ACHIEVING ADAPTATION THROUGH LIVE VIRTUAL MACHINE MIGRATION IN TWO-TIER CLOUDS

HONGBIN LU

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE AND ENGINEERING

YORK UNIVERSITY

TORONTO, ONTARIO

January 2014

©Hongbin Lu, 2014

Abstract

This thesis presents a model-driven approach for application deployment and management in two-tier heterogeneous cloud environments. For application deployment, we introduce the architecture, the services and the domain specific language that abstract common features of multi-cloud deployments. By leveraging the architecture and the language, application deployers author a deployment model that captures the high-level structure of the application. The deployment model is then translated into deployment workflows on specific clouds. As a use case, we introduce a live VM migration framework that maintains the application quality of services through VM migrations across two tier-clouds. The proposed framework can monitor the performance of the applications and their underlying infrastructure and plan and executes VM migrations to eliminate hotspots in a datacenter. We evaluate both the application deployment architecture and the live migration on public clouds.

Acknowledgements

Thanks to my supervisor, professor Marin Litoiu, who guided me on this thesis and provided me with opportunities to research real world problems. Thanks to Mark Shtern, Bradley Simmons and Mike Smit for their invaluable advice. Thanks to my examination committee for spending time to read this thesis and provide me with feedback. Thanks to the Department of Computer Science and Engineering for its support. Thanks to my team members in the Adaptive Software Systems Research Lab.

Table of Contents

Abstract ii

Acknowledgements iii

List of Tables vi

List of Figures vii

List of Algorithms viii

List of Acronyms ix

Chapter 1: Introduction 1

Chapter 2: Background 4

Chapter 3: Related work 9

Chapter 4: Model-driven Deployment and Management 17

4.1 Application Deployment and Management: Requirements and Contributions 17

4.2 Deployment Architecture 20

4.2.1 The Models and the Metamodel 21

4.2.2 The Software Components 27

4.2.3 The Management Operations 29

4.3 Implementation 32

4.3.1 XML-based DSL 33

4.3.2 Implementations of the Software Components 35

4.4 Experiments 36

4.4.1 Experiment 1 37

4.4.2 Experiment 2 39

4.4.3 Threats to validity 39

4.5 Summary 40

Chapter 5: Achieving Adaptation through VM Migration 41

5.1 Overview 42
5.2 Cross-cloud Migration Management 44
5.2.1 Model-driven Migration 44
5.2.2 Architecture 46
5.2.3 Algorithm 49
5.3 Experiments 52
5.3.1 Experiment 1 55
5.3.2 Experiment 2 58
5.3.3 Threats to validity 61
5.4 Summary 62
Chapter 6: Conclusions and Future Work 63
Chapter 7: Publications during Research 66
Bibliography 67

List of Tables

Table 1: The list of parameters used in experiment 1 56

Table 2: List of parameters used in experiment 2 59

Table 3: A list of implemented services 75

Table 4: List of Restful resources 76

Table 5: List of methods in Topology resources 77

List of Figures

Figure 1: A reference architecture for multi-cloud application deployment and management 20

Figure 2: A set of portable abstractions 22

Figure 3: A sample deployment model 25

Figure 4: A sample workflow model that represents a deployment of MySQL database server 26

Figure 5: An overview of workflow generation process 28

Figure 6: An illustration of scale operation 30

Figure 7: An illustration of a VM migration 32

Figure 8: A sample deployment description file 33

Figure 9: A deployment description file that describes a multi-node deployment 34

Figure 10: The performance of PDS in SAVI testbed 37

Figure 11: The performance of PDS in Amazon EC2 37

Figure 12: A break down of performance overhead 38

Figure 13: Performance comparison of different images 39

Figure 14: An overview of the migration management system 43

Figure 15: An architecture of cross-cloud migration management system 46

Figure 16: A high-level view of the algorithm 50

Figure 17: Initial placement of the applications 54

Figure 18: Workloads in the experiments 55

Figure 19: The migration management process 56

Figure 20: Performances of the applications with different migration policies 57

Figure 21: A comparison of SLA violation with different migration policies 58

Figure 22: Performances of the applications by predicting their workloads 60

Figure 23: A comparison of performance of the applications when prediction is used 60

Figure 24: Portion of a system pattern description showing the support of nested virtualization 78

List of Algorithms

Algorithm 1: An algorithm for predicting if a server will be overloaded 51

Algorithm 2: An algorithm for generating migration plan 52

Algorithm 3: An algorithm to construct a graph based on the inputted deployment model 74

Algorithm 4: An algorithm for provisioning cloud resources... 74

Algorithm 5: An algorithm for provisioning cloud resources... 74

List of Acronyms

DSL	Domain specific language
IaaS	Infrastructure as a service
MaaS	Monitoring as a service
PaaS	Platform as a service
PDS	Pattern Deployment Services
PM	Physical machine
SaaS	Software as a service
SPD	System pattern description
SAVI	Smart Applications on Virtual Infrastructure
SLA	Service-level agreement
VM	Virtual machine
VMI	Virtual machine instance
XCAMP	Cross-cloud application management platform
XML	Extensible markup language

Chapter 1: Introduction

The multi-cloud [1] [2] [3], which means a combination of computing resources from many cloud providers (e.g., Amazon EC2, Rackspace, private clouds, etc.) to achieve efficient resource utilization, is an emerging and challenging domain of cloud computing. Research in this domain addresses fundamental concepts such as brokers [4] [5], meta-data services [6], elasticity [7] or security [8].

We use the term *two-tier cloud* to refer to a special type of multi-cloud, where resources from multiple cloud providers are divided into two groups. Resources in one group are limited in quantity, but have some desired properties, such as affordable pricing, great configurability, or low network latency. In contrast, resources in the other group are almost unlimited in quantity, but relatively more expensive, have limited configurability, or higher network latency. Typically, cloud resources in one group are from a small or private datacenter, while cloud resources in the other group are from a large datacenter.

One of the research projects that leverages this two-tier architecture is the Natural Sciences and Engineering Research Council of Canada Strategic Network for Smart Applications on Virtual Infrastructure (SAVI) [9]. The SAVI project outlines a cloud system composed of two cloud types: *core* and *smart edge*. The core represents a cloud with large amount of computing resources, typically a large data centre. From the user's point of view, the computing resources available at core are unlimited, but applications hosted in core have a high network latency due to the assumption that core is geographically far away from end users. The smart edge represents a cloud with a limited set of computing resources, typically a small data centre, but geographically close to end users thus resulting in low network latency. Depending on availability of resources and the state of applications, computation tasks are moved between core and smart edge(s). To facilitate the movement of computation tasks, core and smart edge(s) are connected by a fast network.

For two-tier cloud in general and SAVI cloud in particular, deploying or managing applications is challenging because adding another tier of cloud increases the overall complexity of the system. Taking application deployment as an example, deploying application on a two-tier cloud is harder than a single cloud because users need to instantiate the deployment in different administrative domains and cross-configure application components hosted by different and heterogeneous clouds. A category of widely used deployment approaches is the workflow based one: the application deployer authors a deployment in scripts that contain the workflows for deploying an application. Often the authored scripts are hard-coded for a specific application and a specific cloud provider. Although it is possible to

extend the scripts to support other clouds, it often requires a significant amount of work due to the inherent complexities of scrip-based workflows. Therefore, scripting the deployment process is impractical for two-tier cloud.

Another challenge arises when computing tasks are moved between core and edge to maximize the performance of applications. A popular technique is live virtual machine (VM) migration [10], which allows relocation of the VMs that host the applications without shutting them down, and thus avoids the disruption of service. However, live VM migration is not cheap and often incurs significant performance penalties during the migration process [10]. Depending on application and workload characteristics, the total migration time can be as long as a few hours, which significantly impedes the performance of the whole system. Due to the challenges of live VM migration, work such as [11] [12] [13] [14] [15] [16] [17] focuses on defining optimal migration scenarios to determine, when VM migration is conducted, which VM should be migrated, as well as which server is the desired destination for the migrated VM. Unfortunately, most work focuses on migration within a single datacenter, which is not applicable to the two-tier architecture.

In this thesis, we consider application deployment and management in two-tier cloud environments, and propose solutions to address the related challenges and complexities. For application deployment, we promote a model-driven approach that enables application deployers to specify a deployment that is portable among different clouds. For application management, we particularly focus on the use of live VM migration technique for managing the performance of applications. In particular, we define a *hotspot* as a server whose computing resources (i.e., CPU) are over-utilized, and promote the use of live VM migrations to eliminate hotspots in the underlying infrastructure, thus guaranteeing the performance of applications. We demonstrate that (i) the proposed model-driven approach is able to simplify the tasks of deploying and managing applications in two-tier or multi-cloud environments, and (ii) the use of live VM migration can enable adaptation of applications and thus improve their performance. The main contributions of this thesis are as follows:

- We propose a model-driven approach, in which application deployers can author a deployment model that is automatically transformed to the deployment workflows. By leveraging the model-driven approach, application deployers can focus on the high-level concepts of deployment without being distracted by the low-level complexity.

- We introduce a set of abstractions to capture various concepts of deployment, independent of the underlying cloud, middleware or infrastructure. The abstractions are designed to capture the heterogeneities of the deployment environments at different levels of details. By introducing the abstractions, we hide the heterogeneities of the environments and help application deployers specify portable deployments of their applications.
- We propose a deployment architecture that encapsulates common practices for implementing a multi-cloud deployment solution. The key design principle of the architecture is the separation of concerns: the architecture clearly separates application-specific deployment logic from the logics of provisioning clouds resources, installing middleware components, and executing deployment workflows. In addition, the architecture is designed to be extendable, thus enabling experts from different domains to contribute their knowledge to the architecture.
- We provide a non-trivial implementation of the deployment architecture. The implementation is called Pattern Deployment Service (PDS), which is a web application that provides application deployment as a service. PDS is designed to be flexible and extendable, and supports multi-cloud environments. In addition, PDS provides some operations to facilitate the management of applications management. The deployment service is delivered through an intuitive API.
- We provide a migration management architecture that facilitates the management of VM migrations in two-tier cloud environments. The architecture is able to monitor the deployed applications, predict the occurrences of hotspots, compute migration plans to mitigate the hotspots, and execute the migration plans.
- We provide an algorithm for computing a migration plan that is as optimized as possible. The algorithm can be parameterized with a user-defined migration policy to determine the VMs that need to be migrated, and a user-provided prediction model to forecast the workloads.

The structure of this thesis is as follows: Chapter 2 presents the background; Chapter 3 presents the related work; Chapter 4 presents the model-driven approach for application deployment and management in multi-cloud environments; Chapter 5 presents a migration management framework for achieving application adaptation in two-tier cloud environments; Chapter 6 presents the conclusions and future work.

Chapter 2: Background

Cloud computing, which refers to the delivery of computing resources over the Internet as a service [18], is an emerging computation model. In general, the delivered resources can be categorized into three types: hardware infrastructure, computing platform, and software, which are referred to as *Infrastructure as a service* (IaaS), *Platform as a service* (PaaS), and *Software as a service* (SaaS) respectively.

We use the term *cloud* to refer to the hardware and software in datacenters dedicated to providing computing services to the end users over the internet. Data centers are buildings where multiple servers and communication gear are collocated, thus greatly centralizing and simplifying their administration [19]. Typically, a data center consists of a set of storage systems, networking systems, power systems, cooling systems, other necessary hardware equipment, and a set of software components running on top of this hardware infrastructure, all of which form a large pool of computing resources that are ready to deliver to end users on demand.

Virtualization is an important enabling technology of the cloud. Generally speaking, virtualization means hosting one or more independent virtual machines (VM), which look like unique Operating Systems (OS) instances that share the same physical machine (PM) [20]. One key advantage of virtualization is that it allows different applications to share computing resources efficiently, without interfering with each other (i.e., sandboxing). From an application functionality point of view, a virtualized OS and non-virtualized OS are indistinguishable. Therefore, applications that are successfully hosted by a non-virtualized OS can be migrated to virtualized OS without modification. Virtualization allows the PMs in a datacenter to be efficiently shared by different VMs that are hosting applications from different users.

In more traditional approaches to computing, the application provider needs to plan their purchasing of computing resources according to estimates for peak application load. A common problem that is frequently encountered with this approach is over-provisioning and under-provisioning. We use the term *under-provisioning* to refer to situations in which the amount of provisioned resources is not sufficient to maintain the quality of service of applications. The term over-provisioning means the opposite. The root cause of under-provisioning and over-provisioning is the inherent difficulty in predicting future workload. Furthermore, applications often see a fluctuating workload where the peak

workload is significantly greater than the steady workload. Provisioning for the peak workload of such applications often leads to under-utilization of computing resources.

Due to the limitations of traditional computing, cloud computing is quickly emerging as a solution for the under-provisioned problem. The two most important characteristics of cloud computing are the elasticity of computing resources and the pay-as-you-go pricing model, which enable application providers to provision resources on demands. Therefore, cloud computing is perfectly adaptable for applications that have highly fluctuating or unpredictable workloads.

Given the advantages of cloud computing, more and more application providers have started to host their applications on the cloud. However, there remain challenges with regards to this approach: How to deploy applications to the cloud? Traditionally, most of the deployment approaches were script-based, in which the workflow of a deployment is encapsulated in a script that is dedicated for deploying specific applications. Typically, the deployment script is hard-coded and complex, so porting the script to another application, platform or cloud provider requires substantial modification which incurs a significant amount of work and compromises the stability of the deployment solution.

To address this challenge, several configuration management tools [21] (such as Chef[1], Puppet[2], and CFEngine[3]) have been introduced to help users develop a portable application deployment. These tools employ their own domain specific languages (DSL) which can be leveraged by users to describe their deployments. Deployment solutions written in a DSL have a high portability because the DSL can specify the logic of deployment workflows without hard-coding for a specific execution environment.

Application deployment is also addressed through model-driven deployment approaches [22] [23] [24] [25] [26]. Generally speaking, this type of approach allows users to specify, in models, how to deploy their applications. A user-specified model contains high-level deployment logic and can be automatically translated into low-level workflows that will be executed to realize a deployment. Compared to other approaches, model-driven deployment allows domain experts to pre-implement a set of workflow templates which can be parameterized and composed by users. As a result, from the user's point of view, the complexities of deployments are greatly reduced.

---

[1] http://www.opscode.com/chef/
[2] http://puppetlabs.com/
[3] http://cfengine.com/

After applications have been deployed, they typically need to be managed. Due to the complexities of managing enterprise applications, researchers have begun to focus on creating an autonomic system that refers to a computing system that can manage itself [27]. The key component of an autonomic system is called *autonomic element*, which refers to the system components that manage their internal behavior and their relationships with other autonomic elements, in accordance with specified policies [27] [28] [29]. An autonomic element consists of an *autonomic manager* and a *managed element*. A managed element could be a piece of software or hardware resource that needs to be managed. An autonomic manager is the component that manages the managed element. An autonomic manager consists of a MAPE-k loop (monitor-analyze-plan-execute-knowledge) and two manageability interfaces: *sensor* and *effector* [29]. An autonomic system can gather information about the managed element through its sensor interface. The gathered information is leveraged by the MAPE-k loop to construct an execution plan, which is executed by the effector interface.

An example of managing a deployed application is the maintenance of performance. If application providers have service-level agreement (SLA) agreements [30] [31] [32] with their users, they need to ensure that performance of the application satisfies each criterion specified in the SLAs. One of the main factors that affect application performance is the amount of resources provisioned for the application. As a result, a research question arises: what is the amount of resources allocated for an application, given the intensity of the workload and/or the existing SLA agreements.

A popular approach for solving the question mentioned above is to optimize the resource allocation statically [33] [34] [35], thus guaranteeing or maximizing the performance of the application. A drawback of this type of approach is that the resource allocation decisions are not adjustable or hard to adjust once they have been made. As a result, each application is allocated a fixed amount of resources that cannot be increased or decreased at runtime. Therefore, this category of approaches is not suitable for applications with non-static or non-predictable workloads. Some researchers [36] [37] [38] attempt to extend the static resource allocation approach to handle the dynamic workload. In particular, they have proposed to track the dynamic states of deployed applications and periodically re-allocate resources to applications. However, an adjustment of resource allocation typically requires stopping and restarting VMs which may cause disruption of services.

Recently, researchers have started to investigate the possibilities of leveraging live VM migration technique [10] [39] [40] to achieve seamless resource re-allocation, thus eliminating hotspots. Using this technique, it is possible to adjust allocation of resources without disrupting the applications, so that the performance of applications can be re-optimized to adapt to time-varying workloads. The widely used live VM migration approach is pre-copy based [10] [40]. In this approach, the state of VM is completely transferred to the destination before the switching of execution hosts. The process of pre-copy migration is summarized in the following steps [10]:

1. A VM migration is requested by clients.
2. The memory pages of VM are sent to destination host without stopping the VM.
3. Iteratively re-send the dirty memory pages (i.e., the memory pages that have been written after the last sending) to destination host.
4. The VM is paused. The rest of the dirty pages and the state of CPU are sent to the destination host.
5. The VM is resumed on destination host.

It is assumed that the persistent storage of the migrating VM is shared by the source and destination hosts so that the storage does not need to be migrated. However, this assumption may not always be the case in practice. Because of that, some researchers [40] have proposed an extended approach that supports migration of persistent storage (together with memory and CPU). In the extended approach, a new step is inserted before step 1 in which the state of disk is transferred.

In contrast to pre-copy migration, another type of VM migration approach is post-copy based [39]. In these approaches, the execution host of the VM is switched before the transfer of memory's state. After the switching of execution hosts, it is possible that the VM needs to access a memory page that has not been sent to new host yet. In this case, the required memory page is fetched on demand, at which time the execution of the VM is stopped, waiting for the memory page. Compared to pre-copy migration, post-copy migration is able to switch the execution host in seconds, thus quickly eliminating the hotspot in the source host. However, there is typically a period of performance degradation right after the switching of execution hosts due to a sequence of misses of memory pages. Shribman et al. [41] proposed to combine the pre-copy and post-copy approach to leverage the advantages of both, but that approach has not been widely used so its performance is unknown.

Generally speaking, live VM migration is a mature and practical technique, but there are challenges with regards to utilizing it: First, during migration, the performance of an application hosted by a migrating VM is significantly degraded, because the migration task consumes extra CPU cycles and memory which reduces the amount of CPU and memory allocated for the application. Second, executing a VM migration generates a significant amount of network traffic which potentially causes a congestion of network. Third, in two-tier cloud, VMs need to be migrated over the wide-area network, which is known to be challenging [40] due to the relatively high latency of wide-area network (performance of VM migration is sensitive to the network latency).

In multi-cloud environment, a way to enable VM migration between two clouds is to leverage *nested virtualization*, which refers to the use of virtualization in an already virtualized resource [8] [42] [43]. In other words, there are two layers of virtualization [8]: In layer 0, a hypervisor runs on a PM in datacenter. In layer 1, another hypervisor runs on a VM from layer 0. Applications are deployed to VM(s) in layer 1. Nested virtualization is useful if the users cannot access the hypervisors in layer 0. In particular, if the cloud is owned by a third party, the users are usually not allowed to access the hypervisor in layer 0, which prevents them to execute hypervisor-level operations, such as VM migration. To address this problem, we could build another layer of virtualization, thus enabling the users to access the hypervisors in layer 1 and perform VM migrations. The main challenge of leveraging nested virtualization is the performance penalty and the lack of support by most of the public cloud providers. The reported performance penalties range from 3 to 30% [8] [42] [43]. The difficulties of utilizing nested virtualization on the major public cloud provider, that is Amazon EC2, are reported by Williams et al. [43].

Chapter 3: Related work

In this chapter, we review the existing literature with emphasis on work that proposes (i) a model-driven approach for deploying applications or (ii) an automated approach for managing VM migrations.

To the best of our knowledge, utilizing a model-driven approach for deploying applications was first proposed by Eilam et al. [22] [23]. In this work, they attempted to reduce the complexities of application deployment in a single datacenter by introducing a *logical deployment model* that captures the high-level concepts of a deployment. A logical deployment model consists of two types of components: *node* and *edge* (or *relationship*). A *node* represents a logical component of an application and its requirements. An *edge* represents a connection between *nodes*. Both *node* and *edge* can be annotated with metadata that might be useful for the deployment. After a logical deployment model is authored, a tool is used to transform the logical deployment model into a *deployment plan* that represents the low-level workflows for executing the deployment. The transformation is decided by a set of specified transformation rules, optimization criteria and a datacenter model. Finally, the deployment plan is the input of a provisioning engine that is responsible to execute the deployment.

Arnold et al. [24] proposed to extend the deployment models proposed by Eilam et al. [22] [23] to support partially specified deployment models, which are referred to as *patterns*. Generally speaking, by leveraging a pattern, users may specify an informationally incomplete deployment description of an application, which could be beneficial if the users are not domain experts and do not have the comprehensive knowledge to specify a complete deployment model. A pattern can be automatically validated and concretized by using the tools developed by domain experts. The basic building block of a pattern is called a *Unit*, which represents a generic entity that needs to be deployed or has already been deployed. The *Unit* can be *virtual* or *concrete*. A virtual unit represents a partially specified resource, and a concrete unit represents a completely specified resource. *Units* can be connected by a *Link*, which represents the various kinds of relationships between the *Units*. A special type of *Link* is called a *realization link*, which can be used to connect a *virtual unit* to a *concrete unit*, or connect a *virtual unit* to another *virtual unit*.

The work presented in [22] [23] [24] proposed to use various kinds of deployment models to overcome the complexities of application deployment. However, their approaches were designed for application deployment in a datacenter environment in which applications are deployed directly to physical resources. No consideration for virtualized environments (i.e., cloud) was investigated.

Konstantinou et al. [25] attempted to facilitate application deployment in a cloud environment by leveraging a model-driven approach. In particular, the authors proposed to pre-install the required middleware in a cloud image, and introduced a model to describe the image declaratively. An image together with its configuration operations and its model is called a *virtual appliance*. Basically, a virtual appliance consists of two parts which are called *packaging* and *composition* respectively. The *packaging* represents the configurations of the pre-installed middleware. The *composition* specifies the way that a virtual appliance instance connects to another virtual appliance instance. Furthermore, a virtual appliance includes a set of operations that can be used to reconfigure the image. Given a list of virtual appliances that are available, application deployers can construct a relatively simple deployment model to describe the way to deploy their applications. The deployment model mentioned above mainly specifies the way to select, configure and compose the virtual appliances to achieve a deployment.

Similarly, Papazoglou et al. [44] [45] proposed an approach to integrate services, applications and/or middleware components hosted by different clouds. To facilitate application deployments, domain experts pre-built components of applications in one or more clouds. The functional and non-functional aspects of each pre-built component are specified in a model written in a DSL called *Blueprint Description Language*. The information about the pre-build components is recorded in a central repository so that application deployers can discover them easily. To define a deployment, an application deployer chooses a list of pre-built components and specifies the way to connect them in a model written in a DSL called *Blueprint Manipulation Language*.

Overall, the work in [25] [44] [45] focuses on reducing the complexities of application deployment by pre-installing and pre-configuring the application components in different clouds. In the IaaS model, a pre-built component is typically stored in a cloud image. However, a disadvantage of such approaches is the lack of flexibility because the pre-built components are not portable (or hard to be ported) between clouds. A solution could be uploading each pre-built image to all the clouds that need to be supported, but such a procedure would clearly incur an overhead. Furthermore, the format of the cloud images is not standardized among cloud providers, thus limiting the portability of the cloud images. Finally, managing a number of copies of cloud images in multiple clouds is an additional overhead.

Wettinger et al. [46] outlined the idea of middleware-oriented deployment that intends to decouple the deployment logic of applications from their middleware components. A phenomenon observed by the authors was that the logic for installing specific middleware might be reusable while the

logic for deploying a specific application is not reusable. Based on this observation, the authors proposed creating a set of middleware components that could be reused for deploying different applications. Portable implementations are provided to ensure that the middleware components can be hosted on different clouds. By doing this, users gain flexibility for choosing the middleware components they want, and placing them on one or more clouds. Generally speaking, a key advantage of the proposed approach is to enable application deployers to focus on the logic for deploying applications without being distracted from the logic for deploying the middleware components. However, the authors have not provided a holistic approach to realize the proposed idea (middleware-oriented deployment). For example, they have not proposed models or architecture that could facilitate the proposed deployment approach. Furthermore, given the challenges stemming from a multi-cloud environment (such as the unstandardized APIs among cloud providers, the lack of interoperability and portability between clouds), the proposed idea by itself might not be sufficient to address all the challenges.

Flissi et al. [47] also attempted to overcome the complexities of application deployment by proposing a model-driven approach. They focus on application deployment on the Grid [48], and defined a set of technology-independent abstractions (e.g. Instruction, Procedure, SoftwareType, Personality, etc.) that capture various concepts on an application deployment. In particular, an *Instruction* represents an elementary workflow. A *Procedure* represents a general deployment procedure that is composed of one or more Instructions. A *SoftwareType* represents the artifact that needs to be deployed. A SoftwareType could contain a set of Procedures used for configuring the software. A *Personality* represents the software system that is consists of zero or more SoftwareType. To deploy an application, the users simply must construct a model by using this set of abstractions. Generally speaking, Flissi et al. proposed the usage of technology-independent abstractions to create a portable deployment plan. However, their proposed approach addresses a different problem from ours. Specifically, their method is designed for the Grid, in which applications are deployed on a fixed set of machines. Our method is designed for multi-cloud environments, in which applications are deployed on machines that need to be provisioned dynamically. In addition, a disadvantage with their approach is that to issue a deployment, users need to construct a model that mixes the high-level abstractions (such as SoftwareType) with the low-level ones (such as Instruction), thus exposing users to unnecessary complexity. We believe that an ideal deployment system should effectively hide the low-level details from users to enable them to focus on the high-level concepts.

Paraiso et al. [49] attempted to address the challenges of application deployment in multi-cloud environments by introducing a platform that provides some multi-cloud deployment services. To prepare a deployment, users leverage the service provided by the platform to provision one or more *nodes*, each of which represents a virtual machine instance in a cloud. The platform hides the cloud-specific configurations of each node from users, thus providing a unified view of the deployment environment. After a node is provisioned, a *kernel* is installed on the node, which is used to support a set of operations for configuring the node. After the kernel is installed on a node, users can upload the corresponding artifacts (such as deployment scripts or an application archive file) to the node and instruct the node to run the uploaded deployment scripts. Generally speaking, this platform could effectively simplify application deployment in multi-cloud environments. However, they proposed a script-based approach (i.e., users are required to develop their own deployment scripts), instead of a model-driven approach. The script-based approach does not contain a model for capturing the abstract concepts in a deployment, which unavoidably complicates the application deployment process.

Beside the challenges of application deployment, runtime application management is also a challenging issue. Recently, researchers have proposed using the live VM migration technique to manage the performance of the deployed applications.

Mann et al. [11] [12] proposed the usage of live VM migration to decongest the network traffic in a datacenter. In their work, they define *congestion* as a computing resource whose utilization exceeds a threshold. For example, if the threshold is set to 80%, a communication link whose load exceeds that limit is considered to be congested, and those congested links are then candidates to be decongested. The authors assume that the initial placement of VMs is done optimally so that initially, no link is congested. However, the workload of applications may change over time, which may result in several congested links. As a result, one or more VMs need to be migrated to decongest the links. They modelled a datacenter as a graph, in which the hosts and switches are represented by the vertices of the graph, and the communication links between vertices are the edges. The cost of a VM migration is estimated by the total amount of network traffic incurred by executing the VM migration. The goal of this approach is to minimize the migration cost, while ensuring that no edge is congested during the migration. The authors prove that the migration problem is NP-Hard, and offers a heuristic approach to solve the problem. In particular, the authors implemented an OpenFlow[4] controller that monitor the network traffic to identify a set of congested links. The VMs that contributed most to the congested links

[4] http://archive.openflow.org/

are selected to be migrated. A heuristic is used to choose a destination for each migrated VM. The heuristic is designed to balance the loads of each link as much as possible. By doing that, the network congestion in a datacenter is effectively reduced, thus improving the performance of the hosted applications.

Wood et al. [13] focus on balancing the loads of servers in a datacenter. In particular, they focus on eliminating hotspots in a datacentre by migrating VMs from overloaded servers to under-loaded servers. To achieve this, they implemented a monitoring component that gathers metrics from each physical server and VM, which is leveraged to compute the utilization of each server and the number of SLA violations of each application. In their work, a server is considered to be a hotspot if its utilization exceeds a defined threshold, or one of its hosted applications has too many SLA misses. After a hotspot is detected, the system computes the amount of additional resources that are sufficient to mitigate it. Then the system provisions the amount of needed resources, and migrates some VMs from the hotspots to the newly provisioned resources. To do that, the authors introduce a hotspot migration algorithm that attempts to resolve the hotspots in the most efficient way (e.g. with minimal overhead). The algorithm proceeds as follows: First, a set of hotspot servers is identified by computing a metric called *volume* that is used to measure the utilization a server. They define the *volume* of a server as the product of its CPU, memory and network loads [13]. Next, the algorithm selects a set of VMs to migrate off the set of hotspot servers. The selection of each VM is based on the feasibility and the efficiency of the migration. The feasibility of a VM migration is determined by the existence of a destination that has enough free capacity to host the VM. The efficiency of a VM migration is estimated by dividing its *volume* by its *size*, where the *size* is the estimated amount of data to be transferred during the migration. Finally, the algorithm finds a destination for each VM that is selected to be migrated. The selected destination must have sufficient capacity to host the incoming VM. If more than one server satisfies this requirement, the one with least *volume* will be selected.

Stage et al. [14] focus on controlling the network traffic during VM migrations. The authors proposed to reserve a fraction of bandwidth that is dedicated for the migration traffic. The reserved bandwidth will be allocated for each migration. To optimize the allocation decisions, the system predicts the amount of bandwidth that is needed for each migration. Furthermore, if the workload of a VM is classified to be non-predictive, the system will allocate the bandwidth online. To guarantee the performance, each migration is given a deadline and earliest start time. By doing that, the bandwidth allocation problem is mapped to a real-time scheduling problem, in which they try to find a feasible

allocation plan to ensure that each VM migration can meet its deadline. After the problem is formulated, an existing real-time scheduling algorithm can be leveraged to solve the problem.

Ma et al. [15] attempted to solve the load balancing problem within a datacenter by using VM migration techniques. They focus on the way to minimize the number of VM migrations while maintaining the performance of the hosted applications. To achieve the goal, the authors proposed selecting the VM with the highest CPU utilization to migrate. The goal is to minimize the number of migrations that are needed. After a VM is selected, they utilized the *Technique for Order Preference by Similarity to Ideal Solution* (TOPSIS) [50] to select a destination for the migrated VM. TOPSIS is a traditional multi-criteria decision making approach, which is used to rank the set of destinations that have enough free capacity to host the migrated VM. The authors conducted a few experiments, which demonstrate the effectiveness of this approach.

Van et al. [16] [51] focus on utilizing VM migration to consolidate the VMs in a datacenter. Generally speaking, the authors attempted to answer two questions: (1) how to map a set of applications to a set of VMs; (2) how to map the set of VMs into a set of PMs. To address the first question, the authors formulated the problem as an optimization problem. In particular, it is assumed that each application has a utility function, which is used to map a set of VMs that are provisioned to host the applications, to a utility value that quantifies the satisfaction of each application. The objective is to maximize the sum of the utility values of all applications, while minimizing the cost of VM provisioning. The constraints are that the number of VMs for each application cannot exceed a threshold, and the total number of VMs cannot exceed the capacity of the datacenter. For answer the second question, the authors again formulated the problem as an optimization problem: the objective is to maximize the resource utilization, subjected to the constraints of physical resources capacities. By combining the answers of question one and two, the applications can be statically placed on resources on a datacenter. To extend the approach to support dynamic workloads, the authors proposed to re-compute the resource allocation periodically. In each period, a new resource mapping is computed, and a transition plan is constructed for switching from the old resource mapping to the new resource mapping. The transition is achieved by executing a sequence of VM migrations. To control the migration costs, the authors attempted to compute an optimized transition plan. In particular, they introduced a model to quantify the cost of a VM migration. The optimality of a transition plan is evaluated as the total cost of all its planed migrations.

Approaches proposed by above papers [11] [12] [13] [14] [15] [16] [51] focus on the usage of VM migrations to balance the loads in a datacenter. None of these papers proposed an approach to manage the cross-datacenter migrations. Therefore, their proposed approaches are not directly applicable to two-tier clouds.

To the best of our knowledge, the usage of VM migrations to balance the loads between two datacenters has only been discussed by Guo et al. [17]. This paper proposed to eliminate hotspots of a private datacenter by migrating some VMs to a public cloud. They focused on maximizing the utilization of the private datacenter while minimizing the price charged by the public cloud provider. They assumed that the workloads of the applications are perfectly predictable. When the private datacenter is predicted to be overloaded, some applications are migrated from the private datacenter to a public cloud (they migrate an application by migrating all its VMs). The authors proposed two options to select the migrated applications: (1) directly migrate the overloaded application; (2) migrate an alternative set of applications to release spaces for the overloaded application. They estimated and compared the cost of options one and two, and the one with lower cost was selected. In the experiments, the authors show that option one tends to perform better than option two. Generally speaking, this paper proposed to eliminate hotspots in a datacenter by migrating the overloaded applications out of the local datacenter. This implies that if a hotspot is caused by a single VM of an application then all the VMs of that application need to be migrated out, which incurs unnecessary cost. The situation is worse if the selected application contains a lot of VMs. We believe an ideal migration management system should be able to migrate individual VM, instead of all the VMs of an application.

Karve et al. [36] [37] focus on solving the dynamic application placement problem. In particular, the authors attempted to answer the following two questions:

1. How to decide the initial application placement so that load dependent resource (e.g. CPU) and load independent resource (e.g. memory) are efficiently utilized.
2. How to update the application placement at runtime to adapt to dynamic workloads.

The guide placement decisions, they introduced a metric called *density*. The density of a server is computed by dividing its CPU capacity by its memory capacity, and the density of an application is computed by dividing its CPU demand by its memory demand. To answer the first question, the authors proposed a heuristic to construct the initial application placement. In particular, applications are chosen by decreasing density, and placed on the servers. After an application is chosen, a server that has

enough free capacity is chosen to host the application (or part of the application). If multiple servers have the required capacity, the one with highest density will be chosen. The intension of this scenario is to balance the utilizations of both CPU and memory. To answer the second question, the authors performed the following steps:

- Determine if the existing placement is able to satisfy the latest demands of all applications. This problem is solved by formulating it as a maximum flow problem of a bipartite graph.
- If the existing placement cannot satisfy the demands, it implies that there must be a set of applications with unassigned demands and a set of servers with unused capacities. The heuristic approach (used to construct the initial placement before) is used again to place the set of applications on the set of servers.
- If there is still no feasible solution, the authors proposed to delete an edge with lowest CPU to memory ratio in the bipartite graph and restart the previous steps.

Overall, the key idea of this approach is to utilize each dimension of computing resource equally, thus reducing the potential needs for changing the placement. However, the authors proposed to start and stop the VMs to achieve a change of placement, which may interrupt the services of the applications. We believe that a better approach is to leverage the live VM migration technique which can be used to change the placement of an application without interrupting its service.

Xu et al. [52] focus on solving the mobile agent planning problem in wireless sensor networks. The authors define a *mobile agent* as an entity that contains a unique ID, data buffer, processing task and route of migration [52]. A mobile agent typically departs from a server and migrates among clients to perform computing tasks. The authors particularly focus on computing migration routes of mobile agents. They formulate the mobile agent planning problem as an optimization problem, with the objective to minimize the migration overhead, subject to fulfilling the designated tasks. Although both their work and our work focus on migrations of computing tasks, their work particularly focus on migration of mobile agents in wireless sensor networks while our work focus on migration of VMs in two-tier clouds.

Chapter 4: Model-driven Deployment and Management

In order to address the challenges derived from multi-cloud environments, we propose a model-driven approach for application administrators to deploy and manage their applications. The proposed solution includes a reference architecture and an implementation. The performance of the implementation is evaluated in experiments.

The main contributions of this chapter are as follows:

- A metamodel that contains a set of technology-independent abstractions to capture the concepts or logic of application deployment and management in various levels of granularity.
- A reference architecture which encapsulates common practices for implementing a multi-cloud application deployment and management system.
- An implementation of the reference architecture.
- An experimental evaluation of the implementation's performance in two different clouds.

This chapter is organized as follows: section 4.1 presents the requirements of a multi-cloud deployment and management system; section 4.2 presents the reference architecture; section 4.2.3 presents our implementation of the reference architecture; section 4.4 shows the experimental evaluation.

4.1 Application Deployment and Management: Requirements and Contributions

Today, enterprise applications tend to be large in scale and complex in terms of interactions. The deployment and management of such applications typically spans multiple servers that are provisioned on demand from one or more clouds. The set of provisioned servers are typically heterogeneous in terms of hardware specifications, operating systems (including their configurations), and/or the stacks of pre-installed software. Additionally, in order to host a distributed application, middleware components need to be deployed and then automatically connected together at runtime. Based on the observations mentioned above, an ideal multi-cloud deployment and management system should enable application administrators to specify their deployments or modification of their deployments, without being distracted from the heterogeneities of the underlying environment. In particular, we believe an ideal deployment and management system should meet the following requirements:

Handle heterogeneity: In multi-cloud environments, a deployment and management system needs to handle heterogeneity at different levels. First, there are some features, services, or operations that are common among providers, but different providers deliver them through different APIs. Second, resources provisioned from different providers are unavoidably heterogeneous in terms of states and configurations. Although most providers offer a set of options for users to customize a provisioned resource (e.g. users can customize a provisioned server by choosing from a set of base images, instance flavors, etc.), those options are not yet standardized. Third, it is often the case that the same middleware product needs to be configured differently for different applications. Fourth, there are some OS-level workflows that are logically the same but needs to be executed differently in different OSs. For example, the command to install a package is of the format `apt-get install PACKAGE-NAME` in Ubuntu[5], while it is `yum install PACKAGE-NAME` in CentOS[6]. An ideal deployment and management system should hide the unnecessary heterogeneities from application administrators and enable them to focus on the core concepts of deploying and managing applications.

Hide complex details: In most cases, application administrators can picture how they want an application to be deployed or managed, but they may not have the comprehensive knowledge to realize that vision. For example, application administrators often have an idea about how to place application components across a set of servers provisioned from clouds. In addition, they may also know the set of middleware components that must be installed on each server and how the set of middleware components should be configured and connected. However, they may have difficulties creating the workflows for provisioning servers from clouds, installing or configuring specific middleware components, or cross-configuring the middleware components. An ideal deployment and management system should enable application administrators to specify what the desired result is without knowing the low-level details about how to achieve the desired result.

Separation of concerns: A multi-cloud deployment and management system typically encapsulates various kinds of concepts or logics, such as the logic for deploying specific applications, the logic for provisioning or configuring cloud resources, the logic to deploying specific middleware, and the logic to combine all of them. An inappropriate mix of those logics (e.g. automating the deployment in a single script) will lead to a system that is complex and fragile. We believe an ideal deployment and

[5] http://www.ubuntu.com/

[6] http://www.centos.org/

management system should clearly separate different kinds of deployment logics and enable them to be developed and maintained independently.

Flexibility: To leverage the benefits of multi-cloud (e.g. minimize the cost, increase availability, avoid vendor lock-in, etc.), an ideal deployment and management system should offer application administrators the flexibility for choosing hardware and software resources from different providers to host their applications. In other words, application administrators should be able to define various kinds of mappings between application components and cloud resources. To further increase the flexibility, it is desirable to allow an easy switching between middleware components that are used to host the application.

Deployment orchestration: The workflows for deploying and managing an application include configuring a set of servers used to host the application. Typically, to reduce the total configuration time, the tasks for configuring each server need to be performed in parallel. At runtime, the parallelized processes for executing the configuration tasks may need to communicate remotely with each other to gain the necessary information. For example, to establish a connection from an application server to a database server, the process for configuring the application server needs to communicate with the one for configuring the database server, directly or indirectly, to get the URL and credentials of the database. Because of that, an ideal deployment and management system should employ a mechanism to orchestrate the parallelized deployment processes and hide the related complexities from application administrators.

Extendibility: A multi-cloud deployment and management system typically has a set of supported cloud providers, middleware and OSs, which might be currently sufficient. However, it is possible that the requirements of applications will evolve and the application administrators will ask for extending support for a new cloud provider. Furthermore, application administrators may introduce a new application that requires to be hosted by unsupported middleware or an unsupported OS. To deal with these situations, an ideal deployment and management system should be properly designed and implemented so that it can be easily extended.

Given the requirements defined above, partial solutions have been proposed in the existing literature. For example, solutions presented in [22] [23] [53] focus on reducing the complexities of application deployment in a single datacenter by introducing a model-driven approach. The approach presented in [47] focuses on overcoming the challenges of application deployment on grid computing

middleware by leveraging portable abstractions to capture the concepts of deployment in various levels of details. The solutions mentioned above do not entirely meet our requirements mainly due to the fact that they are not suitable for multi-cloud environments. Other solutions such as [25] [44] [45] proposed to pre-install, pre-configure, and package each application component into specific clouds and use a model or language to describe the pre-built component. Since pre-built components are not shared between providers, the flexibility of a deployment is limited by the availability of specific application components in specific clouds, which make those approaches not ideal for multi-cloud environments. The approach presented in [49] introduces a PaaS infrastructure that is able to provide several services to facilitate application deployment and management in multi-cloud environments. Although that approach is able to hide heterogeneity of clouds, it still exposes several complexities to users (e.g. users need to provide dedicated deployment scripts), which make that approach not easy to be used.

In the rest of this chapter, we propose an approach for constructing an application deployment and management system that satisfies the requirements defined above. The proposed approach includes a reference architecture and an implementation. The architecture and implementation are primarily designed to facilitate application deployment, but they also support some management operations to enable the management of deployed applications.

4.2 Deployment Architecture

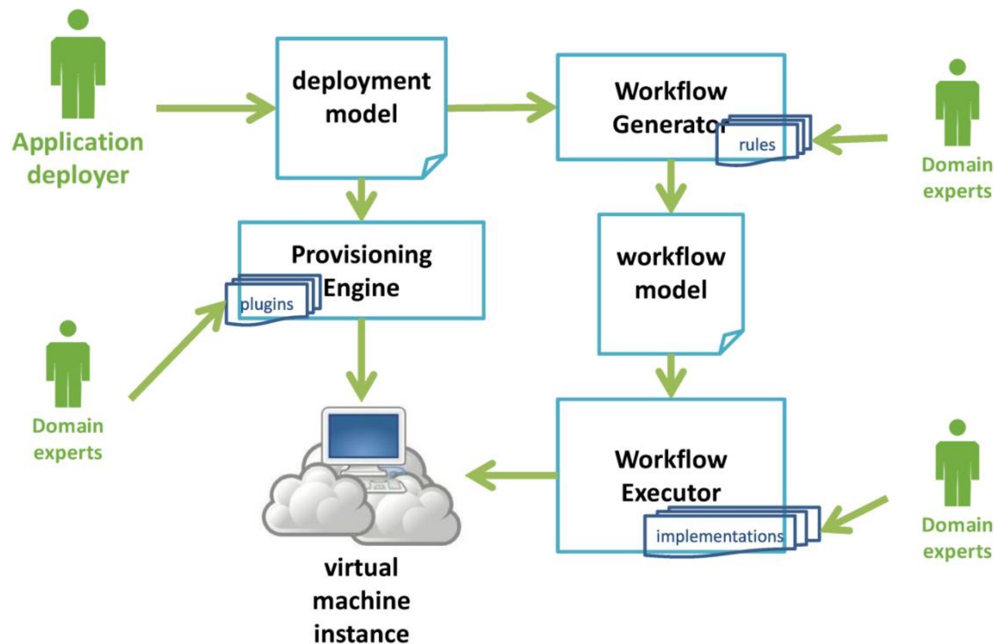

Figure 1: A reference architecture for multi-cloud application deployment and management

This section presents the reference architecture that encapsulates the common practices for implementing a multi-cloud deployment and management system. The reference architecture is showed in Figure 1. In the architecture, the application deployer defines an application deployment through a deployment model. A deployment model specifies (i) requirements of constructing the underlying infrastructure (i.e., what VMs to provision from which providers), (ii) the placement of the middleware components (i.e., on which VMs to place which middleware components), and (iii) the application deployment logic (i.e., how to configure and connect the middleware components to host the application). A deployment model is the input of the Provisioning Engine that is responsible for provisioning cloud resources. The Provisioning Engine is a generic framework for providing a resource provisioning service, but it does not contain any knowledge about specific clouds (e.g. the API specification of a specific cloud). The cloud-specific knowledge is provided by a set of *plugins*, which are typically developed by domain experts. The deployment model is also the input of the Workflow Generator that is responsible for generating a workflow model used to model the deployment workflows. The way to transform a deployment model to a workflow model is decided by a set of *rules*, which are typically defined by domain experts. The workflow model produced by the Workflow Generator is fed into the Workflow Executor that is responsible for executing the deployment according to the inputted model. The Workflow Executor is aware of the underlying environments (e.g. the OS of the host), and is able to executes the deployment workflows correctly in different environments. The knowledge about workflow executions in different types of environments is provided by a set of *implementations*, which is typically developed by domain experts as well.

4.2.1 The Models and the Metamodel

In most of the cases, application deployers have high-level knowledge of the deployment, like the structure of the application, the placement requirements of application components, etc. However, they are usually not familiar with low-level configuration details or domain-specific knowledge, like API specification of specific cloud, configurations of specific middleware, or comprehensive knowledge of specific OS. Based on the observation above, we promote a model-driven approach, in which application deployers specify an application deployment in a high-level deployment model. The deployment model can be automatically translated into a low-level workflow model that is used for executing the deployment. Both the deployment model and the workflow model need to conform to a metamodel.

4.2.1.1 The Metamodel

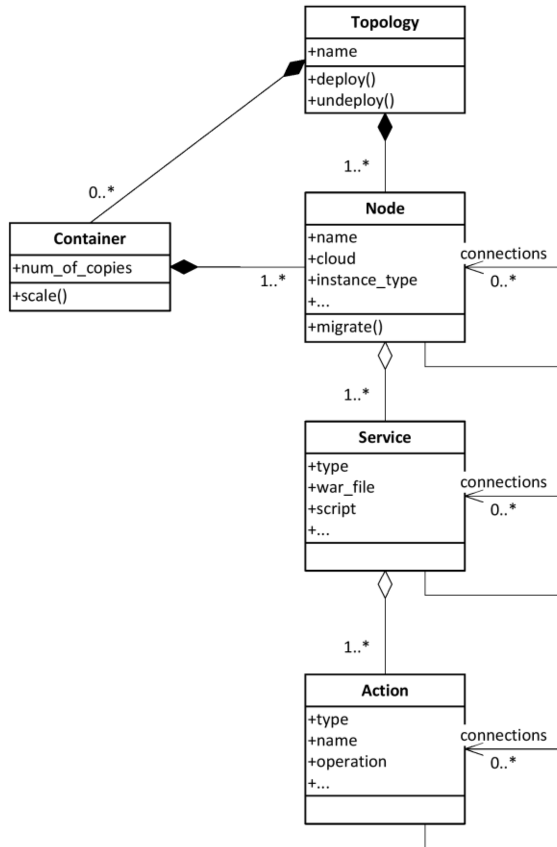

Figure 2: A set of portable abstractions

The metamodel, which is depicted in Figure 2, is composed by a set of *abstractions* and the *connections* between them. An abstraction represents a unit of concept in a deployment. There are five types of abstractions: topology, container, node, service, and action. A **topology** represents a complete application deployment. A topology has one or more nodes, and zero and more containers. A **node** represents a server used to host the application. In cloud environments, the representing server is a virtual machine instance (VMI) that is provisioned from a cloud provider. A node could establish a connection to another node to represent a kind of relationship between the two nodes (e.g. one node is nested inside another node). A **container** contains one or more nodes that need to be scaled as a unit. In a node, there are one or more **services**, each of which represents a software process running on the node for providing a type of service. For example, a MySQL[7] server process is running on a node for providing a database service. From an implementation point of view, deploying a service typically

[7] http://www.mysql.com/

requires installing and configuring a specific middleware component on the containing node, so a service can also capture this requirement. Similar to a node, a service can also establish a connection to other services to express the requirement for cross-configuring two middleware components (e.g. configure an application server to establish a JDBC connection to a database server). A service could be realized by one or more **actions**, each of which represents a unit of workflow to be performed. Examples of an action can be an execution of a Linux shell command to create a folder in the file system. Action is the lowest level building block of a deployment, and it encapsulates a unit of reusable implementation for building different services. An action could also establish a connection to another action, typically representing a communication channel used to send or receive messages at runtime.

In the metamodel, an abstraction (e.g. node, service, etc.) captures an abstracted concept in a deployment, without assuming its underlying cloud or technology. An abstraction could have multiple instances. An instance of an abstraction represents a concrete entity in a deployment. For example, a node represents a generic concept of a server, and an instance of a node represents a specific VMI launched on a specific cloud. From the functional point of view, an abstraction is used as a blueprint from which an individual instance is created. An abstraction supports a set of attributes that can be set by individual instance to describe itself. The attributes are of the key-value pair format, which allows them to be interpreted and modified easily.

Besides attributes, an abstraction optionally supports one or more operations that can be invoked from individual instance to perform specific tasks. The four frequently used operations are deploy, undeploy, scale, and migrate. The operation **deploy** is used to deploy the application represented by a topology. The operation **undeploy** is used to tear down a deployed application, and release the resources taken by the application. The operation **scale** is used to add/remove nodes from/to a cluster (the detail will be discussed in section 4.2.3.1). The operation **migrate** is used to migrate a node in an application (the detail will be discussed in section 4.2.3.2).

The abstractions and its main attributes of this metamodel (see Figure 2) are summarized as follows:

- topology: A topology represents a deployment of an application.
    - name: the name of the topology
- container: A container represents a collection of nodes that are expected to be scaled at runtime.
    - num_of_copies: the multiplicity of the nodes contained by the container

- node: A node represents a server. In cloud environments, it represents a VMI provisioned from a cloud provider.
    - name: the name of the node
    - cloud: the cloud provider of the node
    - credential: the credential used to log onto the cloud
    - image_id: the ID of the image that the node is based on
    - instance_type: the flavor of the cloud instance represented by the node
    - ssh_user: the username used for logging onto the node
    - key_pair_id: the ID of the private key used for logging onto the node
    - security_groups: the list of security groups used by the node
- service: A service represents a software process running on a node for providing a type of service.
    - type: the type of the service
    - war_file: a Java EE application archive file
    - script: the name of the database script, which typically contains SQL statements for creating tables for the application
- action: An action represents a unit of OS-level workflow.
    - type: the type of the action
    - name: the name of the action
    - operation: the operation to perform

4.2.1.2 Deployment Model

A deployment model is leveraged by application deployers to specify a deployment. As discussed before, a deployment model needs to conform to the metamodel. The main goals of using a deployment model is to (i) hide the low-level deployment workflows from application deployers thus enabling them to focus on the high-level deployment logic, and (ii) promote the portability of deployments by allowing application deployers to specify a deployment in a cloud-independent manner. A typical deployment model typically consists of an instance of *topology*, several instances of *nodes* and *services*, and the *connections* between them.

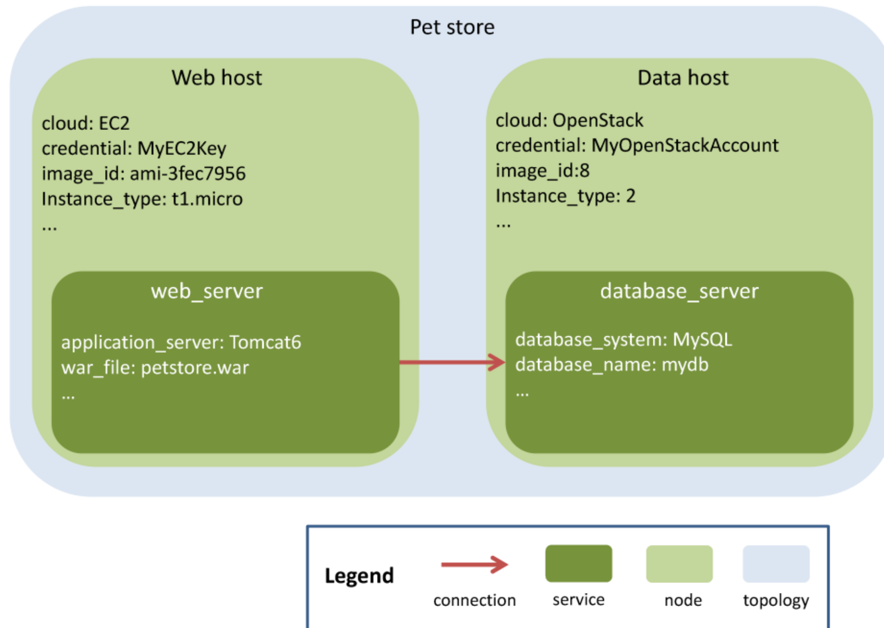

Figure 3: A sample deployment model

A sample deployment model is depicted in Figure 3. In this sample model, there are two nodes contained by the topology `Pet store`, which are called `Data host` and `Web host` respectively. The node `Web host` represents a VMI provisioned from Amazon EC2[8], and the node `Data host` presents a VMI provisioned from a private cloud based on OpenStack[9]. Each node has a set of attributes to specify additional information. For example, the attribute `credential` specifies the credential used for authenticating users against the cloud (the credential is required to be presented in the specified cloud and needs to be registered in the deployment system); the attribute `image_id` specifies the virtual machine image that the node is based on; the attribute `instance_type` decides the hardware capability of the node. A node can contain one or more services (although the case of multiple services in one node has not been demonstrated in the sample model) that represent a set of middleware components that need to be installed. For example, the service `web_server` represents a Tomcat 6 server that is used to host the application, and the service `database_server` represents a MySQL database server used by the application.

4.2.1.3 Workflow Model

A workflow model is used to describe the low-level deployment workflows. Same as a deployment model, a workflow model needs to conform to the metamodel. The main goal of using a workflow

[8] http://aws.amazon.com/ec2/
[9] http://www.openstack.org/

model is to capture the logic of the deployment workflows without assuming details about its execution environments, thus increasing the portability of the deployment. A workflow model needs to be auto-generated instead of being manually created, because it contains several low-level details that are hard to specify manually. A typical deployment model consists of an instance of *topology*, several instances of *nodes* and *actions*, and the *connections* between them.

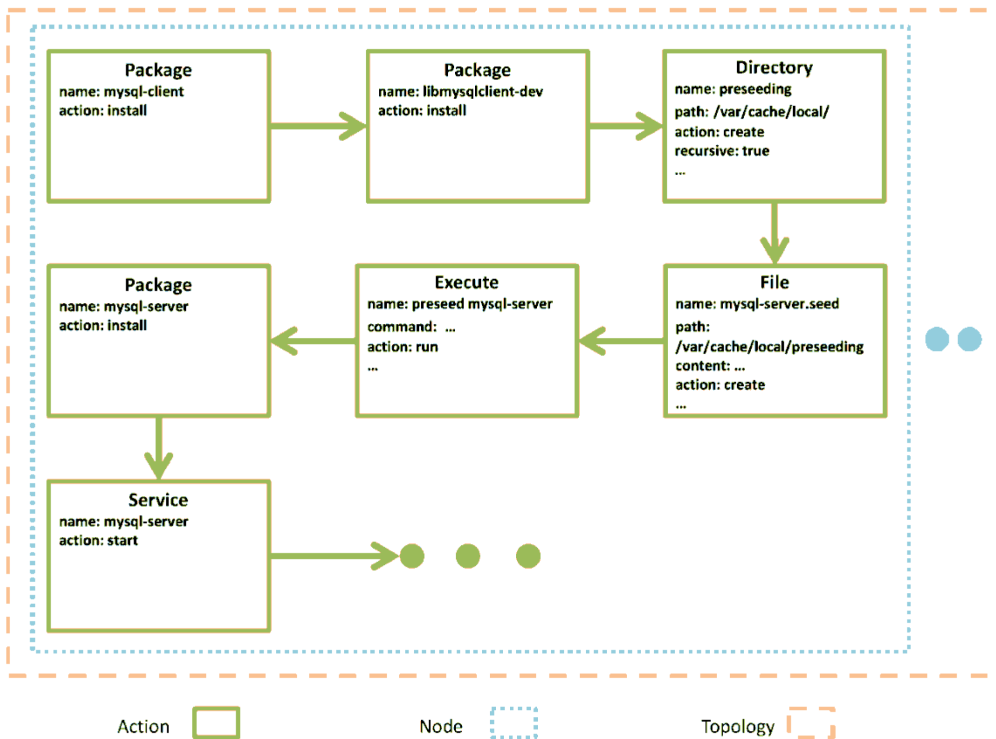

Figure 4: A sample workflow model that represents a deployment of MySQL database server

To gain insight into the workflow model, we show a sample workflow model in Figure 4, which represents the workflows for installing a MySQL database server. The workflow model contains a topology, which contains a set of nodes. In each node, there is a sequence of actions that represents the deployment workflows. In the figure, there are five types of actions: `Package`, `Directory`, `File`, `Execute`, and `Service`, which represents different types of workflows: a `package` represents a software package managed by an OS-specific package management system; a `Directory` and `File` represent, respectively, a folder and a file in the OS; an `Execute` represents an execution of a shell command; a `Service` represents an OS-level software service. Each action contains two common attributes: `name` and `operation`, which specify, respectively, the name and the operation that is going to be performed. The value of the attribute `operation` must be valid, which means the specified

operation must be supported. Different types of actions have different lists of supported operations. For example, an action of type `Package` supports five operations: `install`, `upgrade`, `reconfig`, `remove`, and `purge`. Some actions contain an additional set of attributes to further describe themselves. For example, the `File` contains the attributes `path` and `content` to specify the path of the file and the content of the file respectively.

4.2.2 The Software Components

The architecture contains three software components: the Provisioning Engine, the Workflow Generator, and the Workflow Executor. Each of them has different responsibilities: The Provisioning Engine is responsible for provisioning clouds resources; The Workflow Generator is responsible for generating deployment workflows; The Workflow Executor is responsible for executing the deployment workflows. All of them are designed to be pluggable, thus facilitating the extendibility of the architecture.

4.2.2.1 Provisioning Engine

The input of Provisioning Engine is a deployment model. After receiving the inputted deployment model, the Provisioning Engine parses it to retrieve a set of nodes, and provisions each of them. The attributes of individual node decides how the node will be provisioned. The connections between nodes might be leveraged to coordinate the procedures of provisioning different nodes. For example, if a connection indicates a dependency between two nodes, the depending node cannot be provisioned until the depended node completes its deployment tasks.

Because resources may need to be provisioned from multiple cloud providers, the Provisioning Engine needs to handle the heterogeneity of the API specifications of different clouds. To do that, the Provisioning Engine exposes a standardized interface that defines resource provisioning operations in a cloud-independent manner. The interface can be implemented by different *plugins*, each of which encapsulates the knowledge of resource provisioning in a specific cloud. The Provisioning Engine typically contains multiple plugins to support multiple clouds.

To provision an individual node, the Provisioning Engine reads the specification of the node, and leverages some operations defined in the interface to provision it. The plugin used to implement the interface is selected at runtime. In particular, the Provisioning Engine reads the cloud attribute specified in the node, and attempts to find a plugin that supports the specified cloud (or throws an exception if none is found).

4.2.2.2 Workflow Generator

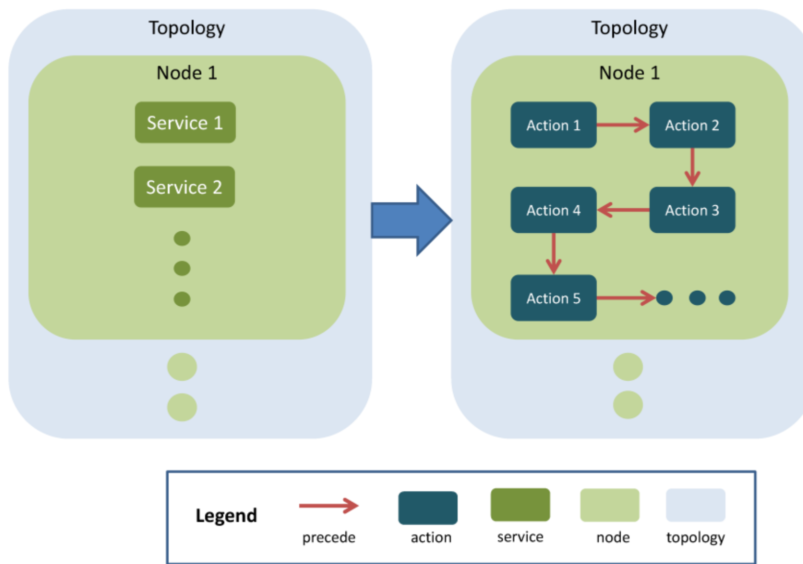

Figure 5: An overview of workflow generation process

The Workflow Generator consumes a deployment model that specifies the high-level structure of a deployment, and generates a workflow model that specifies the low-level workflows for realizing the deployment. An overview of the workflow generation process is shown in Figure 5. The left side of the figure shows a deployment model that is the input, and the right side of the figure shows a workflow model that is the output. Generally speaking, the Workflow Generator is responsible to convert individual service, which represents the requirement to install specific middleware, to a sequence of actions that represents the workflows for installing the middleware. The workflow generation process is guided by a set of *rules*, which encapsulate the knowledge for realizing the services. A rule is actually a template of workflow defined for a specific type of service. The Workflow Generator typically contains multiple rules to support multiple types of services.

The steps taken by the Workflow Generator to generate workflows are as follows:

1. Parse the inputted deployment model to retrieve the set of services in each node.
2. Convert each service to a sequence of actions that represents the workflow for realizing the service. To achieve this, the Workflow Generator reads the type of the service, and attempts to find a rule that matches the service's type (or throw an exception if none is found). Then, the Workflow Generator uses the selected rule, together with the specification of the

service and some properties of the execution environment, to generate a sequence of actions that represents the workflow for the service.
3. Aggregates the workflows generated for each service to produce the outputted workflow model.

4.2.2.3 Workflow Executor

The Workflow Executor takes a workflow model as an input and executes the deployment according to the model. The Workflow Executor contains a set of *implementations*, each of which encapsulates the knowledge for executing a type of action. An implementation contains one or more *executors*, each of which can execute the action in a specific type of environment (e.g. a specific type or version of OS). Portability of an action is achieved by creating different executors to support different execution environments. For example, to enable the portability of an action of type `Package` (which represents an installation of a software package), multiple executors are needed to support different package management systems, such as APT[10], YUM[11], RPM[12]. As a result, the action can be executed different OSs who use a package management system supported by an executor.

To execute the deployment workflows, the Workflow Executor parses the inputted workflow model to retrieve a set of nodes. For each node, the Workflow Executor obtains the sequence of actions specified inside, and executes each of them in order. The execution of individual action is performed as follows:

1. The Workflow Executor attempts to find an implementation that supports the type of the action (or throw an exception if none is found).
2. The Workflow Executor attempts to find an executor in the implementation, which matches the properties of the execution environment as much as possible (or use the default executor if none is matched).
3. The chosen executor executes the action according its specification.

4.2.3 The Management Operations

After applications are deployed, they may need to be managed at runtime. For example, more resources might need to be allocated to an application, when there is a spike of its workload. Like application deployment, application management is often challenging due to the complexity of enterprise

[10] https://wiki.debian.org/Apt
[11] http://yum.baseurl.org/
[12] http://www.rpm.org/

applications. To address the challenges, we introduce two management operations, which can be used to reconfigure deployed applications, thus facilitating the management of them.

A management operation is defined on an abstraction (e.g. a *node*, a *container*, etc.), and exposed by individual instance of the abstraction. A management operation can be invoked by application administrators to modify the state of its associated abstraction instance (e.g. change its attributes or connections). A state change of an instance is automatically translated into the low-level workflows used to reconfigure the application.

In this section, we introduce two management operations: *scale* and *migrate*. The operation *scale* is used to add/remove cloud resources from/to a deployed application. The operation *migrate* is used to migrate a node in a deployed application.

4.2.3.1 Scale

The management operation *scale* is defined in the *container* (as shown in Figure 2), which represents a collection of nodes needed to be scaled at runtime. A container has an attribute `num_of_copies` that specifies the multiplicities of each node in the container. The operation *scale* is used to modify the attribute `num_of_copies` of an instance of container, thus adding/removing nodes to/from the container.

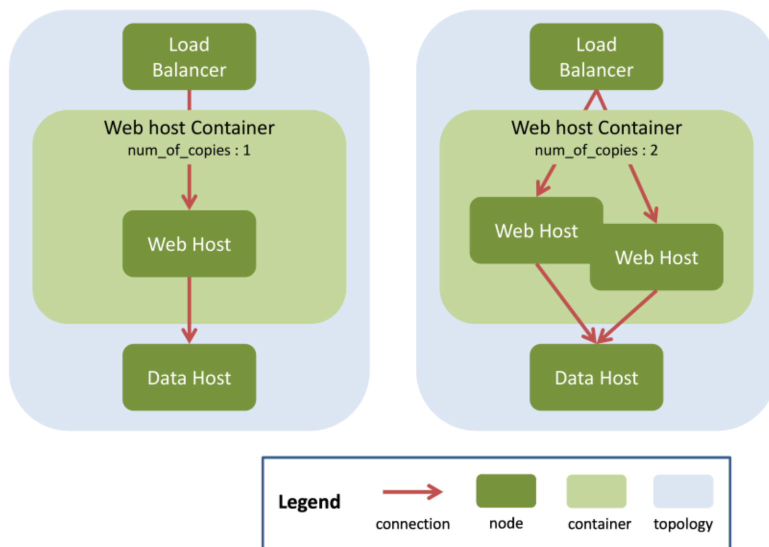

Figure 6: An illustration of scale operation

To gain inside into this operation, we present a sample usage that is illustrated in Figure 6. The left side of the figure shows a topology representing an application deployed on three nodes: the `Load`

Balancer, the Web Host, and the Data Host. The node Web Host hosts an application server. The node Data Host hosts the database of the application. The node Load Balancer hosts a proxy used to dispatch requests from end-users to an application server. To enable elasticity, a container Web Host Container is used to hold the Web Host. Initially, the Web Host Container has its attribute num_of_copies set to value 1, which indicates exactly one application server is deployed. Suppose, at a point in time, an application administrator decides to add one more application server to the deployment. To achieve the goal, he or she invokes the operation *scale* on the Web Host Container to set num_of_copies to 2. After this invocation, another Web Host is provisioned, and another application server is deployed to the new Web Host (as shown at the right side of the Figure 6). The new application server is deployed and configured in the same way as the old one, so they have the same set of configurations (e.g. both of them connect to the same database). After the new application server is successfully deployed, the Load Balancer is reconfigured to dispatch requests equally to the two application servers.

By leveraging this operation, it is possible to scale an application automatically based on a user-specified elasticity policy [54]. A use case has been presented in our paper [55], which introduces a cross-cloud application management platform (XCAMP). In XCAMP, applications are automatically scaled out or scaled in based on (i) the monitored resources' utilizations and (ii) a user-specified management logic. XCAMP relies on this operation to execute the scaling of applications. The details of XCAMP are out of the scope of this thesis and interested readers are directed to [55].

4.2.3.2 Migrate

The management operation *migrate* is defined in the *node* (as shown in Figure 2), and used to migrate an instance of node that is typically used for hosting application components. To leverage this operation, the migrated node must represent a VM, and connect to another node that represents the server for hosting the VM (the server contains a hypervisor that runs the VM). For clarification purpose, we use the term *host node* to refer a node for hosting a hypervisor, and the term *guest node* to refer a node that is hosted by a host node. The operation *migrate* is invoked from a guest node that has a host node connected, and reconfigures the guest node to connect to another host node. After this operation is invoked, the guest node is migrated from a host node to another host node.

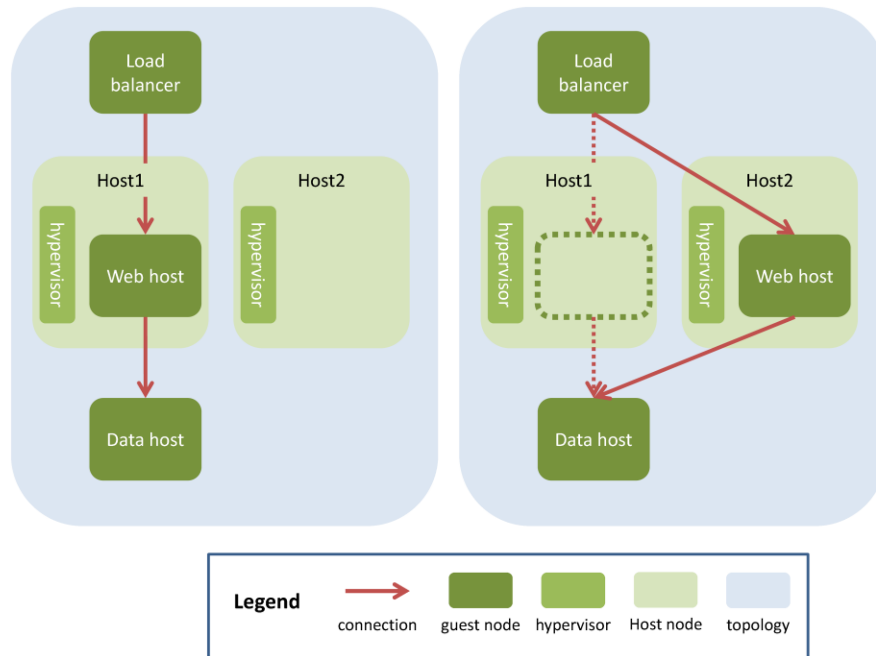

Figure 7: An illustration of a VM migration

To gain insight into this operation, we present a sample usage that is illustrated in Figure 7. The left side of the figure shows a topology representing an application deployed on three nodes (same as Figure 6). Initially, the `Web Host` is hosted by the `Host1`. At a point in time, an application administrator invokes the operation *migrate* on the `Web Host` to migrate it to the `Host2`. After this invocation, the `Host2` is reconfigured to open a channel for receiving the incoming migration traffic. Then, an instruction is sent to the hypervisor running on `Host1` for performing the VM migration. Finally, the `Load balancer` is reconfigured to update the IP address of the migrated `Web host` (since the IP address of the migrated node will change after a migration).

4.3 Implementation

In this section, we present an implementation of the deployment architecture (shown in Figure 1). The implementation is a web application called Pattern Deployment Service (PDS) [56]. To realize the architecture, PDS provides implementations of its software components (e.g. the Provisioning Engine, the Workflow Generator, and the Workflow Executor). Furthermore, the deployment model in the architecture is represented by an XML document in PDS.

PDS is developed for the SAVI project, and has been widely used in the SAVI research community, which currently consists of researchers and engineers from 9 universities and 20 industrial

partners. In this section, we briefly introduce the implementation of PDS. More details can be found in our paper [56] and the online document[13].

4.3.1 XML-based DSL

In PDS, a deployment model is represented by an XML document called a system pattern description (SPD), which is written in a DSL. To specify an application deployment, users can author a SPD file to describe the deployment, and submit the SPD file to PDS. When PDS receives a SPD file, it will parse the file, and create an instance of topology that represents the application. The topology instance contains a set of node instances that represent the set of servers used to host the application. Each node instance contains a set of service instances that represent the middleware components. If container(s) is/are specified in the SPD, a set of container instances are created and added to the topology instance as well. In PDS, an instance of topology, node, container, or service is implemented as a Restful resource, which can be created, read, updated, and deleted through the APIs (the specification of the API is presented in Appendix C). For example, to deploy the application, users can invoke the `deploy` operation through the `modify` method in the topology resource (see Table 5 in Appendix C).

```
<topology id="petstore">
  <node id="webHost">
    <cloud>EC2</cloud>
    <use_credential>MyEC2Key</use_credential>
    <key_pair_id>MyEC2Keypair</key_pair_id>
    <ssh_user>ubuntu</ssh_user>
    <image_id>ami-3fec7956</image_id>
    <instance_type>t1.micro</instance_type>
    <security_groups>PDS-MySQL,PDS-AppServer</security_groups>
    <service name="web_server">
      <database_connection node="webHost"/>
      <war_file>
        <file_name>petstore.war</file_name>
        <datasource>jdbc/pet</datasource>
      </war_file>
    </service>
    <service name="database_server">
      <script>petstore.sql</script>
    </service>
  </node>
</topology>
```

Figure 8: A sample deployment description file

[13] https://github.com/ieraslabs/pattern-deployer/wiki

```
<topology id="petstoreMultiNodes">
  <instance_templates>
    <template id="EC2Instance">
      <cloud>EC2</cloud>
      <use_credential>MyEC2Key</use_credential>
      <image_id>ami-3fec7956</image_id>
      <instance_type>t1.micro</instance_type>
      <ssh_user>ubuntu</ssh_user>
      <key_pair_id>MyEC2Keypair</key_pair_id>
    </template>
  </instance_templates>
  <container id="webHostContainer" num_of_copies="2">
    <node id="webHost">
      <use_template name="EC2Instance"/>
      <security_groups>PDS-AppServer</security_groups>
      <service name="web_server">
        <database_connection node="dataHost"/>
        <war_file>
          <file_name>petstore.war</file_name>
          <datasource>jdbc/pet</datasource>
        </war_file>
      </service>
    </node>
  </container>
  <node id="dataHost">
    <use_template name="EC2Instance"/>
    <security_groups>PDS-MySQL</security_groups>
    <service name="database_server">
      <script>petstore.sql</script>
    </service>
  </node>
  <node id="webBalancer">
    <use_template name="EC2Instance"/>
    <security_groups>PDS-AppServer</security_groups>
    <service name="web_balancer">
      <member node="webHost"/>
    </service>
  </node>
</topology>
```

Figure 9: A deployment description file that describes a multi-node deployment

Two sample SPDs written in the DSL are shown in Figure 8 and Figure 9. Most of the XML elements used in the sample SPDs (e.g. topology, node, service, etc.) can be matched to an entity or an attribute in the metamodel (discussed in section 4.2.1.1). There are two special elements: `template` and `use_template` in the SPD shown in Figure 9. The element `template` is used to define a template of a node, which can be referenced by using the element `use_template`. By properly using these two elements, users can consolidate the repeated declarations into a template, thus improving the maintainability and readability of the SPD.

The design of the DSL and SPD provides several advantages. First, the syntax of the DSL is comprehensive, and the structure of a SPD naturally represents the structure of an actual application deployment, both of which effectively facilitate the deployment specification process. Second, a SPD is of XML format, which enables it to be validated easily by using an XML schema. Validation is useful for capturing human errors before the deployment is executed.

4.3.2 Implementations of the Software Components

Both the Provisioning Engine and the Workflow Generator take a deployment model (which is actually a SPD) as an input. Upon receipt of the deployment model, the Provisioning Engine provisions each node from the specified cloud. Then the Workflow Generator dynamically generates the deployment workflows, which is executed by the Workflow Executor. To facilitate this process, the PDS parses the inputted deployment model and construct a graph in memory. The in-memory graph is used by both the Provisioning Engine and the Workflow Generator to facilitate their respective tasks. The algorithms for creating and using the in-memory graph are showed in Appendix A.

Provisioning Engine: As mentioned before, the Provisioning Engine defines a standardized interface for provisioning of cloud resources, and the interface is implemented differently by different *plugins*. In PDS, we implemented *plugins* by using Chef Plugins. Chef is the configuration management tool we use by us for configuring provisioned nodes. Some Chef Plugins can extend the functionalities of Chef to support resource provisioning on different clouds. Currently, two Chef Plugins have been integrated to PDS: the EC2 plugin and the OpenStack plugin. These two plugins were originally developed by Chef's community and modified by us to meet our needs. Besides EC2 and OpenStack, there are other Chef plugins that can be easily integrated to PDS, such as the IBM smart cloud plugin, the Rackspace plugin, etc.

Workflow Generator: The Workflow Generator takes a deployment model as input and dynamically generates a workflow model based on a set of *rules*. We defined each *rule* in a Chef Recipe, which is a Chef's component for holding a unit of workflow template. At runtime, the Workflow Generator parameterizes each Chef Recipe with the inputted deployment model, as well as the properties of the execution environment. PDS detects the execution environment at runtime by using a third-party tool Ohai[14]. Ohai can detect various kinds of configurations of the host such as OS configurations, network configurations, CPU capacity, Memory capacity, Kernel configurations, etc. In the current state of PDS, we defined 9 rules for supporting 9 services. The details are showed in Appendix B.

Workflow Executor: The Workflow Executor is responsible for executing the deployment according to the inputted workflow model. As mentioned before, the basic building block of a workflow model is *action*, and each type of action has one or more *executors*. In PDS, an action is implemented by

[14] http://docs.opscode.com/ohai.html

using a Chef Resource, and an *executor* is implemented by using a Chef Provider. A Chef Resource is used to specify the desired state after the action is executed, and a Chef Provider defines the steps for realizing the desired state in a specific environment. The set of Chef Resources and Chef Providers provided by Chef is able to cover most of our use cases, and the uncovered use cases are handled by implementing additional set of Chef Resources and Chef Providers.

4.4 Experiments

In this section, we present two experiments we conducted to evaluate the performance of PDS. The main objectives of these experiments are to (i) demonstrate the feasibility of leveraging PDS to deploy applications on different clouds, and (ii) demonstrate the performance of PDS, and (iii) evaluate the scalability of PDS.

In the first experiment, we deployed the testing applications to a set of VMIs based on a standard Ubuntu cloud image. In the second experiment, we deployed the testing applications to a set of VMIs based on a custom image made by us. In both experiments, an instance of PDS is hosted by a VMI with 4 virtual CPU and 8G of memory. We simulated a number of concurrent users by using JMeter[15]. Each simulated user performed the following operations:

1. Deploy: Provisioned 3 VMIs from the cloud, and configured them to host a JEE application called Petstore[16]. In particular, the 3 VMIs were configured as an application frontend, an application server, and a database server respectively. Each provisioned VMI has 1 virtual CPU and 2G of memory.
2. Scale out: Scaled the topology from 3 nodes to 4 nodes. The additional node was used to host a replica of the application server.
3. Scale in: Scaled the topology back to 3 nodes.
4. Undeploy: Terminated all the VMIs that were provisioned.

To evaluate the performance, we profiled the times of performing each operation above. To increase the reliabilities of the result, we repeated each experiment 3 times, and computed the average performance of them.

[15] http://jmeter.apache.org/
[16] http://beehive.apache.org/docs/1.0.2/samples/petstore.html

4.4.1 Experiment 1

This experiment was conducted on the SAVI testbed [57] and Amazon EC2. Generally speaking, the SAVI testbed is a cloud that has limited amount of computing resources. In contract, EC2 is a public cloud that is almost unlimited in capacity. In this experiment, the application was deployed to a set of VMIs based on an Ubuntu cloud image. This is a stripped-down image that basically contains only the OS, so that deploying applications on this image needs to go through all the steps, including (i) downloading scripts and/or artifacts from the PDS instance, (ii) downloading and installing a few software packages by using the built-in package management system, (iii) configuring the installed software packages to host the application.

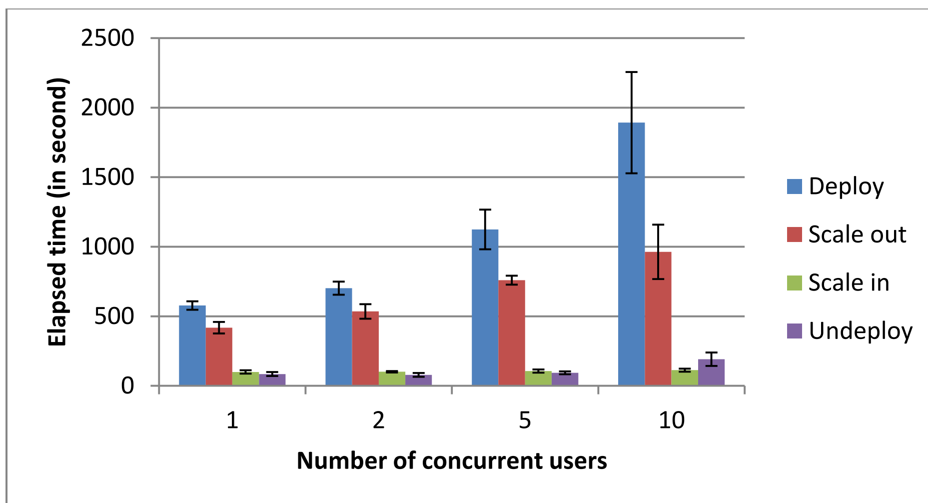

Figure 10: The performance of PDS in SAVI testbed

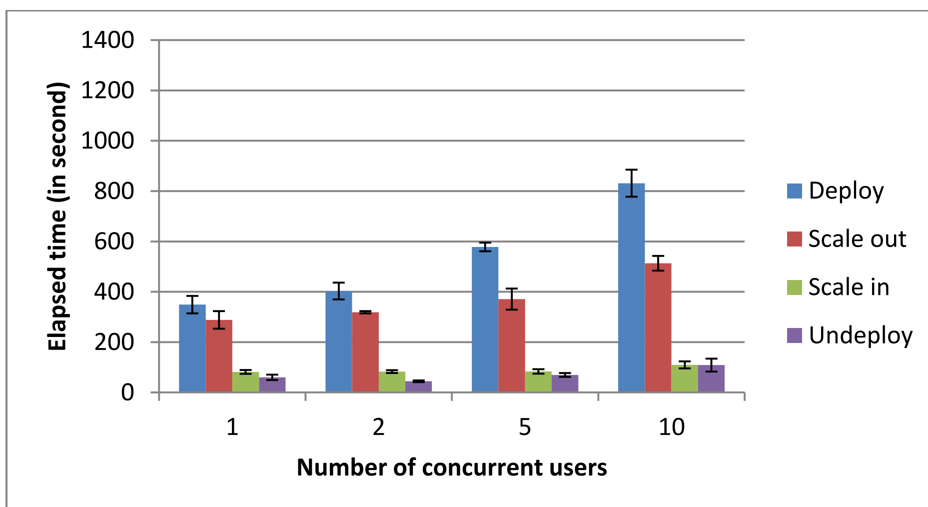

Figure 11: The performance of PDS in Amazon EC2

The results are showed in Figure 10 and Figure 11. These two figures show the performance of the PDS on the SAVI testbed and EC respectively. As showed in the figures, deploying a 3-node topology takes between 500 to 1200 seconds on the SAVI testbed, and between 350 to 800 seconds on the EC2. Scaling out a deployed application takes about 300 to 700 seconds. The other operations take no more than 100 seconds. An observation is that the time spent on `Deploy` and `Scale out` increases gently as the number of users increases.

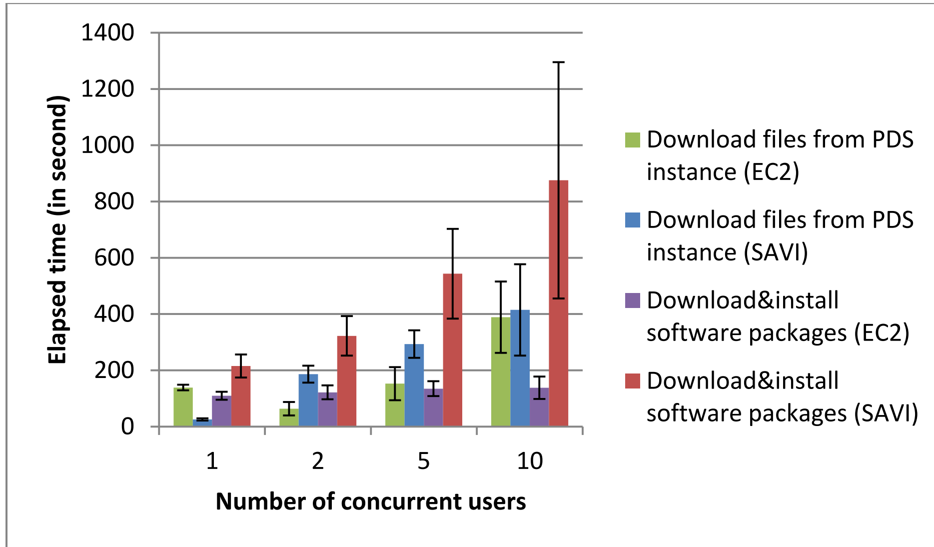

Figure 12: A break down of performance overhead

To break down the performance overheads, we analyzed the logs to find out the particular deployment tasks that contributed most to the deployment time. Our finding is summarized in Figure 12. There are two heavy deployment tasks: file downloading from PDS instance and package installation. As shown in the figure, the time spent on file downloading increases proportionally with the number of users, which implies that the PDS instance could be a bottleneck if the number of users becomes large. In addition, the time spent on packages installations has different characteristics on EC2 and the SAVI testbed. On EC2, the packages installation time is independent of the number of users. On the SAVI testbed, the packages installation time grows dramatically as the number of users increases. For this issue, we consulted with the testbed development team. Based on the feedback from the team, the thrashing of performance is due to the limited capacity of the cloud. This implies that the scalability of the underlying infrastructure might impact the performance of PDS.

4.4.2 Experiment 2

In this experiment, we deployed the testing applications on a custom image that has most of the needed files/packages pre-downloaded/pre-installed. During the deployment, the PDS detects the presence of the needed files and packages, and skip the corresponding steps for downloading or installing them, thus reducing the deployment time. Beside the differences of the based image, the setting of this experiment is the same as the previous experiment. Finally, we compare the result of this experiment with the previous experiment to show the performance differences of using different images.

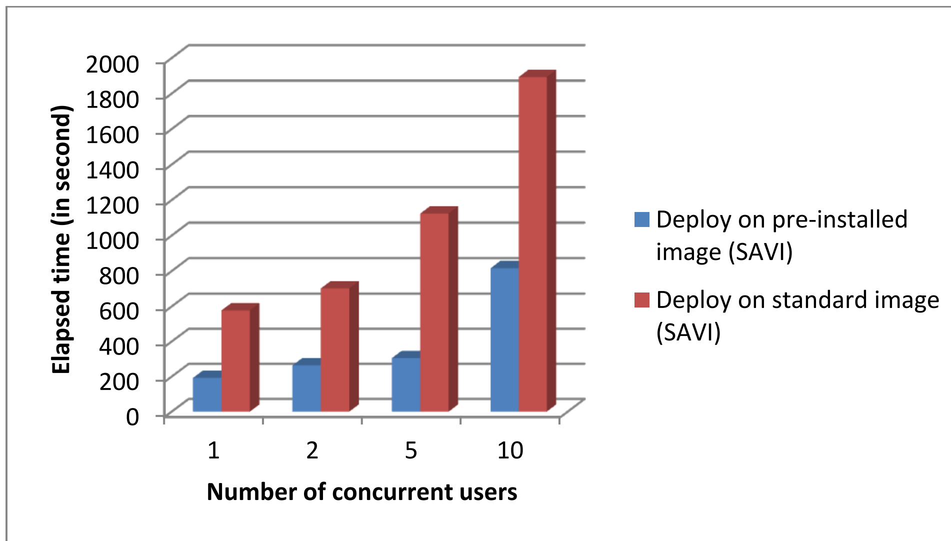

Figure 13: Performance comparison of different images

The results are presented in Figure 13, which demonstrate that we can save about half of the deployment time by caching the needed resources into a cloud image. However, a drawback of this approach might be the extra works to pre-build a dedicated image in individual clouds. Users need to trade off the amount of work and the performance gain.

4.4.3 Threats to validity

This section discusses the threats to the validity of these experiments. In particular, we describe the conditions in which these experiments were taken, and the potential factors which might prevent the results of these experiments from being reproduced.

Probably, the main threat to the validity of these experiments is the versioning of the PDS and its dependencies. For example, it is possible that a newer version of PDS is released whose performance is not consistent with the result of these experiments. Furthermore, PDS depends on a large set of third-

party libraries. It is possible that a newer version of a depending library releases which breaks the build of PDS. In fact, that happened before.

Another threat to the validity might be the use of a specific testing topology. In particular, the testing topology we used is a 3-node topology, which is used to deploy a sample JEE application. Obviously, if different topologies are used or different applications are deployed, users might be observed different performance characteristics.

Lastly, these experiments were conducted on specific clouds. If applications are deployed to a different cloud, users might also observe differences in performance, because different clouds have different hardware and software configurations.

4.5 Summary

This chapter presented a reference architecture that encapsulates the common practices to implement an application deployment solution in multi-cloud environments. In the architecture, an application deployer specifies a deployment model. The high-level deployment model is automatically translated into the low-level deployment workflows that will be executed to achieve the deployment. The architecture consists of several pluggable components that allow users to extend its functionalities on the fly. We implemented the proposed architecture in Pattern Deployment Service, which provides application deployment as a cloud service. We conducted the experiments to demonstrate the performance of PDS.

Chapter 5: Achieving Adaptation through VM Migration

As mentioned before, there are two types of clouds in the SAVI architecture: core and smart edge. To maximize the performance of applications, administrators typically prefer to place their applications on the smart edge, which is closest to the end-users. The core can be leveraged when the amount of resources in the smart edge is insufficient. Placement of applications can be static or dynamic.

In the simplest case (i.e., static application placement), administrators would estimate the peak demand for the application, and determine the amount of resources to provision in order to meet the estimated peak demand. An obstacle for static application placement is the inherent difficulties for estimating demands of applications. In addition, provisioning resources for peak demands typically leads to an underutilization of cloud resources.

Approaches for enabling dynamic application placement can overcome the weaknesses of static application placement approach. In this type of approach, a deployment system decided the amount of needed resources according to the steady demands of applications (instead of the peak demands). At runtime, if a growth of demands yields a hotspot, applications (or components of the applications) are dynamically moved out of the hotspot. The invention of the live VM migration technique provides a seamless way to re-locate VMs that host the applications without disrupting their services. On the other hand, if a drop of in demand leads to a low utilization of resources, VM migration can also be leveraged to dynamically consolidate the applications.

It is possible to manage VM migrations manually. However, due to the scale of the infrastructure and complexities of the management tasks, migrations need to be managed automatically by a migration management system. Applications with their migration management system form a self-adaptive system, in which applications automatically adapt to changes in their demand by triggering VM migrations. In this chapter, we focus on constructing a self-adaptive system described above. We propose a model-driven approach for managing migrations, thus simplifying the migration management tasks.

The main contributions of this chapter are as follows:

- A migration management system for two-tier cloud is presented.
- A model-driven approach for managing the complexities of the VM migration management tasks is described.

- A reference architecture that encapsulates the common practices for implementing a migration management system in a two-tier cloud is presented.
- An algorithm for planning VM migrations is proposed. This algorithm can be parameterized with a user-defined policy and workload-prediction model to optimize the migration-related decisions, such as deciding when a migration is triggered, which VM is selected to be migrated, where the VM is migrated to.
- A prototype implementation and set of evaluation experiments for validating this approach are presented.

This chapter is organized as follows: section 5.1 provides an overview of the migration management system; section 5.2 presents the migration management architecture and algorithm; section 5.3 demonstrates the experimental evaluation.

5.1 Overview

We propose a model-driven approach to facilitate the management of VM migrations in two-tier cloud environments. By definition, a two-tier cloud could consist of multiple cores and multiple smart edges. Without loss of generality, we consider a specific two-tier cloud composed of one core and one smart edge. In a two-tier cloud in this context, VMs are migrated between (i) two machines in the smart edge, which is referred to as *local migration*, or (ii) between a machine in the smart edge and a machine in the core, which is referred to as *cross-cloud migration*. Typically, a local migration is preferred to a cross-cloud migration, because it incurs less performance penalties. However, if the smart edge is over utilized or predicted to be over utilized, cross-cloud migrations need to be used to move some VMs off the smart edge.

Due to the differences between local and cross-cloud migrations, it might be necessary to separate their management logic. Furthermore, to optimize the management decisions, users should be able to employ different models, policies and/or parameters for managing the two types of migrations. To facilitate this, we propose a migration management system in which the logics of migration management are separated into two components: one dedicated for managing the local migrations, and the other dedicated for managing the cross-cloud migrations. These two components can execute independently, but they could communicate with each other to coordinate the management decisions. The goal of this design is to promote the separation of concerns. As a result, the logic for managing one type of migration can evolve independently of the logic of managing the other type of migration.

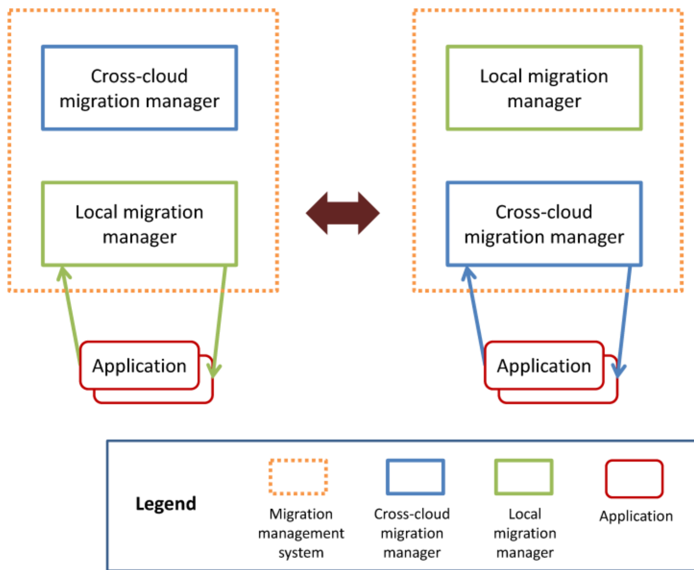

Figure 14: An overview of the migration management system

The proposed design of a migration management system is depicted in Figure 14. In this design, a migration management system consists of two components: the local migration manager and the cross-cloud migration manager. The local migration manager is responsible for detecting potential problems (e.g. over-utilized or under-utilized of resources) in the smart edge, and planning and executing local migrations to resolve those problems. Similarly, the cross-cloud migration manager detects potential problems in the smart edge, but it plans and executes cross-cloud migrations instead (problems in the core might also need to be detected and resolved, but that is out of our considerations). To avoid the possibility of conflicts between management decisions, the management system might employ a mechanism to coordinate the executions of these two managers. In particular, a rule is enforced to ensure that only one migration manager can be running at any given point in time. Under this regulation, there are two possible states of the system, which are showed, respectively, in the left and right side of Figure 14. At the left side of the figure, the local migration manager is running and the cross-cloud migration manager is waiting to be run. In a system in this state, the applications are under the management of the local migration manager, so that only local migrations are planed and executed. At some time, the management system may switch to the other state (as presented at the right side of Figure 14), in which the cross-cloud migration manager is running and the local migration manager is waiting. In a system in this state, only cross-cloud migrations are planed and executed. The decisions for switching from one state to the other depend on an administrator-defined scenario. An

example of such scenario could be: run the cross-cloud migration manager if the utilization of the smart edge is greater than 60% (the goal is to migrate some VMs out of the smart edge). The utilization of the smart edge is defined to be the average utilizations of each host in the smart edge.

This thesis focuses on the cross-cloud migration manager, which is discussed in section 5.2. For the implementation of the local migration manager, the same approach can be leveraged as well, but it is not our focus.

5.2 Cross-cloud Migration Management

In the state-of-the-art, researchers mainly focus on finding an optimal strategy, algorithm or scenario for managing local migrations. However, only a few researchers focus on the management of cross-cloud migrations and on managing frameworks. In this section, we propose a model-driven approach to facilitate the cross-cloud migration management.

5.2.1 Model-driven Migration

The goal of a VM migration is to transfer the state of the VM from one host to another. With existing technologies [10] [40], it is possible to transfer the state of CPU, memory, and disk of the migrated VM, without disrupting the service provided by the migrated VM. However, maintaining the network connections of a migrated VM could be challenging. Clark et al. [10] proposed a method for maintaining the IP address and all the open network connections of a migrated VM, but this method is not applicable to cross-cloud migration (it is applicable to local migration only). Bradford et al. [40] proposed a wide-area network redirection method, in which a dynamic DNS is utilized to resolve the IP address of the migrated VM. As a result, the IP address of the migrated VM can be updated in the dynamic DNS to ensure the consistency. By leveraging this method, it is possible to handle the change of the IP address without reconfiguring the application. However, this method requires a non-trivial setup of the network infrastructure which makes it difficult to use.

To address the challenge, we propose reconfiguring the application to adapt to the change of the IP address due to a migration. Generally speaking, there are two cases to be considered:

1. The migrated VM hosts an internal service. For example, the migrated VM hosts a replica of an application server that is used by a load balancer (not directly used by end-users).
2. The migrated VM hosts a service that is used by end-users directly.

To handle the first case, we expose the structure of the application to retrieve a set of nodes, called dirty nodes, which connects to the migrated VM. After the migration completes, we remotely access the set of dirty nodes and reconfigure the corresponding middleware component(s) to update the IP address. The reconfiguration process typically involves re-writing the IP address in the middleware-specific configuration files, and restarting the middleware component to pick up the new configurations. To handle the second case, we introduce a frontend component that is used to redirect the requests from end-users to the migrated VM. The change of the IP address is updated in frontend component, thus being hidden from the end-users.

To streamline the migration process, we propose a model-driven migration approach, in which application administrators can specify and execute a migration by modifying a model that represents the current structure and configurations of a deployed application in a high-level manner. The model can be modified by leveraging the provided management operations. After the model is modified, the system validates the modification, and automatically translates it to the low-level workflow for executing the migration. After the migration is successfully executed, the modified model is stored in the system to record the current state of the application. The model-driven approach in general has been presented in Chapter 4, and the management operation used to execute VM migrations has been discussed in section 4.2.3.2.

By leveraging the model-driven approach, the low-level details of migration workflows are hidden from application administrators, which enable them to focus on the high-level specifications of migrations. In addition, some post-migration tasks, such as updating the IP address of the migrated VM, can be automated and internalized, which reduce the complexities of managing the migrations.

5.2.2 Architecture

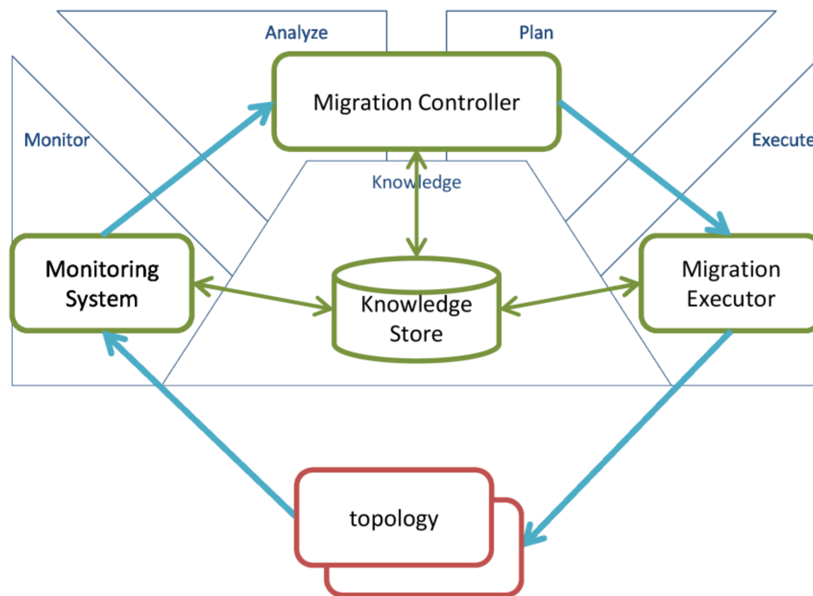

Figure 15: An architecture of cross-cloud migration management system

In this section, we present a reference architecture that encapsulates the common practices for implementing a cross-cloud migration manager. The proposed architecture, which is designed according to the architectural approach of autonomic computing [27] [28], is depicted in Figure 15. The architecture consists of an MAPE-k loop (monitor-analyze-plan-execute-knowledge) and a set of topologies. The set of topologies are the models of the applications that are under management. The MAPE-k loop (i.e., as implemented by the autonomic manager) is responsible for managing the cross-cloud migrations within the topologies. The implementation of the MAPE-k loop consists of four components: Monitoring System, Migration Controller, Migration Executor, and Knowledge Store. The Monitoring System is responsible for gathering necessary metrics of each application and its external environment. The collected metrics are analyzed by the Migration Controller to determine if one or more migrations need to be triggered. Additionally, the Migration Controller generates a migration plan that is passed to the Migration Executor to execute. During the management process, the Monitoring System, Migration Controller, and Migration Executor will read/write information from/to the Knowledge Store, which is a component for storing the needed data.

The **Monitoring System** is a component that is responsible for monitoring the deployed applications and the state of the underlying infrastructures. The Monitoring System collects three kinds of statistics: OS-level metrics, hypervisor-level metrics, and application-level metrics. Examples of OS-

level metrics include CPU utilization, memory utilization, network congestion, etc. For each topology, the OS-level metrics are collected on a per-node basis, and the collected metrics are aggregated for analyzing the state of the managed elements. The Monitoring System also collects hypervisor-level metrics that indicate how resources are shared between different domains, and some application-level metrics that exposes the characteristics of the workloads. Those metrics are useful for deciding and planning the potential migrations.

The metrics can be collected directly or indirectly. Direct metric collection normally requires logging onto the OS of the monitored machine and installing a monitoring agent on it. Administrators can implement their own monitoring agent or leverage existing monitoring tools, such as Nagios[17] or Ganglia [58]. If the machine that needs to be monitored is owned by a third-party, such as a public cloud provider, the required metrics may need to be collected indirectly, possibly through a monitoring-as-a-service (MaaS) interface. An example of MaaS interface is Amazon CloudWatch[18] that exposes various metrics about the resources in EC2 cloud. It is possible that some metrics cannot be collected by the Monitoring System (due to the ownership of the resources, security requirements etc.). In this case, the management system will attempt to use the incomplete set of metrics.

Probably, the most challenging problem for implementing a Monitoring System is the heterogeneity of the two-tier cloud environments. Due to the varieties of monitoring technologies and monitoring targets, the Monitoring System needs to support (or easily add support for) various kinds of monitoring tools or MaaS interfaces. Furthermore, the metrics collected from different sources are possibly of different formats and expose different levels of granularities. The Monitoring System needs to process the collected metrics and expose them in a standardized and consistent manner. Finally, the Monitoring System must be scalable and fault-tolerant. How to develop a system for monitoring heterogeneous collection of resources has been studied intensively by Smit et al. [59]. However, for this study, we implemented a much simpler Monitoring System as a proof-of-concept.

In our prototype, we leverage Ganglia [58] for collecting metrics from each node. In particular, a Ganglia's monitoring agent is installed in each node by the PDS at deployment time. This installation is specified by adding a service `monitor` (see Appendix B) to each node in a deployment model. For each node with the service `monitor`, PDS installs a monitoring agent on it. After the application is deployed, the installed monitoring agent will launch a daemon process for collecting metrics. By default, a

[17] http://www.nagios.org/
[18] http://aws.amazon.com/cloudwatch/

predefined set of OS-level metrics are collected, which includes CPU usage and memory usage. We extended the default behaviour for collecting an additional set of hypervisor-level metrics, such as the CPU allocation for each domain. In each node, the collected metrics are aggregated and exposed through a Web interface, which is used by the Monitoring System to pull metrics from each node remotely. The metrics pulled by Monitoring System are processed and stored in the Knowledge Store for later retrieval.

The **Migration Controller** is the component for managing the cross-cloud migration. It analyzes the metrics collected by Monitoring System to decide whether one or more migrations need be triggered. Migration(s) should be triggered only if necessary, since it perturbs the application's performance. If a migration is triggered, the Migration Controller needs to compute a cross-cloud migration plan. The computed migration plan should be feasible, which means each VM migration specified in the plan should be feasible (e.g. the source and destination server is connected by network, the destination server has enough capacity to host the migrated VM, etc.). In addition, if there is more than one feasible plan, the Migration Controller attempts to choose the plan that is as optimal as possible. The details about when to trigger a migration and how to compute a migration plan will be discussed in section 5.2.3.

The **Migration Executor** is responsible for executing the migration plan computed by the Migration Controller. It also needs to monitor the status of the migration plan when it is executing, and attempts to recover from failures, if any occurs. In our prototype, the Migration Executor incorporates the service provided by the PDS for executing individual VM migration. A VM that is a potential candidate to migrate is represented by a node in a deployment model, and the node contains an operation *migrate* that is used by the Migration Executor for moving the VM from one host to another. The input of the Migration Executor is a migration plan computed by the Migration Controller. The Migration Executor converts the inputted migration plan to a sequence of invocations of the *migrate* operation with a set of parameters, including the ID of the topology that contains the node to be migrated, the ID of the node that is going to be migrated, the ID of the destination node. The *migrate* operation returns immediately without waiting for the completion of the migration. The Migration Executor needs to query the status of the migration periodically, until completion.

The **Knowledge Store** is the centralized storage of the management system. It is used to store various kinds of data that needs to be shared among other components. Examples of such data could be the deployment models of the managed applications, runtime information of the applications (e.g. the

IP addresses of each node), the user-defined migration policy, the historical metrics collected by the Monitoring System, etc.

In this thesis, we assume that there is only one autonomic manager (the MAPE-k loop) which manages all applications in the smart edge and the core. This corresponds to a business case in which the autonomic manager is owned by the cloud provider. The single autonomic manager makes global and centralized decisions, improving the efficiency of resource management. In other business cases, it is possible that more than one autonomic manager are needed. In those cases, decisions made by different autonomic managers may conflict with each other, therefore arbitration mechanisms might be needed to mitigate the conflicts.

5.2.3 Algorithm

In this section, we present an algorithm for managing cross-cloud migrations in two-tier cloud environments. As in section 4.2.3.2, we use the term *guest node* (or *guest* when it does not cause ambiguity) to refer a server that is used to host applications or components of applications, and the term *host node* (or *host* when it does not cause ambiguity) to refer a server that is used to host an application or a component of an application. The input of the algorithm is a subset of metrics that are gathered by the Monitoring System. The metrics are used by the algorithms to learn the state of both the applications and their underlying infrastructure. If a problem (e.g. a hotspot) is diagnosed, the algorithm will attempt to compute an optimal migration plan to fix it.

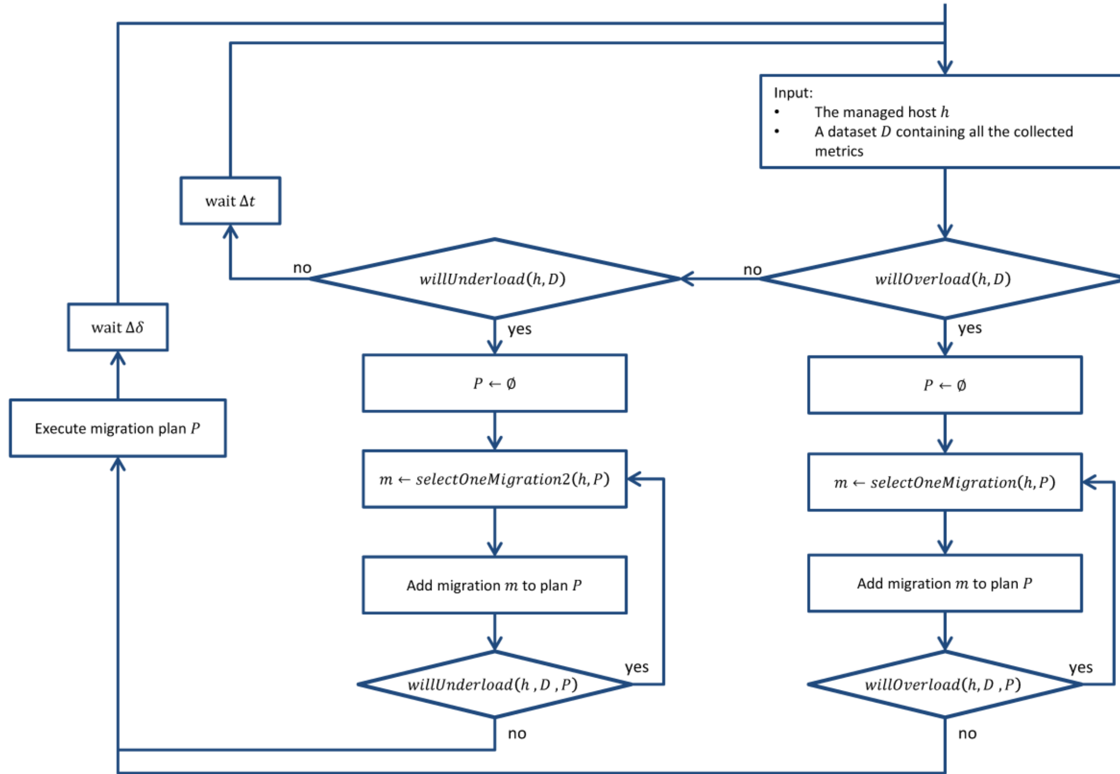

Figure 16: A high-level view of the algorithm

The high-level view of the algorithm is depicted in Figure 16. This algorithm needs to run on every host node in the smart edge. An input of the algorithm is a dataset that contains the metrics provided by the Monitoring System. The dataset is leveraged by the sub-algorithm *willOverload* that is used to predict the occurrences of hotspots. If the current host is predicted to be overloaded, a migration plan is computed to resolve the hotspot. The steps to compute a migration plan are as follows:

- Initially, an empty migration plan is created.
- A migration is computed and added to the migration plan. The computed migration migrates a VM from current host (in the smart edge) to a host in the core. The sub-algorithm *selectOneMigration* is called to select the migrated VM and the destination host.
- The current migration plan is passed to the sub-algorithm *willOverload* to determine if it can mitigate the hotspot. If it is, the migration plan is executed. Otherwise, repeat the previous step to add more migrations into the migration plan.

Similarly, the sub-algorithm *willUnderload* also analyzes the dataset to determine if the resources in the current host will be under-utilized. If it is, a migration plan is computed similarly as above to resolve the problem. The computed migration plan specifies one or more migrations that migrate a VM from a host in the core to current host (in the smart edge). The migrated VM is selected by using the sub-algorithm *selectOneMigration2*.

There are two parameters that need to be traded off by administrators: system cool-down interval Δt and migration cool-down interval $\Delta\delta$. System cool-down interval decides the frequencies for checking the state of the infrastructure and potentially triggering migration(s). In other words, the system cool-down interval determines how fast the management system reacts to occurrences of hotspots (or low resource utilizations): a small value can improve the system's responsiveness to a hotspot, but it increase the resource consumption of the migration management system (due to the frequent executions of the loop); a large value has, on the other hand, the opposite effect. The migration cool-down interval specifies the minimum duration between two consecutive triggering of migrations. This parameter determines the aggressiveness of the management system to execute migrations.

Algorithm 1: An algorithm for predicting if a server will be overloaded

```
   Input: A host h, a dataset D, a prediction model PM, a migration plan P (optional)
   Output: A boolean value that indicates whether host h will be overloaded
 1 begin
 2    Let t denote the administrator-defined utilization threshold
 3    Let i denote the future time at which the workload is predicted
 4    Let u_i^h denote the estimated utilization of host h at time i
 5    Predict u_i^h by using model PM with dataset D
 6    if migration plan P is provided then
 7       Let u_i^P denote the estimated decrease of utilization at time i if migration plan P is executed
 8       Predict u_i^P by using prediction model PM with dataset D
 9       u_i^h ← u_i^h - u_i^P
10    if u_i^h > t then
11       return true
12    else
13       return false
14    end
15 end
```

The algorithm *willOverload* is presented in Algorithm 1 (the algorithm *willUnderload* can be derived similarly). This algorithm is responsible to predict the occurrences of hotspots in a host. In particular, a user-provided prediction model is leveraged to predict the utilization of the host in the near future. The set of observed values, which the prediction is based on, is retrieved from the inputted dataset. If a migration plan is given, the algorithm will take the effect of the migration plan into account. In particular, it subtracts the predicted utilization of the host from the estimated decrease of utilization

(as showed in line 9 of Algorithm 1), which is the expected impact of the migration plan. Finally, the predicted utilization is compared with a threshold to determine if the host will become a hotspot.

An observation is that this algorithm does not assume the usage of specific prediction approach. Instead, it takes a prediction model as a parameter, and leverages it to predict the future's workloads. As a result, administrators have the flexibility for choosing a prediction model that is optimized for their use cases.

Algorithm 2: An algorithm for generating migration plan

```
  Input: A host h, a migration plan P, a migration policy MP
  Output: An migration selected by using migration policy MP
1 begin
2 |  Let G be the set of guests that are hosted by host h but not yet included in migration plan P
3 |  Let g represent the guest selected to be migrated
4 |  g ← MP(G)
5 |  Let D be a set of hosts that are capable to be a destination of guest g
6 |  Let d represent the selected destination
7 |  d ← MP(D)
8 |  return a planned migration for migrating guest g to destination d
9 end
```

The algorithm *selectOneMigration* is responsible to compute a migration that migrates a VM out of a host that has been predicted to be a hotspot. Similarly, the algorithm *selectOneMigration2* is responsible to compute a migration that migrates a VM into a host that is predicted to be under-utilized. The algorithm *selectOneMigration* is presented in Algorithm 2 (the algorithm *selectOneMigration2* can be derived similarly). This algorithm selects a guest node that will be migrated out of the host. This selection is based on a user-provided migration policy. The destination of this migration is selected from a set of hosts in the core, which have sufficient capacity to host the migrated guest. This selection is also based on the user-provided migration policy. Generally speaking, a migration policy is a specification of a few criteria used to select a migration that is expected to have maximum amount of the benefits and minimal amount of costs. Examples of such criterion could be: selecting the VM with smallest disk size to migrate, selecting the VM with smallest memory size to migrate, or selecting the VM with lowest CPU utilization to migrate. A variation is to use a utility function [60] to select a migration (instead of using a migration policy). The utility function should quantify the benefits and the costs of a selected migration, thus identifying the optimal migration among a set of feasible migrations.

5.3 Experiments

To demonstrate the effectiveness of the proposed approach, we conducted experiments with a prototype implementation of the migration management system. The first experiment shows the

performance of applications when different migration policies are incorporated into the management system. The second experiment shows the performance of applications when a workload-prediction method is incorporated into the management system.

The experiments ran on the infrastructure that consists of a smart edge and a core, both of which were emulated by using nested virtualization technology. In particular, we constructed a layer of virtualization overlay on a set of VMIs provisioned from EC2, which is used to emulate the two datacenters (the smart edge and the core). In such a nested virtualized environment, there are two types of node: host node and guest node. A host node is provisioned from EC2, and used to host a QEMU[19] hypervisor. A guest node is launched by a hypervisor hosted by a host node, and used to host an application (or its components). We rely on PDS to construct the nested virtualized environment automatically, and deploy the applications on it (see Appendix D for details).

The application used in the experiments is a simple Java EE web application that provides operations to read/write from/to a database. In the experiments, there are several instances of this application hosted by the underlying infrastructure. There are three software components involved: the prototype migration management system, the PDS and the workload generator. The PDS is responsible for executing the migrations decided by the migration management system, which is actually a Java application. In particular, the migration management system is responsible for monitoring the applications, computing migration plans, and instructing the PDS to execute the migration plans. The migration management system consumes the service of PDS by leveraging a generated Java API client (A feature of PDS is the auto-generation of a language-specific API client that allows an easy integration with other projects). The workload generator is responsible for generating workloads of the applications by using JMeter. In particular, the workload generator simulates a number of users, each of whom sends requests to an instance of application and records the response time of each request. These three components are hosted by 3 different machines, each of which has 2 virtual CPU and 4GB of RAM.

[19] http://wiki.qemu.org/Main_Page

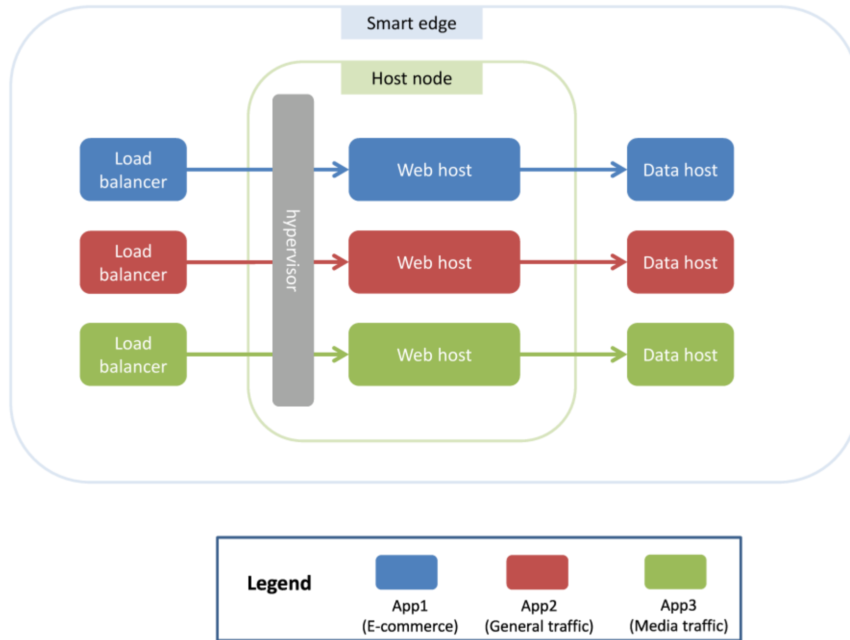

Figure 17: Initial placement of the applications

In each experiment, three instances of application were pre-deployed, which is showed in Figure 17. Each instance of application is deployed to three nodes: the `load balancer`, the `web host`, and the `data host`. The `load balancer` is the application frontend, and the `data host` is the database of the application, both of which are provisioned from EC2 and of instance type `t1.micro` (with 1 virtual CPU and 512MB of RAM). The `web host` is a guest node with 1 virtual CPU and 1GB of RAM, and used to host an application server. All the nodes were initially placed on the smart edge, and the `web servers` of the three applications were collocated on a single host node, which is provisioned from EC2 and of instance type `m1.large` (with 2 virtual CPU and 8G of RAM).

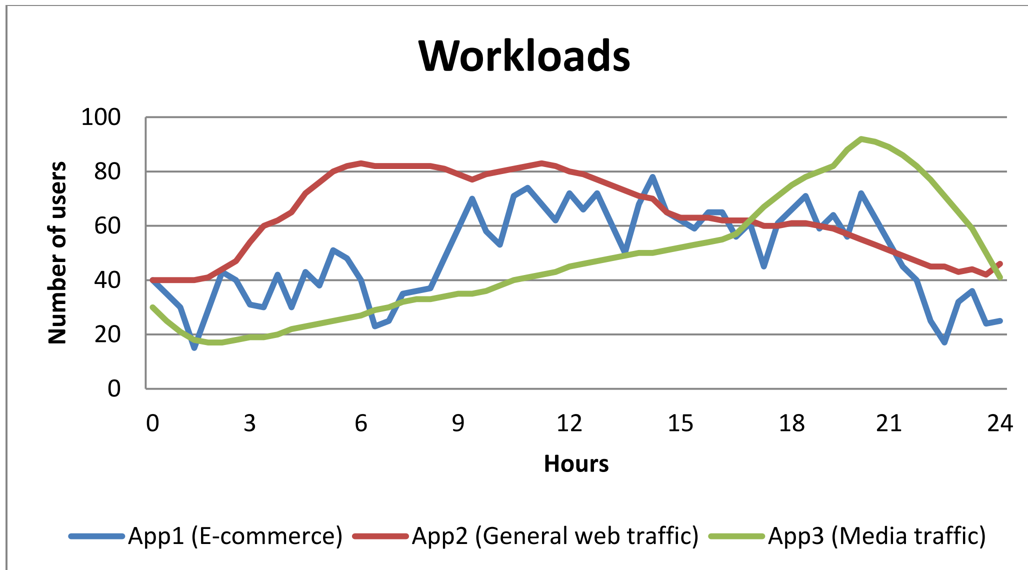

Figure 18: Workloads in the experiments

The workloads of the applications are depicted in Figure 18. As showed in the figure, three classes of workloads were simulated independently, and applied to the three instances of application respectively. For each instance of application, its workload's intensity is decided by the number of simulated users, which is changing over time according to the workload pattern of a specific type of application (E-commerce, General web traffic, or Media traffic). Each simulated user continually sends requests to the application and waits for their responses. After a response is received, the user sleeps for a random amount of time, and then sends another request, and so on. The sleep time is of 3 seconds mean with 1 second derivation. The workloads are originally of 1 day duration, but we scaled them to one hour to save the cost of running the experiments. Given the applications applied to the workloads specified above, we evaluated the performance of each application by monitoring its response to each request.

5.3.1 Experiment 1

The goal of this experiment is to evaluate the performances of different migration policies. To allow us to focus on the goal of this experiment, we disabled the workload prediction feature. The experiment consists of three rounds. In the first round, we do not trigger any VM migration. The intention is to get the baseline performance. In the second round, we used the least-utilized first (LUF) policy, in which the VM with the lowest CPU utilization was selected to be migrated out. In the third round, we use the most-utilized first (MUF) policy, in which the VM with the highest CPU utilization was selected to be migrated out. The destination of the migration is a host node in the core, which has enough capacity to

host the migrated VM (if multiple host node satisfies the capacity requirement, one of them will be chosen). The list of parameters used in this experiment is listed in Table 1. It should be noted that there is clearly room to improve these parameter settings, and the list of values presented in Table 1 is used for demonstration only.

CPU utilization threshold	60%
System cool-down interval	20 seconds
Migration cool-down interval	10 minutes

Table 1: The list of parameters used in experiment 1

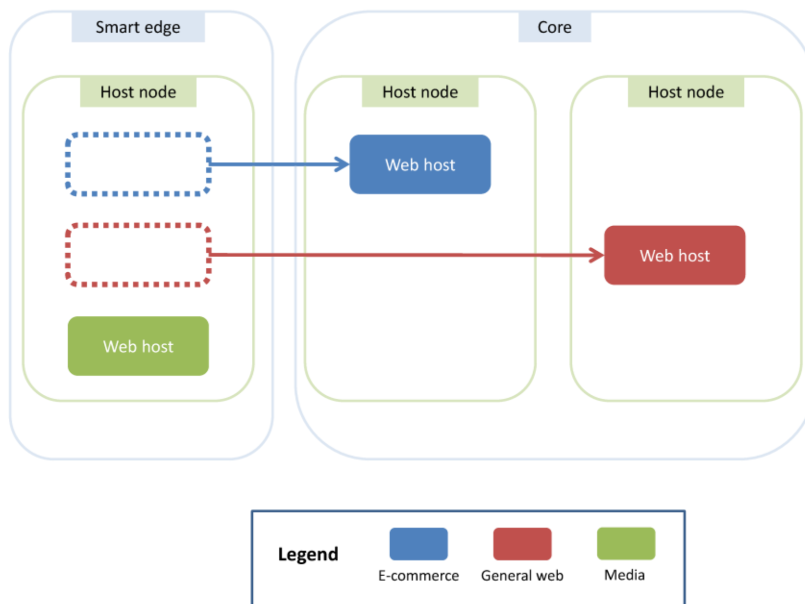

Figure 19: The migration management process

The VM migration process is outlined in Figure 19. As the intensity of the workloads increases, the host node in the smart edge becomes a hotspot. The management system monitored the utilization of the host node and triggered a VM migration when the average utilization in the last 2 minutes is greater than 60%. There are two migrations triggered in each round (except the first round). The choices of the migrated VMs are based on the LUF policy and the MUF policy in the second round and third round respectively.

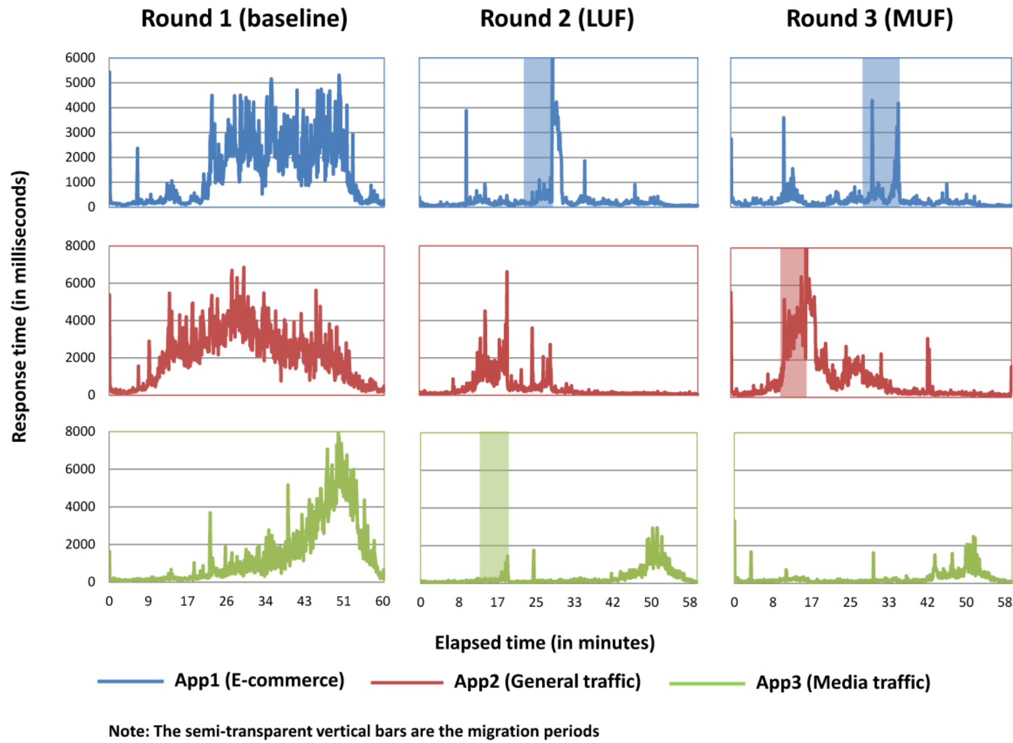

Figure 20: Performances of the applications with different migration policies

The performance of the applications in this experiment is presented in Figure 20, which consists of nine graphs, each of which plots the response time of a specific application in a specific round. The nine graphs are organized in columns and rows, and a graph in row *x* column *y* shows the response time of application *x* in round *y*. If an application has a VM that was migrated during the experiment, the migration period is highlighted in the graph by a semi-transparent vertical bar.

As shown in Figure 20, in the first round (baseline) of the experiment, no VM migration was performed, and the performances of the applications in this round are the worst. In the second round (LUF) and the third round (MUF), we utilized two different migration policies to select the VMs to migrate, and the performances of the applications are better than the first round. This improvement shows the benefit of using VM migrations to manage the performance of the applications. In addition, the overall performance of the applications in the second round (LUF) is better than the third round (MUF). Our best guest of this observation is that the MUF policy selected the heaviest VM to migrate, which seems to incur a greater performance penalty during migration.

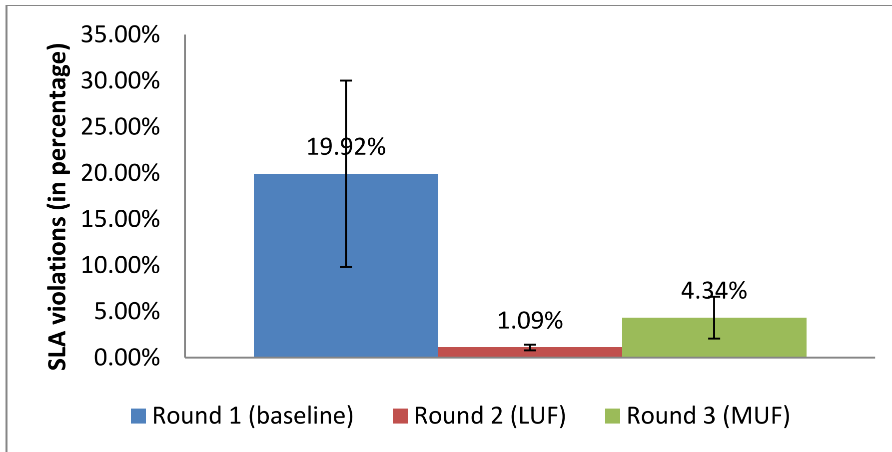

Figure 21: A comparison of SLA violation with different migration policies

To compare different migration policies quantitatively, we repeated this experiment 5 times and computed the average percentages of SLA violations. To compute the number of SLA violations, we parsed the logs to retrieve the response time of each request sent by the simulated users, and counted the number of requests whose response time is longer than 3 seconds (A response longer than 3 seconds is considered to be an SLA violation). The percentage of SLA violations in a round is computed by dividing the number of SLA violations by the total number of requests in this round. The result is presented in Figure 21, which shows that the percentage of SLA violations in the third round (MUF) is roughly four times more than the second round (LUF).

By the result of this experiment, it is not appropriate to conclude that the LUF policy consistently outperforms the MUF policy. However, the result is sufficient to demonstrate that the performance of applications can be improved by optimizing the choice of the migration policy.

5.3.2 Experiment 2

In this experiment, we evaluated the effectiveness of using workload-prediction techniques to predict the occurrences of hotspots, thus helping the system to resolve them in an efficient manner. There are two rounds in this experiment. In the first round, we disabled the workload-prediction to get a baseline performance. In the second round, we enabled the workload-prediction. In both rounds, we used the MUF migration policy.

The workload-prediction is based on a simple linear regression model [61]. The simple linear regression model is a simple prediction model, in which all the observed values are fit into a line so that the sum of square residuals is minimized. The goal is to predict the CPU utilization of a host node in 6

minute after now. To achieve this goal, the first step is to predict the CPU utilization of each guest node in the host node. In particular, the management system continually monitored the CPU utilizations of each guest, and stored them into the system. The future's utilization of a guest is predicted by using the simple linear regression model with its observed values, which are the historical utilizations of the guest in the last 6 minutes. The predicted utilization of each guest is normalized by multiplying the predicted utilization with the percentage of CPU time in the host allocated to this guest. The future's utilization of the host is computed by summing the normalized utilizations of all guests. In addition, the management system can estimate the decrease of utilization due to a migration plan. The estimated decrease of utilization is computed by summing the normalized utilizations of all the guests that were planned to be migrated out. In summary, the parameters used in the second round of this experiment are listed in Table 2.

CPU utilization threshold	60%
System cool-down interval	20 seconds
Migration cool-down interval	10 minutes
Workload-prediction model	Simple linear regression
Windows size	6 minutes before now
How far to predict	6 minutes after now
Migration policy	MUF

Table 2: List of parameters used in experiment 2

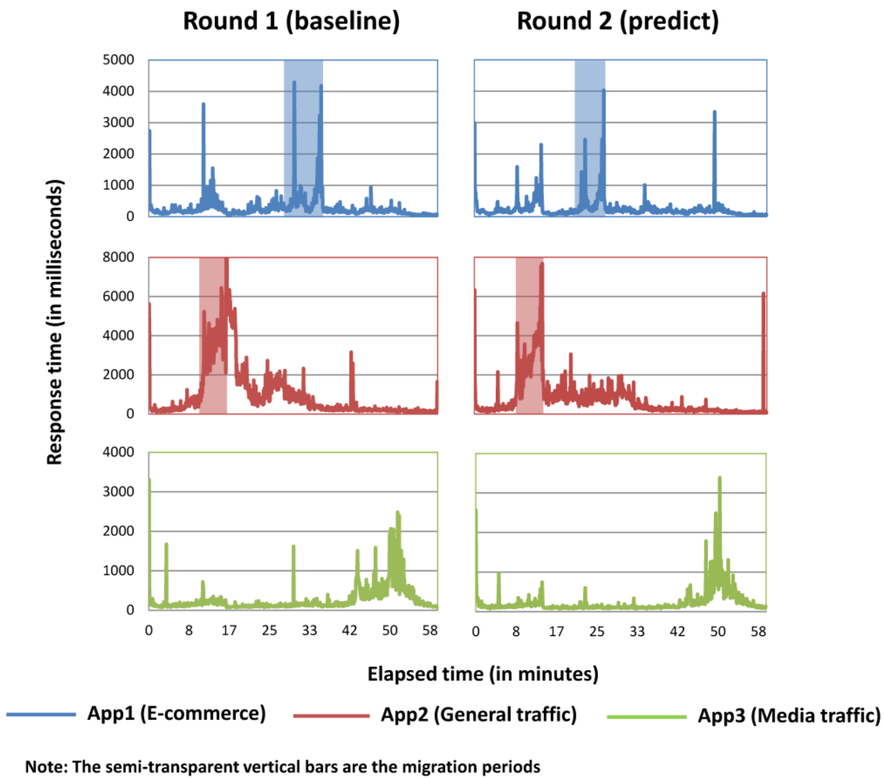

Figure 22: Performances of the applications by predicting their workloads

The result of this experiment is presented in Figure 22. An observation is that in the second round, the migrations were triggered earlier than the first round. As a result, the overall performance of applications in the second round was slightly better than the first round.

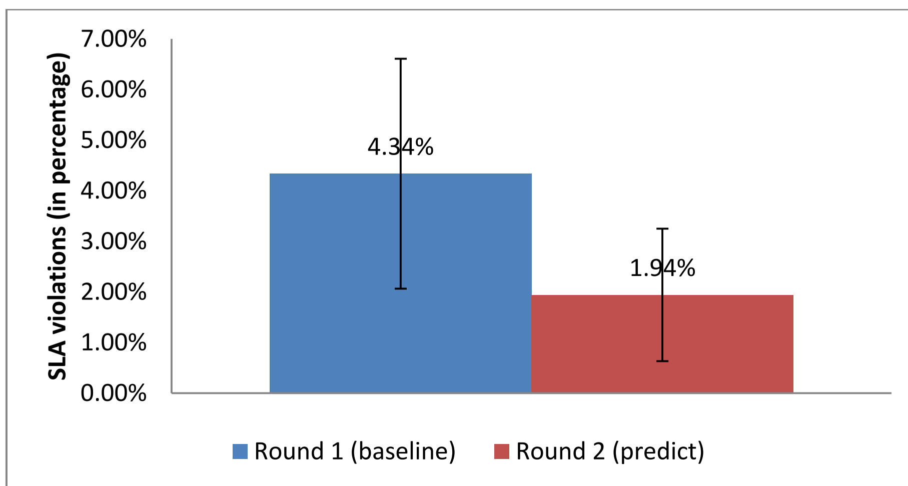

Figure 23: A comparison of performance of the applications when prediction is used

To compare the performance quantitatively, we repeated this experiment 5 times and computed the average percentages of SLA violations (as done in previous experiment). The result is showed in Figure 23. As showed in the figure, the percentage of SLA violations in the second round is roughly half of the first round.

5.3.3 Threats to validity

This section discusses the potential threats to the validity of these experiments. In particular, we describe the conditions in which these experiments were performed, and the potential factors which might prevent the result of these experiments from being reproduced or generalized to the real-world settings.

Firstly, a threat to the validity of these experiments could be the use of nested virtualization technology. On the one hand, utilizing nested virtualization allows us to overlay a layer of virtualization on public cloud, thus emulating the testing infrastructure. In such a nested virtualized environment, we are able to access to the hypervisors and instruct them to execute VM migrations. On the other hand, nested virtualization has not been widely used in the real-world, which might affect the generalization of the experimental results. In addition, we used QEMU hypervisor in these experiments because it is runnable on an EC2's Ubuntu instance (most of existing hypervisors are not runnable on EC2's instances due to various reasons). However, QEMU is not a widely used hypervisor, which might also affect the generalization of the experimental results.

Secondly, the choice of the testing application could also be a threat to the validity. In these experiments, we used a JEE application which is easy to deploy and capable to create the required demands on the underlying hardware resources. However, this application might be simpler than typical enterprise applications, which might affect the representativeness of the experimental results.

Thirdly, the use of specific workloads could be another threat to the validity. The workloads were generated by a JMeter Test Plan with specific parameters to emulate typical workloads. Obviously, using different workloads could result in different migrations and/or migration triggering time, thus impacting the application performance.

Finally, the configurations of the underlying infrastructure could also be a threat to the validity. These experiments were conducted on EC2, which is a major public cloud provider. A change of the underlying infrastructure (e.g. switch to another cloud provider) might result in variations of performance, thus producing different results. Furthermore, we observed significant variations of

performance on EC2 at different points in time, which could also be a threat to the validity. In order to overcome this, we repeated each experiment several times and present the average performance.

5.4 Summary

This chapter presented a migration management framework for two-tier cloud environments. We provided a reference architecture and an algorithm to facilitate the management of cross-cloud migrations. The proposed algorithm attempts to compute optimal migration plans by leveraging a user-defined migration policy and workload-prediction approach. The experimental results validated the effectiveness of our approach.

Chapter 6: Conclusions and Future Work

This thesis focuses on enabling adaptation of applications in two-tier cloud environments. We achieved this goal in two steps:

1. Introduced a model-driven approach to facilitate application deployment.
2. Leveraged the live VM migration technique to relocate VMs dynamically between two datacenters so the application performance is maintained during high loads.

To facilitate application deployment, we proposed a deployment architecture that simplifies application deployment in multi-cloud environments. In the architecture, an application deployer specifies a deployment of an application in a deployment model, and uses the model to deploy the application. The deployment model is composed of a set of high-level abstractions, each of which describes a deployment concept or entity in a technology-independent manner. The high-level deployment model is translated into a low-level workflow model at runtime, and the workflow model is fed into an executor that executes the deployment accordingly.

The deployment architecture has been implemented by PDS which provides application deployment as a cloud service. PDS employs a domain specific language that enables users to specify their deployment in an XML-based document, and provides a Restful interface to simplify the consumption of the deployment service. In addition, PDS also provides operations that can be leveraged by users to reconfigure the deployed applications at runtime, such as scale out, scale in and migrate. Lastly, the PDS supports an application undeployment service that is used to clean up the cloud resources.

The performance of PDS is demonstrated by the experiments presented in section 4.4. The experiments were conducted on two different clouds: Amazon EC2 and SAVI testbed. In experiment 1, we show that deploying a JEE application to a set of VMIs based on a standard Ubuntu cloud image takes no more than 10 minutes, given that the PDS is under a light load. In experiment 2, we show the possibilities to improve the performance of PDS by pre-installing application components into the images.

To enable VM migrations in two-tier cloud environments, we proposed an adaptation framework based on live migration of VMs. We particularly focused on the management of cross-cloud migrations, and proposed a model-driven approach, architecture and algorithm to facilitate the

management tasks. In our proposed model-driven approach, the execution of VM migration is based on a model that describes the deployment of the application. We also proposed a migration management architecture that is able to monitor the deployed applications, predict hotspots in the underlying infrastructure, and plan and execute VM migrations to mitigate the hotspots. The architecture employs an algorithm that is responsible for deciding VM migrations and computing migration plans. The algorithm predicts the future's workloads of the applications, and leverages a user-defined migration policy to select the VMs to migrate.

We conducted two experiments to validate our approach. The result of the first experiment shows that (i) leveraging VM migrations significantly improves the performance of applications, and (ii) the migration policy LUF performs better than the migration policy MUF, under the experimental infrastructure we set up and the workloads we simulated. In the second experiments, we demonstrated that we could improve the performance of the applications by predicting the workloads of applications and trigger VM migrations pro-actively.

In summary, the main contributions of this thesis are as followings:

- We proposed a model-driven deployment approach, in which application deployers can focus on the high-level concepts of deployment without being distracted from the low-level complexities of the deployment workflows.
- We proposed a deployment architecture that is able to fulfill the requirements derived from multi-cloud environments, and allows experts from different domains to contribute to their knowledge independently.
- We introduced the PDS, a non-trivial implementation of the deployment architecture. The PDS is able to provide application deployment as a service, and the deployment service is delivered through an intuitive Restful API.
- We proposed a migration management framework that is able to reduce the complexities of managing VM migrations in two-tier cloud environments.
- We provided a migration management architecture that is able to monitor the state of applications and their underlying infrastructure, and executing migrations to balance the workloads dynamically.

- We provided an algorithm for managing cross-cloud migrations. The algorithm attempts to make good migration decisions by incorporating a user-defined migration policy and a user-provided prediction model.

The research in this thesis can be extended or improved in several directions, which include:

- Validate the model-driven deployment and management across more cloud providers, such as Rackspace[20] (PDS was validated on EC2 and OpenStack-based clouds).
- Extend the PDS to support more deployment patterns, such as GlassFish[21], Hadoop[22] (currently it supports JEE application deployment patterns).
- Improve the scalability of PDS by making it replicable. By doing that, the workloads of the PDS can be distributed to difference replicas of PDS, thus improving the scalability.
- Parameterize the migration algorithm with more migration policies and prediction models, and evaluate their performance.
- Conduct more experiments to validate the migration management approach. In particular, we are going to conduct experiments that run on the SAVI testbed, in which we have permission to execute layer 0 migrations (instead of layer 1 migrations we executed in the experiments).

[20] http://www.rackspace.com/
[21] https://glassfish.java.net/
[22] http://hadoop.apache.org/

Chapter 7: Publications during Research

H. Lu, M. Shtern, B. Simmons, M. Smit and M. Litoiu, "Pattern-based Deployment Service for Next Generation Clouds," in *IEEE 9th World Congress on Services*, Santa Clara Marriott, CA, 2013, pp. 464-471. **This paper and demo was awarded the Silver medal in the *Services Cup* Competition.**

M. Shtern, B. Simmons, M. Smit, H. Lu and M. Litoiu, "Toward an Autonomic Systems Platform for Multi-Clouds," in *Cloud Services,Networking and Management*, IEEE Communications Society, To Appear, 2014.

Bibliography

[1] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond and M. Morrow, "Blueprint for the Intercloud - Protocols and Formats for Cloud Computing Interoperability," in *Proceedings of the Fourth International Conference on Internet and Web Applications and Services*, Venice/Mestre, 2009, pp. 328-336.

[2] R. Buyya, R. Ranjan and R. N. Calheiros, "InterCloud: utility-oriented federation of cloud computing environments for scaling of application services," in *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing*, Busan, 2010, pp. 13-31.

[3] M. Shtern, B. Simmons, M. Smit and M. Litoiu, "Navigating the Cloud with a MAP," in *13th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ghent, 2013, pp. 464-470.

[4] P. Pawluk, B. Simmons, M. Smit, M. Litoiu and S. Mankovski, "Introducing STRATOS: A Cloud Broker Service," in *2012 IEEE 5th International Conference Cloud Computing (CLOUD)*, Honolulu, HI, 2012, pp. 891-898.

[5] N. Grozev and R. Buyya, "Inter-Cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience,* 2012.

[6] M. Smit, P. Pawluk, B. Simmons and M. Litoiu, "A Web Service for Cloud Metadata," in *IEEE Eighth World Congress on Services*, Honolulu, HI, 2013, pp. 361-368.

[7] H. Ghanbari, B. Simmons, M. Litoiu and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *2011 IEEE International Conference on Cloud Computing*, Washington, DC, 2011, pp. 716-723.

[8] M. Shtern, B. Simmons, M. Smit and M. Litoiu, "An architecture for overlaying private clouds on public providers," in *8th International Conference on Network and Service Management*, Las Vegas, NV, 2012, pp. 371-377.

[9] "Strategic Network for Smart Applications on Virtual Infrastructure (SAVI)," 2012. [Online]. Available: http://www.savinetwork.ca/. [Accessed 7 August 2013].

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, Boston, MA, 2005, pp. 273-286.

[11] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar and A. Iyer, "Remedy: network-aware steady state VM management for data centers," in *NETWORKING 2012*, Prague, Springer Berlin Heidelberg, 2012, pp. 190-204.

[12] V. Mann, A. Vishnoi, A. Iyer and P. Bhattacharya, "VMPatrol: Dynamic and automated QoS for virtual machine migrations," in *8th International Conference on Network and Service Management (CNSM)*, Las Vegas, NV, 2012, pp. 174-178.

[13] T. Wood, P. Shenoy, A. Venkataramani and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, Cambridge, MA, 2007, pp. 17-17.

[14] A. Stage and T. Setzer, "Network-aware migration control and scheduling of differentiated virtual machine workloads," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, BC, 2009, pp. 9-14.

[15] F. Ma, F. Liu and Z. Liu, "Distributed load balancing allocation of virtual machine in cloud data center," in *IEEE 3rd International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, 2012, pp. 20-23.

[16] H. N. Van, F. D. Tran and J.-M. Menaud, "Autonomic virtual resource management for service hosting platforms," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, BC, 2009, pp. 1-8.

[17] T. Guo, U. Sharma, T. Wood, S. Sahu and P. Shenoy, "Seagull: Intelligent Cloud Bursting for Enterprise Applications," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, Boston, MA, 2012, pp. 33-33.

[18] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, 2009.

[19] U. Hoelzle and L. A. Barroso, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Morgan and Claypool Publishers, 2009.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, 2003, pp. 164-177.

[21] T. Delaet, W. Joosen and B. Vanbrabant, "A survey of system configuration tools," in *Proceedings of the 24th international conference on Large installation system administration*, San Jose, CA, 2010, pp. 1-8.

[22] T. Eilam, M. H. Kalantar, A. V. Konstantinou and G. Pacifici, "Reducing the complexity of application deployment in large data centers," in *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, Nice, 2005, pp. 221-234.

[23] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing and A. Agrawal, "Managing the configuration complexity of distributed applications in Internet data centers," *IEEE Communications Magazine,* vol. 44, no. 3, pp. 166 - 177, 2006.

[24] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou and A. A. Totok, "Pattern Based SOA Deployment," in *Service-Oriented Computing – ICSOC 2007*, Springer Berlin Heidelberg, 2007, pp. 1-12.

[25] A. V. Konstantinou, T. Eilam, M. Kalantar, A. A. Totok, W. Arnold and E. Snible, "An architecture for virtual solution composition and deployment in infrastructure clouds," in *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing*, Barcelona, 2009, pp. 9-18.

[26] T. Eilam, M. Elder, A. V. Konstantinou and E. Snible, "Pattern-based composite application deployment," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*,

Dublin, 2011, pp. 217-224.

[27] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer,* vol. 36, no. 1, pp. 41-50, 2003.

[28] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess and J. O. Kephart, "An Architectural Approach to Autonomic Computing," in *Proceedings of the First International Conference on Autonomic Computing*, New York, NY, USA, 2004, pp. 2-9.

[29] Y. Brun, G. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè and M. Shaw, "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems*, Berlin, Springer Berlin Heidelberg, 2009, pp. 48-70.

[30] J. Sauvé, F. Marques, A. Moura, M. Sampaio, J. Jornada and E. Radziuk, "SLA design from a business perspective," in *Proceedings of the 16th IFIP/IEEE Ambient Networks International Conference on Distributed Systems: Operations and Management*, Barcelona, 2005, pp. 72-83.

[31] I. Goiri, F. Julia, J. Ejarque, M. d. Palol, R. M. Badia, J. Guitart and J. Torres, "Introducing virtual execution environments for application lifecycle management and sla-driven resource distribution within service providers," in *Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, 2009, pp. 211-218.

[32] M. Comuzzi, C. Kotsokalis, G. Spanoudakis and R. Yahyapour, "Establishing and monitoring slas in complex service based systems," in *Proceedings of the 2009 IEEE International Conference on Web Services*, Los Angeles, CA, 2009, pp. 783-790.

[33] N. Bansal, K.-W. Lee, V. Nagarajan and M. Zafer, "Minimum congestion mapping in a cloud," in *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, San Jose, California, 2011, pp. 267-276.

[34] Z. Li, J. Chinneck, M. Woodside and M. Litoiu, "Fast scalable optimization to configure service systems having cost and quality of service constraints," in *Proceedings of the 6th international conference on Autonomic computing*, Barcelona, 2009, pp. 159-168.

[35] Z. Li, J. Chinneck, M. Woodside and M. Litoiu, "Deployment of Services in a Cloud Subject to Memory and License Constraints," in *IEEE International Conference on Cloud Computing*, Bangalore, 2009, pp. 33-40.

[36] T. Kimbrel, M. Steinder, M. Sviridenko and A. Tantawi, "Dynamic application placement under service and memory constraints," in *Proceedings of the 4th international conference on Experimental and Efficient Algorithms*, Cala Galdana, Menorca Island, Spain, 2005, pp. 391-402.

[37] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko and A. Tantawi, "Dynamic placement for clustered web applications," in *Proceedings of the 15th international conference on World Wide Web*, Edinburgh, 2006, pp. 595-604.

[38] H. Ghanbari, B. Simmons, M. Litoiu and G. Iszlai, "Feedback-based optimization of a private cloud," *Future Generation Computer Systems,* vol. 28, no. 1, pp. 104-111 , 2012.

[39] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Washington, DC, 2009, pp. 51-60.

[40] R. Bradford, E. Kotsovinos, A. Feldmann and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *Proceedings of the 3rd international conference on Virtual execution environments*, San Diego, CA, 2007, pp. 169-179.

[41] A. Shribman and B. Hudzia, "Pre-Copy and Post-Copy VM Live Migration for Memory Intensive Applications," in *Euro-Par 2012: Parallel Processing Workshops*, Springer Berlin Heidelberg, 2012, pp. 539-547.

[42] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman and B.-A. Yassour, "The turtles project: design and implementation of nested virtualization," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, Vancouver, BC, 2010, pp. 1-6.

[43] D. Williams, H. Jamjoom and H. Weatherspoon, "The Xen-Blanket: virtualize once, run everywhere," in *Proceedings of the 7th ACM european conference on Computer Systems*, Bern, 2012, pp. 113-126.

[44] M. P. Papazoglou and W.-J. van den Heuvel, "Blueprinting the Cloud," *IEEE Internet Computing,* vol. 15, no. 6, pp. 74 - 79, 2011.

[45] M. P. Papazoglou, "Cloud Blueprints for Integrating and Managing Cloud Federations," in *Software Service and Application Engineering*, Berlin, Springer Berlin Heidelberg, 2012, pp. 102-119.

[46] J. Wettinger, V. Andrikopoulos, S. Strauch and F. Leymann, "Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment," in *Proceedings of the IEEE 6th International Conference on Cloud Computing*, Santa Clara Marriott, CA, 2013, pp. 478-485.

[47] A. Flissi, J. Dubus, N. Dolet and P. Merle, "Deploying on the Grid with DeployWare," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, Lyon, 2008, pp. 177-184.

[48] I. Foster and C. Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure, San Francisco: Morgan Kaufmann Publishers Inc., 2003.

[49] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy and L. Seinturier, "A Federated Multi-cloud PaaS Infrastructure," in *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, Honolulu, HI, 2012, pp. 392-399.

[50] C.-L. Hwang and K. Yoon, Multiple Attribute Decision Making: Methods and Applications, New York: Springer-Verlag, 1981.

[51] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller and J. Lawall, "Entropy: a consolidation manager for clusters," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Washington, DC, 2009, pp. 41-50.

[52] Y. Xu and H. Qi, "Mobile agent migration modeling and design for target tracking in wireless sensor networks," *Ad Hoc Networks,* vol. 6, no. 1, pp. 1-16, 2008.

[53] K. E. Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar and A. V. Konstantinou, "Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*,

Melbourne, 2006, pp. 404-423.

[54] H. Ghanbari, B. Simmons, M. Litoiu and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *Proceedings of the IEEE International Conference on Cloud Computing*, Washington, DC, 2011, pp. 716-723.

[55] M. Shtern, B. Simmons, M. Smit, H. Lu and M. Litoiu, "Toward an Autonomic Systems Platform for Multi-Clouds," in *Cloud Services,Networking and Management*, IEEE Communications Society, To Appear, 2014.

[56] H. Lu, M. Shtern, B. Simmons, M. Smit and M. Litoiu, "Pattern-based Deployment Service for Next Generation Clouds," in *IEEE 9th World Congress on Services*, Santa Clara Marriott, CA, 2013, pp. 464-471.

[57] J.-M. Kang, H. Bannazadeh and A. Leon-Garcia, "SAVI Testbed: Control and Management of Converged Virtual ICT Resources," in *IFIP/IEEE Integrated Network Management Symposium (IM 2013)*, Ghent, 2013, pp. 664-667.

[58] M. L. Massie, B. N. Chun and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation And Experience," *Parallel Computing,* vol. 30, no. 7, pp. 817-840, 2003.

[59] M. Smit, B. Simmons and M. Litoiu, "Distributed, application-level monitoring for heterogeneous clouds using stream processing," *Future Generation Computer Systems,* vol. 29, no. 8, pp. 2103-2114, 2013.

[60] W. E. Walsh, G. Tesauro, J. O. Kephart and R. Das, "Utility Functions in Autonomic Systems," in *Proceedings of the First International Conference on Autonomic*, New York, NY, 2004, pp. 70-77.

[61] G. Box and G. M. Jenkins, Time Series Analysis: Forecasting and Control, Holden-Day, Incorporated, 1990.

[62] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology,* vol. 2, no. 2, pp. 115-150, 2002.

Appendix A: Algorithms for Graph Creation and Usage

As mentioned in section 4.3.2, PDS parses the inputted deployment model and construct a graph in memory. The algorithm for creating a graph is shown in Algorithm 3. The algorithm for using the graph to provision the set of nodes is shown in Algorithm 4. The algorithm for using the graph to generate and execute deployment workflows is shown in Algorithm 5.

Algorithm 3: An algorithm to construct a graph based on the inputted deployment model

```
   Input: XML document topology
   Output: Graph graph
 1 begin
 2   Let nodes represent a list of nodes obtained from parsing the topology
 3   foreach node in nodes do
 4     Create a vertex v
 5     Assign to vertex v all services, attributes and files associated with node
 6     Add vertex v to graph graph
 7   end
 8   Let dependencies be the set of dependencies obtained from parsing the topology
 9   foreach dependency in dependencies do
10     Let vertex v1 denote the source of the dependency
11     Let vertex v2 denote the destination of the dependency
12     Create an edge e from vertex v1 to vertex v2
13     Assign to edge e the properties of dependency
14     Add edge e to graph graph
15   end
16   Let connections be the set of service-level connections obtained from parsing the topology
17   foreach connection in connections do
18     Let service s1 denote the source of the connection
19     Let v1 denote the vertex that contains the service s1
20     Let service s2 denote the destination of the connection
21     Let v2 denote the vertex that contains the service s2
22     Create an edge e from vertex v1 to vertex v2
23     Assign to edge e the properties of connection
24     Add edge e to graph graph
25   end
26   Ensure graph graph does not have any circular dependency
27   return graph
28 end
```

Algorithm 4: An algorithm for provisioning cloud resources

```
  Input: Graph graph
  Output: Resources Provisioned on Multi-Cloud
1 Let unprovisioned be the set of vertices in graph
2 begin
3   while unprovisioned is not empty do
4     foreach node in unprovisioned where its dependencies are resolved do
5       Instruct Provisioning Engine to provision node
6       Remove node from unprovisioned
7     end
8   end
9 end
```

Algorithm 5: An algorithm for provisioning cloud resources

```
  Input: Graph graph
  Output: Functioning Application Deployment on Multi-Cloud
1 Let undeployed denote the set of vertices that are provisioned and undeployed in graph
2 begin
3   while undeployed is not empty do
4     foreach node in undeployed do
5       Instruct Workflow Generator to generate workflow for node
6       Instruct Workflow Executor to execute the generated workflow on node
7     end
8   end
9 end
```

Appendix B: List of Supported Services in PDS

In the current state of PDS, 9 types of services are supported, which is listed in Table 3.

Service	Description
database_server	Installation of a database server. Current supported database is "MySQL" and "PostgreSQL".
web_balancer	Installation of a load balancer.
web_server	Installation of a web server. Current supported server is "Tomcat6".
virsh	Installation of Qemu hypervisor.
server_installation	Installation of Chef server which is a component of PDS. This is the self-installation feature which means a PDS can install itself in the cloud.
client_installation	Installation of PDS server. This is the self-installation feature.
standalone_installation	Installation of both PDS server and Chef server. This is the self-installation feature.
monitor	Installation of a monitoring agent.
monitoring_server	Installation of a monitoring server.

Table 3: A list of implemented services

Appendix C: API Overview of PDS

To promote the usability of the deployment model, we implemented a Restful [62] web services to expose the functionalities of PDS. By implementing a web service, we eliminate the overhead of local installation for users. The Restful-style of design also enables us to deliver the deployment service through a simple and intuitive interface.

Resource	Description	Path
container	Used to group nodes in a collective unit (e.g. application server tier). Containers can be scaled at runtime.	/api/topologies/{tid}/containers
credential	Used for uploading credentials that are used for authenticating users against their chosen cloud system.	/api/credentials
node	Represents server.	/api/topologies/{tid}/nodes /api/topologies/{tid}/containers/{cid}/nodes
service	Represents an application or service that runs on a node.	/api/topologies/{tid}/nodes/{nid}/services /api/topologies/{tid}/containers/{cid}/nodes/{nid}/services /api/topologies/{tid}/templates/{temid}/services
supporting_service	A set of services shared by multiple topologies, e.g. DNS server, certificate authority, etc.	/api/supporting_services
template	Creates and modifies templates for nodes.	/api/topologies/{tid}/templates
topology	Represents the overall system pattern: how all of the containers, nodes, and services are related. Users can create, query, repair, deploy, or undeploy their system patterns.	/api/topologies
uploaded_file	Used to upload file(s): private keys, application archives (WAR, etc.), SQL scripts.	/api/uploaded_files

Table 4: List of Restful resources

By following the Restful principle, we implemented 8 Restful resources that are listed in Table 4. Most of the resources support five methods: *list*, *get*, *create*, *delete*, *modify*. As an example, the list of methods in *topology* resource is list in Table 5.

Methods	**HTTP**	**Description**	**Path**
list	GET	List all topologies.	/api/topologies
get	GET	Get topology with “tid”.	/api/topologies/{tid}
create	POST	Create a topology. This method requires users to submit their system pattern as an input parameter and create a topology based on the submitted pattern.	/api/topologies
delete	DELETE	Delete the topology with “tid”.	/api/topologies/{tid}
modify	PUT	Modify topology with “tid” using an operation: “deploy”, “undeploy”, “repair”, “rename”, and “update_description”. “deploy” initiates translation from the pattern into a Chef workflow and deploys the application; “undeploy” terminates all resources involved in the topology; “repair” will recover a failed topology; “rename” and “update_description” change the metadata of the topology.	/api/topologies/{tid}

Table 5: List of methods in Topology resources

Appendix D: Application Deployment in Nested Virtualized Environments

To realize a VM migration, users need access to the corresponding hypervisor and invoke a hypervisor-specific command to migrate a VM. To achieve this, the corresponding hypervisor is required to be accessible by users. However, users typically do not have the permission to access a hypervisor that is owned by a third-party (e.g. a hypervisor owned by a public cloud provider). A solution is to leverage nested virtualization technique, which enables users to create another layer of virtualization (layer 1 virtualization) on top of the layer of virtualization constructed by the cloud providers (layer 0 virtualization). As a result, users gain access to the hypervisors in layer 1 thus facilitating the execution of migrations.

To facilitate the deployment process, we extended PDS (our implementation of the proposed deployment approach) to support application deployment on nested virtualized environments. In particular, PDS supports (i) automated construction of layer 1 virtualization, (ii) deploying applications to the VMs from layer 1. The steps taken by PDS to realize a deployment in this context are as follows:

1. Provision a set of VMIs from the specified cloud provider(s).
2. Install hypervisors on the set of VMIs provisioned in step 1.
3. Instruct the hypervisors to launch another set of VMIs that are used to hosts the application.
4. Deploy the application to the VMIs provisioned in step 3.

How to perform the steps above can be specified in a deployment model, which is authored by users and submitted to PDS to execute the deployment. In a deployment model, a VMI, provisioned in either step 1 or step 3, is represented by a *node*. We use the term *host node* to refer a node for hosting a hypervisor (that is a VMI provisioned in step 1), and the term *guest node* to refer a node for hosting an application component (that is a VMI provisioned in step 3). To specify a deployment, users can author a deployment model in the following steps:

1. Create a *topology* to represent the application.
2. Create a set of *host nodes* in the topology to represent the set of VMIs from layer 0.
3. Create a set of *services* in the host nodes to capture the requirements of installing the hypervisors in each host node.

4. Create a set of *guest nodes* in the topology to represent the set of VMIs from layer 1, and connect the guest nodes to their corresponding host nodes.
5. Create another set of *services* in the guest nodes to represent the middleware components used to host the application.

```
<topology id="migrate">
  ...
  <node id="hostNode">
    <use_template name=" EC2Instance"/>
    <service name="virsh"/>
  </node>
  <node id="guestNode">
    <use_template name="guestInstance"/>
    <nest_within node="hostNode">
      <open_port redir_from="5555">22</open_port>
      <open_port redir_from="8080">80</open_port>
      <image_file>ubuntu.img</image_file>
      <image_url>http://cloud-images.ubuntu.com/oneiric/current/oneiric-server-cloudimg-amd64-
disk1.img</image_url>
      <image_format>qcow2</image_format>
      <memory>1048576</memory>
    </nest_within>
        ...
  </node>
</topology>
```

Figure 24: Portion of a system pattern description showing the support of nested virtualization

A sample SPD is depicted in Figure 24. In the SPD, a service `virsh`[23] is defined in the node `hostNode`, which represents an installation of a hypervisor in that node. Furthermore, the node `guestNode` contains an XML element `nested_within` that specifies a connection between the guest node and the host node (the ID of the host node is declared in the attribute `node` of the XML element). The XML element has a list of children elements that specifies the provisioning of the guest node: the XML elements `image_file`, `image_url` and `image_format` specifies the based image of the guest node; the XML element `memory` specifies the size of the memory; the XML elements `open_port` specify the ways to redirect traffic from the host node to the guest node. When PDS receives the SPD showed in Figure 24, it will (1) provision the node `hostNode` from the specified cloud provider, (2) deploy a QEMU hypervisor to the node `hostNode`, (3) configure the hypervisor to provision the node `guestNode`, and (4) deploy the application (or a component of the application) to the guest node. In particular, the guest node provisioning process (step 3) includes downloading the specified virtual machine image, defining a domain according to the specification of the guest node, and running hypervisor-specific commands to launch the defined domain.

[23] Virsh is a cross hypervisor abstraction