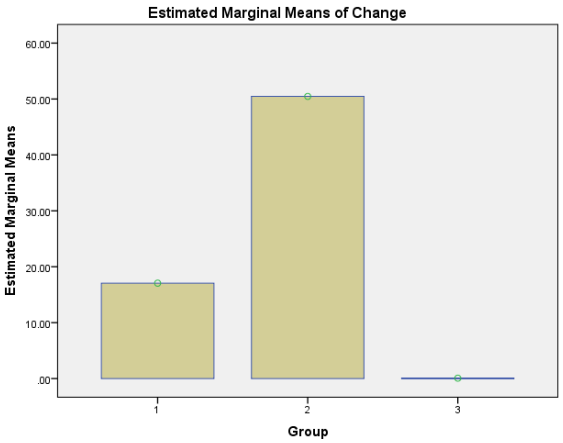

Figure 5.20: Changes in Response Times [CS3]

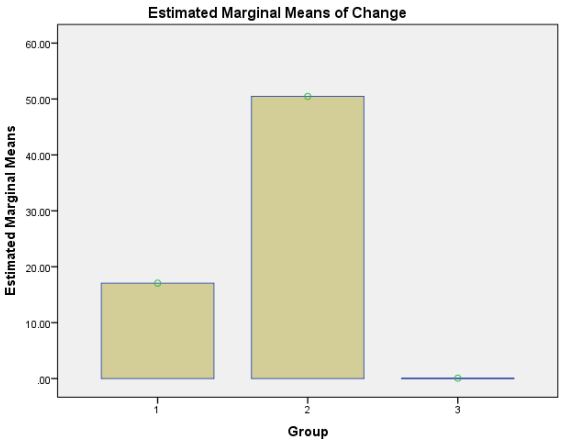

Figure 5.21: Changes in Response Times [CS3]

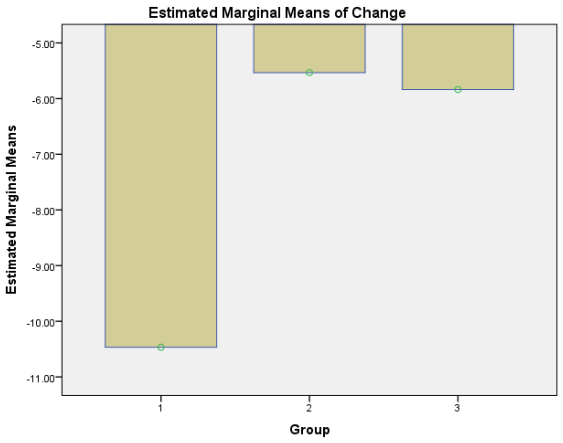

Figure 5.22: Changes in Error% [CS3]

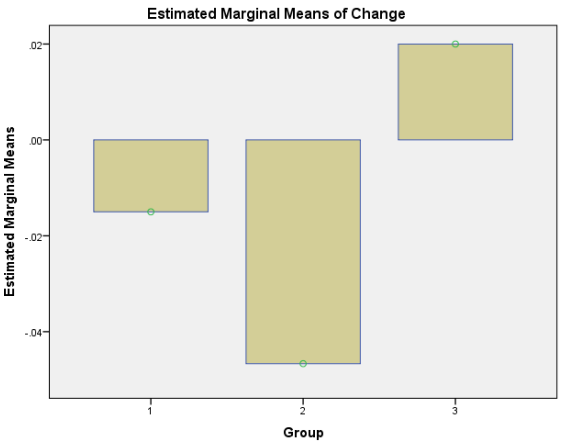

Figure 5.23: Changes in Throughput [CS3]

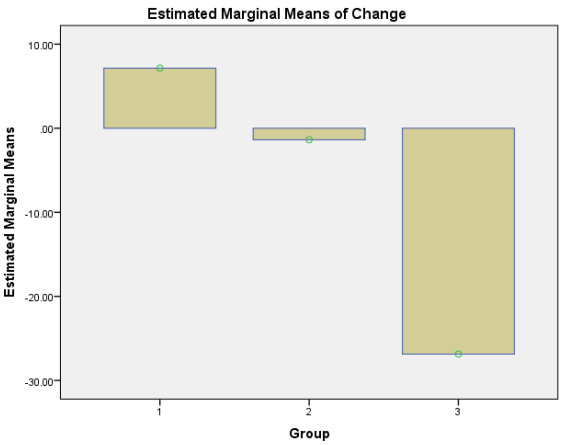

Figure 5.24: Changes in CPU [CS3]

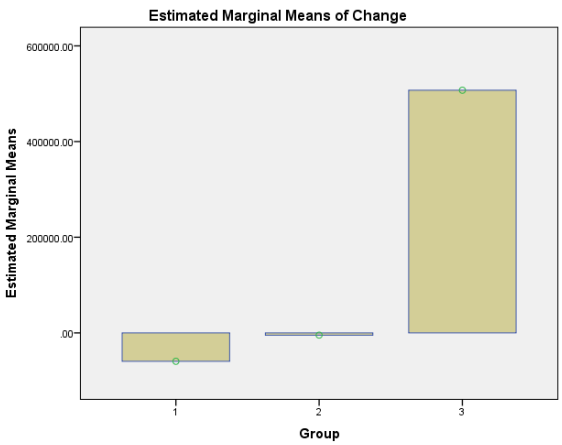

Figure 5.25: Changes in Memory [CS3]

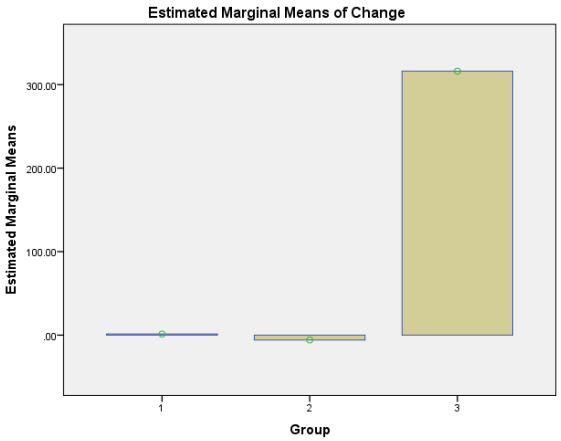

Figure 5.26: Changes in Disk I/O [CS3]

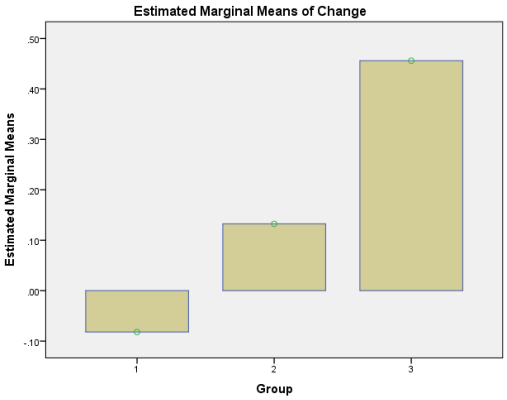

Figure 5.27: Changes in System Load [CS3]

then a high degree of isolation (regarding response times) is expected both for the tenant-isolated component and the dedicated component. For case study 2, the error% (i.e., the number of requests with unacceptably slow response times) was negatively impacted especially when committing a large number of files to a shared repository that interacts directly with the native OS filesystem. Case study 3 revealed that for transactions on bug database where support for locking is enabled, performance isolation between tenants (e.g., regarding response time) could be improved with a dedicated component while resource consumption (e.g., CPU and memory) could be improved with the shared component.

The next chapter will be devoted to showing unexpected and varying results across the three case studies and well as results that are common across the case studies and which can be generalised. In addition, the next chapter will also present (i) an explanatory framework and new insights on multitenancy isolation, and (ii) the trade-offs for consideration in order to achieve the required degree of multitenancy isolation.

Chapter 6

Degrees of Multitenancy Isolation: Synthesis of three Case studies

6.1 Introduction

In chapter three, three separate case studies were presented that applied COMITRE to implement and evaluate the varying degrees of multitenancy isolation in cloud-hosted GSD processes: (i) continuous integration with Hudson; (ii) version control with subversion; and (iii) bug tracking with Bugzilla. The three case studies were carried out because of realisation of the fact that it is usually not possible to adequately investigate all aspects of a phenomenon in one case study, hence the need for more than one case study. As pointed out by Cruzes and Dyba (2011), no matter how well designed and executed, empirical findings from single studies are limited in the extent to which they may be generalised (Cruzes & Dybå 2011).

The aim of this chapter is, therefore, to extend the overall evidence beyond a single case, by synthesising the findings of the three primary case studies that empirically evaluated the varying degrees of multitenancy isolation on cloud-hosted GSD processes. This synthesis will provide a novel explanatory framework and new insights into varying degrees of multitenancy isolation.

The rest of the chapter is organised as follows - Section 6.2 discusses the approach used in synthesising the findings of the three case studies. Section 6.3 presents an explanatory framework and new insights into varying degrees of multitenancy isolation. Section 6.4 discusses the threats to the validity of the case study synthesis. Section 6.5 summarises the chapter.

6.2 Synthesis of Case Studies Findings

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

6.2.1 Cross-case Analysis

In a cross-case analysis, evidence from each primary study is summarised and coded under broad thematic headings, and then summarised within themes across studies with a brief citation of primary evidence (Cruzes et al. 2015). This paper adopts Miles and Huberman's approach for conducting the cross-case analysis. The approach consists of three mains steps: data reduction, data display, and conclusion drawing and verification (Huberman & Miles 2002, Cruzes et al. 2015). These steps were applied in an iterative manner during the analysis to reach the conclusion.

Data Reduction

This mainly involves the identification of items of evidence in the primary studies (Cruzes et al. 2015). In our study, much of the data reduction process was already done in the primary case studies. For each case study the following details were presented: (i) the paired test sample test, (ii) plots of the estimated marginal means of change, (iii) discussion of the findings and recommendations for achieving the required degree of isolation between tenants.

The experimental results in Table 5.1, Table 5.2, and Table 5.3 show the paired sample test result for case study 1, case study 2, and case study 3, respectively. The plots of the estimated marginal means of change (EMMC) are shown in Figure 5.4 - 5.11 for case study 1, Figure 5.12 - 5.19 for case study 2, and Figure 5.20 - 5.27 for case study 3. In addition to this data, a table showing the characteristics of the three cases studies is presented (see Table 6.1).

Data display

This step involves organising and assembling information that allows the drawing of conclusions using tools such as meta-matrices/tables and cause and relationship graphs. The data display steps will be tackled from two approaches to cross-case comparisons: variable -orientated and case-oriented (Ragin 2004).

Table 6.1: Characteristics of the Case Studies

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

Table 6.2: Comparison of different aspects in which the Cases vary

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

(A) Variable oriented approach: This approach focuses on the variables to explain why the cases vary. This study focused on factors such as performance and resource consumption that are known to affect isolation between tenants. The data derived at this stage is a table (see Table 6.2) showing the factors in which the cases vary, to explain why there is variation in the degree of multitenancy isolation across the cases. It is assumed that factors such as performance, resource utilisation, that are known to affect isolation between tenants were already used to evaluate the three cases independently. These factors are captured in the seven metrics used to evaluate the three cases: response times, error%, throughput, CPU, Memory, disk I/O, and system load. Knowing the various aspects in which the cases vary would enable us to explain the variation in the degrees of multitenancy isolation for different GSD processes. The synthesis identified five aspects in which the cases vary: size of data generated, the resource consumption of the GSD process, client's latency and bandwidth, supporting task performed, and error messages due to sensitivity to workload changes. These aspects are summarised below.

1. *Size of Data Generated:* One of the most important factors that account for the variation in the degree of multitenancy isolation is the fact that some GSD tools generate more data than others. For example, several of the problems that occur in version control relate to the fact that version control systems usually create additional copies of files on the repository, especially the ones that use the native operating system (OS) file system directly. This adversely affects performance because these files occupy more disk space than they actually use, and the OS spends a lot of time seeking across many files on the disk.

2. *Effect of GSD process on Resource Consumption:* Another important factor that accounts for the variation in the degree of multitenancy isolation is the effect of the particular GSD process on the cloud infrastructure resources. Some GSD processes consume more of a particular resource than others, and so this is bound to affect the degree of multitenancy isolation required by tenants. As shown in the experiments, continuous integration showed no significant change in CPU consumption when used with most of the patterns compared to version control and bug tracking. Under normal conditions, continuous integration systems being compilers consume huge amounts of memory and disk I/O during high workload. Based on our results, the dedicated component was recommended for performing builds when there is a sudden increase in workload.

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

4. *Type of GSD Process and Supporting operations:* There are several conditions associated with a GSD process that can result in different or varying degrees of isolation. Examples of such conditions include (i) running long builds, (ii) running a large number of builds, (ii) running complex and difficult builds, and (iv) enabling file locking. For example, a complex and difficult build involving lots of inter-dependencies will consume more resources (e.g., CPU) than an ordinary check-out process in a version control system.

5. *Error Messages and Sensitivity of workload Changes:* The cases also vary in terms of their sensitivity to workload changes as manifested in the nature and type of error messages produced by the different GSD processes during the implementation of multitenancy isolation. The experimental results show that when a tenant experiences a high workload, different kinds of error messages were generated depending on the GSD process. The error messages are summarised as follows: for continuous integration, the most common type of error was that of insufficient system resource (e.g., memory); for version control, the common error was that of directory and file permissions; and for bug tracking the common error was database-related errors (e.g., exceeding maximum number of allowed queries, connections and packets etc.)

(i) Case-oriented approach: This approach focuses on the case itself instead of the variables to explain in what ways the cases are alike. The data derived at this stage is a table (see Table 6.3) showing the factors that are alike across the cases, and which appear to lead to similar outcomes when evaluating the varying degrees of multitenancy isolation in cloud-hosted GSD tools. By knowing the aspects in which the cases are alike it is then possible to generalise our findings, for

Table 6.3: Comparison of different aspects in which the Cases are alike

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

example, to identify factors that appear to lead to high (or low) degree of multitenancy isolation with a corresponding effect on resource consumption. The synthesis identified five aspects in which the cases are alike: a strategy for reducing disk space, locking, low consumption of some system resources, use of plugin architecture for extending the GSD tool, and aspects of multitenancy isolation. The various aspects in which the cases are alike are summarised as follows.

1. *Strategy for Reducing Disk Space:* An interesting feature of all the GSD tools is that they have strategies for reducing disk space because of the possibility of the GSD tool generating a large volume of data due to the size, the number of artefacts and the number of users that may be involved in the project. For instance, CI systems can be configured to discard old builds. Version control systems can use *delification* (i.e., a process for transferring differences between versions instead of complete copies) and *packing* to manage disk space. For a bug tracking system, the error and log files can be purged from the database regularly.

2. *Locking Process:* All the GSD tools implement some form of locking whether at the database level or filesystem level. For example, locking is used internally in version control systems to prevent clashes between multiple tenants operating on the same working copy (Collins-Sussman et al. 2004). In Bugzilla, locking is used to prevent conflicts between multiple tenants or programs trying to access the Bugzilla database (Bugzilla 2016). In continuous integration, locking can be used to block builds with either upstream or downstream dependency from starting if an upstream/downstream project is in the middle of a build or the build queue (Moser & O'Brien 2016). When using a version control system that im-

plements locking, fetching large data remotely and finalising a commit operation can lead to unacceptably slow response times (and can even cause tenants' request to time out), and so having the repository together with the working copy located on your machine is beneficial. The results of case study two recommended the shared component to address the trade-off between resource utilisation and the speed of accessing or completing a version control process (e.g., checking out files from a repository).

3. *Low Resource Consumption:* Most GSD tools do not consume much system resources like CPU and memory but can benefit from optimisations when there is a sudden change in workload. For continuous integration, memory and disk I/O will be mostly affected. For Bugzilla, it will be memory especially if locking and database transactions are enabled. For subversion, disk space and disk I/O are the obvious resources that will be most affected. System load and CPU consumption were generally low, and so using any of the patterns would not make much difference.

4. *Mechanism for customization and Use of Plugin Architecture:* All the GSD tools implement a "plugin architecture" for use in customising, modifying and extending the GSD tool. This means that other programs and components can be easily integrated with it (Ochei, Bass & Petrovski 2015a). For example, Hudson is easily extensible using plugins. A series of extension points are provided in Hudson that allows developers to extend its functionality (Hudson 2016b). These extension points are where the GDS tools can be customised to support multitenancy isolation.

5. *Aspects of Isolation:* The results generally showed that performance-related parameters such as response time, %error and throughput had changed significantly for shared pattern compared to system's resources such as CPU, memory, disk I/O and bandwidth. Because of this, the dedicated component is recommended to improve performance related parameters while the shared component was recommended to improve resource utilisation. For example, in version control and bug tracking, the dedicated component is recommended to improve response time while the shared component is recommended to improve utilisation of memory and disk I/O.

Conclusion Drawing

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

Summary of Findings from Statistical Analysis

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

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

2. System load is nearly constant and no variability was found in almost all the case study results. A possible explanation for this is that the configuration of the deployed component, the nature of tasks, and absence of piled-up task queue for a long time being processed resulted in reasonably good throughput. In most cases, if the load average is less than the total number of processors in the system, this suggests that the system is not overloaded and so it is assumed that nothing else influences the load average.

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

4. Throughput changed significantly, and this change was relatively stable for most of the patterns

in all the three case studies, except for case study three where there was no meaningful change. This may be because the system quickly reached peak capacity and so additional requests simply do not add to the throughput. Furthermore, the small private cloud used for the experiments may have contributed to this fairly stable but significant change in throughput.

Summary of Recommended Multitenancy Patterns for Deployment

Table 6.4 shows a summary of the recommended multitenancy patterns for achieving multitenancy isolation between tenants when one of the tenants experiences a high load. These recommended patterns are derived by first checking the paired sample test result and then analysing the plots of the estimated marginal means of change (EMMC) to compare the magnitude of change in each pattern relative to other patterns. The key used in constructing the table is as follows: (i) the symbol "✓"means that the pattern is recommended; (ii) the symbol "**x**" means that the pattern is not recommended; and (iii) the symbol "**-**" implies that there is no difference in effect, and so any of the three patterns can be used.

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

6.2.2 Narrative Synthesis

To further enrich the case study synthesis, the ***narrative synthesis*** was also used. Narrative synthesis is a textual approach to condense and explain the findings from case studies (Cruzes et al. 2015, Miles & Huberman 1994). A condensed summary is provided to (i) explain the effect of performance and resource utilisation on tenants deployed based on different multitenancy

Table 6.4: Recommended Patterns for optimal deployment of components

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

patterns when one of the tenants experiences a sudden change in workload, and (ii) present some recommendations for achieving the required degree of multitenancy isolation.

(1) Response times and Error%: The case studies results showed that response times and error% did not change significantly for the shared component, and so it is recommended for addressing low latency and the bandwidth requirements of tenants. This suggests that a GSD tool should be configured close to the backend storage. For example, CI server (e.g., Hudson) should be configured close to the SCM server when polling a version control repository for changes. The performance of tenants with low bandwidth accessing a version control system can be boosted by minimising the size of the network communications (e.g., reducing file size transferred between shared repositories). When committing large files to a repository residing over a network, delays could arise causing requests to time out (Collins-Sussman et al. 2004). For version control systems, the error% (i.e., requests with unacceptably slow response times) could be negatively impacted when committing a large number of files to a repository that is using a native OS file system (e.g., FSFS). Tenants request could time out while waiting for a response due to delays in finalising a commit operation (Collins-Sussman et al. 2004).

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

(3) CPU and System Load: The case study results show that most GSD tools do not consume much CPU; consumption only slightly increased for some patterns. Therefore, the key in efficient utilisation of CPU while achieving the required degree of isolation lies in avoiding operations that are likely to increase CPU consumption. For continuous integration systems, a build can be run in the background without affecting other resources or processes, but this could increase if builds are difficult and complex (Electric-Cloud 2016). For version control systems, CPU consumption could increase when moving data from one repository into another (e.g., using *svnadmin dump* and *svnadmin* load subcommands in subversion) or switching from a repository that uses a database (e.g., Berkeley DB or MySQL) to a repository that is based on FSFS file format (Bugzilla 2016). Compressing data of large sizes in a bid to improve performance could also consume more CPU. System load was not influenced by any of the patterns, possibly because the number and size of requests did not overload the system to cause any significant change.

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

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

6.3 Explanatory Framework for Degrees of Multitenancy Isolation

This section is used to provide an explanatory framework (in a descriptive form) and new insights into multitenancy isolation. Firstly, this section presents a mapping of different degrees of multitenancy isolation to the GSD processes, the cloud application stack and cloud resources on which

the GSD tools are hosted. Secondly, the trade-offs that should be considered when implementing the required degree of multitenancy isolation are discussed.

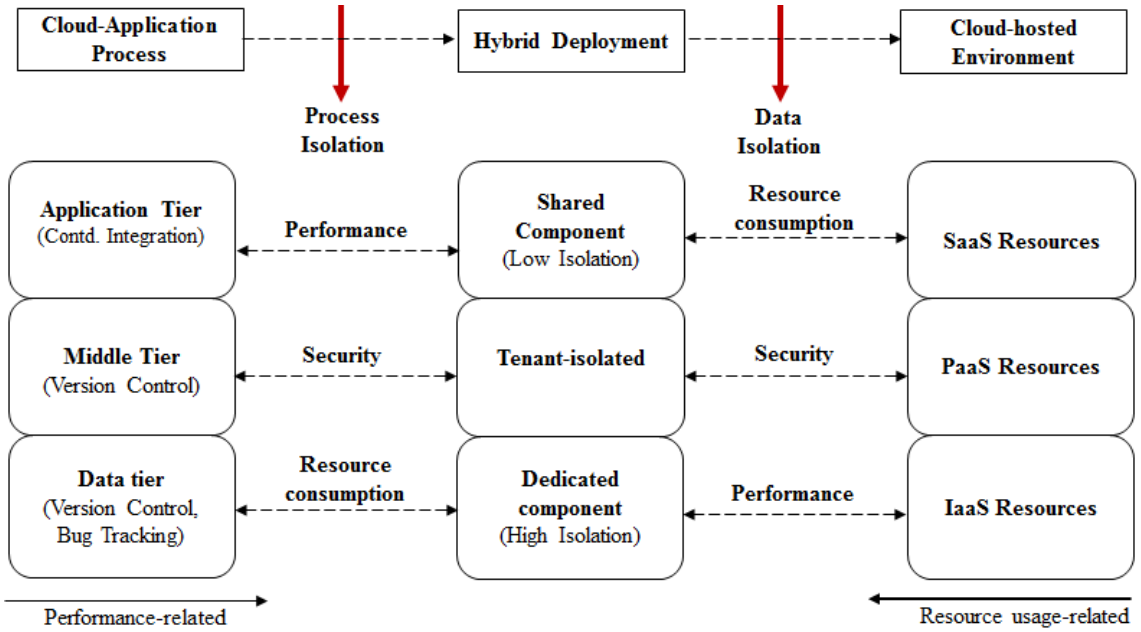

Figure 6.1: Mapping of Degrees of Isolation to Cloud-hosted GSD Process and Resources

6.3.1 Mapping of Multitenancy Isolation to GSD Processes and Resources

Figure 6.1 maps the different degrees of multitenancy isolation to: (i) software processes triggered by the cloud-hosted GSD tools; (ii) cloud application stack; and (iii) cloud resources on which the processes are executed. As shown in Figure 6.1, GSD processes are placed on the left and cloud resources on the right. In this mapping, it is assumed that the ease and flexibility of achieving multitenancy isolation increases vertically, from top to bottom, and horizontally, from left to right.

(1) Mapping Multitenancy Isolation to Layers of a Cloud Stack: The mapping in Figure 6.1, shows that a high degree of isolation can be achieved on the infrastructure layer and vice versa. Therefore, as the required degree of isolation increases, the ability to improve resource consumption reduces when implementing multitenancy isolation. On the other hand, as the required degree of isolation increases, the ability to improve performance increases. This means that it is better to implement resource sharing or efficient resource utilisation using the shared component and reduce performance interference using a dedicated component.

Depending on the layer of the application stack (i.e., application layer, platform layer, or in-

frastructure layer), multitenancy isolation may be realised differently with associated implications. Assuming the component being shared is a database, implementing the shared component on a bug tracking system at the SaaS layer implies allowing multiple tenants to share a single instance of the bug database. This ensures efficient sharing of cloud resources, but isolation is either very low or not guaranteed at all due to possible performance interference. Implementing a dedicated component at the IaaS layer would require installing the bug database for each tenant on its own instance of virtual hardware. This guarantees a high degree of isolation but with limitations of high runtime cost and reduction in the number of tenants that can be served.

(2) Mapping Multitenancy Isolation to GSD Processes: Multitenancy isolation can be implemented at different levels of a cloud application stack depending on the type of component or process being shared. Due to the way in which software processes interact with an operating system, files system and systems resources, the GSD processes can be mapped to varying degrees of multitenancy isolation, and hence the application stack. Figure 6.1, show a mapping of the three GSD processes to different levels of the cloud application stack. Notice that the GSD processes are placed in the following order from top to bottom: continuous integration, version control and bug tracking. In the mapping, the continuous integration process is placed in the top to fit into a situation where it is deployed to multiple users using the SaaS model. However, in a hybrid scenario, it is possible to place continuous integration on the middle tier of the cloud application stack (e.g., based on the PaaS deployment model). This scenario is suitable in a case where the continuous integration system is used as a platform to configure and host other programs. The bug tracking system when used with a database to store bugs would be placed on the bottom layer.

(3) Mapping Multitenancy Isolation to Aspects of Isolation: As shown in Figure 6.1, the mapping of the different aspects of isolation between tenants is done in the following order: performance, security, resource consumption, from top to bottom for process isolation, and vice versa for data isolation. This means, for example, that it is better to use the shared component to improve resource consumption when implementing multitenancy isolation at the data/infrastructure level. On the other hand, it means that it is better to use the shared component to improve performance related requirements when implementing multitenancy isolation at the application level.

Again, the chance of implementing the required degree of isolation increases across the map-

ping (i.e., Figure 6.1) from left to right for performance-related requirements such as response time and throughput, while it increases from right to left for systems resource related requirements such as CPU and disk I/O usage. Issues related to security, privacy, trust and compliance to regulation can mostly be tackled in a hybrid related fashion. For example, data/bugs generated from a bug tracking system could be stored in a certain location to comply with privacy and legal regulations while the architecture of the GSD tool could be modified to restrict exposure of certain data to users located in regions not considered to be of interest to the owners of the hosted data. Architecting the deployment of a cloud service based on this arrangement can be best be tackled using a hybrid approach.

The mapping presented in this chapter is useful for several reasons: (i) it can be used to select a suitable multitenancy pattern or combination of patterns for deploying services to the cloud; (ii) it can be used to select an appropriate layer of the cloud application stack to implement multitenancy isolation in order to obtain the optimal performance and allocation of cloud resources, and minimize the risk to tenants; and (iii) in some cases, the business requirements set by cloud consumers may be either not feasible or too costly to implement. In such a case, the mapping provided will guide an architect in re-considering the business requirements of organisations to cope with available cloud resources.

6.3.2 Exploring Trade-offs for Achieving Multitenancy Isolation

This section discusses the key trade-offs for consideration when implementing the required degree of multitenancy isolation for cloud-hosted software processes. The case study synthesis identified six trade-offs that should be considered while implementing multitenancy isolation: multitenancy isolation versus (resource sharing, number of users/requests, customizability, the size of generated data, the scope of control of the cloud application stack, and business constraints). These trade-offs are explained below:

Multitenancy Isolation versus Resource sharing

The trade-off between multitenancy isolation and resource sharing is one of the most important considerations when deploying services to the cloud for guaranteeing multitenancy isolation. As the degree of isolation increases, the ability to share resources reduces. A low degree of isolation

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

Multitenancy Isolation versus Number of Users

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

Multitenancy Isolation versus Customizability

To implement the required degree of multitenancy isolation on a GSD tool, some level of customization would have to be done depending on the level where the process or component to be customised resides (Khan et al. 2012). The higher the degree of isolation that is required, the easier it is to implement the GSD tool. For example, implementing multitenancy isolation for a GSD tool like Hudson via virtualization on the infrastructure level will not be as difficult as implementing it on the application level in terms of the effort, time and skill required. Implementing isolation on the application level would require good programming skills to modify the source code, and also

address issues of compatibility and interdependencies between the GSD tool and required plugins and libraries (Hudson 2016b). Each time a multitenant application or its deployment environment changes, then a tedious, complex and maintenance process may also be required.

Multitenancy Isolation versus Size of Generated Data

There is also a trade-off between multitenancy isolation and the size of data generated by the GSD tool. The more data is generated, the more difficult it is to achieve a higher degree of isolation. For example, most version control systems (e.g., Subversion, File System SCM plugin) create additional copies of files on the shared repository. Over time, these files will occupy disk space thereby adversely affecting the performance experience by tenants. This will lead to a low degree of isolation between tenants since a lot of time would be spent fetching data from the repository that contains numerous unused or unwanted files. The study recommended a dedicated component for exclusive access to disk space, but again this implies significantly increasing the disk space and other supporting resources allocated to each tenant. In Figure 6.1, the GSD tools mapped to the lower level of the cloud stack (i.e., version control system and bug tracking) generate the most data. It is important to note that other GSD tools can be configured to generate additional data. For instance, Hudson can be configured to archive artefacts to a repository. Because of this, most the GSD tools have mechanisms for removing unwanted files, thereby saving disk space.

Multitenancy Isolation versus Scope of Control

Implementing the required degree of multitenancy isolation to a large extent depends on the "scope of control" of the cloud application stack. The term *cloud application stack* refers to the different layers of resources provided by the cloud infrastructure on which the cloud-hosted service is being hosted. This could either be the SaaS, PaaS or IaaS level (Badger et al. 2012). The architect has more flexibility to implement or support the implementation of the required degree of multitenancy isolation when there is greater scope of control of the cloud stack application. In other words, if the scope of control is restricted to the higher level of the cloud stack (i.e., the SaaS) then the architect may only be able to implement a low degree of isolation (e.g., shared component), and vice versa. Therefore, if an architect is interested in achieving a high degree of isolation (e.g., based on the dedicated component), then the scope of control should extend beyond the higher

level to the lower levels of the cloud stack (i.e., PaaS and IaaS). This would enable an architect to deploy a GSD tool on the IaaS platform so that exclusive access can be provided to the customer together with all the configuration requirements to support any operation that requires frequent allocation and de-allocation of resources. For example, using a version control system to perform operations that involve moving a repository between different hosts and keeping history would require having file system access in both hosts (Subversion 2016).

Multitenancy Isolation versus Business Constraints

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

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

6.4 Validity of the Case Study Synthesis

The validity of case study research can be evaluated using four key criteria: construct validity, internal validity, external validity and reliability (Yin 2014). Construct validity has been achieved by first conducting a pilot study, and after that three case studies using the same experimental procedure and analysis. The results of the study including the plots of estimated marginal means and the statistical results of the three case studies were compared and analysed to ensure consistency. Construct validity was further increased by maintaining a clear chain of evidence from the pri-

mary studies to the synthesised evidence, including the approach for implementation multitenancy isolation, experimental procedure and statistical analysis. Furthermore, the validity of the synthesised information has been increased by involving the authors of the primary studies to review the synthesis.

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

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

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

6.5 Chapter Summary

This chapter presented a synthesis of findings of three case studies (i.e., continuous integration with Hudson, version control with FileSystem SCM Plugin and Bug tracking with Bugzilla) that empirically evaluated the degrees of multitenancy isolation between tenants for components of a cloud-hosted service. The case study synthesis combined cross-case analysis and narrative synthesis approach to produce the commonalities and differences in case studies. Five aspects in which the cases are alike were identified: the strategy for reducing disk space, locking, low consumption of some system resources, and use of plugin architecture for extending the GSD tool, aspects of multitenancy isolation. Five aspects in which the cases differ were identified: size of data generated, the resource consumption of the GSD process, client's latency and bandwidth, supporting task performed, and error messages due to sensitivity to workload changes.

A further contribution of this chapter is an explanatory framework for (i) mapping the multitenancy isolation to different GSD processes, cloud resources and layers of the applications stack; (ii) explaining the different trade-offs to be considered for optimal deployment of components with a guarantee of the required degree of multitenancy isolation. The case studies synthesis identified six trade-offs that should be considered while implementing multitenancy isolation: multitenancy isolation versus (resource sharing, number of users/requests, customizability, size of generated data, scope of control of the cloud application stack and business constraints).

The study confirmed overall that a high degree of multitenancy isolation leads to high resource consumption and the running cost of tenants (or components). On the other hand, a low degree of isolation promotes an efficient utilisation of resources but with a possibility of performance interference. Therefore, there is need to resolve these trade-offs when optimising the deployment of services to the cloud while guaranteeing the required degree of multitenancy isolation for tenants. This is a decision-making problem which will be addressed in the next chapter by developing a model-based decision support system to achieve optimal deployment of components of a cloud-hosted service for guaranteeing multitenancy isolation.

Chapter 7

Optimal Deployment of Components for Guaranteeing Multitenancy Isolation

7.1 Introduction

In chapter 5 and 6, it was established that there are varying or different degrees of multitenancy isolation between tenants (or components) which in turn produce different effects on the required performance and resource consumption of tenants (or components) when there are workload changes (Fehling et al. 2014, Bauer & Adams 2012). A high degree of isolation between components may be necessary to avoid interference and enhance service security, but this usually leads to high resource consumption and running cost per component. A low degree of isolation promotes the sharing of components, thus leading to low resource consumption and running cost, but with a high possibility of performance influence when the workload changes, and the application does not scale up/down.

Therefore, the challenge is how to 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 running cost of tenants. 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 (Martens et al. 2010) (Legriel et al. 2010). Previous work on providing optimal allocation of cloud resources have focused on developing models that minimize the cost of cloud resources. Furthermore, previous work either do not use heuristic for optimisation

(Shaikh & Patil 2014, Westermann & Momm 2010) or use simple heuristic (e.g., hill climbing) in very few cases to find optimal solutions (Aldhalaan & Menascé 2015b).

Motivated by this problem, this chapter presents a model-based *decision support system* (DSS) together with a metaheuristic technique that can be used to provide 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. This chapter is based on preliminary work in (Ochei, Petrovski & Bass 2016b) and therefore duly referenced. Unlike the preliminary work, there are four variants of the metaheuristic in this chapter and the simulation experiments carried out on the model are based on: (i) datasets made up of larger instances of varying sizes and densities, (ii) a large number of function evaluations (i.e., 1000000 function evaluations); and (iii) a higher number of runs or trials (i.e., 20 runs).

The rest of this chapter is organised as follows: Section 7.2 describes and formalises our problem by mapping it to a multichoice multidimensional knapsack problem (MMKP). Section 7.3 discusses the open multiclass queueing network model, while Section 7.4 discusses the metaheuristic solution. In section 7.5, the decision support system (DSS) is described including its architecture, and the algorithm for implementing the DSS. Section 7.6 is the evaluation and results. Section 7.7 summarises the chapter.

7.2 Problem Formalization and Notation

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

7.2.1 System Model and Description of the Problem

Let us assume that there are multiple components of the same tenant on the same underlying cloud infrastructure. A tenant in this context 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 inter-

face components, etc.), purpose or size or some other feature (see Figure 7.1). 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. 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 application as a whole) without having the total resources used to exceed the available resources.

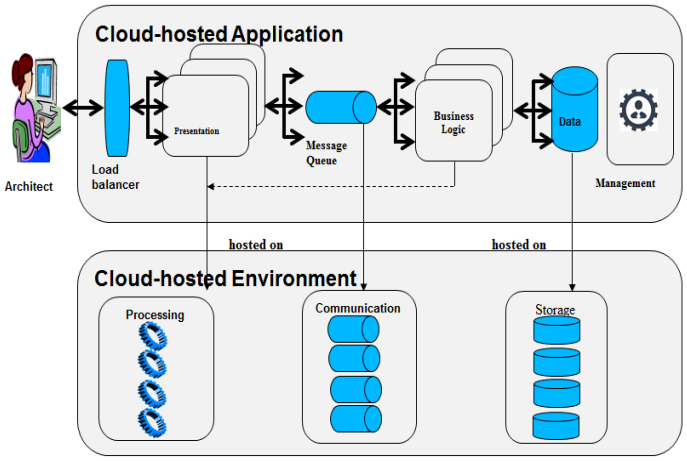

Figure 7.1: System Model of a Cloud-hosted Service with multiple groups of components

7.2.2 System Notations and Assumptions

The following notations(Table 7.1) 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) or hardware (e.g., virtual server). Each component is associated with six parameters: the isolation value, the

Table 7.1: 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
l	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 center 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	-

number of requests allowed to access the component, and resource consumption for CPU, RAM, Disk and Network bandwidth. A component could be located at any level of the cloud stack- SaaS, PaaS or IaaS layer.

2. *Tenant and Multi-tenant:* This study extends the notion of tenant and multitenant from a single user/customer to a team, 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.

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.

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. *Optimal Function:* As stated in the problem description (section 7.2.1), 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 according to the preferences of the decision maker (Talbi 2009). This approach has been widely used in literature for metaheuristic such as genetic algorithm and simulated annealing(Chipperfield, Whidborne & Fleming 1999, Karasakal & Köksalan 2000)

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

Definition 7.1 (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:

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

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

$$P_{ij} = \sum_{j=1}^{m} Rj^{max} \left\{ 0, \left(\frac{R_j - R_j^{max}}{R_j^{max}} \right) \right\}^2 \tag{7.1}$$

where w1,w2,w3 are the weights for isolation value (w1=100), number of requests(w2=1) and penalty(w3=0.1). The weights are chosen based on problem-specific knowledge so that more importance or preference is given to the isolation value and number of requests which are parameters to be maximised in our model. The degree of isolation, I_{ij}, for each component, is set to either 1, 2, or 3 for *shared component, tenant-isolated component and dedicated component*, respectively. 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.

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

For a cloud-hosted service that can be designed to use or integrated with several components in N different groups, and with m resource constraints, the problem of providing optimal solutions that guarantee multitenancy isolation can be mapped to a 0-1 multichoice multidimensional knapsack problem (MMKP) (Martello & Toth 1987, Kellerer, Pferschy & Pisinger 2004). An MMKP is a variant of the Knapsack problem which has been shown to be a member of the NP-hard class of problems (Martello & Toth 1990). Our problem is formally defined as follows:

Definition 7.2 (Optimal Component Deployment Problem): Suppose there are N groups of components $(c_1,..., c_N)$ with each having l_i $(1 \leq \text{i} \leq N)$ components that can be used to design (or integrate with) a cloud-hosted application. Each application component is associated with: (i) the required degree of isolation between components (I_{ij}); (ii) the arrival rate of requests to the component λ_{ij}; (iii) the service demand of the resources supporting the component D_{ij} (equivalent

to $D_{c,k}$ in the QN model as shown in section 7.3); (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 section 7.3) and (v) m resources which are required to support the component, $r_{ij}^{\alpha} = r_{ij}^{1}, r_{ij}^{2},..., r_{ij}^{m}$. The total amount of available resources in the cloud required to support all the application components is $R = R^{\alpha}$ $(\alpha = 1,..., m)$. The objective of an MMKP is to pick exactly one component from each group for a maximum total value of the collected items, subject to m resource constraints of the knapsack (Yu, Zhang & Lin 2007, Akbar, Rahman, Kaykobad, Manning & Shoja 2006). Concerning our problem, the goal is to deploy components of a cloud-hosted service by selecting one component from each group to meet the resource constraints of the system and maximise the optimal function $\mathcal{G}$. There are unique features in our problem that lend to solving it using an MMKP and an open multiclass problem. For example, the resources supporting each component are mapped to the resources required by the object in MMKP and are also mapped to the service centres of each class in the open multiclass QN. The third and fourth columns of Table 7.2 show how some of the key attributes of the multitenancy isolation problem map to the MMKP and 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

$$\sum_{i=1}^{N} \sum_{j \in C_i} r_{ij}^{\alpha}.a_{ij} \leq R^{\alpha} (\alpha = 1, 2, ..., m) \tag{7.2}$$

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

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

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 (Menasce et al. 2004) for each component using the arrival rate of each class of requests, and the service demands of each resource required to support the component (i.e., CPU, RAM, Disk capacity, and Bandwidth). Section 7.3 describes how the average number of requests allowed to access each component is computed.

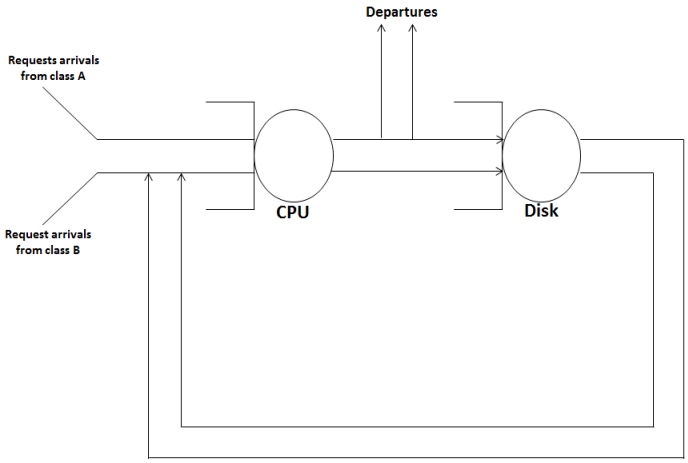

Figure 7.2: Open Multiclass Queuing Network Model

7.3 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 (Menasce et al. 2004). Figure 7.2 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) a component is deployed to support a single cloud application, and so cannot support different applications or applications at different system requirements.

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

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

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

are functional.

The above assumptions allow us to use an open multiclass queuing network (QN) model to determine the average number of requests that can be allowed to access the component while meeting the required degree of isolation and system requirements. In an open multiclass QN, the workload intensity is specified by the request arrival rate. This arrival rate usually does not depend on the system state, that is, it does not depend on the number of other tenants in the system (Menasce et al. 2004).

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

$$U_{c,k}(\overrightarrow{\lambda}) = \lambda_c D_{c,k} \tag{7.3}$$

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:

$$Q_{c,k}(\overrightarrow{\lambda}) = \frac{U_{c,k}(\overrightarrow{\lambda})}{1 - \sum_{i=1}^{N} U_{i,k}(\overrightarrow{\lambda})} \tag{7.4}$$

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

$$Q_c(\overrightarrow{\lambda}) = \sum_{k=1}^{K} Q_{c,k}(\overrightarrow{\lambda}) \tag{7.5}$$

7.4 Metaheuristic Search

The optimisation problem described in Section 7.2.1 and then mapped to an MMKP in Section 7.2.3 is an NP-hard problem which has been known to have a feasible state space that grows in a combinatorial way (Yu et al. 2007). The number of feasible states for our optimal component

deployment problem is given by the following expression:

$$\left\{ \binom{l}{j} \right\}^{N} \tag{7.6}$$

Equation 7.6 above represents the number of ways for selecting one component (j items) from each a group (made of up l items) out of several (N) groups of components to integrate with or designed to use a cloud-hosted application when workload changes in a particular time interval. Thus in response to the workload changes, the number of ways of selecting one component (i.e., j=1) each from twenty groups (i.e., N=20) containing ten items in each group (i.e., l=10) will result in approximately 10.24 x 10^{12} states. Depending on the number of times and frequency with which the workload changes, the number of states could grow very large at a much faster rate.

Therefore, an efficient heuristic is needed to find an optimal solution to the optimisation problem, which must be solved by the decision support system and provided to the SaaS customer (or a cloud deployment architect) in almost real-time. 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 (Rothlauf 2011):

(i) Usually, modern heuristics are defined 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 use no explicit

diversification steps.

Any of the four variants of the metaheuristic solution can be utilised with *OptimalDep* (see line 17 of Algorithm 3). 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 Algorithm 3 and Algorithm 4, respectively. A high-level description of these algorithms is provided below:

Algorithm 3 optimalDep Algorithm

```
1: optimalDep (workloadFile, mmkpFile)
2: optimalSoln ← null
3: Accept workload from SaaS users
4: Load workloadFile, mmkPfile; populate global variables
5: repeat
6:     /*Compute No. of req. using QN Model*/
7:     for i ← 1, NoGroups do
8:         for i ← 1,GroupSize do
9:             Calculate Utilization /*see Equation 7.3*/
10:            Calculate No. of req. /*see Equation 7.4*/
11:            Calculate Total No. of req. /*see Equation 7.5*/
12:            Store fitValue, Isol, qLength of optimal soln.
13:        end for
14:    end for
15:    Update the mmkpFile with qLength
16:    /*Run Metaheuristic*/
17:    SA(GREEDY)( )
18:    /*Display optimal solution for deployment*/
19: until no more workload
20: Return (optimalSoln, fitValue, Isol, qLength)
```

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

$$T_t = T_0 - \eta t \tag{7.7}$$

The above cooling schedule (equation 7.7) is linear and which means that T decreases every t iterations by an amount ηt. Since the introduction of this linear cooling schedule shown in Equation

Algorithm 4 SA(Greedy) Algorithm

1: **SA(Greedy)** (mmkpFile, N)
2: Randomly generate N solutions
3: Set initial temperature T_0 to st. dev. of all optimal values
4: Create greedySoln a^1 with optimal value $g(a^1)$
5: *optimalSoln* $\leftarrow$ $g(a^1)$
6: *bestSoln* $\leftarrow$ $g(a^1)$
7: **for** $i \leftarrow 1, N$ **do**
8: Create neighbouring soln a^2 with optimal value $g(a^2)$
9: Mutate the soln a^2 to improve it
10: **if** $a^1 < a^2$ **then**
11: *bestSoln* $\leftarrow a^2$
12: **else**
13: **if** *random[0,1)* $<$ exp(-($g(a^2)$ - $g(a^1)$)/T) **then**
14: $a^2 \leftarrow$ *bestSoln*
15: **end if**
16: **end if**
17: $T_{i+1} = T_0 - \eta i T$ /*see Equation 7.7*/
18: **end for**
19: *optimalSoln* $\leftarrow$ *bestSoln*
20: **Return** (optimalSoln)

7.7 by the authors in (Kirkpatrick, Gelatt, Vecchi et al. 1983), it has been widely used in several optimization models relying on simulated annealing (Nourani & Andresen 1998, Pirkwieser & Raidl 2008, Zoraghi, Najafi & Akhavan Niaki 2012, Huang 2003). In the above cooling schedule, the variable η is computed as follows:

$$\eta = \left(\frac{T_0}{max(t)} \right) \tag{7.8}$$

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

A simple dry run of the algorithm for the instance C(20,20,4) is as follows: 20 optimal so-

lutions are randomly generated and then the standard deviation of all the solutions is computed. Assuming this value is 5.26, the T_0 is set to 5. At the first iteration, $g(a^2)$ = 151634.9773 and $g(a^1)$ = 151535.7984 and the current temperature then becomes 4.999995. At the next iteration, the current temperature is expected to reduce further (see equation 7.8). After five iterations, the algorithm constructs an initial/first solution with $g(a^1)$ = 151732.4362, a current/second random solution with $g(a^2)$ = 151733.9821 and with a current temperature of 4.999975. The solution a^2 will replace a^1 with probability, P =exp(-(1.5459)/4.999975)=0.7340, because $g(a^2) > g(a^1)$. In lines 13 to 15, a random number between 0 and 1 (i.e., rand = 0.0968) is generated, and since *rand* $<$ *0.7340*, a^2 replaces a^1 and the algorithm continues with a^2. Otherwise, the algorithm continues with a^1. At the next iteration, the temperature T is reduced which now becomes T_6 = 4.99997 (line 17). The iteration continues until N (i.e., the number of iterations set for the algorithm to run) is reached, and so the search converges with a high probability to the optimal solution.

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 based on simulated annealing algorithm (i.e, SA(Greedy and SA(random)) can be converted to a local search based on the hill climbing algorithm by setting the initial temperature to zero (i.e., T=0) so that the simulated annealing is forced to systematically explore the neighbourhood around the current solution and to ensure that the search returns a local minimum.

***HC(Random) and HC(Greedy)*:** The *HC(Random)*, uses 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 out lines 12-15 (i.e., the else part of the if statement) of Algorithm 4.

7.5 Decision Support System for Optimal Deployment of Components

A model-based decision support system (DSS) has been created to provided optimal solutions for deploying components of a cloud-hosted service.

7.5.1 Architecture of the Decision Support System

The DSS can be implemented in different ways, for example, as a web-based application or a desktop application. It can then be deployed directly to the cloud or installed on a Docker container. This section shows the architecture of the DSS which is composed of five main modules (see Figure 2).

Input Interface

This module is used to send input to the decision support system. 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 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 supports 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 from the updated MMKP instance whose id was passed to it. 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 (e.g., as shown in section 7.5.2), the number of requests accessing each component and the total number of requests accessing the whole cloud service

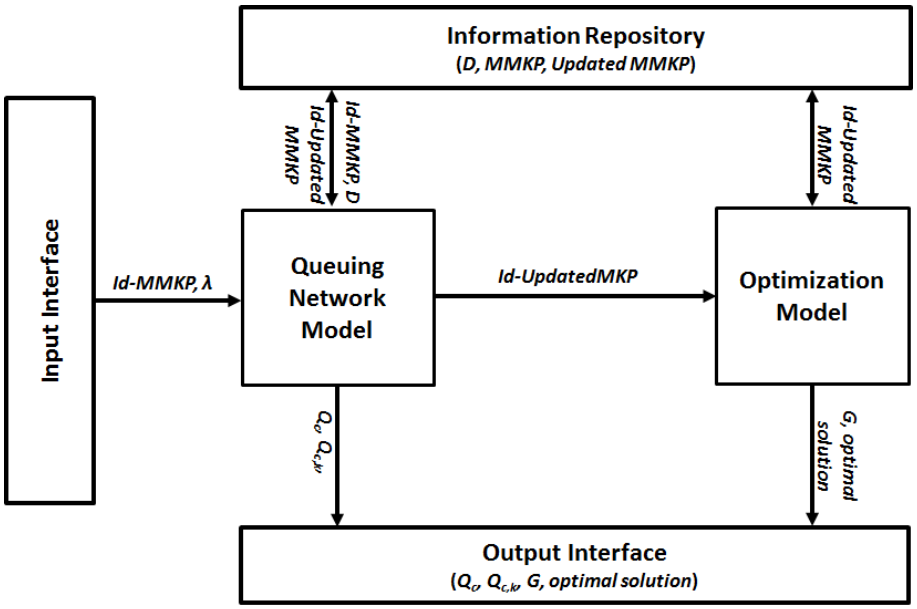

Figure 7.3: Architecture of the Model-based Decision Support System

7.5.2 OptimalDep: An algorithm for Optimal Deployment of Components

This section describes the OptimalDep algorithm and also shows how the open multiclass QN model and the heuristic search fits into the decision support model.

Description of OptimalDep Algorithm

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

Table 7.2: 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: (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		✓		✓		✓

required degree of isolation.

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

OptimalDep Algorithm Example

The following example shows the different solutions evaluated by *optimalDep* combined with the SA(Greedy) algorithm to find a 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 has a maximum number of requests that can be allowed to access it without having a negative impact on the degree of isolation between components of the cloud-hosted service. Each component is supported by four main resources: CPU, RAM, Disk capacity and bandwidth. The service demands for CPU=0.25, RAM=0.23, disk=0.22, bandwidth=0.2, while the maximum capacity of each of these resource is 20.

When a request arrives indicating a change in workload (i.e., in our case, this means an arrival rate between 0 to 3.7 req/min), the QN model equations 7.3, 7.4, and 7.5 is solved to find the average number of requests that can access the components. The ninth row shows the updated problem instance with the current number of requests (i.e., 5.2) that can access the components in each group. The updated problem instance is then solved with the metaheuristic and the state with the highest optimal function value is returned. Solution 1 (in row twelve) shows the optimal value of 515.6 for selecting a solution that deploys the first component from all the groups. This solution results in an optimal value of 515.6 (isolation value=500; and number of request=15.60). Note that no component can be selected for deployment and hence no changes can be effected on the cloud environment until the search is over and a better solution is found.

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

7.6 Evaluation and Results

This section evaluate the optimalDep algorithm which is the main algorithm that drives the model-based decision support system. The optimaDep 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

Table 7.3: 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]
SA(Greedy) Algorithm	
No of Iterations	N=1000000
No. of Runs	20
Temperature	T_0 = st. dev of N randomly generated solutions (N=no. of groups)
Cooling Schedule	$T_t = T_0 - \eta t$ ()see equation 7.7 and 7.8)

algorithm.

The performance evaluation will be presented in terms of the quality of solution, robustness and 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.

7.6.1 Experimental Setup and Procedure

The problem/MMKP instances used for our experiments were generated as described in section 3.3.1. 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 7.3 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.

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

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

The approach presented in this chapter is novel in that it combines a QN model and metaheuristics to find optimal solutions for component deployment with guarantees for 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 percent deviation from the optimal solution. The instance used is C(4,5,4) because it was small enough to cope with the requirements of the machine. The workload (i.e., the arrival rate) for each component was randomly generated between 0.0 and 4.

The results are summarised in Table 7.4. 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), which stand for the value of the optimal function and the number of function evaluations to attain the optimal solution.

As shown in Table 7.4, 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 instance larger than C(4,5,4) due to limitations in its hardware requirements (i.e., CPU and RAM). Notice that the number of function evaluations required to produce the optimal solution for the greedy variations of the algorithm is 0. This is due in part to the small size of the MMKP instance, and the fact that some effort has already been put in to produce the greedy solution and so the optimal solution is attained very quickly with little or no computational effort in terms of the number of function evaluations.

In Figure 7.4, 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

Table 7.4: 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
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

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 limitation regarding the time and amount of resources required to execute the decision support system 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 (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.

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

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

$$TargetSoln = ((n \times max(I) \times w1) + (0.05 \times (n \times max(Q) \times w2))) \tag{7.9}$$

where *n* is the number of groups, *max(I)* is the maximum isolation value, *max(Q)* is the maximum possible number of requests (calculated based on the upper limit of the arrival rate) and *w1* assigned to *I* and *w2* is the weight assigned to the *Q*. This equation when used to compute the target solution of C(4,5,4) with arrival rate of 2.7 req/sec gives 1219.2, which is very close to the optimal solution shown in Table 7.4. The target solution for all instance sizes ranging from C(10,5,4)/C(10,20,4) to C(1000,5,4)/C(1000,20,4) are shown in Table B.1 and B.2 of appendix B.

The optimal values obtained for each instance were the same for all the variants of the metaheuristic (see Table B.1 and Table B.2 in appendix B). The rest of the experiment was conducted with an arrival rate of 3.9 requests per second.

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

Measuring the Quality of Solutions

The quality of the solutions was measured in terms of the percent deviation from the target solution (see equation 3.3). As shown in Table 7.5, the percent deviation for all the variants of the metaheuristic was the same. It was noticed that the percent deviation of solutions is lower when the number of components per group is high. For example, the percent deviation for C(500,5,4) is 3.5 when the number of components is 5 and the percent deviation of C(500,20,4) and then 1.49 when the number of components is increased to 20. This means that the quality of solutions is a function of the number of components per group. The more choices of a particular type of component there are, the better the chance of obtaining an optimal configuration. This is particularly important for large open-source projects that are either designed to use a large number of components within the cloud-hosted service or be integrated with several components residing in other locations.

Table 7.6 shows the percent deviation for a large instance size (i.e., C(500,20,4)). 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 Figure 7.6, the quality of solutions is shown for the first 10000 function evaluations. 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

Table 7.5: Average performance on different instance sizes(m=5; m=20)

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

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.

Measuring the Robustness of the Solutions

Robustness refers to how sensitive the solutions are, against small deviations in the input data or other parameters; the lower the variability, the better the robustness(Talbi 2009). The standard deviation was used as a measure of this variability. Table B.1 and B.2 (in Appendix B) show the standard deviation for all instance sizes in the variable, Opt/Std. For example, 0.27 is the value for standard deviation for C(1000,20,4). It was observed that the standard deviation for SA(Random) and SA(Greedy) was higher than that of HC(Random) HC(Greedy) in most cases. This means that metaheuristic based on hill climbing was more stable and robust than the other variants based on simulated annealing, especially for large instances.

Table 7.6 shows that although the minimum, maximum, average values of solutions produced by the HC(Greedy) when applied to a large instance (i.e., C(500-2-4)) for the first 10000 function

Table 7.6: Comparing Solution Quality with Number of Function Evaluations

ITRN	HC(Rand)	HC(Greedy)	SA(Rand)	SA(Greedy)	HC(rn)	HC(gr)	SA(rn)	SA(gr)
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

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 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.

Measuring the Computational Effort

The computational effort was measured using several metrics: success rate, performance rate and average execution time required to produce a solution. Table 7.7 presents the success rate and performance rate for C(500-20-4) after running the algorithms for 10000 function evaluations. 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 7.5 shows the run length distribution of the C(500-20-4) instance for all the variants of our metaheuristic. As expected, the variants that start with the initial greedy solution have a

Table 7.7: Success Rate and Performance Rate based on Target Solution (C(500-20-4))

ITRN	**HC(rn)**	**HC(gr)**	**SA(rn)**	**SA(gr)**	**HC(rn)**	**HC(gr)**	**SA(rn)**	**SA(gr)**
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

Table 7.8: Function Evaluations to attain Target Solution.

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

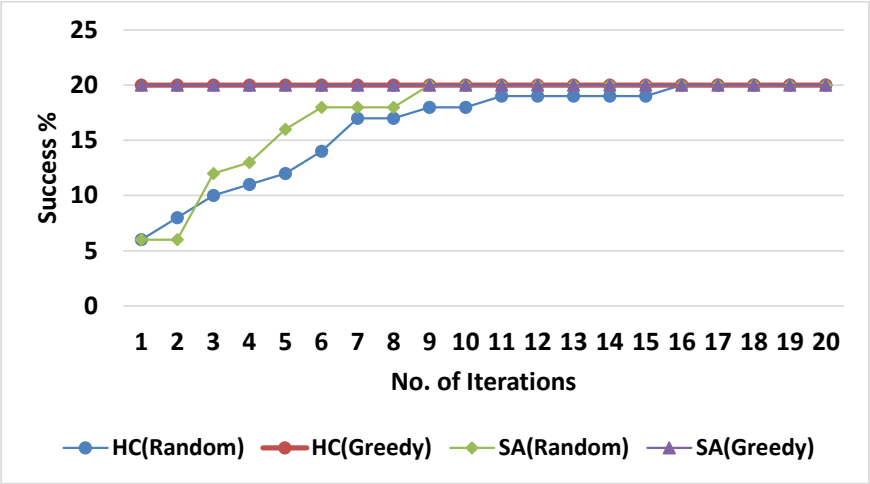

Figure 7.4: Run Length Distribution for Small Instance (C(4-5-4))

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 7.8 shows the number of iterations reached for each run before attaining the target solution.

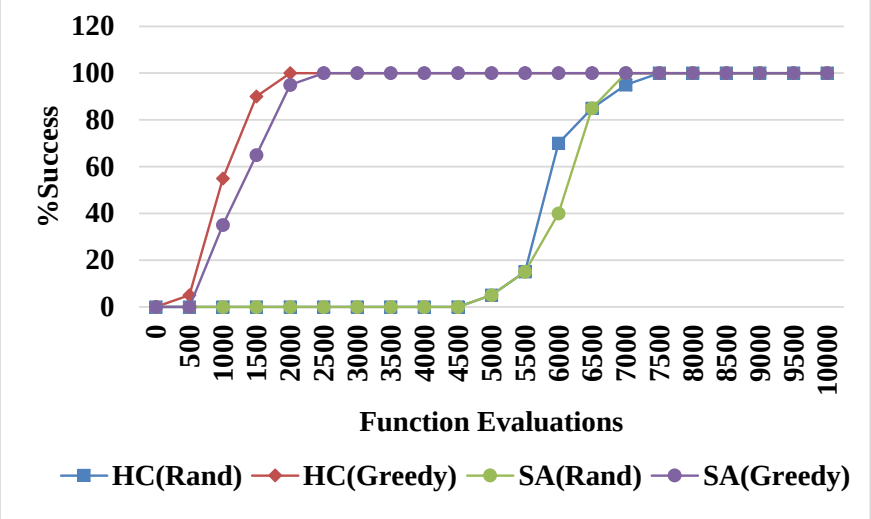

Figure 7.5: Run Length Distribution for a Large Instance(C(500,20,4))

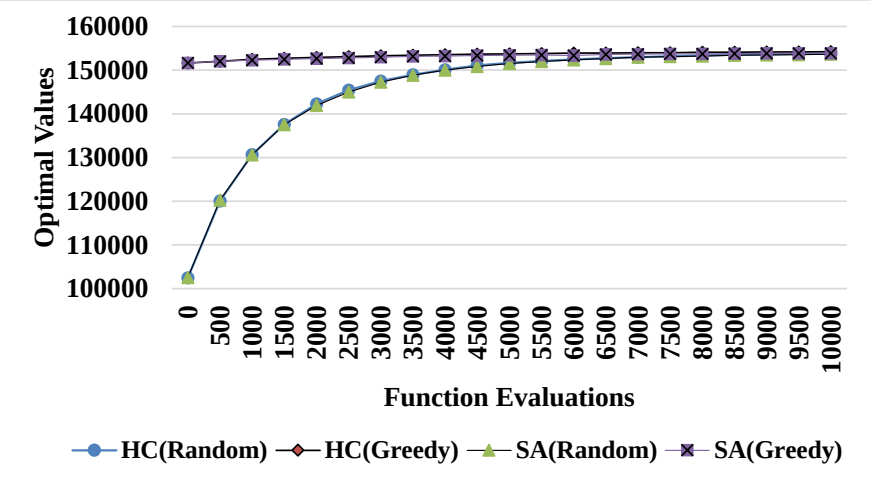

Figure 7.6: Quality of Solution for a large instance size(C(500-20-4))

In addition to the above metrics, the estimated execution time required by each variant of the metaheuristic to reach the target solution for different instance sizes, was also computed. The estimated execution time is calculated using the formula given in equation 3.4 (section 3.4.2). Table 7.9 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

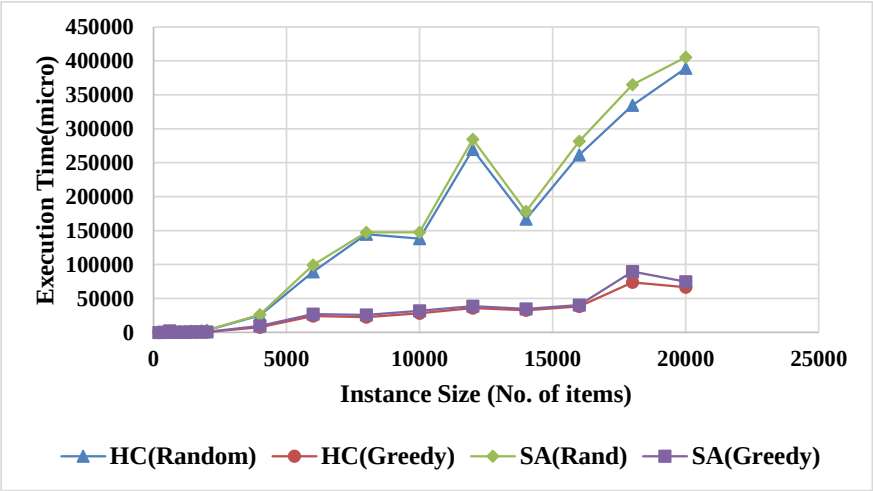

Figure 7.7: Computational Effort for a large instance size(C(500-20-4))

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

The results show that it takes the variants of the metaheuristic that start with the greedy solution a far less number of functions evaluations to reach the target solutions. For example, Table 7.8 shows that 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). As expected, Table 7.9 shows that the average execution times for producing the initial greedy solution is larger than that of the random solution. Surprisingly, as illustrated in Figure 7.7, 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 7.9 shows that the execution time required to produce an initial greedy solution is 400 times in most cases over that of random solutions. However, because the average number of function evaluations required by the metaheuristic that start with greedy solutions (i.e., HC(Greedy) and SA(Greedy)) is far less than those that start with random solutions, 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. For large instances, percent deviation of

Table 7.9: Computational Effort (in seconds) 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

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 10000 in the 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.

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

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

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

Solutions from hill climbing were more stable and robust than that of simulated annealing, especially for 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.

7.6.4 Statistical Analysis

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 if there is an interaction between the two independent variables on the dependent variable (Laerd.com 2017). The variables of interest are (i) percent deviation (for testing quality of solution); (ii) standard deviation of a set of optimal solutions(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 (i.e., C(10,20,4) - C(60,20,4)) and large instances (i.e., C(500,20,4) - C(1000,20,4)) 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 (α = 0.005) for the hypothesis. This study assumes typical conditions of normality, independence and equality of variance (Cohen 1995, Talbi 2009).

In the design, the *type of instance* and the variant of metaheuristic has two and four levels, respectively. In all there are 2 x 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/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 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 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 running for 1000000 function evaluations, which ensured that the search converges to an optimal solution.

(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 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.

(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(Random) and SA(Greedy)(p=0.000). There was no difference between HC(Random) and SA(Random)(p=0.561) and HC(Greedy) and SA(Greedy)(p=0.843). This means that for large instance sizes, there is no difference in execution time if a random solution is 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.

7.7 Chapter Summary

This chapter presents the implementation of the model-based decision support system (DSS) for providing optimal solutions for deploying components designed to use (or 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 DSS works as follows: when a request arrives 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 optimalDep when combined with metaheuristic that starts with an initial greedy solution, provides solutions that are robust and of better quality when compared with the metaheuristic that starts with random solutions. For large projects, starting the metaheuristic with an initial solution with a greedy solution can boost the model-based DSS. Also, for large in-

stances, 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.

Chapter 8

Discussion

8.1 Introduction

In the previous chapters, the findings from the different strands of this work were presented in general terms. This chapter will first discuss the findings of the different aspects of the thesis, followed by some recommendations for architecting the deployment for cloud-hosted services for guaranteeing multitenancy isolation. Finally, the different areas where our work can be applied will be discussed.

8.2 Discussion of Findings: Exploratory, Case Studies, Synthesis and Modelling

This section discusses the findings from the exploratory study, case studies, case studies synthesis and modelling.

8.2.1 Discussion on Findings from Exploratory Study on Deployment Patterns

The results clearly suggest that by positioning a set of GSD tools on our proposed taxonomy, the purpose of the study has been achieved. The overarching result of the study is that most deployment patterns should be combined with others during implementation. The findings presented here support previous research suggesting that most patterns are related and so two or more patterns can be used together (Bass et al. 2013, Vlissides et al. 1995).

(1) Combining Related Deployment Patterns: Many deployment patterns are related and cannot be fully implemented without being combined with others, especially to address hybrid deployment scenarios. This scenario is very common in collaborative GSD projects, where a GSD tool either requires multiple cloud deployment environments or components, each with its set of requirements. Our taxonomy, unlike others (Wilder 2012, Homer et al. 2014), clearly shows where to look for hybrid-related deployment patterns (i.e., the space demarcated by thick lines in Table 4.1) addresses this challenge. For example, when using Hudson, there is usually need to periodically extract the data it generates to store in an external storage during the continuous integration of files. This implies the implementation of a hybrid data pattern. Hudson can be used in combination with other GSD tools, such as Subversion (for version control) and Bugzilla (for error tracking) within a particular software development project, each of which may also have their deployment requirements.

(2) GSD Tool Comparison: The taxonomy gives us a better understanding of various GSD tools and their cloud-specific features. While other taxonomies and classifications use simple web applications (Wilder 2012) to exemplify their patterns, a mixture of commercial and open-source GSD tools is used. For example, commercial GSD tools (i.e., JIRA and VersionOne) are offered as a SaaS on the public cloud, and they also have a better chance of reflecting the essential cloud characteristic. Their development almost coincides with the emergence of cloud computing, allowing new features to be introduced into revised versions. The downside is that they offer less flexibility regarding customization (Sommerville 2011).

On the other hand, open-source GSD tools (i.e., Hudson, Subversion) are provided in the public cloud by third party providers, and they rely on API/plugins to incorporate support for most cloud features. The downside is that the developer's community does not maintain many of the plugins available for integration and, so consumers use them at their risk. The taxonomy also revealed that open-source GSD tools (e.g., Hudson, Subversion) are used at a later stage of a software life-cycle process in contrast to commercial tools, which are used at the early stages.

(3) Support for API/Plugin Architecture: Another interesting feature of our taxonomy is that by positioning the selected GSD tools on it, it was discovered that support for the implementation of

most deployment patterns is practically achieved through API/Plugin integration (Musser 2012). This is no coincidence, as a typical cloud application is composed of various web-based related technologies such as web services, SOA and n-tier architectures. Therefore, a GSD tool with little or no support for APIs/Plugins is unlikely to attract interest from software developers. For example, JIRA's Elastic Bamboo support for Blob storage on Windows Azure is through an API (Atlassian.com 2016). JIRA has a plugin for integrating with Hudson, Subversion and Bugzilla (Atlassian.com 2016) and vice versa.

(4) Maintaining State and Exchanging Information Asynchronously: Our taxonomy also highlights the technologies used to support the software processes of GSD tools, unlike others, which focus mostly on the design of cloud-native applications (Fehling et al. 2014). Web services (via REST) and messaging (via message queues) are the preferred technologies used by cloud deployment patterns (e.g., stateless pattern, message-oriented middleware) to interconnect GSD tools and other components. Public cloud platforms favour the REST style. For example, JIRA's support for SOAP and XML-RPC is depreciated in favour of REST (Atlassian.com 2016). This trend is also reported in (Wilder 2012, Musser 2012).

(5) Accessing Data stored in Cloud Storage: Some GSD tools (e.g., Subversion) handle significant amounts of data (images, music, video, documents/log files) depending on the nature of the software development project. This data can be stored in cloud storage to take advantage of its ability to scale almost infinitely and store large volumes of unstructured data. The downside is that the application code of the GSD tool has to be modified to enable direct HTTP-based REST API calls. Cloud storage's object architecture requires REST API to be either integrated as a plugin into the GSD tool or coded separately. Storing data on the cloud is invaluable in a case where the GSD tool runs in a static environment, and the data it generates is to be archived on an elastic cloud.

(6) Patterns for Cloud-application Versus Cloud-environment: Our taxonomy can be used to guide an architect in focusing on a particular architectural deployment component of interest - that is, either a cloud-hosted application or cloud-hosted environment. Other taxonomies (Homer et al. 2014, Wilder 2012) are concerned with the design of cloud-native applications. Assuming an architect is either interested in providing the right cloud resources or mapping the business

requirement to cloud properties that cannot be changed (e.g., location and ownership of the cloud infrastructure), then Taxonomy A would be more relevant.

However, if the interest is in mitigating certain cloud properties that can be compensated for at an application level (e.g., improving the availability of the cloud-hosted GSD tool), then Taxonomy B should be considered. Fehling et al. describe other cloud properties that are either unchangeable or compensatable for deploying cloud applications (Fehling et al. 2014).

8.2.2 Discussion of Findings from Case Studies and Case Study Synthesis

This section presents some recommendations for implementing varying degrees of multitenancy isolation based on the results of the case studies and case study synthesis carried out. These recommendations are summaries below:

(1) Volume of Data Generated : The case studies revealed that the volume of data generated could have a significant impact on the required degree of isolation. The continuous integration process and version control process can potentially generate more data than bug tracking systems. For example, several of the problems that occur in version control relate to the fact that version control systems usually create additional copies of files on the repository (especially the ones that use the native operating system (OS) filesystem directly) (Collins-Sussman et al. 2004). This adversely affects performance because these files occupy more disk space than they use, and the OS spends a lot of time seeking across many files on the disk.

(2) Optimising Cloud Resources while Guaranteeing Isolation: Apart from the continuous integration process (that is, build/compilation process), which are known to consume much memory and disk I/O, most GSD processes do not consume much IT resources. For example, Hudson which is used to simulate continuous integration does not consume much CPU because builds can be set-up to run in the background without interfering with other processes. Therefore, to optimise the cloud resources while guaranteeing multitenancy isolation, the architect should avoid certain operations that would increase system's resource consumption. Generally, operations that would lock processes for a long time should be avoided. For example, in continuous integration, it would be recommended to avoid as much as possible (i) carrying out difficult and complex builds (i.e., builds that have many interdependencies with other programs or systems), and (ii) running a large number of builds concurrently. For the version control, it would be recommended to avoid moving

data from one repository to another, data compression and packing. For bug tracking process that uses a relational database to store data, it would be recommended to avoid caching database transactions and using run-time libraries (e.g., model_perl) that are known to consume much memory. Data generated in the cloud environment are mostly unstructured. This seems to suggest that relational databases are not ideal for multitenancy and so it is recommended to use simpler databases (e.g., Amazon Simple DB, SQL Azure, and Google Big Table) provided by large cloud producers (Sommerville 2011, Doddavula, Agrawal & Saxena 2013a).

(3) Sensitivity to Workload Interference: Our experience seems to suggest some of the GSD processes are sensitive to workload interference. For example, bug tracking (with Bugzilla) was susceptible to increased workload especially if locking is enabled for the bug database. It was noticed that frequent crashes of the Bugzilla database occurred in our experiments which required recovery. There were also numerous database related errors. It would be recommended to increase the maximum size of a file that can be stored in the database. It may also be necessary to remove restrictions on the maximum number of allowed queries, connections and packets, etc. There are several cloud patterns which can be used to minimise workload interference in multitenant applications. For example, Doddavula et al. present several cloud computing solution patterns for handling application workloads such as applications with highly variable workloads in public clouds, and workload spikes with cloud burst (Doddavula, Agrawal & Saxena 2013b)

(4) Interacting with GSD Processes with Sufficient Latency and Bandwidth: When a client network has high latency and bandwidth, it can have a negative impact on the performance of other tenants and hence the required degree of multitenancy isolation. For bug tracking, if a client with a low network bandwidth is interacting with a GSD process (e.g., bug tracking), it may be advantageous to compress large bug attachments before moving the data across the network, though with limitations of high CPU consumption. For the continuous integration (CI) process, it is recommended that the network connection has low latency especially if the CI server is configured to publish artefacts to a source control server automatically.

(5) Implementing Multitenancy on Different layers of the Application Stack: Depending on the layer of the application stack, achieving multitenancy isolation for cloud-hosted GSD tools and

supporting processes may be realised differently with associated implications. For example, implementing the shared component on the SaaS layer for a bug tracking process that uses a database to store data would ensure efficient sharing of cloud resources, but isolation is either very low or not guaranteed at all. Implementing a dedicated component on the IaaS layer would require installing the bug database for each tenant on its instance of virtual hardware, thus increasing the runtime cost and limiting the number of tenants that can be served. Previous research work focused on implementing multitenancy at the data tier (Wang et al. 2008, Vanhove et al. 2014). The approach we have presented in this thesis addresses this challenge by capturing and analysing tenants requests/transactions at the application tier and then using this to adjust the behaviour of the cloud-hosted service based on the required degree of isolation (Ochei, Bass & Petrovski 2015c, Ochei, Petrovski & Bass 2015).

8.2.3 Discussion of Findings from Modelling and Simulation

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

(1) Quality of the solutions: The model-based decision support system (DSS) can be used to obtain high-quality solutions with any of the four variants of metaheuristic when dealing with small instances. The DSS would perform well both in 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 work to improve the quality of solutions. Many variants of metaheuristic often use initial solutions generated randomly (Rothlauf 2011).

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 the solutions (Rothlauf 2011).

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 agrees with the conclusions from (Sliwko & Getov 2015) where the author developed a prototype metaheuristic load balancer to allocate services on a cloud system without overloading the nodes and maintaining the system stability with minimum cost. The author recommended that better results can be achieved if 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) was more robust and stable on 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 or control such interdependencies 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 (Fehling et al. 2014), unpredictable and aggressive workloads (Doddavula et al. 2013b, Walraven et al. 2012).

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

(3) 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 less 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 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 work, 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 (Sliwko & Getov 2015).

Our approach assumes that the initial solution is computed first before running the metaheuristic, 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 the metaheuristic that start with the greedy solution. However, the results show 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 DSS when supported with metaheuristic that starts with greedy solutions would be suitable for handling large scale projects that may have a significant number of interdependent components.

There are several situations where there is 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 multi-tenancy isolation. Another situation is when there is 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 for a whole cloud service (Fehling et al. 2014). 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.

8.3 Challenges and Recommendations

This section presents a general discussion that focuses on the challenges of implementing multi-tenancy isolation and the recommendations that can be followed to achieve the required degree of isolation.

8.3.1 Type and Location of the application component or process to be shared

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

On the application level, multitenancy isolation can be implemented by introducing a tenant-id field to tables so that tenants can only access data that is associated with their tenant-id. An approach (i.e., COMITRE) has been developed for evaluating the required degrees of multitenancy which is anchored on shifting the task of routing a request from the server to a separate component (e.g., Java class or plugin) at the application level of the cloud-hosted GSD tool (Ochei, Bass & Petrovski 2015c). One of the advantages of using this approach is that it can be used at the application level to optimise the utilisation of the underlying cloud resources in a resource constrained environment, for example, where there are limited CPU, Memory and disk space. The drawback is the effort and skill required in modifying the GSD tool before implementing the COMITRE logic.

8.3.2 Customizability of the GSD tool and supporting process

Most GSD tools would have to be customised to implement the required degree of multitenancy isolation. This can be a big challenge if the GSD has several components that are being shared. Different application components can be implemented at different levels to address the problem between aspects of the GSD that can be customised with ease and those that cannot. For example, Bugzilla interface can be exposed as an integrated component to different tenants working on other GSD tools like JIRA while the Bugzilla database can be implemented as a dedicated component to ensure proper isolation of bugs belonging to various tenants.

Another major challenge associated with customization is that a GSD tool can have many

inter-dependencies on different levels of the application itself and with other applications, plugins, libraries, etc., deployed with other cloud providers. There is also a serious risk of using incompatible plugins and libraries required to modify, customise and execute these GSD tools. This could corrupt the GSD tool and stop other supporting programs/processes from running. An easy way to address this challenge on the cloud is to push the implementation of multitenancy isolation down the lower levels of the cloud stack, where the architect can, for example, install the GSD tool on a PaaS platform. Issues of middleware and methods for customizability of Saas applications have been discussed in (Walraven, Van Landuyt, Truyen, Handekyn & Joosen 2014, Walraven 2014).

8.3.3 Optimization of Cloud Resource due to changing Workload

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

Under-utilisation of resources in a cloud environment is possible for two main reasons: (i) when there are excess resources available on the cloud infrastructure, and (ii) when there are a small number of tenants accessing the application component (and data stored). The type of cloud deployment model and the particular type of resource that the architect intends to optimise could significantly affect the degree of multitenancy isolation. For example, elasticity can be restricted to a significant extent in a case where the GSD tool is deployed on a private cloud to support a small number of users.

As pointed out in the case study involving continuous integration with Hudson, CPU consumption of tenants changed significantly for the shared component. Therefore, on a private cloud which supports a small number of tenants, the shared component can be used to optimise CPU utilisation. However, there would be no guarantee of a high degree of isolation between tenants. In a continuous integration system, builds are known to consume a vast amount of disk I/O, and so running a large number of builds concurrently and running builds that are too complex should be performed on a dedicated component (Moser & O'Brien 2016). If the shared component is used to deploy Bugzilla (where mod_perl is enabled) on the same type of cloud infrastructure described

above, then minimising RAM consumption would be the main issue of concern, and not CPU (Bugzilla 2016).

Our recommended approach for optimising cloud resources while guaranteeing the required degree of isolation is through the use of a model-based decision support system. Models used in previous work have focused on minimising the cost of using cloud resources. Metaheuristics were not used for the optimisation and only in a few cases were simple heuristics used (Aldhalaan & Menascé 2015b)). The model integrated into our decisions considers optimising the required degree of isolation and metaheuristics were used to solve the model (Ochei, Petrovski & Bass 2016b).

8.3.4 Hybrid Cloud Deployment Conditions

There are situations where combining more than one multitenancy pattern is more suitable for implementing isolation between tenants. One example may be to handle hybrid deployment scenarios, for instance, integrating data residing on different clouds and static data centres. There are several cloud offerings such as Dropbox [1] and Microsoft's Azure StorSimple [2] that allow customers to integrate a cloud-based storage with a company's storage area network (SAN). Another scenario that is suitable for combining more than one multitenancy pattern is when different settings are applied concurrently to a particular software process. For example, settings could be applied to vary the frequency with which code files are submitted to a shared repository or lock certain software processes to prevent clashes between multiple tenants.

For example, builds or commits to a repository could be configured to run concurrently or at regular intervals. Running such builds as a long complete integration build in a slow network environment could take a lot of time and resources. To achieve a high degree of isolation while guaranteeing efficient resource utilization, the integration build can be split into two different stages, so that: (i) the first stage creates a commit build that compiles and verifies the absence of critical errors when each developer commits changes to the main development stream, and (ii) the second stage creates secondary build(s) to run slower and less important tests (Fowler 2016). As the result of case study one involving continuous integration showed, CPU consumption of

[1] Dropbox is a file hosting service that offers cloud storage, file synchronization, personal cloud, and client software (http://www.dropbox.com)

[2] Azure StoreSimple is a cloud storage service that integrates primary storage data deduplication, automated tiered storage of data across local and cloud storage (https://www.microsoft.com/en-gb/cloud-platform/azure-storsimple)

tenants changed significantly for the shared component; and so, the first stage of the build can use a dedicated component while the second stage of the build can use the shared component. This will ensure that secondary builds do not consume many resources and even if they fail, it will not also affect other tenants. Several other hybrid cloud deployment scenarios can be utilised to guarantee the required degree of multitenancy isolation[1].

8.3.5 Tagging Components with the Required Degree of Isolation

Components designed to use or integrated with a cloud-hosted service should be tagged as much as possible when there is need to implement the required degree of multitenancy isolation. Our approach achieves tagging by mapping our problem to an MMKP instance and associating each component with its required degree of isolation, thus allowing us to monitor and respond to workload changes efficiently. Tagging can be a challenging and cumbersome process and may not even be possible under certain conditions (e.g., in a case where the component is integrated into other services and are not within the control of the customer). Therefore, instead of tagging each component with an isolation value as required, this can also be predicted in a dynamic way. In our previous work (Ochei, Petrovski & Bass 2016a), an algorithm was developed which learns the features of existing components dynamically in a repository and then uses this information to associate each component with the appropriate degree of isolation. This information is crucial for making scaling decisions and optimisation of resources consumed by the components, especially in a real-time or dynamic environment.

Our approach is related in many ways to existing cloud offering such as Amazon's Auto Scaling and EC2 (Amazon 2016), and Microsoft Azure's Web Role (Microsoft 2016) where users can specify the different configurations for a component, for example, the number of components that can be deployed for a certain number of requests. However, users cannot tag each component with the required degree of isolation before deployment, as has been proposed in our approach.

8.3.6 Error Messages and Security Challenges during Implementation

Multitenancy isolation introduces significant error and security challenges in the cloud especially when the resources are shared among multiple tenants. Sharing of resources is closely associated with implementing the shared component, which does not guarantee a high degree of isolation

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

between the tenants. In this situation, if an error is triggered in the cloud service due to overload or insufficient resources, or if one of the tenants on the network is malicious, it can cause a denial of service and performance degradation for other tenants (Bass et al. 2013).

The type of error messages received from the case studies is a pointer to the key resources to consider in achieving the required degree of multitenancy isolation. For continuous integration, the error messages were related to insufficient system resources. For example, while implementing multitenancy isolation with Hudson, the most common error experienced was that of insufficient memory allocation. The cloud infrastructure did not cause this but it was partly caused by Hudson as it is not very optimised and also by the demands of the continuous integration process.

For the bug tracking process where bugs are stored in a database, the most common errors were related to resolving database errors, for example, exceeding the limit of file size, query, connections, etc. Therefore, it is necessary to modify the bug database to remove these restrictions. The bug database running on the VM instance can be quite sensitive to workload changes depending on the size, the volume of bugs, and the bug database isolation level. For the version control process, the most common error was that of insufficient memory and file or directory permission issues (e.g., when setting FTP request configurations). This problem becomes more acute when moving the VM image instance (whose file permission had been set on a local machine) to the cloud infrastructure. Therefore, it is necessary to get repository ownership and permission right before conducting the experiments.

8.4 Practical Applications

Some practical applications of our work are summarised below.

(1) Applicability of the Taxonomy of Cloud Deployment Patterns

Taxonomies are applied in software engineering to document theories that accumulate knowledge on software engineering (Sjoberg et al. 2007), and to carry out comparative studies involving tools and methods, for example, software evolution (Buckley et al. 2005) and Global Software Engineering (Smite et al. 2012). Our taxonomy can be used by a cloud architect to know the type of cloud deployment pattern to select and where the cloud pattern is targeting, for example, either the cloud-application or the cloud environment.

As stated earlier, most cloud patterns are related and have to be combined with others during implementation (Vlissides et al. 1995, Bass et al. 2013). One possible area of application is in selecting deployment patterns for deploying cloud services in a hybrid deployment scenario. This will be the case if data stored in different locations is to be integrated to form one cloud solution. For example, the cloud solution might require architecting the deployment to store some data in a particular format or to comply with certain regulations. These complexities would require a careful selection of cloud deployment patterns that would map the requirements to properties of the cloud infrastructure that cannot be changed and those that can be changed at the application level of the cloud-hosted solution.

The taxonomy can reduce the learning times of software architects. Collaborative work is made easier as similar cloud patterns for a particular project/task can be identified for use by practitioners in the organisation. Knowledge sharing can be automated more readily because similar and compatible cloud patterns would have been used while working on similar projects/tasks.

(2) Applicability of COMITRE Approach and Case Study Findings

One of the areas of applying COMITRE (a key outcome of the case study) is in software development, particularly in customising existing open-source tools and legacy systems. Most of these applications were not implemented using any multitenant architecture, and so they should be customised to support varying degrees of multitenancy isolation before deploying them to the cloud.

The several case studies allow us an in-depth understanding of the behaviour of different cloud-hosted services and the effects of varying degrees of multitenancy isolation of tenants or components. The case study synthesis has provided an explanatory framework and new insights into how multitenancy isolation affect a variety of software processes used to support Global Software Development. For example, the explanatory framework can help architects to (i) select suitable patterns to deploy services depending on the type of software processes they support; (ii) optimize cloud resources (e.g., CPU, memory) in a resource-constrained environment; and (iii) understand the trade-offs to consider when implementing multitenancy isolation.

(3) Applicability of the Model-based Decision Support System

Our proposed decision support system has several applications in the real cloud computing envi-

ronment. 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. Our decision support system can achieve this by setting a limit on the resources that are used to support each component (i.e., CPU, RAM, Disk and Bandwidth). After that, the decision support system 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 decision support system 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 [1] and Microsoft Azure Traffic Manager [2]), 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.

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

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

Our model-based decision support system can be customised to provide information for decommissioning of components 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.

8.5 Chapter Summary

This chapter has presented a discussion of results from each aspect of the study and after that the challenges and recommendations for achieving the required degree of isolation. Firstly, by creating a taxonomy and applying it to position a set selected of software tools, it demonstrates that appropriate deployment patterns can be identified together with the supporting technologies for deploying cloud-hosted services. It is observed that most deployment patterns are related and can be implemented by combining with others, for example, in hybrid deployment scenarios to integrate data residing in multiple clouds. Secondly, by empirically evaluating the varying degrees of multitenancy isolation in different case studies, it means that the approach developed in this study (i.e., COMITRE) has been applied not only to implement multitenancy but the required degree of multitenancy isolation between tenants. Thirdly, our model-based decision support system provides a model as well as a metaheuristic solution for providing near-optimal solutions for deploying components of a cloud-hosted service.

This chapter has also presented some concrete examples and application scenarios where our work can be applied. Our research can be applied in situations where there is need to select and combine two or more cloud deployment patterns together with their supporting technology for deploying cloud-hosted services. The case studies and their synthesis provide information to architects on a range of issues, such as how to implement multitenancy isolation in cloud-hosted services (e.g., GSD tools), recommended multitenancy patterns and general recommendations for implementing the required degree of multitenancy isolation. The model-based decision support system can be used to provide near-optimal solutions in (i) resource constrained environment where architects work with limited resources (e.g., CPU, memory); and (ii) a real-time/dynamic environment, where monitoring components and responding to workload changes is critical.

Chapter 9

Conclusion

9.1 Summary

This thesis has investigated how to architect the deployment of components of a cloud-hosted service for guaranteeing the required degree of multitenancy isolation. In previous chapters, this thesis has presented how this problem has been addressed from different perspectives. Chapter one introduced the thesis, while chapter two discussed the theoretical concepts used in the thesis and relevant related work. Chapter three focused on the methodology used for the research. This thesis applied a multimethod research approach by combining exploratory study, case study and case study synthesis, and simulation based on a model. Chapter four explains how to create and use a taxonomy of cloud deployment patterns to guide architects in selecting suitable deployment patterns together with associated technology for deploying cloud-hosted services. Chapter five presents the three case studies conducted to empirically evaluate the varying degrees of multitenancy isolation in different case studies of GSD processes. To generalise the results and explain observed variations and exceptions, a synthesis of findings from the three case studies in chapter six was carried out.

Chapter seven provides a model-based decision support system that combines a Queueing Network and optimisation model to provide near-optimal solutions for deploying components of a cloud-hosted service. Chapter eight first presents a discussion of each aspect of the work followed by recommendations for achieving the required degree of multitenancy isolation. The rest of this chapter is organised as follows: Section 9.2 revisits the research contributions of this thesis.

Section 9.3 discusses the scope and limitations of the research. Section 9.4 reflects on the PhD by discussing the challenges and lessons learned, while Section 9.5 is the future work. Section 9.6 is the conclusion of the thesis.

9.2 Research Contributions Revisited

This section revisits the main contributions of the thesis.

1. *A novel taxonomy and a general process for selecting applicable cloud deployment patterns (chapter 4):* Previous taxonomies of cloud patterns were not benchmarked to existing classifications and also not applied to applications in a particular domain. Current research has been extended by developing a taxonomy that is benchmarked to existing classifications: ISO/IEC 12207 taxonomy of software life cycle processes, components of a cloud model based on NIST cloud computing definition, NIST SP 800-145, and components of an architectural deployment structure (i.e., cloud-application and cloud environment).

   A further contribution is the application of our taxonomy to position a set of Global Software Development (GSD) tools and process, which also allowed us to identify technologies that can support the selected deployment patterns. Our taxonomy, unlike others (Wilder 2012, Homer et al. 2014), clearly shows where and how to select deployment patterns for addressing hybrid deployment scenarios, for example, integrating data residing in multiple cloud environments.

   The findings presented here support previous research that most patterns are related and so two or more patterns can be used together (Bass et al. 2013, Vlissides et al. 1995). By applying the taxonomy, recent technological trends in cloud deployment have been identified, for example, the use of plugin architectures for customization (Musser 2012, Atlassian.com 2016), preference for REST and messaging to interconnect GSD tools and other components. (Wilder 2012, Musser 2012)

2. *A novel approach, COMITRE, together with supporting algorithms for implementing varying degrees of multitenancy isolation (Chapter 5):* Previous research on implementing multitenancy assumes two extreme cases of isolation: shared isolation and dedicated isolation which are mostly implemented at the data tier (Chong et al. 2017, Vanhove et al. 2014,

Schneider & Uhle 2013, Schiller 2015), and so ignore the effect of varying degrees of isolation between tenants. Unlike previous research which focuses more on performance isolation (Krebs 2015, Krebs et al. 2013), our approach also applies to other aspects of isolation including resource consumption of tenants (e.g., CPU, RAM). Specifically, our approach extends the current research by considering the effect of varying degrees of multitenancy isolation for individual components of a cloud-hosted service under different cloud deployment conditions. For example, by capturing and configuring tenants request/transaction (i) implementation can be done at different levels (although more flexible at the application level for optimizing cloud resources); (ii) individual components can be monitored and adjusted to reflect changing workload; and (iii) trade-offs between the performance and resource consumption of tenants can be resolved easily depending on the required degree of isolation. One of the key benefits of our approach is that the underlying middleware and infrastructure do not necessarily need to be multitenant aware as the isolation is handled on the application level.

Similar to our approach, many providers implement techniques that capture client transactions/requests and decide what level of isolation is required. However, these approaches do not guarantee the availability and multitenancy isolation of specific components/individual IT resources (e.g., a particular virtual server or disk storage), but for the offering as a whole (e.g., starting new virtual servers) (Amazon 2017, Fehling et al. 2014). To address this problem, our approach initially tags each component and after that decides which isolation level is suitable for deploying a component based on the metadata of existing components in a component repository.

3. *Evaluating varying degrees of Multitenancy Isolation (Chapter 5):* This study extends current research by empirically evaluating the varying degrees of multitenancy isolation. Three case studies were conducted: continuous integration with Hudson, version control with File System SCM plugin and bug tracking with Bugzilla. Conducting more than one case study helps not only to strengthen the validity of our results but generalises our findings. (Cruzes & Dybå 2010, Cruzes & Dybå 2011). Previous research has applied multitenancy patterns/architectures (and cloud patterns generally) to simple web applications, for example, weblog applications (Moyer 2012, Homer et al. 2014) and largely ignored key factors that

could influence pattern selection such as application processes, workload and resource demands imposed on cloud service and cloud infrastructure on which the service is hosted.

The research makes several contributions in relation to multitenancy isolation:
(i) In case study 1, it was discovered that the shared component provides the lowest degree of isolation between other tenants when one of the tenants is exposed to demanding deployment conditions (e.g., large instant loads). There was no significant difference between the implementation of the tenant-isolated component and the dedicated component for a small number of build processes. The study concludes that when code files are checked into a shared repository at a low frequency to trigger a build process, then a high degree of isolation (regarding response times) is expected both for the tenant-isolated component and dedicated component. For the shared component, the degree of isolation is lower which means that it is more prone to performance effect when exposed to high load (Ochei, Bass & Petrovski 2015c).

(ii) In case study 2, it was discovered that when using a version control system, the dedicated component provides the highest degree of isolation between tenants (compared to the shared component and tenant-isolated component) regarding error% (i.e., the percentage of errors with unacceptably slow times) and throughput. While response times, CPU and memory consumption had the most negative impact on tenant isolation when exposed to large instant loads, system load of tenants showed no variability, and hence did not influence the degree of tenant isolation for all the three multitenancy patterns (Ochei, Petrovski & Bass 2015).

(iii) In case study 3, it was discovered that when requests/transactions are sent to the bug database where support for locking is enabled, performance isolation between tenants (e.g., in terms of response time) can be improved with a dedicated component while resource consumption (e.g., CPU and memory) can be reduced with a shared component. The study recommends that during bug tracking, the storage space should be reasonably large enough to accommodate bugs with large attachments. Bugs can be stored directly on disk while the file paths to the bugs are stored in the database table (Ochei, Bass & Petrovski 2016).

4. *An explanatory framework and new insights on multitenancy isolation (Chapter 6):* This study extends previous research by carrying out a synthesis of findings from the three case studies to find out: (i) commonalities that could be generalised; and (ii) differences that

would help explain the exceptions and variations. The commonalities include generation of additional data, use of locking, use of back-end storage, use of disk saving strategies, use of web server and runtime library, the size of users and project, and system load and CPU consumption. The differences include resource consumption, storage space, latency and bandwidth of clients accessing the server, type of GSD processes, storage format of the backend server, and inter-dependencies with other tools.

It has been argued that the shared component is better for resource utilisation while the dedicated component is better in avoiding performance interference (Fehling et al. 2014). And yet, as this experiment shows, there are certain GSD processes where that might not necessarily be so, for example, in version control, where additional copies of the files are created in the repository, thus consuming more disk space. Over time, performance begins to degrade as more time is spent searching across many files on the disk (Ochei, Petrovski & Bass 2015, Collins-Sussman et al. 2004).

A further contribution of this study is an explanatory framework for (i) mapping multitenancy isolation to different GSD processes, cloud resources and layers of the cloud service stack; (ii) explaining the different trade-offs to be considered for optimal deployment of components for guaranteeing the required degree of multitenancy isolation. Six trade-offs were identified for consideration while implementing multitenancy isolation: multitenancy isolation versus (resource sharing, number of users/requests, customizability, the size of generated data, the scope of control of the cloud application stack and business constraints.

5. *A model-based decision support system (DSS) (Chapter 7):* A clear contribution of this thesis to knowledge is the fact that the problem of tenants requiring varying degrees of multitenancy isolation is first modelled as a optimization model (which combines a QN model and combinatorial optimization) and then wrapped into a decision support system.

   This study focuses on an aspect of multitenancy isolation in a way that has not been done before. Previous research in multitenancy isolation has focused on a scenario where multiple tenants are accessing a component or cloud-hosted service and behave as if they were different tenants (Fehling et al. 2014, Krebs 2015, Walraven et al. 2012). Our research considers a particular case where multiple components of one tenant behave as if they were components of different tenants and, thus, are isolated from each other.

It was learnt that components designed to use (or integrated) with a cloud-hosted service should be tagged (or associated with a set of desired properties) as much as possible to achieve the required degree of multitenancy isolation especially when there is a possibility of frequent or sudden workload changes. This will allow each component (and the whole cloud-hosted service) to easily monitor and respond to workload changes in a timely and efficient manner.

6. *Metaheuristic solution for solving the optimisation model(Chapter 7):* This study contributes to knowledge and thus extends previous research by developing four variants of a metaheuristic technique and applying it to a new area, namely, provision of optimal solutions for deploying components of a cloud-hosted service for guaranteeing multitenancy isolation. Previous research focused on minimising the cost of using cloud infrastructure resources and used no metaheuristics (Shaikh & Patil 2014, Westermann & Momm 2010). In some cases, simple heuristics, and not metaheuristic were used (Aldhalaan & Menascé 2015b), to provide optimal solutions in a way that guarantees the required degree of multitenancy isolation.

   Performance evaluation showed that the variants of the metaheuristic that start with an initial greedy solution produce solutions that are robust and of better quality compared to metaheuristic that start with initial random solutions. However, there is a price to pay regarding the time and resources (i.e., computational effort) required to produce the optimal solutions, therefore making them unsuitable in real-time or dynamic environments where workload changes frequently.

9.3 Research Scope and Limitations

The scope and limitations of this research are summarised below:

1. The focus of this thesis is on using architectural patterns or cloud patterns to solve problems facing the deployment of components of a cloud-hosted service. More specifically, this thesis focuses on multitenancy architecture/patterns that capture the varying degrees of isolation between tenants (or components). Furthermore, the approach and the associated algorithms that are presented in this thesis apply to multitenancy patterns and other related

patterns that target the cloud-hosted service at the application level, and so are implemented almost at runtime.

2. The three case studies conducted to evaluate the varying degrees of multitenancy isolation empirically focused on well-known software processes (and not tools) used to support Global software development processes. Based on these processes, open-source GSD tools/plugins were selected that could be used to trigger these processes: Hudson for continuous integration, File System SCM Plugin for version control, and Bugzilla for bug tracking. The reason for using open-source tools is obvious because our interest was to modify the source code to allow us to implement multitenancy isolation.

3. This study focused on cloud-hosted GSD tools (e.g., Hudson) used for large-scale distributed enterprise software development projects. Therefore, other cloud-hosted services (e.g., cloud storage services, and document, video, audio, image sharing services) are outside the scope of this study, and so the findings of this study do not apply to all cloud-hosted services. Large software projects are usually executed with stable and reliable GSD tools, whereas for small software projects (with few developers and short duration), high performance and low cost may be the primary consideration in tool selection.

4. The number and size of requests sent to the application component during the experiments were within the limit of the private cloud used (i.e., Ubuntu Enterprise Cloud). Therefore, the results of this study apply to private clouds and should not be generalised to large public clouds. This study assumes that a small number of users send multiple requests to components of the GSD tool. For example, some GSD tools like Hudson (which is well known to consume a lot of memory) are not very optimised to accept a large number of requests, and so the most common error experienced during the experiments was that of insufficient memory allocation. Therefore, it is necessary to properly vary the setup values to get the maximum capacity of the software process triggered by the GSD tool (e.g., Hudson's build processes) running on the private cloud before conducting experiments.

5. The dataset (i.e., MMKP instances) used for the simulation experiments on the model-based decision system (DSS) were generated randomly following a standard approach used for similar problems. Also, we could not measure the overall computation time of the Op-

timalDep 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.

9.4 Reflection on the PhD

This section reflects on the conduct of the PhD by highlighting some challenges (e.g., choice of research methods, conducting the experiments, setting up a private cloud, etc.) and lessons learned in the process.

1. *Choosing a suitable research methodology:* The first major challenge was how to select an appropriate methodology for the research. The choice of the multimethod research strategy evolved over time during the research. After each phase of the work, it was realised that different methodologies were needed to solve the problem. By the time, the research was in the last phase (i.e., modelling and simulation), it was discovered that three key methodologies had been used: exploratory study, case study (with case study synthesis) and simulation based on a model.

   During the writing-up of the thesis, there was the challenge of pulling together the results and outcomes of the different research methodologies to form a coherent thesis. This problem was addressed with the guidance of the supervisory team. Over time, the required research methods could be selected and applied in the right order. It was also easy to determine how each method contributed to the overall research process in an interlinked fashion.

   It was learned that there should be careful consideration of the research questions and expected contributions early in the research to determine whether or not a single research method or multiple research methods would be adopted. It would be recommended that if a research student decides to use multimethod (i.e., multiple research methods) research, then it is important to carefully consider the time frame for the research and the availability of expertise to support the research process. Furthermore, it would be helpful to develop a flowchart early on during the research process to show how each method contributes to the

overall research process.

2. *Learning different programming languages and GSD tools:* The second major challenge was that of learning several programming languages and GSD tools (together with supporting plugins and libraries) at each phase of the PhD. This was very difficult and frustrating sometimes because I had to learn some of these tools concurrently. For example, the different GSD tools used were developed using different programming languages: Hudson is developed in Java, Subversion is developed in Python, and Bugzilla is developed in Perl. I also had to learn Linus/Ubuntu commands, Eucalyptus (i.e., an open-source software platform) administration commands for setting up and managing the private cloud, the database schema used for some GSD tools (e.g., Bugzilla bug database), load generator and testing tool (i.e., JMeter) and statistical tools (e.g., SPSS).

   In terms of having a sense of the implementation effort (e.g., man-hour, lines of code, number of classes) required to modify the GSD tools, it would be advised that researchers who may want to either repeat the experiments or experiment with other GSD tools should focus on the recommended approach by the original developers for modifying such tools. It is normal to expect that most (if not all) would present a detailed procedure for modifying and extending such tools. For example, the standard procedure recommended by Mozilla Foundation for modifying Bugzilla is presented on its website and associated manuals and textbooks (i.e., *the Bugzilla Guide*). The extension procedure defines a consistent API for extending the standard templates and source files in a way that separates standard code from extension code. This means that it is possible to write extensions that work across multiple versions of Bugzilla, making upgrading a Bugzilla installation with installed extensions easier as each extension is basically a simple directory structure (Bugzilla 2016). Therefore the implementation effort for modifying Bugzilla simply translates to knowing what to add/modify and then carefully studying the source code to identify the hook (i.e., a named place in a standard source file) where the extension code for that hook get processed. Thereafter, hooking an extension source file to the hook is simply putting the extension file into the extension's code directory.

3. *Setting up UEC Private Cloud:* Setting up a private cloud for the experiments was very difficult for me since there has been no one previously in RGU. I had to refer to several

reference materials and online sources to figure out how to setup a UEC private cloud for experiments. Because of this, I ended up spending so much time and effort moving back and forth until it was finally done.

It would be recommended that a PhD student intending to conduct experiments in a cloud environment should insist on having a private cloud setup within the school so that the IT staff can support the student with the technical aspects of managing the private cloud. I envisage that if I were to do the PhD again, I would insist that an alternative arrangement be put in place with another University or research centre that has a private cloud, at least to offer some technical support.

4. *Conducting the experiments in a private cloud:* Before conducting the experiments, I had to study the source code of each of the GSD tools to identify the extension points or modules that can be modified to implement multitenancy isolation. Many times, the code documentation did not match the actual source code that was downloaded from the code repository. There were also permission and login issues before being able to download and update modified versions of these tools.

   During the experiment, there were several conflicting error messages that were triggered while running the modified GSD tool on the cloud. Some of the errors were associated with: (i) the GSD tool (ii) the plugin used by the GSD tool (iii) operating system used (iv) the database used and (iv) the load generating and testing tool. To resolve these errors, I had to check and analyse comments from users on the online developer community that were working actively on the tool.

   It was learnt that knowing the sequence of what to learn is as important as knowing what to learn during the PhD process, especially when using a multimethod research strategy. For example, I spent too much time learning how to use publicly available cloud platforms like Amazons Elastic Bean Stalk and Microsoft Azure instead of the programming language, database schema used to design the GSD tools and the commands for managing the private cloud.

5. *Incompatible Plugins and Libraries:* Some plugins and libraries used to install and configure GSD tools (e.g., Hudson) are either obsolete or no longer maintained by developers

community. This is because the development team of some open-source GSD tools (e.g., Hudson) have split and so some of the tools/libraries and procedures for installing and configuring the plugins have also changed.

It would be recommended that when using software tools that require re-configuration with plugins, it would be necessary to use: (i) stable versions of the plugins/libraries before the split; (ii) plugins certified by the developers community; and (iii) and plugins that have a well-defined set of procedures for installation and configuration. Furthermore, downloading and installing incompatible plugins and libraries required to execute software tools like Hudson and Bugzilla could corrupt the source code of the GSD tools and even stop other supporting programs/processes from running. To avoid this, it would be advisable to prevent automatic system updates and also shield the private cloud from public access.

6. *Frequent Crashes of Private Cloud and Software Tools:* One of the most difficult challenges I encountered while conducting the experiments were the frequent crashes; for instance, the GSD tool shutting down, a plugin ceasing to run, a database being corrupted and even the private cloud shutting down. Each time this happened, I would have to restart the whole process and in some cases, re-configure some GSD tools before continuing with the experiments. For example, before conducting the experiments with Hudson, I varied the setup of the testing environment by sending load/requests to the GSD tool to determine the capacity of the UEC private cloud and whether it could give the required effect for the experiment.

   It was initially difficult for me to address the technical and network issues of the private cloud because of lack of expertise in setting up and managing a private cloud. By referencing online developer communities such as askubuntu, stackoverflow etc., I was able to get help in resolving errors while setting up and conducting the experiments.

9.5 Conclusion

This thesis has investigated how to architect the optimal deployment (i.e., regarding performance, resources consumption and access privileges) of components of a cloud-hosted service to serve multiple users in a way that guarantees the required degree of isolation between tenants (or components) when one of the tenants (or components) experiences a high workload.

The key solution created for the multitenancy isolation problem can be looked at as a *framework* composed of three main components: (i) a taxonomy and general process for selecting applicable deployment patterns together with supporting technologies for deploying services to the cloud, (ii) a COMITRE approach for implementing the varying degrees of multitenancy isolation for cloud-hosted services, and (iii) a model-based decision support system together with metaheuristics for providing for optimal solutions to deploy service to the cloud.

Several lessons have been learnt in relation to implementing the required the degree of multitenancy isolation. Firstly, it was learned in this study that deploying components of a cloud -hosted application would require reference to the taxonomy to help in deciding whether to target a cloud-hosted application or cloud-hosted environment. The positioning of a set of GSD tools on the taxonomy has shown how our taxonomy can guide architects in selecting suitable cloud deployment patterns for deploying services to the cloud.

Secondly, it was learnt that before deploying a component to the cloud for guaranteeing the required degree of multitenancy isolation, there is need to evaluate the effect of the varying degrees of isolation of tenants (components) on the performance and resource consumption of components. The COMITRE approach allows us to implement and evaluate the varying degrees of multitenancy isolation, provide explanations for commonalities and differences that exist in different case studies as well as the trade-offs to consider when implementing multitenancy isolation.

Thirdly, the optimal deployment of components can be achieved by using a model-based decision support system (DSS) to maximise both the required degree of multitenancy isolation and the number of requests allowed to access the components of a cloud-hosted service. Different variants of the metaheuristic solutions have also been presented to support the model-based DSS in providing optimal solutions for deploying components of cloud-hosted service for guaranteeing multitenancy isolation.

9.6 Future Work

There are many future directions for our research as explained in the following section.

9.6.1 Multitenancy Isolation Problem: Exploring other Models and Metaheuristics

The model created in this study is an analytic model composed of a set of formulas and computational algorithms to solve the problem instances. This work can be extended by developing a simulation model (or a *simulator*) which is based on computer programs that emulate the different dynamic aspects of a system as well as their static structure (Menasce et al. 2004). Although many providers offer similar functionality in the form of rule-based algorithms (Amazon's Auto-Scaling [1]) and Microsoft's Windows Azure Traffic Manager [2]) to configure the scaling functionality of the cloud-hosted services (Amazon.com 2017, Microsoft.com 2016), these offerings do not implement the varying degrees of multitenancy isolation for individual components.

In our case, an architect can specify that a new set of components be selected for deployment either once an average utilization of components/whole system exceeds a defined threshold or once the arrival rate of requests exceeds a defined threshold. The simulation model will allow the different behaviour or aspects of the system to be captured and evaluated when there are workload changes. The workload for testing the model can be generated randomly or an observed trace script from a real cloud-environment can be used. Consider the following questions which can be answered using the simulation model:

- How many requests can be allowed to access a specific component or the whole application based on the required degree of isolation.
- Assuming the required degree of isolation is the shared component pattern, what is the correct ratio between CPU and RAM for optimal deployment of components.
- What is the optimal solution (i.e., a suitable configuration of components) for deployment once an average utilization of a particular component (or group of components) exceeds a defined threshold.

[1] Auto Scaling ensures that the correct number of Amazon EC2 instances is available to handle application load.

[2] Microsoft Azure Traffic Manager allows you to control the distribution of user traffic for service endpoints in different datacenters.

These types of questions would be easier to answer using simulation models that pure mathematical models because several architectural parameters can be turned into constants and ranges. Also, it can be used to analyse the architectural design space of cloud-hosted services in situations where requirements are often difficult and complex to interpret and could change suddenly due to workload interference (Bass et al. 2013).

It would also be interesting to consider integrating other types of metaheuristics into the OptimalDep algorithm (i.e., the main algorithm driving the decision support system) or combining simple heuristics with more advanced metaheuristics. Several research work have developed algorithms that combine genetic algorithm (GA)(i.e., a population-based algorithm) with simulated annealing (SA) for solving various optimization problems (Gan, Huang & Gao 2010, Chen, Jiang, Chen & Zhang 2012). For example, the authors in (Chen et al. 2012) have developed the GA-SA-combined algorithm, an algorithm that combines genetic algorithm with simulated annealing for optimization of wideband antenna matching networks. In their algorithm, the GA starts the initial phase of the optimization and provides its values as the initial parameters of SA which forms the second phase of the optimization; as a result, fast convergence with an optimized solution is expected. The basis for this combination is that GA is not sensitive to initial starting parameters, and so it can iterate fast to nearby optimal solution for the SA to take over. Due to the good initial parameters from GA, only a small number of iterations are needed for SA to obtain the optimal solution.

9.6.2 Predicting QoS of Components based on Required Degree of Isolation

Another area where our research can be extended is to incorporate a module into our decision support system for evaluating and predicting the effect of the required degree of multitenancy isolation on different Quality of Service (QoS) of functional properties of components. Apart from performance (which as considered in this study), another QoS attribute whose analytic framework is well-understood is availability (Bass et al. 2013). Availability can be used to indicate the uptime of a system (or components of a system) over a sufficiently long duration. Availability can be expressed as:

$$\frac{MTBF}{(MTBF + MTTR)} \tag{9.1}$$

where MTBF is the *mean time between failure*, which is derived based on the expected value of the implementations' failure probability density function (PDF), and MTTR refers to the *mean time to repair*.

Increasing the availability of a component (or whole system) deployed using a particular multitenancy pattern can be achieved using different types of fault recovery tactics. Most fault recovery tactics rely on introducing a backup copy of a component that will take over in case the primary component suffers a failure (Scott & Kazman 2009). For example, in an active redundancy, a redundant spare, which possesses an identical state to the active processor, can take over from a failed component in a matter of milliseconds. These tactics differ primarily in how long it takes to bring the backup copy up to speed, and so the MTTR will be where the difference among the tactics shows up. The availability of components is of particular importance because many cloud providers do not guarantee the availability of individual cloud resources (e.g., CPU, RAM, disk servers, bandwidths, virtual servers), but only for the whole cloud offering (e.g., the ability to start a new virtual server) (Fehling et al. 2014, Ochei, Petrovski & Bass 2016a).

9.6.3 Multitenancy Isolation: exploring different scenarios, tools and processes

Our research focused on evaluating the effect of multitenancy isolation on a software process invoked by a cloud-hosted GSD tool. Furthermore, our approach to achieving multitenancy isolation was implemented at the application level where the request ID is captured and re-routed to a separate component that adjusts the configuration of the system to the required degree of isolation. This research can be extended in the following directions:

1. *Conducting case studies with other cloud-hosted software tools and processes:*. An interesting option would be agile management tools and support processes. The agile software development process is now widely used by large scale distributed enterprises with rapidly evolving requirements and short time-to-market constraints (Fairbanks 2010, Bass et al. 2013). Code refactoring, for example, is the mainstay practice of agile development projects. Based on the required degree of multitenancy isolation, there may be an interest in comparing the performance and resource consumption of each code refactoring task to the base system during every proposed improvement before deploying it to the cloud.

2. *Conducting case studies with other cloud-hosted services:* There are several classes of widely used cloud-hosted services/tools that can be experimented with. These tools are deployed on the cloud to serve multiple users and so would require multitenancy isolation. Notable examples include customer relationship management systems (e.g., SalesforceIQ, and Sales cloud) and content management system (e.g., WordPress). In this case, the focus will be on evaluating the effect of multitenancy isolation on the data generated from these tools.

3. *Conducting case studies with other cloud deployment scenarios and indicators:* There are other performance indicators that could affect multitenancy isolation that could also be explored. These include the effect of different file system formats; the number and size of data generated or stored; and concurrent running processes. For example, the performance of version control systems like Subversion can be affected by the type of file system format used to store artefacts (Ben Collins-Sussman 2011).

4. *Conducting case studies using human subjects:* The different multitenancy patterns can be evaluated in multi-user collaborations involving cloud-hosted GSD tools such as Hudson and Bugzilla. For example, developers can carry out a collaborative task dealing with a software project where developers are working on multiple branches simultaneously using a version control tool such as (e.g., subversion). Several research work have evaluated the performance/response times of operations made by remote users of a collaborative system such as a LiveMeeting/Webex shared application, instant messenger, and checkers (Junuzovic & Dewan 2006, Junuzovic, Chung & Dewan 2005). In addition to configuring the multitenancy patterns as a network, this can also be done in terms of the order in which the messages are exchanged between tenants and the system. This will help in selecting a suitable type of collaboration architecture to be used for certain types of services or processes that are shared by multiple users.

Bibliography

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

Aiken, L. (2017). Why multi-tenancy is key to successful and sustainable software-as-a-service (saas). [Online: accessed in February 2017 from http://www.cloudbook.net/resources/stories/.

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

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

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

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

Amazon (2017). Amazon elastic compute cloud (ec2) documentation. [Online: accessed in February 17, 2017 from https://aws.amazon.com/documentation/ec2/].

Amazon.com (2017). What is auto scaling? [Online: accessed in March 8, 2017 from http://docs.aws.amazon.com/autoscaling/.

Arista.com (2014). Cloud networking: Design patterns for cloud-centric application environments.
URL: *http://www.arista.com/assets/data/pdf/CloudCentric...pdf*

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., 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.
URL: *http://doi.acm.org/10.1145/1721654.1721672*

Aspray, W., Mayadas, F., Vardi, M. Y. et al. (2006). Globalization and offshoring of software, *Report of the ACM Job Migration Task Force, Association for Computing Machinery* .

Atlassian.com (2016). Atlassian documentation for jira 6.1. Online: accessed in November, 2015 from https://www.atlassian.com/software/jira/.

Babar, M. A. & Zahedi, M. (2012). Global software development: A review of the state-of-the-art (2007–2011), *Technical report*, IT University of Copenhagen.

Badger, L., Grance, T., Patt-Corner, R. & Voas, J. (2012). Cloud computing synopsis and recommendations, *NIST special publication* **800**: 146.

Banati, H. & Bajaj, M. (2013). Performance analysis of firefly algorithm for data clustering, *International Journal of Swarm Intelligence* **1**(1): 19–35.

Barrero, D. F., Muñoz, P., Camacho, D. & R-Moreno, M. D. (2015). On the statistical distribution of the expected run-time in population-based search algorithms, *Soft Computing* **19**(10): 2717–2734.

Bass, J. (2014). How product owner teams scale agile methods to large distributed enterprises, *Empirical Software Engineering* pp. 1–33.

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

Bauer, E. & Adams, R. (2012). *Reliability and availability of cloud computing*, John Wiley & Sons.

Bazeley, P. (2006). The contribution of computer software to integrating qualitative and quantitative data and analyses, *Research in the Schools* **13**(1): 64–74.

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

Ben Collins-Sussman, Brian W. Fitzpatrick, C. P. (2011). Version control with subversion. for subversion 1.7(compiled from r4543), [Online: accessed in November 2013 from http://www.svnbook.red-bean.com/en/1.7/svn-book.pdf].
URL: *svnbook.red-bean.com/en/1.7/svn-book.pdf*

Braga, R. M., Mattoso, M. & Werner, C. M. (2001). The use of mediation and ontology technologies for software component information retrieval, *ACM SIGSOFT Software Engineering Notes*, Vol. 26, ACM, pp. 19–28.

Braga, R. M., Werner, C. M. & Mattoso, M. (2006). Odyssey-search: A multi-agent system for component information search and retrieval, *Journal of Systems and Software* **79**(2): 204–215.

Brandle, C., Grose, V., Young Hong, M., Imholz, J., Kaggali, P. & Mantegazza, M. (2014). Cloud computing patterns of expertise. International Technical Support Organization.
URL: *ibm.com/redbooks*

Buckley, J., Mens, T., Zenger, M., Rashid, A. & Kniesel, G. (2005). Towards a taxonomy of software change, *Journal of Software Maintenance and Evolution: Research and Practice* **17**(5): 309–332.

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

Buyya, R., Broberg, J. & Goscinski, A. (2011). *Cloud Computing: Principles and Paradigms*, John Wiley & Sons, Inc.

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

CERMSEM (2017). Index of /pub/cermsem/hifi/mmkp/. [Online: accessed in May 14,2017 from ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/MMKP/.

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

Chen, A., Jiang, T., Chen, Z. & Zhang, Y. (2012). A genetic and simulated annealing combined algorithm for optimization of wideband antenna matching networks, *International Journal of Antennas and Propagation* **2012**.

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

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

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].

Chong, F., Carraro, G. & Wolter, R. (2017). Multi-tenant data architecture. [Online: accessed in February 15, 2017 from https://msdn.microsoft.com/en-us/library/aa479086.aspx.

Cohen, P. (1995). Empirical methods for artificial intelligencemit press, *Cambridge, MA* .

CollabNet (n.d.). Subversionedge for the enterprise. [Online: accessed in November, 2015 from http://www.collab.net/products/subversion].

Collier, D. & Elman, C. (2008). Qualitative and multi-method research: Organizations, publication, and reflections on integration.

Collins-Sussman, B., Fitzpatrick, B. & Pilato, M. (2004). *Version control with subversion*, O'Reilly.

Connolly, B. (2004). Capturing and analyzing client transaction metrics for .net-based web services, *MSDN Magazine* .

URL: *https://msdn.microsoft.com/magazine/msdn-magazine-issues*

Corbet, J. (2009). Distributed bug tracking. [Online: accessed in December, 2016 from https://lwn.net/Articles/281849/].

Cruzes, D. S. & Dybå, T. (2010). Synthesizing evidence in software engineering research, *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM, p. 1.

Cruzes, D. S. & Dybå, T. (2011). Research synthesis in software engineering: A tertiary study, *Information and Software Technology* **53**(5): 440–455.

Cruzes, D. S., Dybå, T., Runeson, P. & Höst, M. (2015). Case studies synthesis: a thematic, cross-case, and narrative synthesis worked example, *Empirical Software Engineering* **20**(6): 1634–1665.

Doddavula, S. K., Agrawal, I. & Saxena, V. (2013a). *Cloud Computing: Methods and Practical Approaches*, Springer, chapter Cloud Computing Solution Patterns: Infrastructural Solutions, pp. 197–219.
URL: *http:// www.springer.com/series/4198*

Doddavula, S. K., Agrawal, I. & Saxena, V. (2013b). Cloud computing solution patterns: Infrastructural solutions, *Cloud Computing: Methods and Practical Approaches*, Springer, pp. 197–219.

Dupuis, R. (2004). Software engineering body of knowledge.

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

Electric-Cloud (2016). Build automation: Top 3 problems and how to solve them. [Online: accessed in November, 2016 from http://electric-cloud.com/plugins/build-automation].

Erinle, B. (2013). *Performance Testing with JMeter 2.9*, Packt Publishing Ltd.

Erl, T. & Naserpour, A. (2014). *Cloud Computing Design Patterns*, Prentice Hall.

Fairbanks, G. (2010). *Just enough software architecture: a risk-driven approach*, Marshall & Brainerd.

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

Field, A. (2013). *Discovering statistics using IBM SPSS statistics*, Sage.

Fowler, M. (2016). Continuous integration - white paper.
URL: *http://www.martinfowler.com/....html*

Fowler, M. (2017). Continuous integration. [Online: accessed in May 14,2017 from https://www.thoughtworks.com/continuous-integration.

Fox, A., Patterson, D. A. & Joseph, S. (2013). *Engineering software as a service: an agile approach using cloud computing*, Strawberry Canyon LLC.

Gan, G.-n., Huang, T.-l. & Gao, S. (2010). Genetic simulated annealing algorithm for task scheduling based on cloud computing environment, *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*, IEEE, pp. 60–63.

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

Glass, R. L. & Vessey, I. (1995). Contemporary application-domain taxonomies, *Software, IEEE* **12**(4): 63–76.

Google (2017). Google cloud platform and the eu data protection directive. [Online: accessed in February, 2017 from https://cloud.google.com/security/compliance/eu-data-protection/.

Goth, G. (2008). Software-as-a-service: The spark that will change software engineering?, *IEEE Distributed Systems Online* **9**(7).

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

Han, B., Leblet, J. & Simon, G. (2010). Hard multidimensional multiple choice knapsack problems, an empirical study, *Computers & operations research* **37**(1): 172–181.

Hanmer, R. (2013). *Patterns for fault tolerant software*, John Wiley & Sons.

Hansma, S. (2012). Go fast and don't break things: Ensuring quality in the cloud., *Workshop on High Performance Transaction Systems(HPTS 2011), Asilomar, CA, October 2011. Summarized in Conference Reports column of USENIX; login 37(1), February 2012.*

Hellstrom, D. & Nilsson, F. (2006). Combining case study and simulation methods in supply chain management research, *15th Annual IPSERA Conference*.

Herbsleb, J. D. (2007). Global software engineering: The future of socio-technical coordination, *2007 Future of Software Engineering*, IEEE Computer Society, pp. 188–198.

Herbsleb, J. D. & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development, *Software Engineering, IEEE Transactions on* **29**(6): 481–494.

Herbst, N., Krebs, R., Oikonomou, G., Kousiouris, G., Evangelinou, A., Iosup, A. & Kounev, S. (2016). Ready for rain? a view from spec research on the future of cloud metrics, *arXiv preprint arXiv:1604.03470* .

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

Hohpe, G. & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*, Addison-Wesley Professional.

Homer, A., Sharp, J., Brader, L., Narumoto, M. & Swanson, T. (2014). *Cloud Design Patterns*, Microsoft.

Hon, K. & Millard, C. (2017). Eu data protection law and the cloud. [Online: accessed in February, 2017 from https://iapp.org/resources/article/.

Hoos, H. H. & Stützle, T. (2004). *Stochastic local search: Foundations & applications*, Elsevier.

Hoos, H. & Stutzle, T. (1998). Characterizing the run-time behavior of stochastic local search, *In Proceedings AAAI99*, Citeseer.

Howitt, D. & Cramer, D. (2011). *Introduction to research methods in psychology*, Pearson Education.

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

Huberman, M. & Miles, M. B. (2002). *The qualitative researcher's companion*, Sage.

Hudson (2016a). Apache software foundation. [Online: accessed in January, 2016 from http://wiki.hudson-ci.org//display/HUDSON/Files+Found+Trigger].

Hudson (2016b). Hudson-ci/writing-first-hudson-plugin. [Online: accessed in November, 2016 from https://wiki.eclipse.org/Hudson-ci/writing-first-hudson-plugin].

Hudson (2016c). Hudson extensible continuous integration server. [Online: accessed in November, 2015 from http://www.hudson-ci.org].

Hudson (2017). Hudson extensible continuous integration server. [Online: accessed in February, 2017 from http://hudson-ci.org/.

IEEE (1990). Ieee standard glossary of software engineering terminology, *Office* **121990**(1): 1.

IEEE (2017). Cloud profiles working group (cpwg), [Online: accessed in February 2015 from http://standards.ieee.org/develop/wg/CPWG-2301...WG.html].

Jamshidi, P., Pahl, C., Chinenyeze, S. & Liu, X. (2015). Cloud migration patterns: a multi-cloud service architecture perspective, *Service-Oriented Computing-ICSOC 2014 Workshops*, Springer, pp. 6–19.

Johnson, D., Kiran, M., Murthy, R., Suseendran, R. & Yogesh, G. (2016). Eucalyptus beginner's guide - uec edition. [Online: accessed in February, 2017 from http://www.csscorp.com/eucauecbook.

Junuzovic, S., Chung, G. & Dewan, P. (2005). Formally analyzing two-user centralized and replicated architectures, *ECSCW 2005*, Springer, pp. 83–102.

Junuzovic, S. & Dewan, P. (2006). Response times in n-user replicated, centralized, and proximity-based hybrid collaboration architectures, *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ACM, pp. 129–138.

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

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

Khan, M. F., Mirza, A. U. et al. (2012). An approach towards customized multi-tenancy, *International Journal of Modern Education and Computer Science* **4**(9): 39.

Khan, S., Li, K. F., Manning, E. G. & Akbar, M. M. (2002). Solving the knapsack problem for adaptive multimedia systems, *Stud. Inform. Univ.* **2**(1): 157–178.

Khazaei, H., Misic, J. & Misic, V. B. (2012). Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems, *Parallel and Distributed Systems, IEEE Transactions on* **23**(5): 936–943.

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

Krebs, R. (2015). *Performance Isolation in Multi-Tenant Applications*, PhD thesis, Karlsruhe Institute of Technology.

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

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

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

Krishna, R. & Jayakrishnan, R. (2013). Impact of cloud services on software development life cycle, *Software Engineering Frameworks for the Cloud Computing Paradigm*, Springer, pp. 79–99.

Kurmus, A., Gupta, M., Pletka, R., Cachin, C. & Haas, R. (2011). A comparison of secure multi-tenancy architectures for filesystem storage clouds, *Proceedings of the 12th International Middleware Conference*, International Federation for Information Processing, pp. 460–479.

Laerd.com (2017). Two-way anova in spss statistics. [Online: accessed in February, 2017 from https://statistics.laerd.com/spss-tutorials/.

Lanubile, F. (2009). Collaboration in distributed software development, *Software Engineering*, Springer, pp. 174–193.

Lanubile, F., Ebert, C., Prikladnicki, R. & Vizcaíno, A. (2010). Collaboration tools for global software engineering, *Software, IEEE* **27**(2): 52–55.

Larman, C. & Vodde, B. (2010). *Practices for scaling lean and agile development: large, multisite, and offshore product development with large-scale Scrum*, Pearson Education.

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

Leymann, F., Fehling, C., Mietzner, R., Nowak, A. & Dustdar, S. (2011). Moving applications to the cloud: an approach based on application model enrichment, *International Journal of Cooperative Information Systems* **20**(03): 307–356.

Li, C. (2012). *A Holistic Semantic Based Approach to Component Specification and Retrieval*, PhD thesis, Edinburgh Napier University.

Lilien, L. (2007). A taxonomy of specialized ad hoc networks and systems for emergency applications, *Mobile and Ubiquitous Systems: Networking & Services, 2007. MobiQuitous 2007. Fourth Annual International Conference on*, IEEE, pp. 1–8.

Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L. & Leaf, D. (2011). Nist cloud computing reference architecture, *NIST special publication* **500**: 292.

Mahmood, Z. (2013). *Cloud Computing: Methods and Practical Approaches*, Springer-Verlag London.

Martello, S. & Toth, P. (1987). Algorithms for knapsack problems, *North-Holland Mathematics Studies* **132**: 213–257.

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

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

Mehta, A. (2017a). Implementing a multi-tenancy architecture, tier by tier. [Online: accessed in January, 2017 from http://www.devx.com/architect/Article/47708/.

Mehta, A. (2017b). Multi-tenancy for cloud architectures: Benefits and challenges. [Online: accessed in January, 2017 from http://www.devx.com/architect/Article/47798/.

Mehta, A. (2017c). Successful strategies for a multi-tenant architecture. [Online: accessed in January, 2017 from http://www.devx.com/architect/Article/47662.

Mell, P. & Grance, T. (2011). The nist definition of cloud computing, *NIST special publication* **800**(145): 7.

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

Mendonca, N. C. (2014). Architectural options for cloud migration, *Computer* **47**(8): 62–66.

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

Microsoft.com (2016). Overview of traffic manager. [Online: accessed in March 2017 from https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-overview.

Mietzner, R., Unger, T., Titze, R. & Leymann, F. (2009). Combining different multi-tenancy patterns in service-oriented applications, *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, IEEE, pp. 131–140.

Milenkoski, A., Iosup, A., Kounev, S., Sachs, K., Rygielski, P., Ding, J., Cirne, W. & Rosenberg, F. (2013). Cloud usage patterns: A formalism for description of cloud usage scenarios, *Technical report*, Standard Performance Evaluation Corporation(SPEC) Research Cloud Working Group.

Miles, M. B. & Huberman, A. M. (1994). *Qualitative data analysis: An expanded sourcebook*, Sage.

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, *Journal of Network and Systems Management* **22**(4): 517–558.

Momm, C. & Krebs, R. (2011). A qualitative discussion of different approaches for implementing multi-tenant saas offerings., *Software Engineering (Workshops)*, Vol. 11, pp. 139–150.

Morse, J. M. (2003). Principles of mixed methods and multimethod research design, *Handbook of mixed methods in social and behavioral research* **1**: 189–208.

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

Moyer, C. (2012). *Building Applications for the Cloud: Concepts, Patterns and Projects*, Addison-Wesley Publishing Company, Pearson Education, Inc, Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, USA.

MSDN (2016). Multi-tenant data architecture. [Online: accessed in December, 2016 from https://msdn.microsoft.com/en-gb/library/hh534480.aspx].

Musser, J. (2012). Enterprise-class api patterns for cloud and mobile.

Nourani, Y. & Andresen, B. (1998). A comparison of simulated annealing cooling strategies, *Journal of Physics A: Mathematical and General* **31**(41): 8373.

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

Ochei, L. C., Bass, J. M. & Petrovski, A. (2015b). Taxonomy of deployment patterns for cloud-hosted applications: A case study of global software development (gsd) tools, *The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2015)*, IARIA, pp. 86–93.

Ochei, L. C., Bass, J. & Petrovski, A. (2015c). Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools, *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, IEEE, pp. 101–112.

Ochei, L. C., Bass, J. & Petrovski, A. (2016). Implementing the required degree of multitenancy isolation: A case study of cloud-hosted bug tracking system, *13th IEEE International Conference on Services Computing (SCC 2016)*, IEEE.

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

Ochei, L. C., Petrovski, A. & Bass, J. (2016a). An approach for achieving the required degree of multitenancy isolation for components of a cloud-hosted application, *4th International IBM Cloud Academy Conference (ICACON 2016)*.

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

Oracle (2017). Oracle database concepts 10g release 1 (10.1). [Online: accessed in February, 2017 from http://docs.oracle.com/.

Pantić, Z. & Babar, M. A. (2012). Guidelines for building a private cloud infrastructure, *IT University of Copenhagen, Denmark, Copenhagen, Denmark* .

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

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

Pesola, J.-P., Tanner, H., Eskeli, J., Parviainen, P. & Bendas, D. (2011). Integrating early v and v support to a gse tool integration platform, *Global Software Engineering Workshop (ICGSEW), 2011 Sixth IEEE International Conference on*, IEEE, pp. 95–101.

Pirkwieser, S. & Raidl, G. R. (2008). A variable neighborhood search for the periodic vehicle routing problem with time windows, *Proceedings of the 9th EU/meeting on metaheuristics for logistics and vehicle routing, Troyes, France*, pp. 23–24.

Pope, C., Mays, N. & Popay, J. (2007). *Synthesising Qualitative and Quantitative Health Evidence: A Guide to Methods: A Guide to Methods*, McGraw-Hill Education (UK).

Portillo-Rodriguez, J., Vizcaino, A., Ebert, C. & Piattini, M. (2010). Tools to support global software development processes: a survey, *Global Software Engineering (ICGSE), 2010 5th IEEE International Conference on*, IEEE, pp. 13–22.

Ragin, C. C. (2004). Turning the tables: How case-oriented research challenges, *Rethinking social inquiry: Diverse tools, shared standards* **123**.

Robson, C. & McCartan, K. (2016). *Real world research*, John Wiley & Sons.

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

Runeson, P. & Host, M. (2009). Guidelines for conducting and reporting case study research in software engineering, *Empirical software engineering* **14**(2): 131–164.

Runeson, P., Host, M., Rainer, A. & Regnell, B. (2012). *Case study research in software engineering: Guidelines and examples*, John Wiley & Sons.

Sawant, N. & Shah, H. (2013). *Big Data Application Architecture - A problem Solution Approach*, Apress.

Schiller, O. (2015). Supporting multi-tenancy in relational database management systems for oltp-style software as a service applications.

Schneider, M. & Uhle, J. (2013). Versioning for software as a service in the context of multi-tenancy.

Scott, J. & Kazman, R. (2009). Realizing and refining architectural tactics: Availability, *Technical report*, Technical Report CMU/SEI-2009-TR-006.

Seacord, R. C., Hissam, S. A. & Wallnau, K. C. (1998). Agora: A search engine for software components, *IEEE Internet computing* **2**(6): 62.

Serrano, N. & Ciordia, I. (2005a). Bugzilla, itracker, and other bug trackers, *Software, IEEE* **22**(2): 11–13.

Serrano, N. & Ciordia, I. (2005b). Bugzilla, itracker, and other bug trackers, *Software, IEEE* **22**(2): 11–13.

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

Sheridan, J. C. & Ong, C. (2011). Spss version 18.0 for windows-analysis without anguish.

Shull, F. & Feldmann, R. L. (2008). Building theories from multiple evidence sources, *Guide to Advanced Empirical Software Engineering*, Springer, pp. 337–364.

Singh, R. (1996). International standard iso/iec 12207 software life cycle processes, *Software Process Improvement and Practice* **2**(1): 35–50.

Sjoberg, D. I., Dyba, T. & Jorgensen, M. (2007). The future of empirical methods in software engineering research, *Future of Software Engineering, 2007. FOSE'07*, IEEE, pp. 358–378.

Sliwko, L. & Getov, V. (2015). A meta-heuristic load balancer for cloud computing systems, *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, Vol. 3, IEEE, pp. 121–126.

Smite, D., Wohlin, C., Galvina, Z. & Prikladnicki, R. (2012). An empirically based terminology and taxonomy for global software engineering, *Empirical Software Engineering* pp. 1–49.

Sommerville, I. (2011). *Software Engineering*, Pearson Education, Inc. and Addison-Wesley.

Stephens, R. (2015). *Beginning software engineering*, John Wiley & Sons.

Stol, K.-J., Avgeriou, P. & Babar, M. A. (2011). Design and evaluation of a process for identifying architecture patterns in open source software, *Software Architecture*, Springer, pp. 147–163.

Strauch, S., Andrikopoulos, V., Leymann, F. & Muhler, D. (2012). Esb mt: Enabling multi-tenancy in enterprise service buses, *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, IEEE, pp. 456–463.

Strauch, S., Breitenbuecher, U., Kopp, O., Leymann, F. & Unger, T. (2012). Cloud data patterns for confidentiality, *Proceedings of the 2nd International Conference on Cloud Computing and Service Science, CLOSER 2012, 18-21 April 2012, Porto, Portugal*, SciTePress, pp. 387–394.

Subversion (2016). Working copy metadata storage improvements (client). [Online: accessed in November, 2016 from http://subversion.apache.org/docs/release-notes/1.7.html].

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

Teddlie, C. & Tashakkori, A. (2003). Major issues and controveries inthe use of mixed methods in the social and behvioral sciences, *Handbook of mixed methods in social & behavioral research* pp. 3–50.

Unterkalmsteiner, M., Feldt, R. & Gorschek, T. (2013). A taxonomy for requirements engineering and software test alignment, *ACM Transactions on Software Engineering and Methodology* **Vol. V, No. N, Article A**.
URL: *http://doi.acm.org/10.1145/0000000.0000000*

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, *Network Operations and Management Symposium (NOMS), 2014 IEEE*, IEEE, pp. 1–4.

Varia, J. (2014a). Architecting for the cloud: best practices. Online: accessed in November, 2015 from http://aws.amazon.com/whitepapers/.

Varia, J. (2014b). Migrating your existing applications to the cloud: a phase-driven approach to cloud migration, Amazon Web Services (AWS). [Online: accessed in November, 2014 from http://aws.amazon.com/whitepapers/].

Vengurlekar, N. (2012). Isolation in private database clouds. [Online: accessed in March, 2015 from http://www.oracle.com/technetwork/database/database-cloud/.

Verma, J. (2015). *Repeated Measures Design for Empirical Researchers*, John Wiley & Sons.

Versionone.com (2017a). Top 10 reasons to choose versionone. [Online: accessed on March 3, 2017 from http://www.versionone.com/whyversionone.pdf].

VersionOne.com (2017b). Versionone-agile project management and scrum. [Online: accessed on March 3, 2017 from http://www.versionone.com.

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

Walraven, S. (2014). *Middleware and Methods for Customizable SaaS*, PhD thesis, PhD thesis, Department of Computer Science, KU Leuven, 6 2014.

Walraven, S., Monheim, T., Truyen, E. & Joosen, W. (2012). Towards performance isolation in multi-tenant saas applications, *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, ACM, p. 6.

Walraven, S., Van Landuyt, D., Truyen, E., Handekyn, K. & Joosen, W. (2014). Efficient customization of multi-tenant software-as-a-service applications with service lines, *Journal of Systems and Software* **91**: 48–62.

Wang, Z. H., Guo, C. J., Gao, B., Sun, W., Zhang, Z. & An, W. H. (2008). A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing, *E-Business Engineering, 2008. ICEBE'08. IEEE International Conference on*, IEEE, pp. 94–101.

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

Wiest, S. (2017). Hudson your escape from integration hell. [Online: accessed in March, 2017 from http://www.methodsandtools.com/tools/tools.php?hudson.

Wilder, B. (2012). *Cloud Architecture Patterns*, first edn, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Wilkes, L. (2011). Application migration patterns for the service oriented cloud. [Online: accessed in November, 2017 from http://everware-cbdi.com/ampsoc.

Wood, M., Daly, J., Miller, J. & Roper, M. (1999). Multi-method research: An empirical investigation of object-oriented technology, *Journal of Systems and Software* **48**(1): 13–26.

Yin, R. K. (2014). *Case Study Research: Design and methods*, Vol. 5, 4th edition edn, Sage Publications, Inc., Thousand Oaks, California.

Yu, T., Zhang, Y. & Lin, K.-J. (2007). Efficient algorithms for web services selection with end-to-end qos constraints, *ACM Transactions on the Web (TWEB)* **1**(1): 6.

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

Zeng, J. (2016). Resource sharing for multi-tenant nosql data store in cloud, *arXiv preprint arXiv:1601.00738* .

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

Zoraghi, N., Najafi, A. A. & Akhavan Niaki, S. T. (2012). An integrated model of project scheduling and material ordering: a hybrid simulated annealing and genetic algorithm, *Journal of Optimization in Industrial Engineering* **5**(10): 19–27.

Appendix A

Published Papers

- L. C. Ochei, J. M. Bass, and A. Petrovski, A novel taxonomy of deployment patterns for cloud-hosted applications: A case study of global software development (gsd) tools and processes, International Journal On Advances in Software., vol. volume 8, numbers 3 and 4, pp. 420434, 2015.
- L. C. Ochei, J. M. Bass, and A. Petrovski, Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools, in 2015 International Conference on Cloud and Autonomic Computing (ICCAC). IEEE, 2015, pp. 101112.
- L. C. Ochei, A. Petrovski, and J. Bass, Evaluating degrees of isolation between tenants enabled by multitenancy patterns for cloud-hosted version control systems (vcs), International Journal of Intelligent Computing Research, vol. 6, Issue 3, pp. 601 612, 2015.
- L. C. Ochei, A. Petrovski, and J. Bass, An approach for achieving the required degree of multitenancy isolation for components of a cloud-hosted application, in 4th International IBM Cloud Academy Conference (ICACON 2016), 2016.
- L. C. Ochei, J. Bass, and A. Petrovski, Implementing the required degree of multitenancy isolation: A case study of cloud-hosted bug tracking system, in 13th IEEE International Conference on Services Computing (SCC 2016). IEEE, 2016.
- L. C. Ochei, A. Petrovski, and J. Bass, Optimizing the Deployment of Cloud-hosted Application Components for Guaranteeing Multitenancy Isolation, 2016 International Conference on Information Society (i-Society 2016).

Appendix B

Numerical Results from Simulation Experiments

Table B.1: Optimal values and standard deviation of different instances(m=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 B.2: Optimal values and standard deviation of different instances(m=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

Appendix C

Graphical and Statistical Results from Simulation Experiments

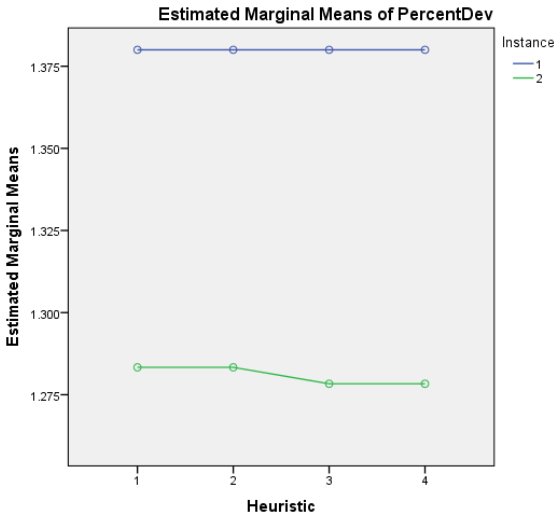

Figure C.1: Quality of Solution- 1

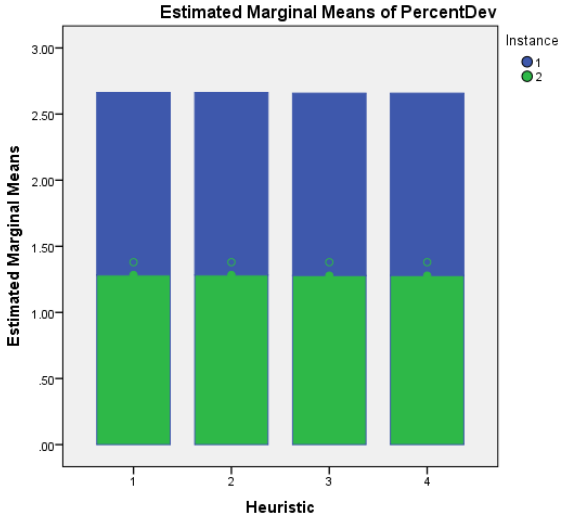

Figure C.2: Quality of Solution- 1

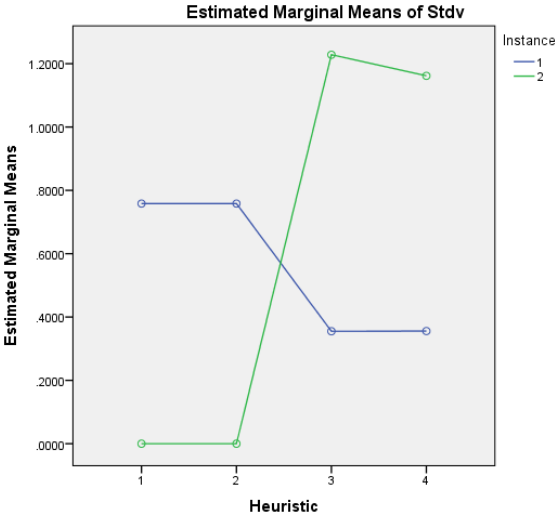

Figure C.3: Robustness of Solution - 1

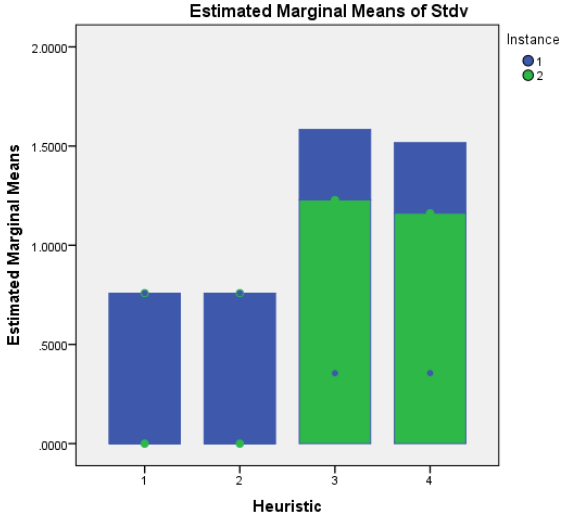

Figure C.4: Robustness of Solution - 2

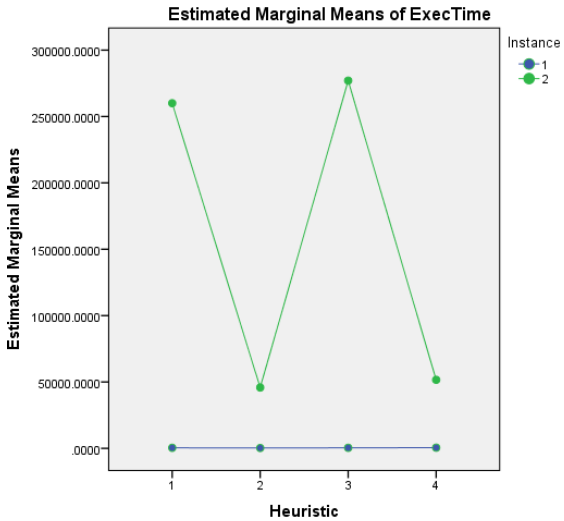

Figure C.5: Computational Effort - 1

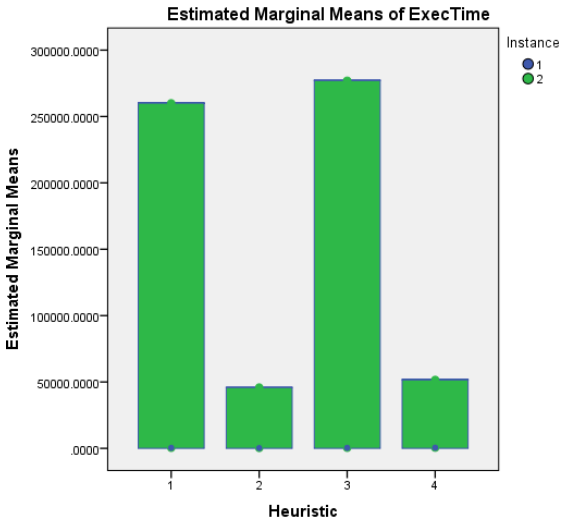

Figure C.6: Computational Effort - 2

Figure C.7: Estimated Marginal Means for 2-way ANOVA