Automating the Upgrade of IaaS Cloud Systems

Mina Nabi

A Thesis

In the Department

of

Electrical & Computer Engineering

Presented in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy (Electrical & Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

July 2019

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mina Nabi

Entitled: Automating the Upgrade of IaaS Cloud Systems

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Luis Amador

______________________________ External Examiner
Dr. Michel Dagnenais

______________________________ External to Program
Dr. Olga Ormandijieva

______________________________ Examiner
Dr. Juergen Rilling

______________________________ Examiner
Dr. Wahab Hamou-Lhadj

______________________________ Thesis Co-Supervisor
Dr. Ferhat Khendek

______________________________ Thesis Co-Supervisor
Dr. Maria Toeroe

Approved by ______________________________
Dr. Rastko R. Selmic, Graduate Program Director

September 4, 2019 ______________________________
Dr. Amir Asif, Dean
Gina Cody School of Engineering & Computer Science

ABSTRACT

Automating the Upgrade of IaaS Cloud Systems

Mina Nabi, Ph.D.

Concordia University, 2019

The different resources providing an Infrastructure as a Service (IaaS) cloud service may need to be upgraded several times throughout their life-cycle for different reasons, for instance to fix discovered bugs, to add new features, or to fix a security threat. An IaaS cloud provider is committed to each tenant by a service level agreement (SLA) which indicates the terms of commitment, e.g. the level of availability, that have to be respected even during upgrades. However, the service delivered by the IaaS cloud provider may be affected during the upgrade. Subsequently, this may violate the SLA, which in turn will impact other services relying on the IaaS. Our goal in this thesis is to devise an approach and a framework for automating the upgrade of IaaS cloud systems with minimal impact on the services and with respect to the SLAs.

The upgrade of IaaS cloud systems under availability constraints inherits all the challenges of the upgrade of traditional clustered systems and faces other cloud specific challenges. Similar challenges as in clustered systems include the potential dependencies between resources, potential incompatibilities along dependencies during the upgrade, potential system configuration inconsistencies due to the upgrade failures and the minimization of the amount of used resources to complete the upgrade. Dependencies of the application layer on the IaaS layer is an added challenge that must be handled properly. In addition, the dynamic nature of the cloud

environment poses a new challenge. A cloud system evolves, even during the upgrade, according to the workload changes by scaling in/out. This mechanism (referred to as autoscaling) may interfere with the upgrade process in different ways.

In this thesis, we define an upgrade management framework for the upgrade of IaaS cloud systems under SLA constraints. This framework addresses all the aforementioned challenges in an integrated manner. The proposed framework automatically upgrades an IaaS cloud system from a current configuration to a desired one, according to the upgrade requests specified by the administrator. It consists of two distinct components, one to coordinate the upgrade, and the other one to execute the necessary upgrade actions on the infrastructure resources. For the coordination of the upgrade process, we propose a new approach to automatically identify and schedule the appropriate upgrade methods and actions for implementing the upgrade requests in an iterative manner taking into account the vendors' descriptions of the infrastructure components, the SLAs with the tenants, and the status of the system. This approach is also capable of handling new upgrade requests even during ongoing upgrades, which makes it suitable for continuous delivery. In case of failures, the proposed approach automatically issues localized retry and undo recovery operations as appropriate for the failed upgrade actions to preserve the consistency of the system configuration.

In this thesis, to demonstrate the feasibility of the proposed upgrade management framework we present a proof of concept (PoC) for the upgrade IaaS compute, and its application in an OpenStack cluster. In this PoC, we target the new challenge of upgrade of the IaaS cloud (i.e. unexpected interference between the autoscaling and the upgrade processes) compared to the clustered systems. In addition, the prototype of the proposed upgrade approach for coordinating the upgrade of all kinds of IaaS resources has been implemented and discussed in this thesis. We also provide an informal validation and a rigorous analysis of the main properties of our

approach. In addition, we conduct experiments to evaluate our approach with respect to SLA constraints of availability and elasticity. The results show that our approach avoids the outage at the application level and reduces SLA violations during the upgrade, compared to the traditional upgrade method used by cloud providers.

Acknowledgments

I would like to express my sincere thanks and gratitude to both Prof. Dr. Ferhat Khendek and Dr. Maria Toeroe, specifically for their supervision, encouragement, and continuous support. This thesis would not have been possible without their tireless guidance, knowledge, and mentorship along the way.

This work has been conducted within the NSERC/Ericsson Industrial Research Chair in Model-Based Software Management, which is supported by Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson and Concordia University.

My sincere thank goes to my Ph.D. committee members for their advice, comments, and their efforts in the final evaluative process.

I express my gratitude to my colleagues at MAGIC (past and present) for their support and friendship, which helped me during the challenging path throughout my Ph.D. program.

My heartfelt gratitude to my family, especially my beloved parents (Ahad Nabi and Farideh Abdi), for their support, patience, and endless love throughout my life. My deepest thanks to my husband, Daniel Mastine, for his love, support, patience and his help during the past few years. I cannot thank them each enough for all the sacrifices they made throughout my journey.

Table of Contents

List of Figures

List of Tables

List of Flowcharts

List of Acronyms

AMF	Availability Management Framework
AWS	Amazon Web Services
CAMP	Cloud Application Management for Platforms
CG	Control Graph
CIMI	Cloud Infrastructure Management Interface
CPU	Central Processing Unit
GUI	Graphical User Interface
HA	Highly Available
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
NRG	New Request Graph
OCCI	Open Cloud Computing Interface
OS	Operating System
OVF	Open Virtualization Format
PoC	Proof of Concept
PPU	Partial Parallel Universe
RG	Resource Graph
SDK	Software Development Kit
SLA	Service Level Agreement
SMF	Software Management Framework
SR-IOV	Single Root I/O Virtualization
TOSCA	Topology and Orchestration Specification for Cloud Applications
VEPA	Virtual Ethernet Port Aggregator
VM	Virtual Machine
VNIC	Virtual Network Interface Controller
VSAN	VMware Virtual Storage Area Network

Chapter 1

Introduction

1.1 Thesis Motivation

Over time, systems need to be upgraded for different reasons, for instance to fix discovered bugs, to add new features, or to fix a security threat. Infrastructure as a Service (IaaS) cloud [1] system, is not exempt from the necessity of such upgrades. The resources of an IaaS cloud system may be upgraded multiple times during their lifecycle, which may impact the services provided by the IaaS (i.e. induce an outage), and the services relying on them. Some of the services, such as carrier grade services, have limited tolerance for service interruption as they are required to be highly available (HA), i.e. available 99.999% of the time. Availability requirement specifies the percentage of the time a system or a service is accessible, thus the allowed outage time for HA services should not exceed five minutes and 26 seconds per year [2][3]. Indeed, a cloud provider is committed to a tenant by a service level agreement (SLA) which indicates the terms of commitment, e.g. the level of availability, that have to be respected even during upgrades [4] . Therefore, the upgrade of IaaS cloud system has to be carried out with minimal impact on the services with respect to their availability requirement indicated in the SLAs.

Many of the challenges of maintaining availability during the upgrade of IaaS cloud systems are similar to traditional clustered systems, while others are specific to the cloud. As in clustered systems, handling the existing dependencies is important to prevent service outages during the upgrade. In the cloud environment different service models (i.e. layers) like IaaS, Platform as a Service (PaaS), and Software as a Service (SaaS) are potentially built on top of each other [5]. For example, a SaaS cloud can be built on an IaaS or PaaS cloud. As a result, the upgrade of IaaS layer can impact the other layers relying on the IaaS. Besides the dependencies between the layers, there are also dependencies between resources within the IaaS layer. The functionality and lifecycles of these resources are tied to each other and their upgrades have to be orchestrated properly to prevent service outage.

Moreover, during the upgrade process incompatibilities that do not exist in the current or in the desired configuration may arise during the transition and break the dependencies. As in traditional clustered systems, this may induce service outage. Therefore, during the upgrade, special consideration should be given to the potential incompatibilities along the dependencies. The specific upgrade methods handling the potential incompatibilities may require additional resources. Considering the scale of IaaS cloud system, minimizing the amount of additional resources for the upgrade process purpose is a more significant challenge in comparison to clustered systems. Moreover, the upgrade actions (e.g. installing software) may fail. In order to guarantee the consistency of the system configuration, these failures have to be carefully handled during the upgrade.

The dynamicity of cloud systems introduces additional challenges for the upgrade of IaaS. Cloud systems adapt to the workload changes by provisioning and de-provisioning resources automatically according to the workload variations. This mechanism is referred to as *autoscaling* [6][7] or *elasticity* [8]. The autoscaling feature may interfere with the upgrade process in different ways. The service capacity of the system decreases during the upgrade when resources are taken out of service for upgrade. In the meantime, the system may have to scale out in response to workload increase. Furthermore, the autoscaling may undo or hinder the process of the upgrade when scaling in releases newly upgraded resources (e.g. VMs), or when scaling out uses the old (i.e. not yet upgraded) version of the resources. To avoid these interferences, it is generally recommended to disable the autoscaling during upgrades as done in [9][10]. However, disabling this feature during the upgrade, deactivates one of the inherent characteristics of the cloud instead of properly addressing the interferences. Moreover, due to the large scale of IaaS cloud systems, it may take an extended period of time to perform the upgrades. Thus, disabling the autoscaling is inappropriate. An upgrade approach must mitigate this interference.

1.2 Contribution of the Thesis

The main objective of this thesis is to devise an approach and a framework for automating the upgrade of IaaS cloud systems, according to the upgrade requests specified by the administrator, and under SLA constraints for availability and elasticity. The proposed approach and framework address all the aforementioned challenges of IaaS cloud upgrade in an integrated manner and it is applicable to upgrade of all kinds of IaaS resources.

The main contributions of this thesis are as follow:

- An approach for the upgrade of IaaS cloud systems under SLA constraints for availability and elasticity. The proposed approach determines and schedules the necessary upgrade methods and actions appropriate for the upgrade requests in an iterative manner, while handling all the challenges. To prevent service outage due to existing dependencies, at the runtime it identifies the resources that can be upgraded without violating dependency requirements according to the configuration of the system. The potential incompatibilities along the dependencies are determined using information coming from cloud vendors and handled using appropriate upgrade methods according to the types of dependencies. In addition, the amount of additional resources is minimized by identifying only the subsystems where additional resources are required for the upgrade process. This approach avoids interferences between the upgrade and the autoscaling processes by regulating the pace of the upgrade according to the state of IaaS cloud system with respect to SLAs. Accordingly, the upgrade starts/resumes if and only if resources can be taken out of service and upgraded without jeopardizing the availability of the IaaS services. To maintain the consistency of the system configuration, in case of failures during the upgrade, the necessary retry and undo operations are identified and issued automatically, as appropriate for the failed upgrade actions. This approach is also capable of handling new upgrade requests even during ongoing upgrades, which makes it suitable for continuous delivery. So, by tackling all of the challenges in an integrated manner, it automates the entire upgrade process for IaaS cloud systems. This contribution has required several investigations and sub-contributions:

a) Infrastructure resource information models for the IaaS cloud system, for the purpose of upgrade: Since during the upgrade, the IaaS cloud system is transferred from a source configuration to the desired one (according to the upgrade requests), it is important to identify the configuration information necessary to facilitate such an upgrade. We identified the necessary information by defining infrastructure resource domain models for the upgrade.

b) Characterization of infrastructure resource dependencies: To carry out the upgrade of IaaS cloud systems with minimal impact on the services with respect to their availability, it is essential to identify the dependencies between IaaS resources. We characterized the existing dependencies in the IaaS cloud layer.

- An upgrade management framework for upgrading IaaS cloud systems. We propose a framework with two main components, an *Upgrade Coordinator* (realizing the proposed upgrade approach) to coordinate the process of the upgrade, and an *Upgrade Engine* to execute the necessary upgrade actions on the resources of the IaaS cloud system. The upgrade coordinator automatically generates *Runtime Upgrade Schedule*(s), each of which indicates upgrade actions and the set of resources on which to apply them. The upgrade engine executes the upgrade actions indicated in the schedules, and provides feedback to the upgrade coordinator indicating the results of the execution. The feedback is used by the upgrade coordinator, to coordinate the remaining upgrades, and generate additional schedules to bring back the system to a consistent configuration in case of failures.

To demonstrate the feasibility of our proposed framework in a real deployment, we first developed a proof of concept (PoC) for upgrading IaaS compute and its application in a virtualized OpenStack cluster. In this PoC, we specifically tackle the additional challenge of upgrade of IaaS cloud system, i.e. dynamicity in the cloud, compared to clustered systems.

We also implemented a prototype of our proposed approach for the coordination of IaaS cloud upgrade, which is applicable to all kinds of IaaS resources. In this implementation, in order to demonstrate the progress of the upgrade, we simulated the behavior of the upgrade engine, which is responsible for applying the schedules generated by the proposed approach in the real system.

To give more confidence on the correctness of our approach, we prove informally, but in a rigorous manner, four main properties of our approach:

1) A given change set in an upgrade request will be applied successfully or will be undone, while keeping the system configuration consistent. Failed resources are isolated to keep the system configuration consistent,

2) If there is no new upgrade request, all of the previously issued upgrade requests will be completed,

3) If the tenants scale out with respect to SLAs, and if the probability function for failure estimation gives accurate results, our approach will respect the SLA constraints of elasticity and availability, and

4) our approach uses minimum additional resources during the upgrade.

Furthermore, we evaluate our approach by conducting experiments to demonstrate how our approach works to respect the SLA constraints of availability and elasticity during the upgrade, compared to the traditional upgrade method used by cloud providers.

1.3 Thesis Organization

The rest of the thesis is organized as follow: In Chapter 2, we lay out the background for our work before reviewing the related work. In Chapter 3, we present the infrastructure resource models, the possible changes in the IaaS layer and the characterized IaaS dependencies. In Chapter 4, after providing definitions of related concepts, we elaborate on the principles used for tackling the aforementioned challenges and we provide an overview of our proposed upgrade management framework. In Chapter 5, we provide definitions and the necessary notations for our proposed approach and we elaborate on the approach for the upgrade of IaaS cloud systems. In Chapter 5 we provide an informal validation and a rigorous analysis of four aforementioned properties of the proposed approach. In Chapter 6, we discuss the proof of concepts developed for demonstrating the feasibility of the proposed approach and the framework. In Chapter 6, we also present an experimental evaluation of our approach for the upgrade of IaaS compute. In Chapter 7, we conclude the thesis.

Chapter 2

Background and Related Work

In this Chapter, we first layout the background of our work by providing an introduction and overview for cloud computing, scaling, OpenStack cloud platform, and availability. Then, we review the work related to this thesis.

2.1 Background

2.1.1 Cloud Computing

Cloud computing is defined, by U.S National Institute of Standards and Technology (NIST) [1], as a model for enabling on-demand access to a pool of configurable computing resources (e.g. network, servers, storage) which can be provisioned and de-provisioned rapidly. In the cloud, the costumers pay the cloud provider based on their consumption of services [11], which is referred to as pay-as-you-go pricing model [12].

NIST [1] defined five essential characteristics of the cloud as follow:

- On-demand self-service: The cloud computing services (e.g. VM, network, and storage) can be provisioned by consumers without human interaction with cloud service providers.

- Broad network access: All of the services are available and accessed through the network.
- Resource pooling: The resources of cloud providers (e.g. compute, storage, and network) are pooled to provide services to multi tenants according to demand for each customer.
- Rapid elasticity: The cloud system is capable of provisioning and de-provisioning services according to the consumer's workload requirements.
- Measured service: The usage of cloud resources can be monitored and controlled using some metering capabilities.

Cloud systems may be categorized under one of the following three main service models based on the type of services provided to the consumers [1]:

- Infrastructure as a Service (IaaS): In this service model infrastructure resources (e.g. computing, storage, and network) are provided to the consumers as services. Here in this model, the consumer has limited access to the underlying infrastructure resources. However, the services provisioned by cloud consumers can be tailored by consumers' requirements [1]. Amazon EC2 [13] is an example of IaaS cloud.
- Platform as a Service (PaaS): In this service model, a predefined development platform and environment is provided to the consumer which allows them to deploy their applications on these platforms. Here, consumers only have control on their deployed application [1]. Google AppEngine [14] is an example of this cloud service model.
- Software as a Service (SaaS): In this service model, consumers are provided by the application running on the infrastructure. Here, the cloud consumers does not have any control on the application and underlying infrastructure layers [1]. SalesForce.com [15] and Google Doc are examples of the SaaS cloud service model.

2.1.2 Scaling

The infrastructure underlying the cloud services is a component based distributed system. The different components may have capacity limits. Violating these limits may cause performance degradation in some circumstances; for example, in case of retransmissions. Other severe violations may cause different types of failures such as VM failure. *Auto scaling,* also referred to as elasticity, is a mechanism for provisioning and de-provisioning resources on demand, based on a schedule or the changes in the workload [6][8]. It optimizes resource utilization while providing protection against overload for any computational resource [16].

There are two different type of scaling: *horizontal scaling* can occur by adding or removing resources from the system, referred to as scaling out or scaling in, respectively. *Vertical scaling* accommodates the workload changes by increasing or decreasing properties of the resource (e.g. increasing/decreasing the size of VM); referred to as scaling up or down, respectively [16].

Most cloud providers use reactive policy-based mechanisms for autoscaling. They define autoscaling groups to control the scaling process, and a scaling policy is associated with each scaling group, which among others indicates the conditions which trigger scaling [17][7]. In a scaling policy, some parameters, such as the maximum size, the minimum size, the cooldown and the scaling adjustment are defined. The maximum and the minimum sizes indicate respectively the maximum and the minimum number of instances in the autoscaling group. The cooldown period is the minimum amount of time between two subsequent autoscaling operations; and the scaling adjustment is the size of the adjustment in terms of instances in a scaling operation [7]. For example, for a system with a CPU threshold of 70% with a cooldown period of 60 seconds and a scaling adjustment of one, if the CPU utilization goes beyond 70% for 60 seconds, one new resource (herein VM) will be added.

2.1.3 OpenStack

OpenStack [18] is an open source cloud platform built up from different components. Different services given by these components are responsible for managing infrastructure resources and building IaaS cloud. Some of the main services of OpenStack are *Nova* responsible for management of compute instances, *Keystone* identity service responsible for authentication and authorization, *Swift* responsible for management of object storage by storing and retrieving unstructured data objects, *Glance* in charge of controlling VM images, *Cinder* for providing block storage, and *Heat* as the orchestration service [18].

Heat [19] is the orchestration service in OpenStack, which deploys VMs on the OpenStack platform based on the configuration described in the heat template. It also provides autocaling service in OpenStack which manages the VMs in the scaling group specified in the template. Heat can be integrated with the configuration management tools to manage the infrastructure resources [19].

2.1.4 Availability

Availability is a non-functional requirement specified in terms of the percentage of time a system or a service is accessible. This percentage determines the allowed outage time for a given period [3]. More specifically in [2] availability is defined as “the degree to which a system is functioning and is accessible to deliver its services during a given time interval.” It is the percentage of time a system is ready to perform its functions and is calculated as follows:

$$Availability = MTTF/MTBF = MTTF/(MTTF + MTTR) \tag{1}$$

where MTTF (Mean Time To Failure) is the mean time it takes for the system to fail; MTBF (Mean Time Between Failures) is the mean time between two failures and represents the sum of MTTF and MTTR (Mean Time To Repair) [2].

One can view the availability of a system through the availability of its services. Service availability can be defined as:

$$ServiceAvailability = ServiceUptime/(ServiceUptime + ServiceOutage) \quad (2)$$

where service uptime is the duration during which the system delivers the given service, while service outage (or also referred as downtime) is the period during which the service is not delivered [2].

High availability (HA) is a strict requirement and refers to an availability of at least 99.999% of the time, which permits for approximately five minutes of downtime per year including scheduled and unscheduled maintenance [3]. Telecommunication services have this HA requirement, which they should not experience a downtime of more than five minutes and 26 seconds per year including downtime due to upgrade.

In order to maintain the availability of the service, different mechanisms may be considered. Protective redundancy is one of the main mechanisms, in which redundant elements are used in the system to protect the system against failures. These redundant elements would not be necessary if the system functions correctly, however it is necessary to guarantee HA. The redundant elements may be organized in different ways and collaborate following different rules. A redundancy model represents this logical organization and the related rules. The element providing the required service to the users plays the active role, while the redundant element, which can be used to take over the active role, is referred to as standby [3]. The literature distinguishes different types of standbys:

Hot Standby: is an instantiated standby that can take over the service of the failed active with no or little downtime. A hot standby can have different levels of state synchronization with the active element. Accordingly, it may be referred to as "updated" and "not updated" hot standby. In the case of an updated hot standby, the state is pushed from the active to the standby. However, in the case of a not updated hot standby, it may be an instantiated spare, which is idle in an initial state, or there may be state synchronization between the active and standby. The synchronization in the not updated hot standby takes place periodically as it is pulled by the standby rather than continuously pushed by the active (or on behalf of the active) as state changes take place [4] [3]. Note that in some papers [20] [21] not updated hot standby is referred to as warm standby. A hot standby can be used for both stateless and stateful applications [3].

Standbys also can be dedicated or shared. A dedicated standby is associated with a single active element. While a shared standby is associated with a number of active elements and takes over the active role of any of them [3].

Spare: a redundant element which can be instantiated or uninstantiated. The uninstantiated spare element is also referred to as Cold Standby. Since a cold standby is unistantiated, it needs some time before it can take over the active role, and thus typically resulting in some downtime. With a cold standby, the state of the active element is stored by check pointing or other techniques. After instantiation, the standby element is synchronized by pulling the state information at the moment it needs to take the active role [3].

By different combinations of roles, and associations, different levels of availability can be achieved. The most relevant combinations have been defined as redundancy models. The SA Forum Availability Management Framework (AMF) [2] defines the following redundancy models:

- 2N: In the 2N redundancy model there are at least two redundant elements one of which provides actively all the protected services while the other element protects all these services as a hot standby.
- N+M: The task of providing the protected services is distributed among multiple (N) active elements, which are protected by one or more (M, M<N) standby element(s). An element may only take either the active or the standby role for all the services it provides or protects.
- N-way: In contrast to N+M, a redundant element may actively provide some services while it protects other services as a standby. Each service is provided by one element in the active role and may be protected by one or more standby elements.
- N-way-Active: In this redundancy model there is no standby element assigned for any of the services. All elements provide the protected services in the active role in a load sharing manner. That is the same service may be provided by more than one element.
- NoRedundancy: A single element is assigned the active role for a given service. An element can provide one service at most. The services are protected by spares.

The proactive redundancy is being used for maintaining the availability in the cloud. Different cloud models have different responsibilities but do rely on each other on a top down manner. SaaS relies on PaaS, and SaaS and PaaS rely on IaaS. The availability of each model depends on the availability of the model(s) it relies on. Note that the availability of the application deployed in the VMs is out of scope for the IaaS layer [3]. However, Placement of the physical nodes of the VMs hosting the application is also important to enhance the availability of a system. VM placement is one of the placement strategies in which VMs are placed on different nodes using specified availability constrains. This technique has been used in [22][23][24][25]. VM placement can increase the availability in case of host failure, however this technique cannot guarantee high availability and protection against application failures. Hence it needs to be used as part of HA mechanism and in combination with a redundancy model to ensure the handling of application failures.

In general, two types of VM placement policies can be used: affinity and anti-affinity. The affinity placement policy is used to enhance performance, while the anti-affinity placement policy is used to ensure availability. Note that the anti-affinity policy is referred to as availability placement policy in the Open Virtualization Format (OVF) [26]. For our purposes, the anti-affinity placement groups are important as they indicate the VMs that cannot be placed on the same host. Moreover, considering the application level redundancy, VMs of the same anti-affinity group must not be upgraded at the same time, otherwise the availability of the application layer may be impacted.

2.2 Related Work on Upgrade

There are different upgrade methods proposed for maintaining HA during the upgrade of cluster-based systems. However, none of these methods alone is sufficient to overcome all the challenges faced in the upgrade of IaaS cloud systems. These methods were designed for cluster-based HA systems and they address in isolation from one another the different challenges of upgrading such systems. To upgrade the IaaS cloud systems, these methods can be used in an upgrade orchestration framework which handles the cloud specific aspects of upgrade (e.g. dynamicity of the system).

In [27] three methods have been proposed for upgrading cluster based giant-scale systems, "*fast reboot*", "*rolling upgrade*", and "*big flip*". Fast reboot is proposed as the simplest upgrade method by quickly rebooting the entire system simultaneously into the new versions, however it cannot satisfy service continuity and HA of the applications running on the system. To maintain HA during the upgrade when there is no incompatibility between the versions of the nodes rolling upgrade is recommended. The nodes are upgraded one at a time like a wave rolling through the cluster. Although rolling upgrade is also introduced in [28] as one of the industry best practice, it is also criticized in [29][30][28] as it may introduce incompatibilities (referred as mixed-version inconsistencies) during the upgrade. Moreover, applying the rolling upgrade to a large system may take very long time. In addition, the rolling upgrade has to be applied separately to upgrade of different kinds of IaaS resources. This adds to the duration of upgrade. In our approach, to minimize the duration of the upgrade, we identify the resources that can be upgraded simultaneously (while respecting dependencies and SLA constraints) and apply the rolling upgrade with dynamic batches.

In the presence of incompatibilities, [27] recommends the use of the big flip method, which overcomes this challenge by upgrading one half of the system first and then flipping from the old version to the new one to prevent running two different versions at the same time [27]. Note that big flip is referred to as split mode in our and some other papers [31]. Although this method is powerful to avoid incompatibilities, it reduces the capacity of the system to its half during the upgrade, which is an issue if there is not enough redundancy in the system. In our approach we apply the split mode method to subsystems where the incompatibilities might be an issue, instead of applying it to the entire system. To improve this upgrade method, delayed switch is proposed in [32] for upgrading cloud systems, where first the nodes are upgraded one at a time and remain deactivated after the upgrade to avoid incompatibility. When half of the system is upgraded, a switch is performed by deactivating the remaining old nodes and reactivating all the upgraded ones. Then, the remaining old nodes are upgraded simultaneously [32]. Another solution proposed for avoiding backward incompatibilities during upgrades is to use explicit embedded versioning at the development time of the software [30]. However, in [30] this solution is applied to a limited set of resources, i.e. which modifies persistent data structures. It is not applicable to upgrade the different kinds of IaaS resources.

To address backward incompatibility, other techniques similar to big flip have also been proposed. In [29][33][34], the Imago system (also referred to as "parallel universe") is presented to perform online upgrades. In this method, an entirely new system is created with the new version of the software, while the old system continues to run. Similar to split mode (or big flip) first, persistent data is transferred from the old system to the new one to be able to test the new system before switching over. Once the new system is sufficiently tested, the traffic is redirected to the new system [29]. Since an entire new IaaS cloud system has to be created with the new version of the resources, the used resources during the upgrade are doubled by this method. Thus, this solution is expensive and may not apply to the upgrade of all cloud system. To minimize the amount of additional resources used during the upgrade, in our approach, instead of bringing up a complete IaaS system as a parallel universe, we use this method locally to upgrade the infrastructure resources supporting VM operations.

Despite the challenge of incompatibility associated with rolling upgrades, this method is still widely used by cloud providers. Windows Azure storage uses rolling upgrades to upgrade the storage system [35] by upgrading one upgrade domain (i.e. a set of evenly distributed servers and replicated storages) at a time in a rolling manner. To maintain the HA of the system during the upgrade, enough storage replicas are kept in the remaining upgrade domains [35]. Rolling upgrades are also used by Amazon Web Services (AWS) to update or replace Amazon Elastic

Beanstalk (PaaS) [36], or Amazon EC2 (IaaS) [9] instances. In Amazon EC2, the rolling upgrade is applied to instances of autoscaling groups in which the batch size can be predefined. To avoid interference between upgrade and autoscaling, it is recommended to suspend autoscaling during the upgrade [9]. Disabling the autoscaling feature during upgrades, disables one of the most important features of the cloud and therefore does not appropriately address the challenge of interference between the upgrade and the autoscaling. In our work, instead of disabling the autoscaling feature, we make it regulate the upgrade process.

Rolling upgrade is also used in [37] to upgrade the VM instances. In this work, the optimization problem of rolling upgrades with multi objectives of minimizing the completion time, the cost and the expected loss of service instances (i.e. VMs) is targeted. They formalized the rolling upgrade considering different iterations of upgrade with a fixed batch size (or granularity in their terms) defined by the operator. Considering potential failures during upgrades, the number of successful upgrades may be less than the predefined batch size, resulting in a longer completion time [37]. In contrast to our work, [37] does not consider changes in the number of VM instances during the upgrade due to autoscaling; so, it does not address the challenge of interferences between autoscaling and the upgrade process.

In [38] a rolling upgrade is used for the reconfiguration of cluster membership using quorums. A subset of the servers, which have the same replicated information, are organized into a quorum. Any member of the quorum can become the candidate leader to initiate a configuration change. The proposed configuration with the largest ballot is selected as the target configuration. In each iteration of the upgrade, a predefined batch of servers is upgraded simultaneously. This approach is suitable for upgrading distributed state-full services (e.g. database service) except for the distributed locking service [38]. Likewise, in this paper, the dynamicity of the system due to autoscaling is not considered.

In [39] state aware instances are suggested to address the incompatibility issues in the rolling upgrade. The instances are upgraded from the old version to the new one using rolling upgrades. However, only instances with the old version are active until a point where the switch-over is performed to the new version while deactivating the old version. This method is similar to the delayed switch method, with the difference in the switching point, which can happen at different points. According to this paper, the switch point has to be determined based on the availability and scalability requirements of the system and the impact of the switching point on the availability and the capacity of the system [39]. Although this paper quantifies the risk associated with the version switching during the rolling upgrade, it considers neither the possible interference of the upgrade process with the autoscaling feature, nor the upgrade of different kinds of IaaS resources.

In [40], an approach is proposed for controlling the progress of the rolling upgrade based on failures, which is referred to as "robust rolling upgrade in the cloud (R^2C)". In this paper, which is an extension of [39], the rolling upgrade controller controls the progress of the upgrade based on inputs from an error detection mechanism. Based on the type of the failure during the upgrade (e.g. platform/infrastructure failures and operation failures), the rolling upgrade controller decides whether to replace the problematic instance or to suspend the upgrade process if the errors impacts the process of the upgrade. Similar to [39], since they replace the failed resource with the old version of the instance, in each iteration during the upgrade, the number of upgraded resources can increase or decrease. This paper provides a prediction model for the expected completion time of the rolling upgrade based on the probability of the different failures and using the different batch sizes (or granularities in their terms) for different runs of the upgrade. Note that in [39], the batch size is fixed during the upgrade, and it is set by the administrator. In addition, the administrator also selects the switching point to the new version.

Since the batch size is not adjusted at runtime to the current state of the system with respect to the SLAs of the tenants, autoscaling may interfere with the upgrade process. In contrast, our approach regulates the upgrade process based on considerations for potential scaling out requests to minimize such interference. More importantly, both [40] and [39] target only the upgrade of VM instances, while our approach handles the upgrade of IaaS resources.

In [41], the upgrade of software deployed on the VM and VMimage is targeted from the user perspective. The goal of this work is to automatically apply the upgrades according to a user request. The proposed software system, referred to as UaaS (Update as a Service), is designed for IaaS clouds. In order to upgrade the software, the user submits the update service request and then the provider applies the requested upgrades. On each VM, an agent is installed. The agent is responsible for collecting software package information on the VM. In order to have low overhead of data collection from the agents, instead of using a pull mechanism, they use a push mechanism where agents submit the software package status of the VM to the master whenever a change (install/uninstall) happens. This information includes the list of installed software packages, and the version of the installed packages. For the offline VMs, an agent outside the VM is considered which can access the VMimage to mount the image and parsing files to get the required information. A central controller (master) is used to collect all of this information and identify the target VMs that need to be upgraded. Subsequently, the master will notify the agents to perform the upgrade action on online or offline VMs [41]. While this work considers the automatic upgrade of the VMimages and software deployed on the VMs, it considers neither maintaining availability, nor the upgrade of different kinds of IaaS resources.

Although all the above-mentioned upgrade methods address the problem of maintaining HA and in some cases the challenge of incompatibilities, they do not address all the challenges the upgrade of cloud systems poses. In particular they do not address the different kinds of dependencies and the dynamicity of the cloud. In contrast, in our work we propose an upgrade management framework for handling all these aspects of upgrades of the cloud in an integrated manner.

Chapter 3

Infrastructure Resource Information Models and Dependencies

During the upgrade, the IaaS cloud system is transferred from a current configuration to a new configuration according to the upgrade requests specified by the administrator. To manage such an upgrade, the configuration information of the IaaS layer has to be identified.

We investigated the existing cloud management standards to see if any of them has all the information necessary for this purpose. We examined the Cloud Infrastructure Management Interface (CIMI) [42], the Open Virtualization Format (OVF) [26], the Open Cloud Computing Interface (OCCI) [43], the Cloud Application Management for Platforms (CAMP) [44], and the Topology and Orchestration Specification for Cloud Applications (TOSCA) [45]. CIMI defines an API for the management of virtual resources within IaaS. OVF defines the format for packaging and distributing virtual appliances. OCCI is again a management API for IaaS which can also be extended to the PaaS and SaaS. CAMP is a management API for the PaaS. Finally, TOSCA is developed for the management of application layer services (SaaS). Through our investigations, we came to the conclusion that none of these cloud management standards today has all the information needed to facilitate upgrades. The main reason is that cloud management standards are mainly developed for the users managing the resources provided by the cloud as services. They lack the information necessary to manage the system configuration as a provider, which is necessary for an upgrade. Therefore, we identified provider side infrastructure resource models for the upgrade. We also identified the possible changes that can be performed on the infrastructure resources.

As mentioned in the introduction, breaking exiting dependencies during the upgrade is the main reason for service outage during the upgrade. To carry out the upgrade of IaaS cloud systems with minimal impact on the services with respect to their availability, it is essential to identify and characterize all the potential dependencies between IaaS resources.

In this chapter, we present the infrastructure resource models and the possible changes applicable in the IaaS layer before characterizing potential IaaS dependencies.

The contents of this chapter have been published partially in [5].

3.1 IaaS Resource Information Domain Models

At the infrastructure level, we identified three types of resources: *physical resources*, *virtualization facility* resources and *virtual resources*. The physical resources are the hardware of the infrastructure on which the rest of this layer is running. Virtual resources are resources provided as services built on top of the physical resources by using the virtualization facilities. Thus, the virtualization facilities enable the creation of virtual environments on top of the physical resources. Therefore, the virtual resources depend on the virtualization facilities, which in turn depend on the physical resources.

We defined a domain model for each of these resource types. It is important to note that these resource domain models are defined for the purpose of upgrade, which determines the level of granularity to consider. Our infrastructure resource domain model for the physical resources,

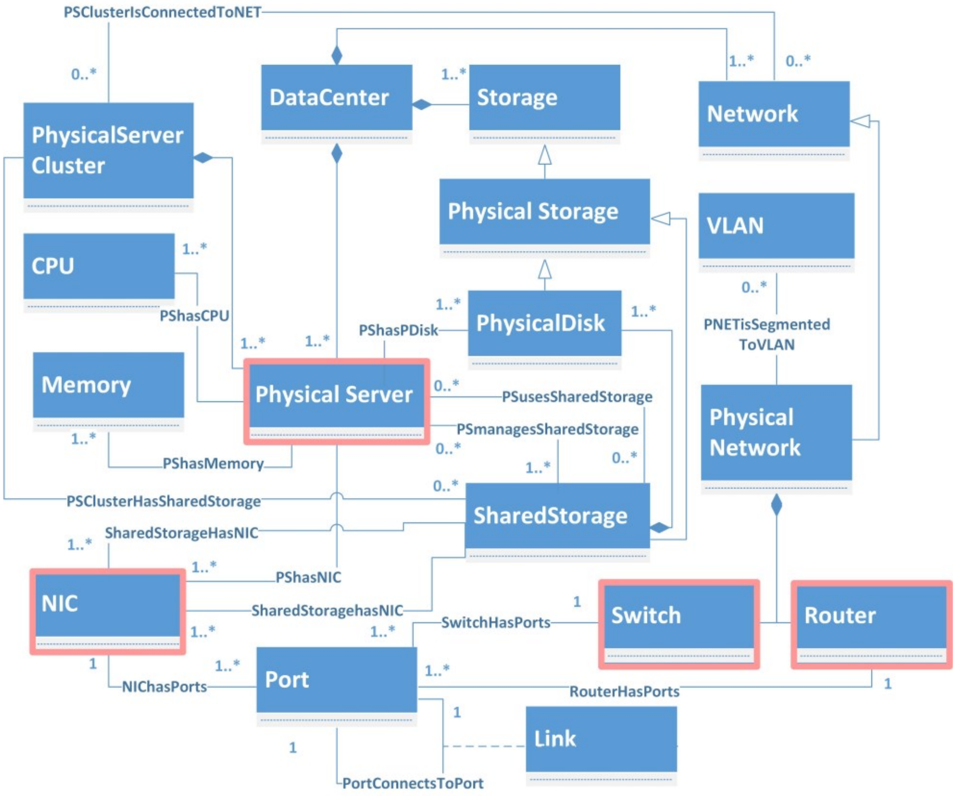


Figure 3.1. Infrastructure resource domain model for the physical resources

the virtualization facility resources[1] and the virtual resources[2] are shown in, Figure 3.1, Figure 3.2 and Figure 3.3, respectively. Note that the connecting elements of the different domain models are highlighted in these figures. The connecting elements of the domain models for physical resources (represented in Figure 3.1) and virtualization facilities (represented in Figure 3.2) are shown in pink, while the connecting element of domain models for virtualization facilities (represented in Figure 3.2) and virtual resources (represented in Figure 3.3) are shown in green.

As shown in Figure 3.1, a cloud data center is composed of physical servers, storage, and network. A physical server has CPU(s), memory(s), physical disk(s), and network interface cards

[1] It should be noted that in this domain model physical entity resources (physical server, NIC, switch, or router) are used for showing the relationship of the virtualization facility resources to the underlying physical resources.

[2] The virtual resources are connected to the underlying virtualization facility resource layer through the hypervisors, or other virtualization facilities (NIC, Switch and Router Firmware).

(NICs). Several physical servers form a physical server cluster, in which physical servers can share a shared storage(s). In this case one or more physical servers may act as controller to manage the shared storage and enable the other physical servers in the cluster to access the shared storage(s). Also, the physical server clusters are connected to the network. The physical network is composed of switches and routers which are connected to other switches and routers through links. The endpoint of a link is a port belonging to a switch, a router or a NIC. In addition, the physical networks can be segmented to VLANs which allows several physical networks to work as a local area network.

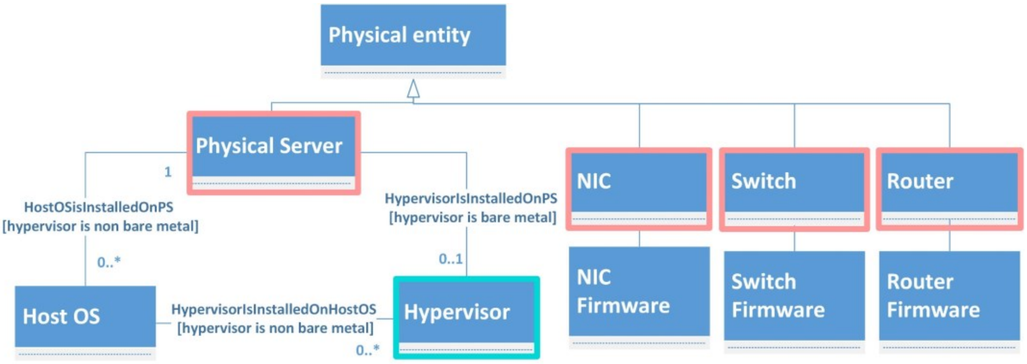

Physical entity
Physical Server
1
HostOSisInstalledOnPS
[hypervisor is non bare metal]
0..*
HypervisorIsInstalledOnPS
[hypervisor is bare metal]
0..1
NIC
Switch
Router
NIC Firmware
Switch Firmware
Router Firmware
Host OS
HypervisorIsInstalledOnHostOS
[hypervisor is non bare metal]
0..*
Hypervisor

Figure 3.2. Infrastructure resource domain model for the virtualization facilities

Virtualization facility resources, shown in Figure 3.2, enable the creation of virtual resources. The hypervisor is the essential resource in this category and based on its type it may be installed directly on the physical server or on a host operating system (OS). Virtualization is not specific to compute resources only, it can also be done for the network and storage resources. Some of the virtual resources are provided by hypervisors; however virtualization technologies may also be built into the firmware of the NIC, switch and router. Additionally, in some cases they can be provided as a virtual appliance running on a hypervisor. In these cases, they need specific technology support from the hypervisor. A vSwitch can be combined with a hypervisor as a single piece of software or provided as a standalone software package or virtual appliance running on top of the hypervisor. When the vSwitch comes as a virtual appliance, one of the vSwitches will act as a vSwitch controller to control and manage the other vSwitches. A virtual switch can also be embedded into the NIC hardware [46] [47] (in the NIC firmware) with the Single Root I/O Virtualization (SR-IOV) technology. Additionally, a vSwitch can be embedded in a switch with the Virtual Ethernet Port Aggregator (VEPA) technology and VN-tag technology [48]. Similar to vSwitches, vRouters can be deployed in or on top of the hypervisor.

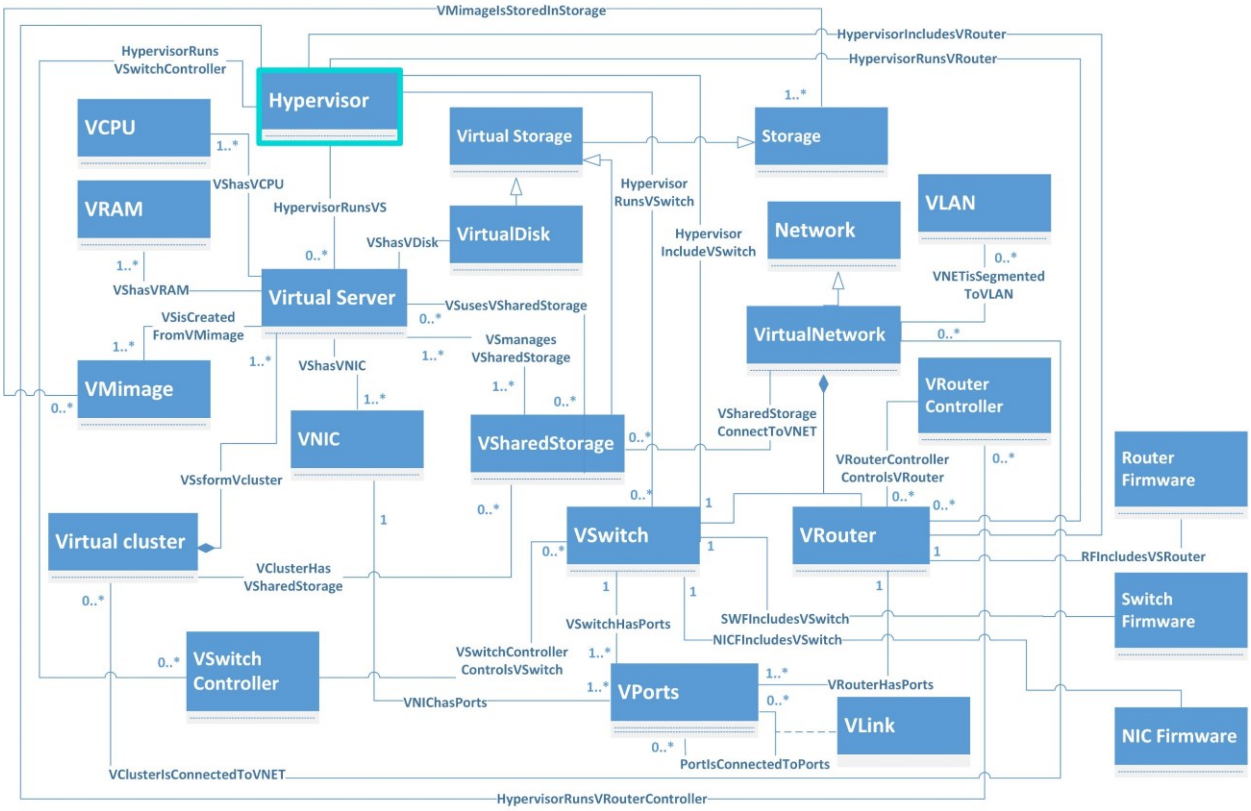


Figure 3.3. Infrastructure resource domain model for the virtual resources

The virtual resources, shown in Figure 3.3, are connected to the underlying virtualization facility resource layer through the hypervisors. Each hypervisor can run multiple virtual servers and allocate VCPUs, VRAMs, virtual disks, and virtual NICs (VNIC) to these virtual servers. Meanwhile, virtual servers can form a virtual cluster, share virtual shared storage(s) and so on as discussed for the physical resources. Accordingly we have virtual shared storage(s), virtual network(s) with virtual switches (vSwitches) and virtual routers (vRouters) connected through virtual links (vLink) with virtual ports (Vports). Note that when it comes to the interconnection the isolation between the physical and virtual resources is primarily administrative and comes from the fact that the virtual resources represent the service offered by the IaaS, which are managed by their users.

In addition, we identified necessary redundancy information for the IaaS provider resources and placement constraints information imposed by upper layer (application layer) for IaaS service resources. We distinguished IaaS resources as *Service* and *Provider* resources. Service resources are the resources given by the IaaS layer as a service to the users of IaaS (e.g. virtual server/VM, virtual switch and virtual router). Provider resources are the resources that are used in the IaaS layer itself for providing the services (e.g. physical server, storage, network resources). Note that virtual storage and virtual network resources can be categorized under service or provider resource whether they are used as provider to give the service, or are provided as a service to the IaaS users.

Figure 3.4 shows the redundancy and placement information in the infrastructure resource domain model. Most of the redundancy configuration in the cloud system is dynamic. So, we defined *PotentialProtectionGroup* and *ProtectionGroup* which consists of provider resources

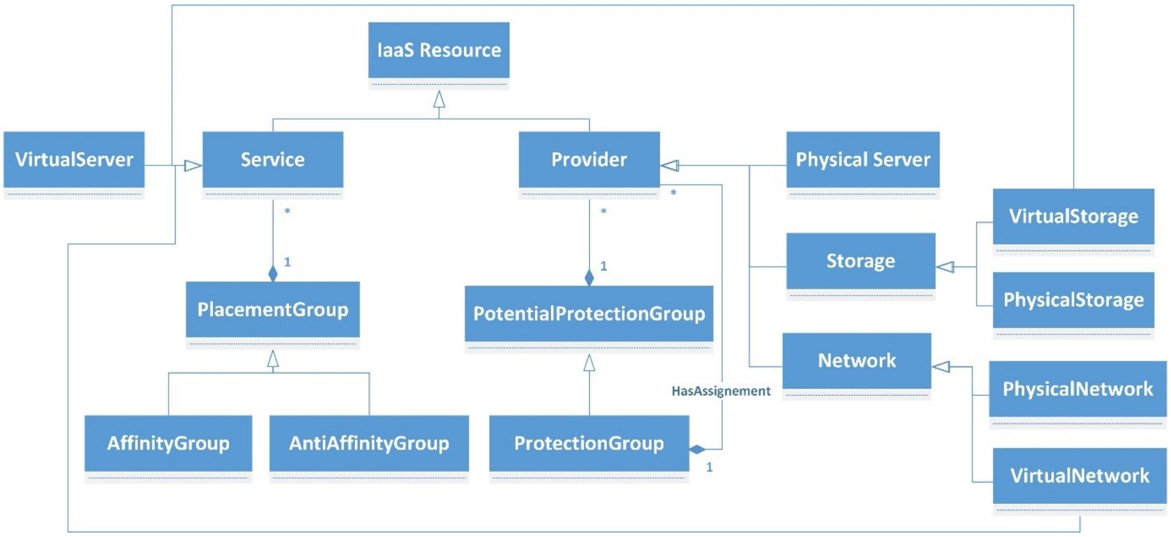


Figure 3.4. Redundancy and placement information in infrastructure resource domain model

which are eligible to get the assignment, and the provider resources within a PotentialProtectionGroup which get the assignment, respectively. For example, set of hypervisors running on physical hosts which are eligible to host the virtual servers are in a PotentialProtectionGroup, and the hypervisors that get the virtual server assignment and host the virtual servers are in a ProtectionGroup. Service resources can belong to a *PlacementGroup* which specifies the group of services that have to follow specific placement constraints imposed by upper layer for IaaS. For instance, VM services which are requested to follow a placement policy belongs to a PlacementGroup. In general, two types of VM placement policies can be used: affinity and anti-affinity. The affinity placement policy is usually used to enhance performance, while the anti-affinity placement policy is used to ensure availability. Thus, we defined *AntiAffinityGroup* and *AffinityGroup* as subcategory for PlacementGroup.

3.2 Possible Changes in the IaaS layer

Depending on the type of the IaaS resource, different types of changes (i.e. add/remove/upgrade) can be performed on a resource. Physical resources can be added, removed, or the firmware of the existing ones can be upgraded (e.g., add/remove physical shared storage, upgrade shared storage firmware). Similarly, the change related to the virtual facility resources can be the addition, removal, or the upgrade of the resource or its firmware (e.g., add/remove/upgrade hypervisor, upgrade switch firmware). Whereas, considering the software nature of the virtual resources, the resource itself can be added, removed or upgraded (e.g., add/remove/upgrade virtual disk).

We defined the list of changes applicable at the infrastructure level for each of the resource types, as shown in Figure 3.5 to Figure 3.7.

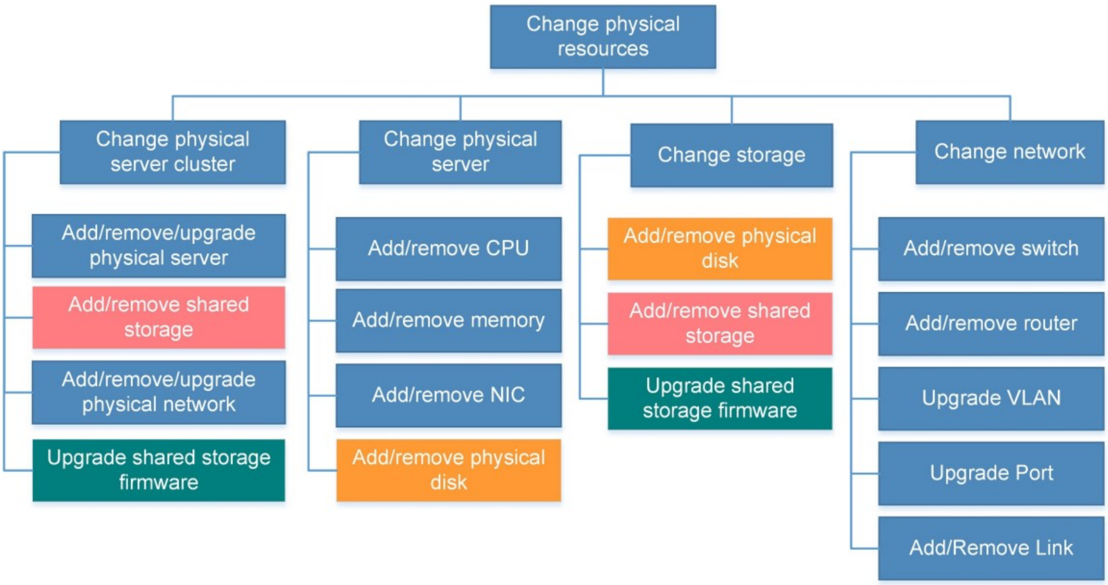

Change physical resources
Change physical server cluster
Add/remove/upgrade physical server
Add/remove shared storage
Add/remove/upgrade physical network
Upgrade shared storage firmware
Change physical server
Add/remove CPU
Add/remove memory
Add/remove NIC
Add/remove physical disk
Change storage
Add/remove physical disk
Add/remove shared storage
Upgrade shared storage firmware
Change network
Add/remove switch
Add/remove router
Upgrade VLAN
Upgrade Port
Add/Remove Link

Figure 3.5. Changes applicable for physical resources

Figure 3.5 shows all the changes related to physical resources. Changing physical resources can be by the change of physical server cluster, physical server, storage resources and network resources. We can breakdown each of these changes into an atomic change, for example, changing the physical server can be done by adding/ removing of CPU, memory, NIC, and physical disk. Note that the same colors (orange, pink, and green) in Figure 3.5 are used for demonstrating the common atomic changes related to different resource types. For example, adding or removing a physical disk (shown with orange color) in Figure 3.5 can be considered as an upgrade of both physical server and storage.

In Figure 3.6, changes related to virtualization facilities resources are illustrated. Change to virtualization facilities can be add/remove/upgrade of hypervisor and host OS, and upgrading firmware of NIC, switch, and router.

Figure 3.7 shows all the possible changes to virtual resources. Add/remove/upgrade of virtual shared storage can be a change to both the virtual cluster and virtual server, which are highlighted in pink.

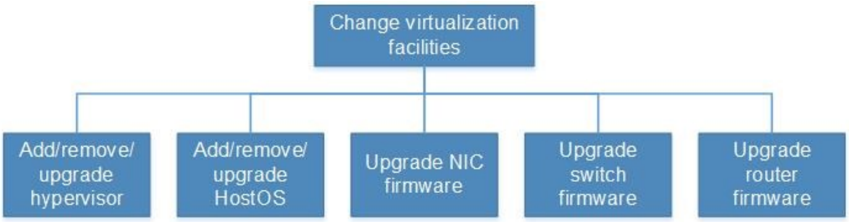

Change virtualization facilities
Add/remove/ upgrade hypervisor
Add/remove/ upgrade HostOS
Upgrade NIC firmware
Upgrade switch firmware
Upgrade router firmware

Figure 3.6. Changes applicable for virtualization facility resources

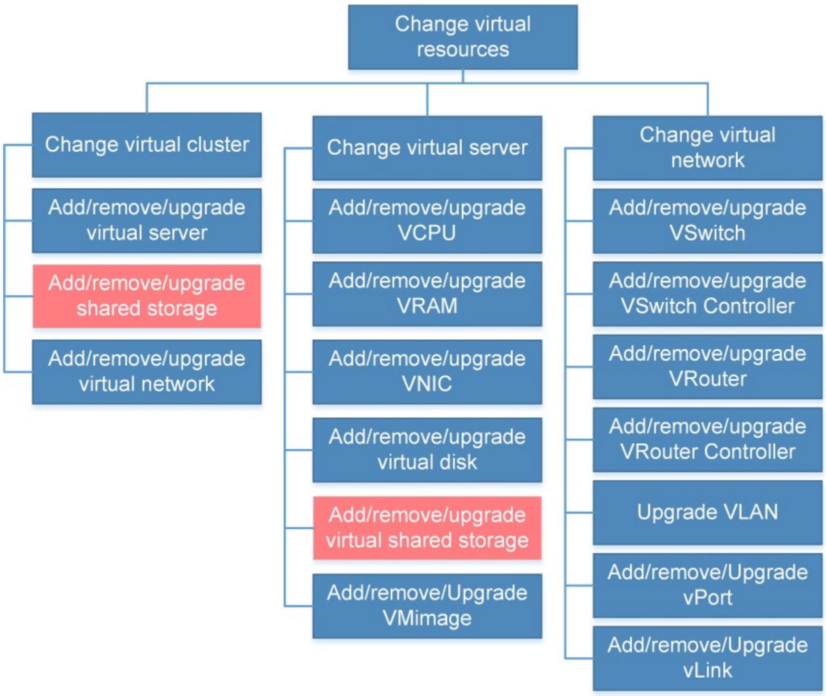

Change virtual resources
Change virtual cluster
Add/remove/upgrade virtual server
Add/remove/upgrade shared storage
Add/remove/upgrade virtual network
Change virtual server
Add/remove/upgrade VCPU
Add/remove/upgrade VRAM
Add/remove/upgrade VNIC
Add/remove/upgrade virtual disk
Add/remove/upgrade virtual shared storage
Add/remove/Upgrade VMimage
Change virtual network
Add/remove/upgrade VSwitch
Add/remove/upgrade VSwitch Controller
Add/remove/upgrade VRouter
Add/remove/upgrade VRouter Controller
Upgrade VLAN
Add/remove/Upgrade vPort
Add/remove/Upgrade vLink

Figure 3.7. Changes applicable for virtual resources

Each of these changes can impact other resource(s) in the system, which may lead to service outage. This is due to the dependencies between the involved resources. Therefore, we have characterized the dependencies at the infrastructure level.

3.3 IaaS Dependency Characterization

With the help of our infrastructure resource domain models, we have characterized the potential dependencies present in the infrastructure layer. Figure 3.8 shows the characterized potential IaaS dependencies. These dependencies can be grouped into two main categories of *Sponsorship dependencies* and *Symmetrical dependencies*.

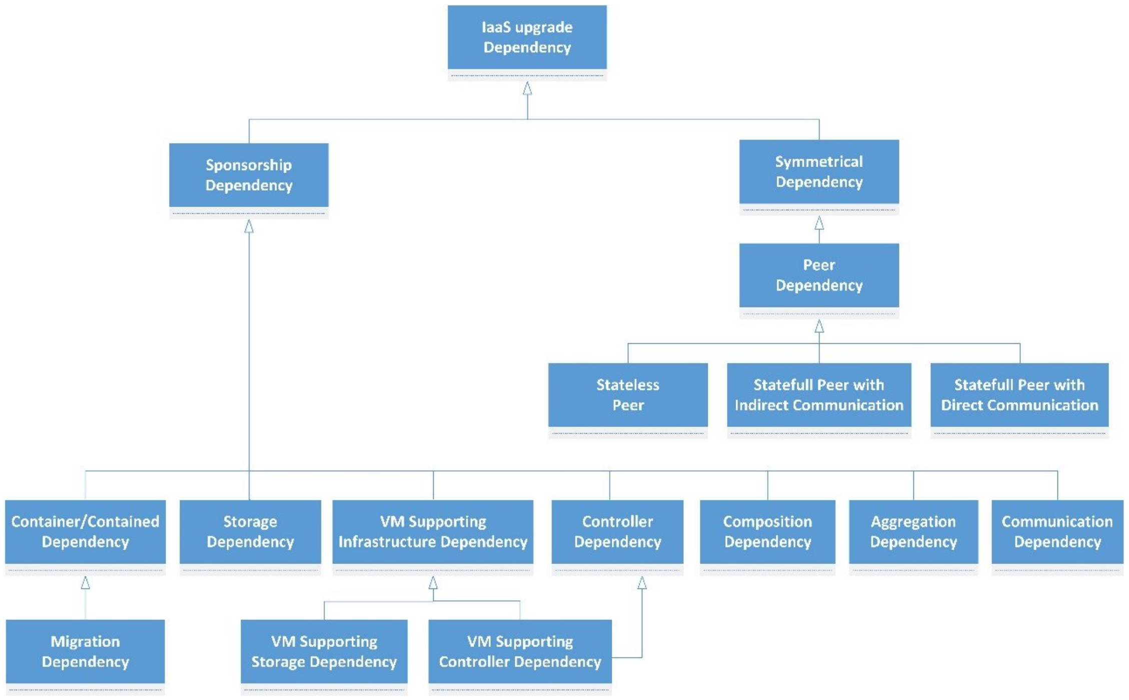


Figure 3.8. Classification of IaaS upgrade dependencies

A sponsorship dependency is a directed dependency which captures the relation between a sponsor and a dependent, in which the dependent cannot function without the sponsor. We defined different subcategories of sponsorship dependency: container/contained, migration, storage, controller, VM supporting infrastructure (VM supporting controller or VM supporting storage), composition, aggregation, and communication dependencies. The second main category of dependencies is referred to as symmetrical dependency. This dependency is a bi-directional dependency, which exists between two or more resources. We defined peer dependency as a subcategory of the symmetrical dependency.

Subcategories of sponsorship dependencies are as follow:

- The container/contained dependency exists between two resources, when the lifecycle of a resource (the contained) depends on the lifecycle of the other (the container). During the upgrade of the container, the contained resources are impacted and experience outage. As an example, this dependency exists between a physical server and its hosted bare metal hypervisor, or between a hypervisor and the vSwitch/vRouter provided by that hypervisor. The upgrade of the physical server causes an outage for the hosted hypervisor, or the upgrade of the hypervisor causes outage of the vSwitch/vRouter it provides.

- Migration dependency is a specialization of the container/contained dependency. In case of migration dependency, the sponsorship relation is dynamic and if not satisfied it triggers a migration. The dependency between virtual servers and a hypervisor is migration dependency. A hypervisor provides VCPU, VRAM, virtual disk and VNIC to its hosted virtual servers. There is a constraint in terms of capacity of CPU, memory, disk and NIC of the hypervisor to host virtual servers (VMs). Thus, only a hypervisor with enough capacity for running the virtual server can host the virtual server. If at any point in time the hypervisor does not have enough capacity, the virtual server will be migrated to another candidate hypervisor (provided there is one), which can provide enough resources for the virtual server. As a result, upgrading the hypervisor has an impact on the virtual server, but it does not necessarily result in an outage of the virtual server.
- Composition dependency exists between multiple resources in which a resource is composed of different resources. If any of the constituent resources goes down, the composite resource will go down. As a result, during the upgrade of the constituent resources, the composite resource is impacted. This dependency exists, for example, between a physical server and its CPU, memory, physical disk and NIC resources, or between a virtual server and its VCPU, VRAM, virtual disk, and VNIC.
- Aggregation dependency exists also between multiple resources in which a resource is composed of multiple resources, but they are of the same type. The difference from composition dependency is that, the dependent resource (the aggregate resource) can

function as long as a minimum number of constituent resources are still available, that is, constituent resources are peers. The aggregate resource will experience an outage whenever the number of available constituent resources drops below the minimum required number. For example, this dependency exists between a virtual shared storage and physical disks, when a virtual shared storage is built using a cluster of physical disks and the data is replicated on different physical disks to maintain the HA of the virtual shared storage.

- Storage dependency exists between two resources, in which a dependent resource is using a storage resource. In this dependency the storage resource is the sponsor. When the storage resource goes down, the dependent resource(s) which is/are using the storage resource may not go down, but their functionality might be impacted. The dependency between the physical (or virtual) server(s) of a cluster and the physical (or virtual) shared storage is such a dependency.
- Controller dependency exists between multiple resources, in which one of the resources controls other resources. If the controller resource goes down, it will lose the control over the controlled resources, which may cause outage. As an example, this dependency exists between vSwitch controller and the vSwitches managed by the vSwitch controller. As mentioned earlier, when vSwitch comes as a virtual appliance, one of the vSwitches act as controller.

- The VM supporting infrastructure dependency indicates a dependency of set of physical servers to an infrastructure resource, in which the infrastructure resource provides infrastructure services for supporting the VM operations running on the dependent physical hosts. If the VM supporting infrastructure resource goes down, all the services provided by the dependent physical servers will have an outage. Note that the physical servers providing VM services are referred to as compute host resources as well. We identified two subcategories of VM supporting infrastructure as VM supporting storage and VM supporting controller dependency.

  - The VM supporting storage dependency is specialization of VM supporting infrastructure dependency. It indicates a dependency of a cluster of physical servers providing VM services (i.e. compute hosts) on a storage infrastructure resource. The storage infrastructure resource provides storage service for supporting the VM operations running on the dependent physical servers.
  - VM supporting controller dependency is another kind of VM supporting infrastructure dependency and it is also specialization of controller dependency. It indicates dependency of a cluster of physical servers providing VM services (i.e. compute hosts) on a controller physical server controlling VM operations on the dependent physical servers. The dependency between OpenStack controller host and OpenStack compute hosts is controller dependency.

- Communication dependency exists between a network resource and other resources. In the system it is realized with physical or virtual link. In this dependency the dependent resource communicates with the external world using the sponsor network resource. If the sponsor network resource goes down, the dependent resource may lose the connection to the network and become isolated. An example of this dependency exists between a physical server and a switch, in which physical server depends on a switch for communication. In this example, the dependency between physical server and the switch is realized with a physical link. If the switch goes down, the physical server is isolated.

Subcategories of symmetrical dependencies are as follow:

- Peer dependency exists between redundant elements which are configured for maintaining the availability of some service(s). To maintain service availability during an upgrade, peer dependent resources should not be upgraded all at the same time. The upgrade should follow their redundancy pattern/requirements, which may be expressed as a minimum required number of in-service peers, or as a maximum number of peer that can be taken out simultaneously. For example, peer dependency exists between redundant storage nodes. We identified three types of peer dependencies:
    - Stateless peer dependency: In this type of dependency, there is no state protection between redundant resources – hence it is stateless – and the peer resources do not exchange state information with each other. This type of peer dependency exists between two NICs providing redundant network connection for a physical node.
    - Statefull peer dependency with direct communication: In this type of dependency, the peer resources are communicating directly with each other to protect the state. This type of dependency exists between two redundant routers that use the virtual router redundancy protocol.
    - Statefull peer dependency with indirect communication: In this type of dependency the peer resources are communicating indirectly through another resource (e.g. database on a host or storage) to keep their state. This type of dependency exists between two peer compute hosts which use a shared storage to keep the images/snapshots of VMs they host.

3.4 Summary

In this chapter, we introduced the infrastructure resource information models for the purpose of upgrade, and the IaaS dependency characterization. In the process of defining the infrastructure resource information models, we have investigated and examined different cloud standards (i.e. CIMI [42], OVF [26], OCCI [43], CAMP [44], TOSCA [45]) to identify if any of them can fulfill the information requirements for the upgrade of IaaS cloud systems. Through our investigations, we came to the conclusion that none of the cloud management standards today has all the information we need in the configuration for the upgrade purpose. Therefore, we identified all the necessary information for upgrading the IaaS cloud system by defining infrastructure resource models. In addition, we characterized all the potential resource dependencies at the infrastructure level to be able to handle in the next chapters the upgrade of different IaaS resources with minimal impact on the service availability.

Chapter 4

Overview of the Framework for IaaS Cloud Upgrade and Principles

Due to the size of cloud deployments and for supporting zero-touch operations, automation of the entire process for the upgrade of IaaS cloud systems is crucial. We defined an upgrade management framework which automates the upgrade of IaaS cloud systems while avoiding or at least limiting service disruptions during the upgrade. This framework addresses in an integrated manner different challenges related to the maintenance of availability during the upgrade.

In this chapter, we first provide the definitions of the concepts used throughout this chapter and then we elaborate how we handle the different challenges of IaaS cloud system upgrade, i.e. the principles of our approach. In addition, we present our evaluation of several configuration management tools that can be potentially used to apply changes to the IaaS resources. At the end of this chapter, we present an overview of our upgrade management framework.

4.1 Definitions

4.1.1 Infrastructure Components

An *infrastructure component* is a piece of software, firmware, or hardware delivered by a vendor as part of a product. The product itself can be a single component (e.g. ESXi [49] hypervisor) or a compound product consisting of different components (e.g. Ceph [50] storage with different components). When a product is fully installed in the IaaS system, this installation becomes a resource (e.g. ESXi hypervisor, Ceph storage) and may consist of the installation of multiple components. Thus, multiple IaaS resources can be mapped to the same infrastructure component (e.g. ESXi hypervisor installed on different hosts) and multiple infrastructure components can be mapped to a single IaaS resource (e.g. Ceph storage with components running on different hosts). We assume that each product delivered by a vendor and therefore each infrastructure component is accompanied with a file – the infrastructure component description – describing among others the component's service capabilities, configuration constraints, hardware management capabilities, delivering software/firmware bundle with their installation/upgrade/removal scripts/commands, estimated time required for their installation/removal, and hardware/software dependencies.

4.1.2 Actions, Operations and Units

To deploy a change in the IaaS cloud system one or more *upgrade actions* may need to be executed. We define an *upgrade action* as an atomic action that can be executed by a configuration management tool (e.g. Ansible [51]) on a resource (e.g. a command for installing ESXi on a host), or performed by an administrator on a resource (e.g. removing a host). An upgrade action is always associated with one or more *undo actions*. Undo actions revert the effect of the upgrade actions on the resource. We use the term *upgrade operation* to represent an ordered list of upgrade actions. We use similarly, the term *undo operation;* while a *retry operation* is defined as a retry of an upgrade operation. A *recovery operation* is defined as undo and/or retry operations.

We define an *upgrade unit* as a group of resources that have to be upgraded using an appropriate upgrade method, for example, for handling the incompatibilities. The resources of an upgrade unit are selected based on the possible incompatibilities along the dependencies of the resources. The upgrade of the resources in an upgrade unit are ordered based on the associated upgrade method, which prevents communication between incompatible versions during the upgrade. An *undo unit* consists of a group of resources on which an upgrade operation has to be applied on all together. Otherwise, the undo operation is triggered. The goal of this grouping is to preserve the consistency of the system configuration with respect to the changes to the IaaS cloud system.

4.1.3 Upgrade Request

The system administrator initiates an upgrade by specifying an *upgrade request*, which is a collection of *change sets*, i.e. a set of change sets. Each change set in the collection specifies a set of tightly coupled changes on the IaaS resources that should either succeed or fail together to maintain the consistency of the system configuration. Within each set each change indicates the addition, removal, or upgrade of an infrastructure component of some resources, some resources themselves, or a dependency between two resources or their sets. Note that the change sets in an upgrade request are independent of each other, and a failure of a change set does not impact the consistency of the system with respect to other change sets.

A system administrator may not be aware of all the dependencies and therefore may not specify all the necessary changes in a change set, i.e. a change set may be incomplete. To satisfy the hardware and/or software dependencies indicated in the infrastructure component description by the vendor, an upgrade request initiated by a system administrator may require complementary changes. To address this issue, we check the completeness of each change set with respect to the infrastructure component description(s) provided by the vendor(s) and derive any missing changes, which then are added to the same change set as complementary changes. For each change, the necessary upgrade actions have to be derived from the infrastructure component description. It is expected that the description contains the scripts used to install and remove a software component, while for a hardware component the scripts is used for its management.

The administrator can also specify four additional parameters in the upgrade request with respect to retry and undo operations. To ensure the completion of the upgrade process, i.e. limit its time, for each change set a max-retry threshold and a max-completion-period can be specified. The *max-retry threshold* parameter controls the retry operations; it specifies the maximum allowed number of upgrade attempts on each resource to which a change in that change set is applied. The *max-completion-period* specifies the maximum time allotted to complete all the changes of the set. To ensure the consistency of the system for each change (in a change set), an undo-threshold parameter and an undo version can be specified. The *undo-threshold* specifies the minimum required number of resources in the set of resources of the requested change that should be operational after applying the requested change. The *undo version* parameter specifies the desired version for the undo operation. By default, this version is the version at which a resource is at the moment the change is applied. This may not be deterministic for upgrade requests issued during an ongoing upgrade. Therefore the default undo version can be overridden by explicitly specifying the undo version in the change request. Note that for complementary changes the undo-threshold and the undo version are derived from the changes requested in the upgrade request.

We keep track of upgrade requests using an *upgrade request model*. This model includes all the information necessary to track the process of applying the changes to the system including failure handling. The execution status of change sets and of changes within each set indicates whether they are new, scheduled, completed, or failed. Whenever a new upgrade request is issued, its change sets, including their respective complementary changes, are added to the upgrade request model. For each change in each change set, the target resources, their source, target and undo versions are reflected, and the execution status is maintained. The target resources and their source versions are identified from the current configuration.

4.2 Principles for Handling Upgrade Challenges

As mentioned in the introduction, we consider several challenges for maintaining availability during IaaS cloud upgrade: (1) dependency of the application (SaaS) layer on the IaaS layer, (2) resource dependencies, (3) potential incompatibilities along the dependencies during the upgrade process, (4) upgrade failures, (5) the dynamicity of the cloud environment, and (6) keeping the amount of additional resources at minimum.

The challenge of the dependency of the application layer on the IaaS layer

As mentioned in the introduction, upgrading the IaaS cloud system can impact the other cloud layers –such as application layer – relying on the IaaS layer. Thus, handling the existing dependency between layers is important to prevent service outages during upgrades. We distinguish between availability management responsibilities of IaaS layer versus application layer. IaaS is not responsible for providing availability solution for protecting availability of the application deployed in the VMs [3]. We assume that the availability of the application deployed in the VMs is maintained by an availability management solution such as the Availability Management Framework (AMF) [52], as proposed in [53] for instance. To handle the dependency of the application layer on the IaaS layer, we assume that the requirements of the application level redundancy are expressed towards the IaaS cloud as VM placement constraints (i.e. as anti-affinity groups). To respect these requirements, during upgrade, VM migration or VM consolidation, the VMs of the same group will always be placed on different physical hosts and at most a specified number (typically one) of VMs of an anti-affinity group will be impacted at a time.

The challenge of resource dependencies

To be able to handle resource dependencies, we have identified the different kinds of IaaS resources and the dependencies between them (as described in Chapter 3). IaaS resource dependencies fall into two main categories, *Sponsorship* and *Symmetrical* dependencies with different subcategories. During upgrade, to avoid breaking any resource dependencies the upgrade has to be performed in a specific order based on the nature of the dependencies. Moreover, to maintain availability we cannot upgrade all the resources at the same time. As a solution, we use an *iterative upgrade process* to select at the beginning of each iteration, the resources that can be upgraded without violating any dependency in that iteration. We re-evaluate the situation at the beginning of each subsequent iteration before continuing with the upgrade. For this selection, first we group together the resources that have to be upgraded at the same time, and then we identify the resource groups that can be upgraded in the current iteration using a set of rules, referred to as *elimination rules*. This results in an initial selection referred to as the *initial batch*, in which the resource groups are selected only based on their dependencies.

The challenge of potential incompatibilities along resource dependencies during upgrade

Even though the source and the target configurations on their own have no incompatibilities, during the transition from one to the other incompatibilities may occur as we need to maintain the availability of services. That is, during the upgrade version mismatch may happen along some of the dependencies for some of the resources. To avoid such incompatibilities these resources have to be upgraded in a certain order using an appropriate upgrade method. Thus, we identify automatically the resources that might have incompatibilities along their dependencies and we group them into *upgrade units*. Note that the information regarding the possible version mismatch can be obtained from infrastructure component descriptions provided by the cloud vendor and considering the existing dependencies in the system configuration. Each upgrade unit groups together the resources that have to be upgraded using an appropriate upgrade method, which avoids incompatibilities by preventing any communication between resources of the incompatible versions. Thus, within an upgrade unit the upgrade of resources is ordered according to the associated upgrade method and the elimination rules used for the batch selection ensure that the resources of the same upgrade unit are selected according to the associated upgrade method. For example, we use split mode [31] to avoid incompatibilities along certain dependencies. In this method, the resources of an upgrade unit are divided into two partitions which are upgraded one partition at a time similar to rolling upgrade. The elimination rules ensure that only one partition is selected at a time, and that the order of deactivation and activation of the partitions is such that it avoids any incompatibilities by having only one version active at any given time until both partitions are upgraded.

Due to ordering constraints, the required upgrade actions on a resource may be required to be applied in different iteration. We defined *execution-level* as an ordered list of upgrade actions to be executed on a resource in a single iteration. Also, we defined *actions-to-execute* as an ordered list of execution-levels to be executed on the resource through different iterations. Thus, the execution-levels order the upgrade actions on a resource, among others, to handle incompatibilities. Each execution-level on a resource is associated with an upgrade unit. In each iteration based on the upgrade unit the elimination rules may or may not remove the resource from the initial batch as appropriate for the order required by the associated upgrade method. Whenever a resource remains in the *final batch* of the iteration (i.e. the resource batch to be upgraded in this iteration), the upgrade actions of its first execution-level will be executed in that iteration. After a successful execution of all the upgrade actions of the first execution-level, the execution-level (with all its upgrade actions) is removed from the list of execution-levels of the actions-to-execute of the resource. Therefore, the next execution-level becomes the first one to be executed in a subsequent iteration whenever the resources is selected again for the final batch.

Upgrade units are also used to handle, for instance, potential incompatibilities introduced by new upgrade requests. Even if the new upgrade requests target the same resources as previous upgrade requests, the new upgrade requests may introduce new incompatibilities. To prevent occurring incompatibilities, new upgrade units different from existing ones are created. The upgrade actions associated with the new upgrade request can only be executed on a resource after finalizing the upgrade actions of the ongoing upgrade requests. To achieve this, upgrade actions associated with a new upgrade unit are grouped into a new execution-level.

The challenge of handling upgrade failures

In case of upgrade failure, recovery operations are performed to bring the system to a consistent configuration. Since changes in a change set are dependent, there are two main criteria to guarantee a consistent configuration: First, all the upgrade actions deploying a change set on a resource must either be applied successfully, or none of them should be applied at all. Second, all the changes of a change set have to be successful without violating their undo thresholds; otherwise, they have to be undone all together.

According to the first criterion, in case an upgrade action of a change set fails on a resource, the effects of the already executed upgrade actions of that set need to be reverted. This is referred to as resource level undo, which take the resource to the version before applying the upgrade actions of the change set. If this is successful and the retry operation is permitted on the resource, i.e. max-retry threshold is not reached yet, another attempt can be made to re-execute the upgrade actions of the set. Otherwise if reverting the upgrade actions was successful (i.e. the previous stable configuration is reached), but the retry operation is not permitted, the resource will be isolated from the system. However, if reverting the upgrade actions fails, the resource needs to be isolated and marked as failed. Hereafter, we refer to a resource, which is isolated but not failed, as an *isolated-only resource*.

If the number of isolated-only and failed resources in the set of resources to which a change is applied violates the undo-threshold value, all changes of the change set will be undone on all applicable resources to preserve the system consistency. Note that since this undo operation is performed in the system level with respect to the change set, we referred to it as system level undo. To account for this, we defined *undo unit* indicating a group of resources on which the undo recovery operation has to be applied together. Thus, an undo unit is assigned to each change set and its targeted resources to maintain the relation of changes applicable to those resources that either need to be deployed or undone all together. The undo operation could be triggered as discussed: if the undo-threshold for a set is violated; if all the upgrade actions of the set cannot be finalized within the indicated max-completion-period; or if the administrator explicitly issues an undo operation for a change set that has not been completed yet. Once a change is completed it cannot be undone, instead a new change can be requested.

When undoing a change in the system level with respect to a change set, all the targeted resources will be taken to the undo version of that change. Note that this undo version specified by the administrator indicates the desired version for the undo operation of the change set and it may be different from the original version of the resource before applying the upgrade actions of the change set. The isolated-only resources may or may not be at the undo version. This is because the isolated-only resources which had a successful resource level undo operation, is taken to the version at the moment the change is applied (not the undo version). If isolated-only resources are at the undo version, they are released from the isolation. Otherwise an attempt is made to take them to the undo version. If this is unsuccessful, they are marked as failed resources.

Note that, there may be several change sets impacting a single resource. Each resource may be associated with several undo units. In our approach when an undo operation is required (e.g. due to an upgrade failure) we perform it locally on the resources targeted by the originating change set instead of undoing all the changes made in the system by the other change sets. The undo operation itself is represented as a change set on the relevant resources and, thus, it can be performed while other change sets are being applied to other parts of the system. Note that the undo actions for the undo operation are organized into the first execution level of the resources so that they will be executed first.

The challenge of dynamicity of the cloud environment

To handle the interferences between autoscaling and the upgrade process, we regulate the pace of the upgrade process. To respect the SLA commitments (scaling and availability), in each iteration the current configuration of the system is taken into consideration and only a certain number of resources is taken out of service for upgrade. Based on the current configuration we consider in each iteration the number of resources necessary for accommodating the current SLA commitments (with respect to scaling) , and we determine the number of resources necessary for any potential scaling out requests and for recovering from potential failures for the

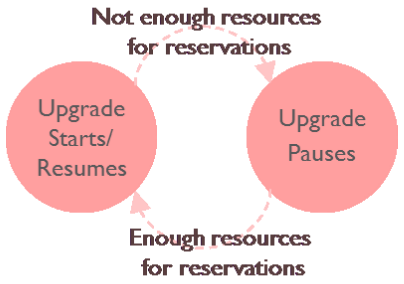

Not enough resources for reservations
Upgrade Starts/ Resumes
Upgrade Pauses
Enough resources for reservations

Figure 4.1. Upgrade process

duration of that iteration. These resources cannot be upgraded without potential violation of availability. So, from the initial batch of resources selected with respect to their dependencies, these resources are eliminated from the final batch. Thus, the upgrade process starts/resumes (as shown in Figure 4.1) if and only if we can take out at least one resource (i.e. the final batch is not empty) and upgrade them without violating the availability and elasticity constraints due to resource failures or valid scaling requests. Otherwise, the upgrade process is suspended until there is enough resource freed up through the process of scaling in.

The challenge of minimizing the amount of required additional resources

Since upgrade takes out resources from the system, providing additional resources to the system may become temporarily necessary for progressing with the upgrade. The amount may depend on the upgrade method, the number of resources the upgrade is applied to and the spare capacity in the system at the moment it is applied. It may be necessary to add resources to enable the use of certain techniques to maintain service continuity and service availability especially in the presence of incompatibilities. As discussed in the related work, some of the upgrade solutions [29][33][34] use the parallel universe method to avoid incompatibilities. Applying the parallel universe method at the system level is expensive in terms of resources. The goal is to use only the minimum necessary additional resources to keep the cost of the upgrade as low as possible. As a solution to this challenge, we identify the subsystem where additional resources are required, and we only use the minimum amount necessary.

To maintain the continuity of the infrastructure services supporting VM operations (e.g. storage, controller), when their resources need to be upgraded and when the new and the old versions are incompatible, we propose to use a *Partial Parallel Universe (PPU)* method. This method applies the parallel universe method locally to a subsystem (e.g. VM supporting infrastructure storage or controller subsystem) instead of creating a complete IaaS system as a parallel universe. With the PPU method we create a new configuration of the VM supporting infrastructure resources with their new version while (in parallel) we keep the old version of such infrastructure resources and their configuration until the new one can take over the support for all the VMs. To achieve the transfer, the physical hosts providing the VM service of the IaaS (i.e. the compute hosts) are also divided into two partitions. The old partition hosts VMs compatible with the old version of the VM supporting infrastructure resources and it hosts all the VMs initially. The new partition, which is empty initially, hosts the VMs compatible with the new version of the VM supporting infrastructure resources. As soon as the new version of the VM supporting infrastructure resources is ready, we migrate the VMs from the old to the new partition potentially in multiple iterations as appropriate for their SLAs. Once all the VMs have been migrated from the old partition to the new one, the configuration of the VM supporting infrastructure resources with the old version can be safely removed. This means that to guarantee the continuity of the VMs supporting services, the requirements for both versions of the configurations of VM supporting infrastructure resources have to be satisfied simultaneously during the upgrade and until the completion of the VM migrations. If these requirements cannot be satisfied using existing resources, additional resources may be required. So, we keep the number of required additional resources to a minimum by trying to use available resources as much as possible during the upgrade and request for additional resources only if they are necessary.

4.3 Evaluation of Configuration Management Tools for Upgrade

Tackling different challenges for maintaining availability during the upgrade in isolation from one another does not assure minimizing the service disruption during the upgrade. All the mentioned principles have to be used in an integrated manner to handle the challenges effectively. For this purpose, we defined a framework which orchestrates the entire process of the upgrade using the principles. To apply the necessary upgrade actions on the infrastructure resources an upgrade engine is required. In this section before presenting our upgrade management framework, we present our evaluation of several configuration management tools as potential upgrade engines.

Puppet [54] is a Ruby [55] based configuration management utility that has two different types of architectures: master/agent and standalone. The configurations of the system are stored in the Manifest and the Catalog. Manifests are the main files containing the Puppet code, and the catalog describes the desired state of each managed node. The agent nodes download the Catalog (which is compiled from the Manifest) from the master node and apply the changes to get to the desired state as specified in the Catalog. In the standalone architecture, the Puppet master applies the changes itself.

Chef [56] is another Ruby based configuration management tool. It is similar to Puppet, it also has the master/agent and standalone architectures. In the master/agent mode, *Cookbook(s)* and *Recipe(s)* are used to tell the Chef Client (agent) how each node has to be configured. Additionally, a Chef installation requires a workstation to control the master. The standalone version of chef is referred to as Chef-solo and it allows for the use of Cookbooks without accessing the Chef Server. In this architecture the Cookbooks need to be located locally on the node.

Salt [57] is based on Python [58]. It uses Python ZeroMQ messaging library for network communication and as a result, it is faster than Puppet or Chef. Similar to Puppet and Chef, Salt can be used in the master/agent or in the standalone mode. In the master/agent architecture, agents (Minions) follow the desired configuration (referred to as States) as provided by the master. There is also a SaltCloud component that can be used to manage Salt Minions in the cloud environment and integrate Salt with cloud providers in a way that Minions can be provisioned and configured.

Ansible [51] is another Python based configuration management tool. Contrary to the other configuration management tools (Puppet, Chef, and Salt), it only uses the standalone architecture and no node agent installation is required. The configurations are defined in Playbook(s), which represents the desired state of the managed instance(s). Ansible has a collection of modules that can be used for management of resources.

The mentioned configuration management tools are all effective in their main goal of applying the configuration and the changes to multiple nodes of the system simply by issuing a single command. Although one can use these configuration management tools to upgrade a system, the coordination mechanism in some of them (especially Puppet and Chef) is limited. Coordination is necessary for maintaining availability during upgrades.

We also examined Mistral [59] and TaskFlow [60] as potential candidates. Mistral is a task management service in OpenStack and it allows for scheduling of any number of tasks. It has a domain specific language based on YAML [61] which allows for the description of *Workflows*, *Actions*, and *Cron-triggers*.

TaskFlow [60] is a Python library for OpenStack and it is used for task execution. Although TaskFlow is not a tool or a service itself, it can be used as a basis for an upgrade engine. It includes an engine which is the core component to run the tasks and it has a mechanism for tracking the actions, tasks and their associated states to correctly track resource modifications. This facility makes possible to resume or revert a task, which can be useful to implement upgrade rollback.

Although Mistral and TaskFlow have many similarities, they are independent projects as they target different use cases. The difference between these two is in the way they decide the execution path of the tasks. Mistral relies on the name of the task, while TaskFlow relies on the dataflow.

After examining each of the configuration management tools, we examined required features of candidate engine for the cloud to see which one of them suits our needs best. An engine needs to have coordination mechanism to control the upgrade by specifying steps and procedures, as well as their order. In order to be able to perform the rolling upgrade, the candidate engine needs to have the capability to indicate the batch size of the upgrade. The other features that potential engine needs to have are: error handling features to indicate alternative procedures, back-up capabilities to be able to roll-back to the old configuration, and tracking features to be able to do undo and redo the upgrade steps. Table 4.1 shows the summary of our evaluation for candidate engines. Among the evaluated potential upgrade engines, Ansible and Salt are the most suitable candidates, since they have powerful orchestration support with a push mechanism, and the bases for designing an upgrade engine. In addition, they are not limited to OpenStack cloud platform, unlike Mistral and TaskFlow. Note that the push mechanism for managing the configuration of resources provides the advantage of taking immediate actions

Table 4.1 Evaluation of candidate upgrade engines

	Coordination	Error handling	Back-up capabilities	Tracking Undo/Redo	Architecture
Puppet	Orchestration features available in Puppet Enterprise version	Error logging, no built-in reactor	Back up file and restore (puppet-filebucket)	Undo/redo code needs to be added	Master/agent and standalone
Chef	No orchestration	Exception (on failed run) and report (on successful run) handlers	Backup and restore	Undo/Redo recipes can be added	Master/agent and standalone
Salt	SaltStack orchestration runner	Built in exceptions, event listeners, and reactors	Backup and restore	Undo/redo can be added	Master/agent and standalone
Ansible	Orchestration	Handlers based on file changes	Backup and restore (Ansible tower)	Undo/redo can be added	Standalone
Mistral	Orchestration	Error handling using on-error clause	Can be added	Keeps the state, and undo/redo can be added	Standalone
TaskFlow	Orchestration	Handles flow failures	Backup and restore tasks can be added	Rollback, retry, and check pointing	Library

on resources, rather than the scheduled-based pull mechanism used by chef and puppet. Note that any other engine capable of running upgrade action on the IaaS resources can be used.

4.4 Upgrade Management Framework

Figure 4.2 depicts our proposed upgrade management framework for IaaS cloud systems, which takes into account the SLA constraints of availability and elasticity. It includes two components, the *Upgrade Coordinator* to coordinate the process of the upgrade, and the *Upgrade Engine* to execute the upgrade actions necessary to deploy in the system the requested upgrade.

The upgrade coordinator keeps track of the upgrade requests and decides about the upgrade process in an iterative manner. For each iteration it generates one or more *Runtime Upgrade Schedule(s)*, each of which is a collection of upgrade actions and the set of resources on which

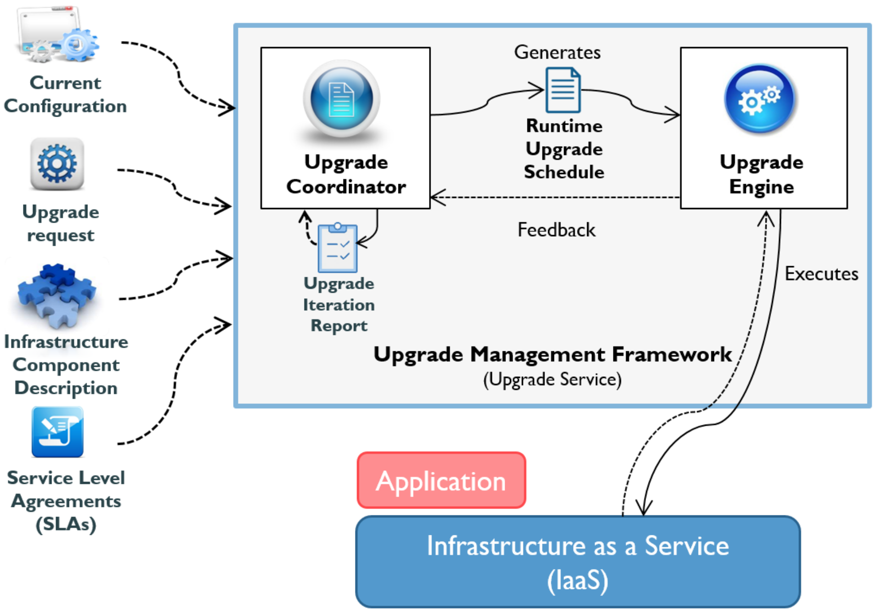


Figure 4.2. Upgrade management framework for IaaS cloud systems

they need to be applied. The upgrade coordinator uses as input the current configuration of the system, the change sets indicated in the upgrade request(s), the infrastructure component descriptions provided by the vendors, and SLAs of the existing tenants as input to generate the schedule. To keep track of the upgrade requests the upgrade coordinator creates an upgrade request model. This model includes the change sets including the complementary changes and their execution status for each upgrade request. Based on the infrastructure component descriptions provided, it infers any complementary changes necessary to satisfy all the dependencies and it identifies all the upgrade actions needed to deploy the different change sets and generates the runtime upgrade schedule(s).

The upgrade engine, an engine capable of running upgrade actions on IaaS resources (e.g. Ansible [51] cloud configuration management tool), executes the upgrade actions specified in the runtime upgrade schedule received from the upgrade coordinator. In section 4.3, we provided our evaluation of different configuration management tools that can be used as upgrade engine in the upgrade management framework. Note that in case of hardware resources the upgrade engine may be limited and may require administrative assistance for actions such as replacement of a piece of hardware. However, it can bring the resources to the required state and signal when the assistance is necessary and on which piece of hardware.

After the execution of an upgrade schedule, the upgrade engine provides a feedback to the upgrade coordinator indicating the results including any failed upgrade action. Based on this feedback, the upgrade coordinator may create a new runtime upgrade schedule to handle the failed upgrade actions at the resource level, i.e. to bring them into a stable configuration. Once all failures are handled for the iteration the upgrade coordinator creates an *Upgrade Iteration Report* as an additional (to those used for the first iteration) input for the next iteration of the runtime upgrade schedule(s) generation. The upgrade iteration report indicates the failed and/or isolated-only resources and failed undo units of the iteration. Based on these, in the subsequent iteration(s) the upgrade coordinator can issue the retry or undo operations as appropriate at the system level considering all the relevant dependencies including those defined by the grouping of requested changes in the upgrade request.

Our proposed upgrade management framework also supports continuous delivery. That is, new upgrade requests may be requested at any time during an ongoing upgrade. The upgrade coordinator takes into account these new upgrade requests, adds them to the upgrade request model, infers the complementary changes as necessary, and extracts the upgrade actions corresponding to the changes. The new requests will be applied to the system in subsequent iterations as applicable.

4.5 Summary

In this chapter, we elaborated how we handle different challenges posed by dependencies and possible incompatibilities along dependencies, by upgrade failures, by the dynamicity of the

IaaS cloud system, and by the amount of used extra resources. As a requirement to automate the upgrade process, we presented our evaluation of several configuration management tools that can be used to apply the upgrade actions in an IaaS cloud system. We also presented an overview of our proposed framework, to manage the whole process of the IaaS cloud upgrade while considering SLA constraints of availability and elasticity. Since this framework automates the entire process of the upgrade, it handles all the aspects of upgrades of the IaaS cloud systems in an integrated manner. It generates the runtime upgrade schedules and executes the upgrade actions indicated in the upgrade schedule to carry out the upgrade requests specified by the administrator in an iterative manner.

In the next chapter, we propose an approach for the coordination of the upgrade process.

Chapter 5

Approach for IaaS Cloud Upgrade

The approach we propose for the upgrade of IaaS cloud systems is used by the upgrade coordinator in our proposed upgrade management framework, to coordinate the upgrade of all kinds of IaaS resources under SLA constraints for availability and elasticity. In this approach, the upgrade requests of the system administrator are handled in accordance with the SLAs with the tenants, the current status of the system, and the infrastructure component descriptions accompanying the products delivered by vendors. The proposed approach identifies the necessary upgrade actions for each IaaS resource to be upgraded and the upgrade methods appropriate for applying those actions in an iterative manner. In case some upgrade actions fail during execution, the recovery operations (i.e. retry and undo) are handled automatically to bring the system to a consistent configuration. This approach is capable to handle new upgrade requests even during an ongoing upgrade, which makes it suitable for continuous delivery.

Note that in our work, similarly to the Software Management Framework (SMF) [52], we classify the upgrade operations into two categories, online and offline. Online operations can be performed without taking the resources out of service and without any impact on availability of the system. In contrast the offline operations require the resources to be taken out of service, and this may impact the availability of the system. Hence, we assume online upgrade operations can be done at any time without being scheduled through our approach, however offline upgrade operations have to be coordinated through our proposed approach.

Before devising our proposed approach applicable to all kinds of IaaS resources, we first defined an initial method [62] for upgrading the IaaS compute (such as the hypervisor, the host OS, or the physical host) while maintaining its availability according to some SLA parameters. In this method we tackled the challenge of dynamicity of the cloud environment by regulating the pace of the upgrade process according to the state of the IaaS cloud system. This method uses the rolling upgrade method with dynamic batch sizes to eliminate the interferences between autoscaling and the upgrade process. The upgrade starts/resumes if and only if resources can be taken out of service and upgraded without jeopardizing their availability even in case of resource failures and requests to scale out within the limit of the SLAs. We generalize this method [62] to handle the upgrade of all kinds of IaaS resources with various dependencies and we defined our final proposed approach accordingly.

In this chapter, we present our approach for the coordination of the upgrade process for all kinds of IaaS resources, after providing the definitions of related concepts. In addition, we prove informally, but in a rigorous manner, four main properties of the proposed approach. This is done through the analysis of the steps and the flowcharts defining the method.

5.1 Definitions

5.1.1 IaaS Cloud System

We view an IaaS cloud system as: a set of physical hosts providing compute services ($M_{compute}$), a set of physical hosts providing virtual storage ($M_{storage}$), a set of physical hosts dedicated to network services ($M_{network}$), another set dedicated to controller services ($M_{controller}$), and a set of other physical resources for networking (e.g. switch, router) and storage (physical storage).

Note that $M_{compute}$ and $M_{storage}$ may intersect. The size of any of these sets may change over time and during the upgrade due to failures and/or the upgrade itself. We assume that all the physical hosts in $M_{compute}$ have a capacity of K VMs. Table A.1 of the Appendix I lists the definitions of all the parameters used in our proposed approach.

The number of tenants may also vary over time including during upgrade. As we apply the changes in an iterative manner, we denote by N_i the number of tenants served by the IaaS cloud at iteration i. A given tenant has a number of VMs which may vary between min_n and max_n. They represent, respectively, the minimum and the maximum number of VMs of the n^{th} tenant that the IaaS provider agreed to provide in the respective SLA. The SLA of each tenant also specifies a scaling adjustment s_n value and a cooldown duration c_n, which represent the maximum size of the adjustment in terms of VMs in one scaling operation to be satisfied by the IaaS provider and the minimum amount of time between two subsequent scaling operations, respectively. These parameters define the SLA elasticity constraints.

As mentioned earlier, we assume that the availability of the applications deployed in the VMs is managed by an availability management solution. The requirements of the application level redundancy are expressed towards the IaaS cloud as VM placement constraints (i.e. as anti-affinity groups), which must be respected during the upgrade. Note that the VMs of each tenant may form several anti-affinity placement groups.

In our work we assume that the IaaS cloud system is configured as highly available system and the availability of VMs is maintained by an orchestration service (e.g. heat service in OpenStack platform) or any VMM which has the capability of bringing up a new VM in its initial state to replace the old one using VM image whenever a VM goes down. Note that since the availability of the application is maintained by an application level HA management solution, the state of the VM is not our focus. More insight can be found in [3].

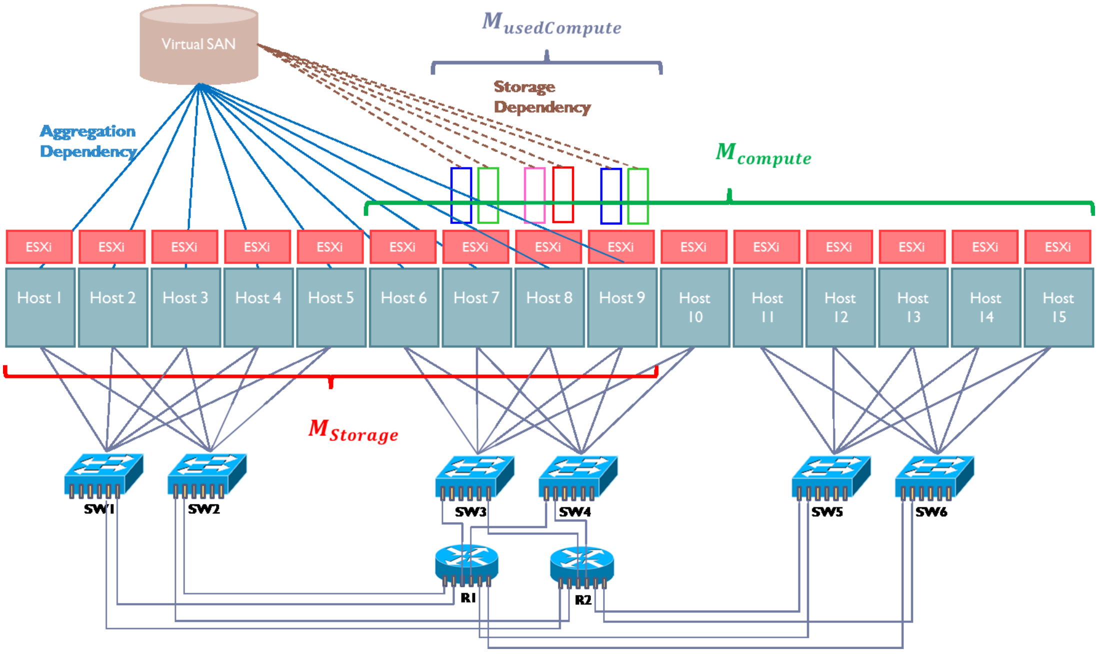


Figure 5.1. An illustrative example

Figure 5.1 shows an example of a system with 15 hosts. Nine of these hosts participate in the creation of a VMware Virtual Storage Area Network (VSAN) [63] – the storage infrastructure supporting VM operations in the system ($|M_{Storage}|$=9) such as migration, while 10 of the hosts provide compute services ($|M_{compute}|$=10). Thus, host 6 through host 9 belong to both sets. In addition to these resources, there are dedicated network resources: switches and routers shown at the bottom of the figure. The example assumes four tenants each with their scaling policy. Note that the controller hosts (VM supporting controllers) are not shown in Figure 5.1.

As mentioned in Chapter 4, an upgrade request is specified as collection of *change sets*, i.e. a set of change sets, to be performed on the resources of the IaaS system. Considering our illustrative example of Figure 5.1, an administrator may want to make two changes: (1) upgrade the virtual shared storage from VSAN to Ceph [50]; and (2) upgrade the networking infrastructure from IPv4 to IPv6. These changes of the virtual shared storage and the networking infrastructure are independent of each other, therefore the administrator separates them into two change sets that compose the upgrade request. For each set, the complementary changes will be inferred automatically from the infrastructure component descriptions provided by the infrastructure vendors. For example, the second change implies the upgrade of all routers, switches and hosts to IPv6. These are added as complementary changes to the second change set.

5.1.2 Resource Upgrade Catalog

To collect all the information necessary for the upgrade of the IaaS cloud system, we define and use a *Resource Upgrade Catalog*. This catalog includes all the infrastructure component descriptions provided by the vendors for all the components already deployed in the system and the products (aka resources) to be added to the system. Whenever an upgrade request referring to a new product (as a target version of a change) is initiated by an administrator, the product and its accompanying infrastructure component descriptions are added to the resource upgrade catalog.

In our illustrative example, the resource upgrade catalog includes the infrastructure component descriptions for VSAN, Ceph, IPv4, and IPv6. Using these infrastructure component descriptions, the scripts for upgrading the virtual shared storage from VSAN to Ceph, as well as upgrading the networking infrastructure from IPv4 to IPv6 can be derived. The same applies also for downgrading the virtual shared storage from Ceph to VSAN and the networking infrastructure from IPv6 to IPv4, should an undo become necessary.

5.1.3 Resource Graph

To coordinate the upgrade process and to create the runtime upgrade schedule(s), one has to be aware of the configuration of the system as well as the status of the ongoing upgrades. For this purpose, we define the *Resource Graph (RG)* which maintains the state of the upgrade process with respect to IaaS resources and their dependencies.

A *RG* is a directed graph *(R, D)*, where *R* is the set of vertices and *D* is the set of edges. The vertices represent the resources in the system (existing or to be added). A vertex (resource) is characterized by the following attributes:

- *Resource/id*: the id of the resource. It is created when a new resource is added to the RG. For existing resources it is collected from the configuration.
- *Resource-kind:* the kind of resource (e.g. compute host, switch, router, etc.) in the infrastructure resource models as described in Chapter 3 (section 3.1).
- *Modification-type*: it indicates whether the resource is to be upgraded, added, or removed by the requested change, or it remains unchanged. It can have one of the following values: "Upgrade", "Add", "Remove", or "No-change". As the upgrade proceeds, the value of this parameter is updated to reflect the first one among the remaining changes to be applied to the resource.
- *Activation-status:* the activation status of the resource may be active (i.e. in service) or deactivated (i.e. out of service).
- *Undo-unit-ids*: the set of undo units the resource belongs to. Since there may be several change sets impacting the same resource, each resource may be associated with several undo units.
- *Actions-to-execute:* is an ordered list of execution-levels where each execution-level is an ordered list of upgrade actions to be executed on the resource. This allows to define two levels of ordering for upgrade actions, within an execution-level and between execution-levels.
- *Number-of-failed-upgrade-attempts*: is the counter of the failed upgrade attempts for the resource per undo unit.

- *Related-resource*: indicates the relation between a new and a current resource, where the new resource is replacing the old one. Note that this parameter is only used to control the process of PPU, where we keep both configurations of a VM supporting infrastructure resource for the time of its upgrade to maintain the continuity of its service. The related resource of the old resource will be the new resource, and vice versa.
- *Is-isolated:* indicates whether the resource is isolated or not.
- *Is-failed:* indicates whether the resource is failed or not.

D is a set of edges, each representing a dependency between resources, either in the current or in the future configuration. The edges can be of different types to capture the different types of dependencies in an IaaS cloud system, as defined in Chapter 3: container/contained dependency, migration dependency, composition dependency, aggregation dependency, communication dependency, controller dependency, storage dependency, VM supporting infrastructure (VM supporting controller or VM supporting storage), and peer dependency between resources.

An edge d_{ij} denotes a dependency of resource R_i on resource Rj, i.e. it is directed from the dependent to the sponsor resource. A symmetrical dependency (peer) is represented by a pair of edges between two resources, i.e. d_{ij} and d_{ji}. Each edge has two main parameters of:

- *Presence:* it indicates whether a dependency exists in the current configuration, in the future configuration, or in both. It is used to properly handle the requirements of existing and future dependencies in the system. It can hold the values of "future", "current", or "current/future".
- *IncompatibilityFactor*: it indicates an incompatibility along the dependency, which needs to be resolved during the upgrade. Note that an incompatibility can only occur

along a dependency with a presence value of “current/future”. It is used to identify the upgrade units. It can hold the values “true” or “false”.

Note that in general we do not upgrade the dependencies, except the ones that realize IaaS resources. The edges representing such dependencies will include additional parameters similar to vertices (e.g. modification-type, actions-to-execute) for managing their upgrade. Since communication dependency realize a physical or virtual link, the edges representing a dependency from this type will have additional parameters representing the upgrade status of the link.

Figure 5.2 shows an example RG reflecting our illustrative example given in Figure 5.1, after the upgrade request was received. In this RG, for example, vertices of R1 to R15 represent the hypervisors running on host1 to host15 represented by vertices R16 to R30. This hosting relation (i.e. container/contained dependency) is represented by the edges between the vertices e.g.

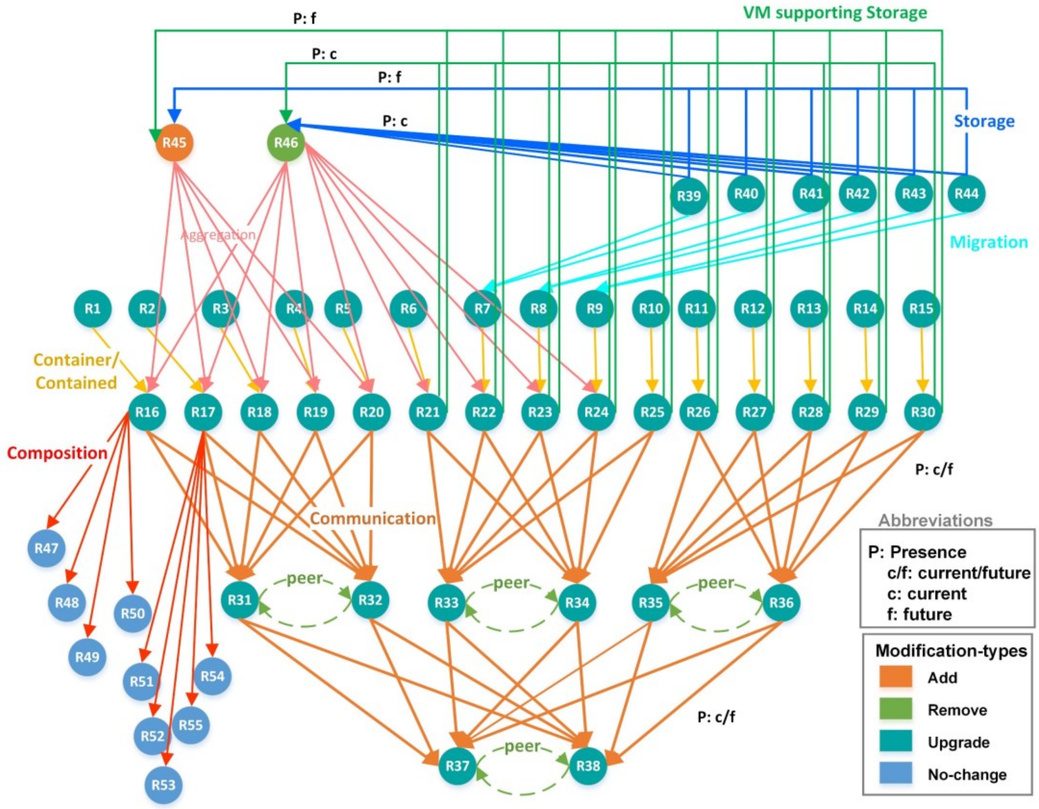


Figure 5.2. Partial resource graph for the illustrative example

R1 and R16. For readability in this graph only part of the configuration of the system and the modification-types for the requested upgrade are represented.

As we mentioned in Chapter 4, a product (e.g. Ceph) delivered by a vendor may be mapped to one or more IaaS resources. In this example, we aim to upgrade the existing VSAN virtual shared storage (represented by R46) to Ceph (represented by R45), which are both compound products delivered and described by their vendors. In the current configuration, storage hosts R16 to R24 are aggregated into the virtual shared storage of R46, while in the future configuration R16 to R20 will be aggregated into R45. R46 serves as a VM supporting storage to the compute hosts R21 to R30 and needs to be replaced by R45. The resources for the current configuration are mapped to the VSAN product and its infrastructure components, while those for the future configuration are mapped to the Ceph product and its components.

Since the virtual shared storage is an infrastructure resource supporting the VM operations, and the VSAN cannot be upgraded to Ceph in place due to incompatibilities, the upgrade coordinator uses the PPU method for the upgrade. As mentioned in Chapter 4, this method applies the parallel universe method locally to a subsystem instead of creating a complete IaaS cloud system as a parallel universe. We use two vertices for representing the resource, one for the old configuration with modification-type of remove (e.g. R46), and one for the new configuration with modification-type of add (e.g. R45). To deploy the Ceph product in our IaaS system the mapping of the IaaS resources is identified based on the requested change, the RG and the requirements indicated in the Ceph component descriptions. The different components of the new Ceph product will be mapped to the storage hosts (represented by R16 to R20), the compute hosts (represented by R21 to R30), and to the new shared storage (represented by R45). After a successful mapping any additional changes required for consistency will be derived and added to the change set. Otherwise, the change set cannot be applied and marked as failed.

5.1.4 Upgrade Methods

As we defined in Chapter 4, an *upgrade unit* identifies a group of resources that have to be upgraded using an appropriate upgrade method to handle the potential incompatibilities during the transition between the current and future configuration (the process of identifying upgrade units will be elaborated in Section 5.2.1). Each upgrade unit may include several resources with different dependencies. According to the types of existing dependencies on which incompatibility issues may arise, a specific upgrade method has to be selected to prevent communication between resources of the incompatible versions. For this purpose, we defined upgrade method templates as follow:

5.1.4.1 *Upgrade Method Templates*

Split mode: we use split mode [31] to avoid incompatibilities along certain dependencies when the resources in an upgrade unit have possible incompatibilities along peer dependency and /or along sponsorship dependency (except communication dependency). In both situations following two conditions have to be valid: 1) there must be no incompatibilities along communication dependency in the whole upgrade unit, and 2) there must be no more than two constituent resources participating in an aggregation dependency in the whole upgrade unit. Otherwise, other upgrade methods have to be used depending on the situations.

As mentioned in Chapter 4, in split mode the resources of an upgrade unit are divided into two partitions which are upgraded one at a time. The order of deactivation and activation of the partitions is orchestrated to avoid introducing incompatibilities, by having only one of the partitions active at any given time until both partitions are upgraded.

Since in our work we aim to upgrade the system with least impact on the availability of the services given by the IaaS cloud system, we try to minimize the impact of the upgrade of resources in an upgrade unit by keeping at least half of the resources of the upgrade unit in service. To account for this, following rules have to be valid for each partition while considering the other partition out of service: 1) the number of in service resources in the partition has to be floor/ceiling of half of the total number of in service resources of the whole upgrade unit, and 2) at least one resource out of each peer resources (direct or indirect) remains in service in the partition. Note that since aggregate resources (i.e. constituents) are considered peer resources, there must be only one aggregate resource in each partition.

For more clarification, let us consider a few examples of resource partitioning for upgrade units with split mode, as shown in Figure 5.3:

a) The upgrade unit includes four peer resources (R1, R2, R3, and R4) with possible incompatibilities along the peer dependencies, as shown in Figure 5.3.a. According to the aforementioned partitioning rules for split mode, each partition will include at least two out of four resources. One possible partitioning for this upgrade unit is to have R1 and R2 in partition 1, and R3 and R4 in partition 2.
b) The upgrade unit includes two peer resources (R7 and R8), with six sponsorship dependent resources (R1, R2, R3, R4, R5, and R6) with possible incompatibilities along all dependencies as shown in Figure 5.3.b. Note that, the sponsorship dependencies are any subcategories of sponsorship dependency except communication dependency. In this example, each partition has to include one of the peer resources of R7 and R8, and floor/ceiling of half of the number of dependent resources (i.e. three dependent resources). Since there is no peer dependencies between dependent resources, different

combination of dependent resources can be in each partition, as long as including floor/ceiling of half of the number of dependent resources.

c) The upgrade unit includes similar resources as of example b, with the difference of having peer dependencies between some of sponsorship dependent resources, as shown in Figure 5.3.c. Here, we have to avoid having the peer resources in the same partition. So, the same partitioning as example b is not valid for this example. One of the possible partitioning will be grouping R7, R1, R3, and R5 into partition 1, and grouping R8, R2, R4, and R6 into partition 2.

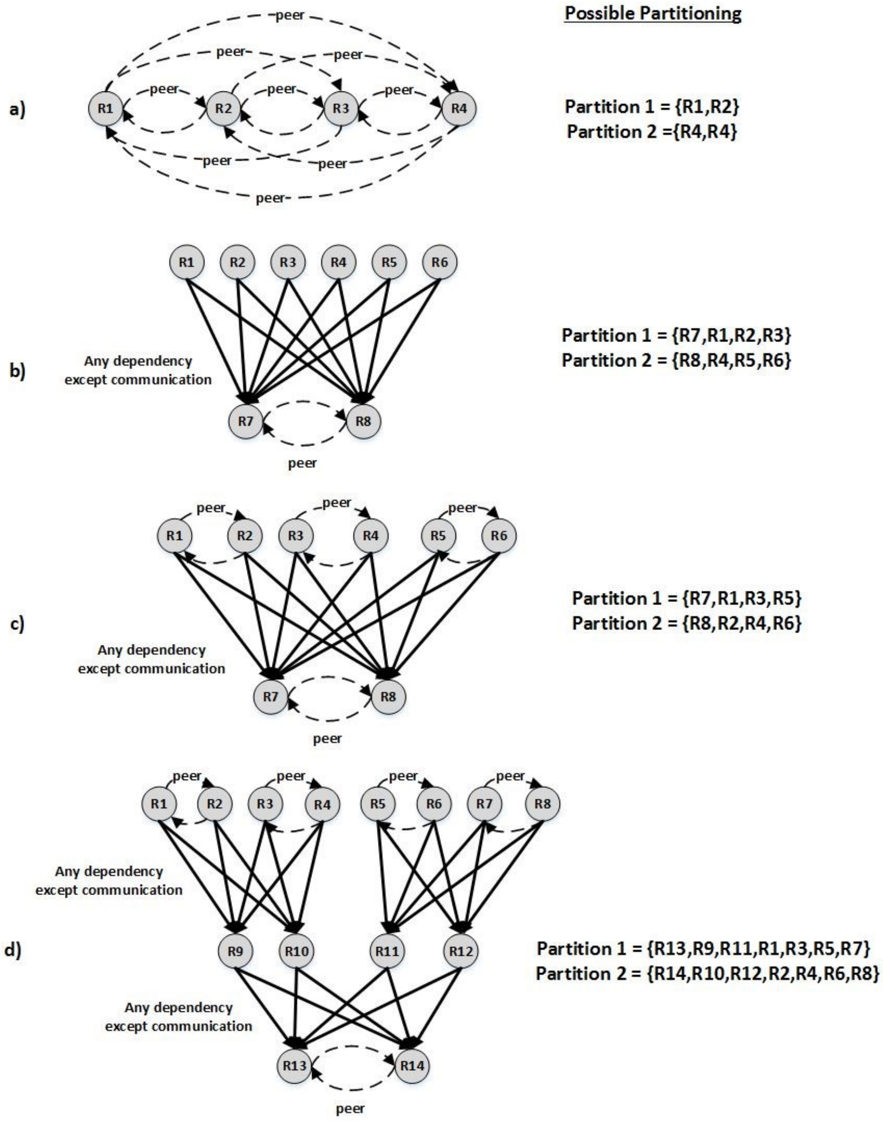


Figure 5.3. Examples of resource partitioning for upgrade units with split mode

d) The upgrade unit includes two levels of sponsorship dependencies (any type except communication dependency) with possible incompatibilities along them, as shown in Figure 5.3.d. To keep at least half of the resources of the upgrade unit in service and to maintain the availability of the services provided by the peer resources, each partition will include one of the most relative sponsor resources (R13 and R14) and half of their direct or indirect dependent resources (R1 to R12), while considering the constraints of peer dependencies between resources.

The steps of the split mode are as follow:

1) Take the first partition out of service (i.e. deactivating) and upgrade it.
2) Take the second partition out of service (i.e. deactivating the second partition) and put back the first partition in service (i.e. activating the first partition). Then, upgrade the second partition, and put them back in service.

Modified split mode: We use this method, when there are resources with possible incompatibilities along communication dependencies in an upgrade unit, and there is no more than two constituent resources participating in an aggregation dependency in the whole upgrade unit. This method implements the split mode upgrade method with some modifications in the partitioning of resources, and activation/deactivation of them.

As mentioned earlier, split mode can be used for handling possible incompatibilities along most sponsorship dependencies, except communication dependencies. When there are incompatibilities along communication dependencies, the application of split mode is problematic. In the partitioning of the split mode, communication dependent resources, as well as others, will be divided between two partitions to keep at least half of the resources of the upgrade unit in service. The problem arises in applying the second step of split mode, when the old version of the communication dependent(s) resources have to be upgraded at the same time as remaining old version communication sponsor(s) in the second partition. The old version communication dependent(s) resources will not be reachable from the sponsor(s) of the new version (due to incompatibilities) and nor from the remaining sponsor(s) with the old versions (due to their presence in the same partition). Indeed, this is caused by the difference of communication dependency and other subcategories of sponsorship dependencies; the communication dependency realizes the physical or virtual link between resources and the dependent resources may lose the connectivity to the network without the sponsor resource.

To resolve the problem while addressing the possible incompatibilities along this type of dependency, we split the second partition (to be upgraded in step 2 of split mode) into two or more partitions depending on the existing levels of communication dependencies (with possible incompatibilities along) in that partition. When there are possible incompatibilities along communication dependency, the communication dependent and sponsor resources have to be in separate partitions. Similar to split mode, at least one resource out of each peer resources have to be in a separate partition. Note that first partition will be the same as first partition in split mode. We do not need to split the first partition, since the communication dependent resources in the first partition are reachable from their communication sponsors of the old version residing in the other partitions during upgrade of the first partition. For more clarification, let us consider a few examples of resource partitioning for upgrade units with modified split mode, as shown in Figure 5.4. In the example upgrade units, we assume there are incompatibilities along the communication dependencies and there are no more than two constituent resources in each upgrade unit; thus the modified split mode have to be used.

a) The upgrade unit includes two peer resources (R7 and R8), with six communication dependent resources (R1, R2, R3, R4, R5, and R6) with possible incompatibilities along all dependencies, as shown in Figure 5.4.a. Since the upgrade unit includes one level of communication dependency, the resources will be divided into three partitions. One of the possible partitioning will be grouping R7, R1, R2, and R3 into partition 1, grouping R4, R5 and R6 into partition 2, and having R8 in partition 3. Note that in partition 1, the communication dependent resources (R1, R2, and R3) can be grouped and upgraded

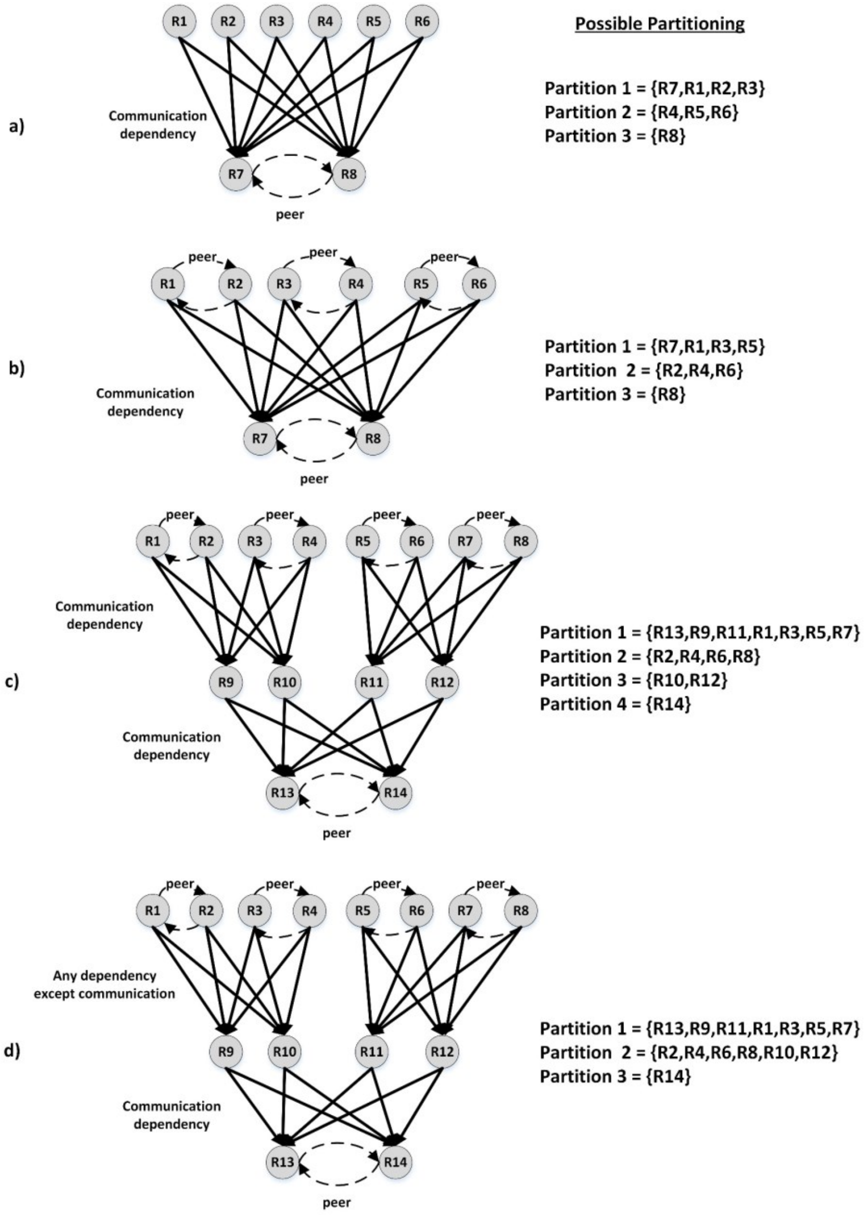


Figure 5.4. Examples of resource partitioning for upgrade units with modified split mode

in the same partition as one of their communication sponsors (R7), since they can be reached through their other communication sponsor (R8) at the time of upgrade.

b) This example is similar to example a, with the difference of having peer dependencies between some of communication dependent resources, as shown in Figure 5.4.b. We have to avoid having the peer resources in the same partition. Thus, the same partitioning as example a will not be valid for this example. One of the possible partitioning will be grouping R7, R1, R3, and R5 into partition 1, grouping R2, R4, and R6 into partition 2, and having R8 in partition 3.

c) The upgrade unit includes two levels of communication dependencies with possible incompatibilities along them, as shown in Figure 5.4.c. Thus, the resources will be divided into four partitions having the communication dependent and sponsor resources in separate partitions, expect for partition 1. Note that we have to take into account the partitioning constraints regarding peer resources. One of the possible partitioning will be as follow: partition 1 including one of the most relative sponsor resources (R13) and half of their direct or indirect dependent resources (R9, R11, R1, R3, R5 and R7), partition 2 including the remaining most relevant communication dependent resources (R2, R4, R6 and R8), partition 3 including the remaining direct communication sponsors of partition 2 (R10 and R12), and partition 4 including the remaining direct communication sponsors of partition 3 (R14).

d) In this example, upgrade unit includes several levels of sponsorship dependencies, as shown in Figure 5.4.d. In contrary to the example c, there is only one level of communication dependency in the upgrade unit, while the other level is any subcategory of sponsorship dependency except communication. Thus, the resources will be divided into three partitions. One of the possible partitioning will be grouping R13, R9, R11, R1, R3, R5 and R7 into partition 1, grouping R2, R4, R6, R10 and R12 into partition 2, and having R14 in partition 3. Note that R2, R4, R6, and R8 can be in the same partition as R10 and R12, since there are no communication dependencies between these two sets of resources. However, R10 and R12 have to be in the separate partition from R14, since communication dependent resources cannot be in the same partition as their communication sponsors, expect for partition 1.

The partitions will be upgraded according to their numbers; the first partition (i.e. partition 1) will be upgraded first and then the partition with most relative communication dependent resources of the old version (i.e. partition 2) will be upgraded next. The upgrade process will continue by upgrading the partition including the communication sponsors of the previous partition, until reaching the last partition including the most relative communication sponsor resources.

In addition to the different resource partitioning in the modified split mode, the prerequisite actions for handling incompatibilities during the upgrade of each partition differs from the split mode. The modified split mode can be applied in two different ways based on the availability of remote link management in the system (i.e. enabling/disabling the link):

- *Modified split mode without remote link management:* when remote management on the communication links is not available, we prevent introducing the incompatibilities by deactivating and activating of the resources of the incompatible versions. After upgrading each partition, the resources of the partitions will remain deactivated, until upgrading the last partition (which includes the remaining most relative communication sponsor resources of the old version). The last partition will be taken out of service while putting back all the previously upgraded partitions in service. However, the deactivation of the last partition is prerequisite for activation of the other partitions. An upgrade unit will have complete outage while applying the modified split mode without remote link management. Thus, additional resources have to be used to compensate the impact of such upgrade.

- *Modified split mode with remote link management:* when remote management on the communication links is available, we prevent the possible incompatibilities during the upgrade of partitions by deactivating or activating the communication links between resources of the incompatible versions. Before upgrading each partition we disable the communication link between the resources being upgraded in the current partition with their communication dependent resources in the other partitions. After upgrading each partition and before putting them back in service, we disable the communication link between the upgraded resources (i.e. new version) of the partition with their communication sponsor resources (i.e. old version) in the other partitions. Subsequently, the communication link of the upgraded resources towards other upgraded partition will be enabled, before enabling the upgraded resources.

Modified split mode with multiple constituent resources: This method is used when we have incompatibilities along peer or sponsorship dependencies, however we are unable to use split mode or modified split mode due to existence of more than two constituent resources participating in an aggregation dependency in the upgrade unit. Since there is restriction to take more than one constituent resource out of service at a time, no more than one constituent resource can stay in the same partition, hence the same partitioning cannot be applied. In modified split mode with multiple constituent resources, we group the resources into partitions similar to modified split mode, exception for the constituent resources. Each constituent resource will be in a separate partition. In other words several partitions in the presence of constituent resources, will correspond to a single partition in modified split mode with similar number of resources without being constituents.

The upgrade order of the partitions will be similar as the corresponding partitions in the modified split mode, while upgrading constituent resource partitions one at a time. Note that depending on the availability of remote link management, the possible incompatibilities will be avoided by either enabling/disabling the resources itself or the communication link between them.

Rolling upgrade: As mentioned in the background, in rolling upgrade the system is partitioned into subsystems and upgraded one at a time while the others provide the services [27]. In our approach we use this method when we do not have incompatibilities. Since we group the resources into upgrade units when there are incompatibilities along dependencies, the resources without possible incompatibilities along their dependencies will be in separate upgrade units. In other words, such an upgrade unit will include a single resource, and have to be upgraded using rolling upgrade method. Note that in a given iteration, depending on the current state of the system and the SLA constraints for availability and elasticity, multiple upgrade units with rolling upgrade method can be selected for the upgrade at the same time.

5.1.4.2 *Required Additional Resources for Upgrade Methods*

All of the upgrade methods handling possible incompatibilities, except the modified split mode with remote link management, prevent the incompatibilities by keeping the resources of each partition deactivated after the upgrade. This result in service degradation or service outage for the upgrade units. The split mode will reduce the service capacity of the upgrade unit to its half, while modified split mode without link management (including modified split mode with multiple constituent resources) will result in the outage of upgrade unit for duration of the upgrade. On one hand side additional resources are required as prerequisite for supporting the upgrade methods handling incompatibilities. On the other hand, the amount of required additional resources has to be minimized to reduce the cost of the upgrade. We assume there are minimum dedicated additional resources in the system to be used for handling incompatibilities. We calculate this minimum number based on the existing upgrade units of the system and considering the amount of service degradation (in terms of compute hosts) while applying the appropriate upgrade methods. To account for this, we identify the upgrade unit with the maximum service degradation in terms of compute hosts, and we consider this amount of compute hosts as the minimum required additional resources dedicated for handling incompatibilities throughout all the upgrades in the system. Thus, the upgrade of some of the upgrade units may be delayed due to limitation of available extra resources.

In the next section, we elaborate the detailed approach for upgrading IaaS cloud system. The flowcharts of our approach are given from Flowchart 5.1 to Flowchart 5.4.

5.2 Detailed IaaS Upgrade Approach

To maintain availability, the IaaS cloud system has to be upgraded using an iterative process. Figure 5.5 illustrates the iterative aspect of our IaaS upgrade approach used in the upgrade coordinator to coordinate the upgrade process. In each iteration, the upgrade coordinator goes through the following four steps to identify the resources to upgrade in the current iteration:

- Step 1- create/update the resource graph,
- Step 2- group the IaaS resources for upgrade,
- Step 3- select the batch of IaaS resources for upgrade, and
- Step 4- select the batch of VMs for migration.

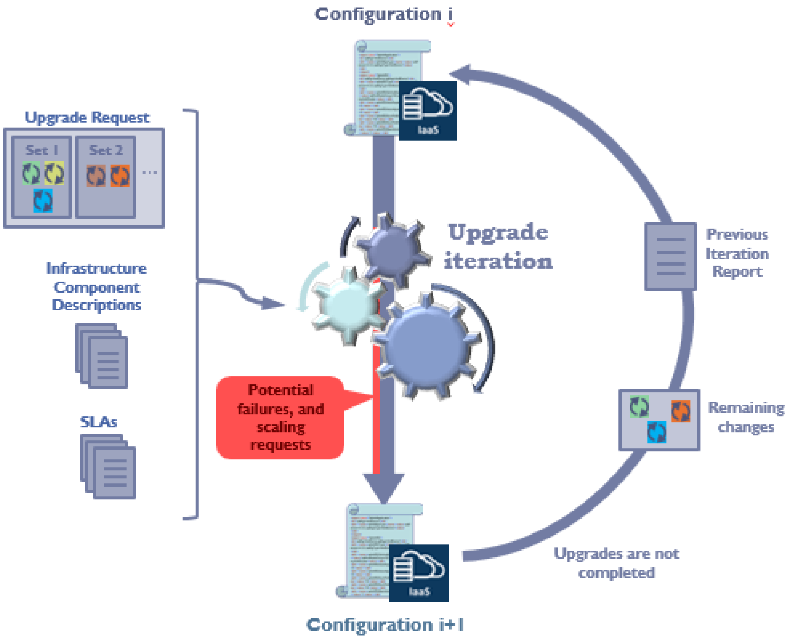

Configuration i
Upgrade Request
Set 1
Set 2
...
Infrastructure Component Descriptions
SLAs
IaaS
Upgrade iteration
Previous Iteration Report
Remaining changes
Potential failures, and scaling requests
Upgrades are not completed
IaaS
Configuration i+1

Figure 5.5. The iterative process of the IaaS upgrade approach

In each iteration, Step 1 identifies the information necessary for the upgrade of the IaaS resources by creating or updating the RG. This graph is created in the initial iteration and then updated in each subsequent one. The inputs for this step in the initial and in the subsequent iterations, while similar, are not the same. In the initial iteration, the RG is created according to the current configuration of the system, the upgrade request, and the infrastructure component descriptions provided by vendors. In the subsequent iterations, the upgrade request model including the state of ongoing upgrade requests and the *upgrade iteration report* indicating the results of the previous iterations are used as additional inputs. Among others the upgrade iteration report indicates the failure of upgrade actions of the previous iteration, as well as the failed and isolated-only resources, based on which undo/retry operations can be initiated as necessary.

As mentioned earlier, the configuration of the system may also change between two subsequent iterations independent of the upgrade process due to live migrations, failures, and scaling operations. Thus, in each iteration the RG is updated based on the current configuration of the system. The RG update also takes into account any new upgrade request and other updates to the upgrade request model.

In Step 2, from the RG the resources that need to be upgraded at the same time are identified based on their dependencies. The vertices of these resources are merged to coarsen the RG into an upgrade *Control Graph (CG)*, where each vertex represents a grouping of one or more resources that need to be upgraded at the same time. The CG is created in the initial iteration and updated in the subsequent ones to reflect the updates of the RG. A vertex of the CG maintains all the information of the vertices of the RG from which it was formed. For example, for the resource groups the actions-to-execute attribute is formed by merging per execution level the actions-to-execute attributes of the resources forming the group. In the subsequent steps the resources that can be upgraded in the current iteration will be selected according to the resource groups of the CG and their dependencies.

In Step 3, first the IaaS resource groups that can be upgraded without violating any of their dependency requirements are selected to form an initial batch. However, because of SLA constraints maybe only a subset of the initial batch can be upgraded resulting in a final batch. Accordingly, a runtime upgrade schedule is generated consisting of the upgrade actions for the final batch. This upgrade schedule is provided to the upgrade engine for execution. After the execution of the upgrade schedule, the upgrade engine provides feedback, including any failed upgrade action, to the upgrade coordinator. Based on this feedback, the upgrade coordinator may create a new runtime upgrade schedule to handle the failed upgrade actions at the resource level, i.e. to bring them into a stable configuration. Once resource level actions are not appropriate or necessary for the given iteration, the upgrade coordinator proceeds to Step 4.

In Step 4 the VMs hosted by the infrastructure are considered. Whenever during upgrade the compute hosts have been partitioned, a batch of VMs may be selected in this step for migration and possibly upgrade. Since the upgrade of both the VM supporting infrastructure resource and the hypervisor affect the compute hosts on which the VMs are hosted, while they are upgraded the IaaS compute hosts are partitioned into an old and a new partitions. If these upgrades do not necessitate VM upgrade, in this step a selected batch of VMs is migrated from the old partition to the new one as appropriate. If VM upgrade is also necessary due to incompatibilities between the versions, then the VMs are also upgraded in the process. The selection of the batch of VMs takes into account the results of Step 3. To respect application level redundancy, we can impact at a time only a limited number of VMs per anti-affinity group (one or as appropriate for the SLA). This means that the selected batch of VMs might need to be upgraded/migrated in sub-iterations. Thus, the upgrade coordinator generates an upgrade schedule for each sub-iteration. As in Step 3, the upgrade coordinator sends each schedule to the upgrade engine for execution and based on feedback received generates the next schedule. If an upgrade action fails, the new upgrade schedule also includes the actions reversing the effects of completed upgrade actions for the failed action. The process continues until all the VMs in the selected batch have been handled. If the compute hosts are not partitioned, this step is skipped all together.

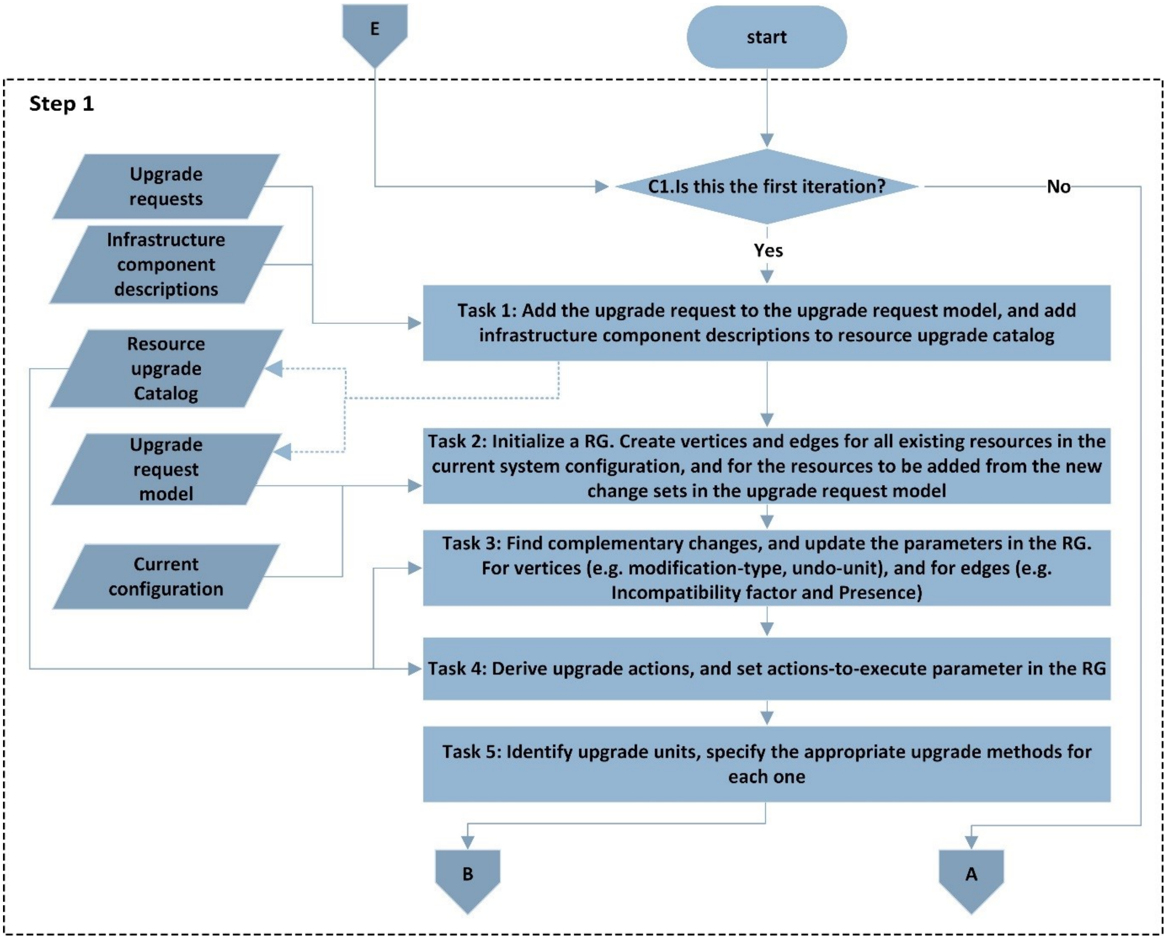
At the end of each iteration the upgrade coordinator updates the upgrade request model, the RG and the CG, and generates the upgrade iteration report to reflect the execution results of all schedules within that iteration. The upgrade iteration report indicates the failed and/or isolated-only resources and failed undo units of the iteration. Based on this report, in the subsequent iteration(s) the upgrade coordinator can issue the retry or undo operations as appropriate at the system level considering all the relevant dependencies including those defined by the grouping of requested changes in the upgrade request.

If new upgrade requests are issued during the iteration, they will be taken into account in subsequent iterations as applicable. The upgrade process terminates when all upgrade requests indicated in the upgrade request model have been handled and no new upgrade request has been received, i.e. all change sets of all the upgrade requests received have been applied successfully or undone unless their target resources failed.

Hereafter, we elaborate more on each of the steps.

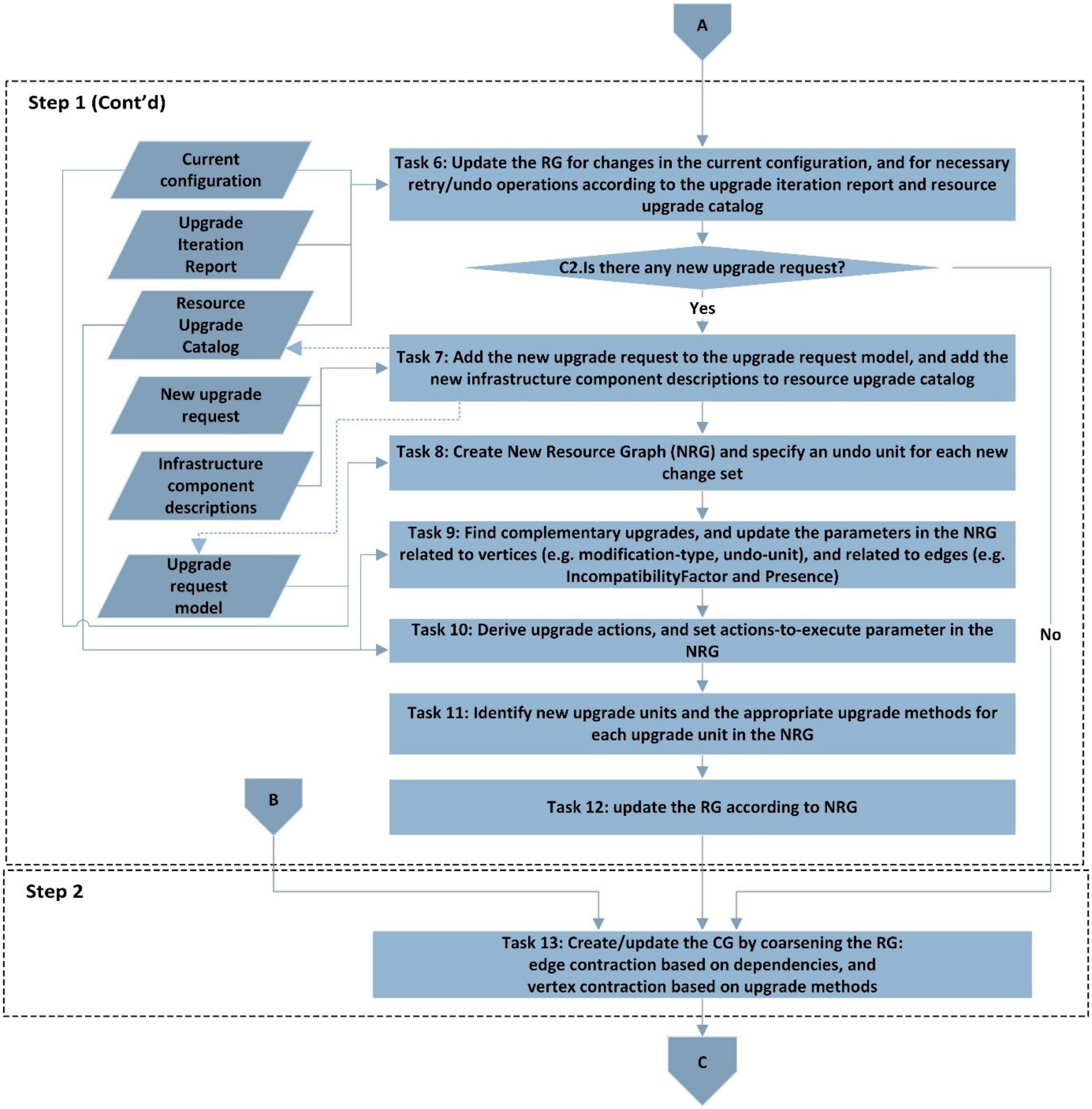
5.2.1 Step 1 - Creating/Updating the Resource Graph

The tasks for creating/updating the RG in this step are indicated from task 1 to 12, within flowcharts given in Flowchart 5.1 and Flowchart 5.2. As we mentioned earlier, the upgrade


E
start
Step 1
Upgrade requests
Infrastructure component descriptions
Resource upgrade Catalog
Upgrade request model
Current configuration
C1.Is this the first iteration?
No
Yes
Task 1: Add the upgrade request to the upgrade request model, and add infrastructure component descriptions to resource upgrade catalog
Task 2: Initialize a RG. Create vertices and edges for all existing resources in the current system configuration, and for the resources to be added from the new change sets in the upgrade request model
Task 3: Find complementary changes, and update the parameters in the RG. For vertices (e.g. modification-type, undo-unit), and for edges (e.g. Incompatibility factor and Presence)
Task 4: Derive upgrade actions, and set actions-to-execute parameter in the RG
Task 5: Identify upgrade units, specify the appropriate upgrade methods for each one
B
A

Flowchart 5.1. Creating the RG in the initial iteration in Step 1 requests received from the administrator are processed and aggregated into the upgrade request model, which is used as input to create and update the RG.

For creating the RG, all existing resources (i.e. vertices) and dependencies (i.e. edges) are extracted from the current configuration of the system. Their parameters are derived from the system configuration (e.g. resource-id) and the upgrade request model (e.g. modification-type). The resources to be added are determined from the change sets in the upgrade request model.


A
Step 1 (Cont'd)
Current configuration
Upgrade Iteration Report
Resource Upgrade Catalog
New upgrade request
Infrastructure component descriptions
Upgrade request model
Task 6: Update the RG for changes in the current configuration, and for necessary retry/undo operations according to the upgrade iteration report and resource upgrade catalog
C2.Is there any new upgrade request?
Yes
No
Task 7: Add the new upgrade request to the upgrade request model, and add the new infrastructure component descriptions to resource upgrade catalog
Task 8: Create New Resource Graph (NRG) and specify an undo unit for each new change set
Task 9: Find complementary upgrades, and update the parameters in the NRG related to vertices (e.g. modification-type, undo-unit), and related to edges (e.g. IncompatibilityFactor and Presence)
Task 10: Derive upgrade actions, and set actions-to-execute parameter in the NRG
Task 11: Identify new upgrade units and the appropriate upgrade methods for each upgrade unit in the NRG
B
Task 12: update the RG according to NRG
Step 2
Task 13: Create/update the CG by coarsening the RG: edge contraction based on dependencies, and vertex contraction based on upgrade methods
C

Flowchart 5.2. Updating the RG in the subsequent iterations in Step 1 and grouping the IaaS resources in Step 2

For them the parameters and dependencies are derived from the upgrade request model and the infrastructure component descriptions provided by the vendor.

For example, whenever the VM supporting infrastructure resources cannot be upgraded in place and we use PPU, in the RG two vertices are created to represent the old and the new configurations of the VM supporting infrastructure. Their modification-type is set respectively to remove and to add. Thus, the old configuration of the VM supporting infrastructure resource(s) will be replaced by the new one as a result of the upgrade.

To satisfy the requirements indicated by the vendors, each change set is verified for completeness and any missing changes are added to the upgrade request model. These are also reflected in the RG. In this process each change set is assigned to a unique undo unit.

The actions-to-execute attribute of each resource is determined using the infrastructure component descriptions kept in the resource upgrade catalog. If the required upgrade actions cannot be applied to a resource in a single iteration due to ordering constraints, the upgrade actions are split into different execution levels to enforce the ordering.

To avoid the communication between resources of incompatible versions during their upgrade, the upgrade of dependent resources with incompatibilities need to be carried out using an upgrade method, which handles appropriately these incompatibilities. For this, we first identify such resources by traversing the RG and then group them into an upgrade unit with which we associate an appropriate upgrade method. We start from several entry points in the RG. These entry points are the leaves that do not have sponsorship dependency towards other resources within RG, and their modification-type is not “No-change”. For each of these entry points, a unique upgrade unit id is assigned. Note that the resources with symmetrical dependencies (peer resources) will belong to the same upgrade unit (with the same upgrade unit id), if there are incompatibilities along symmetrical dependencies or there are incompatibilities along the sponsorship dependencies of the peer resources. We traverse through the sponsorship dependencies to assign upgrade unit id to the remaining resources in the RG. If there is no incompatibilities along sponsorship dependency (IncompatibilityFactor is false), the dependent resource will belong to a new upgrade unit. Otherwise, dependent resource will have the same upgrade unit of its sponsor. Note that, since we handle incompatibilities during the upgrade of VM supporting infrastructure resources (i.e. VM supporting storage and controller) in a global way throughout our approach using PPU method, we exclude VMs and VM supporting infrastructure resources from the upgrade unit assignment process. After identifying the upgrade units, the appropriate upgrade method for each upgrade unit will be selected according to the upgrade method templates describes in Section 5.1.4.1.

To update the RG in a subsequent iteration, first the current configuration of the system is reflected in the RG for any changes that occurred in the system. The upgrade iteration report of the just completed iteration helps in identifying any retry and system level undo operations needed. The RG is updated to include upgrade actions necessary for a retry operation on a resource with a failed upgrade attempt, if the number of failed upgrade attempts is less than the retry thresholds of the related undo unit. Otherwise, the resource is isolated. Whenever, the number of isolated-only and failed resources for an undo unit reaches the undo threshold, all the changes already applied to the resources of the undo unit has to be undone. In addition, the RG is updated to include upgrade actions for an undo operation for any undo unit whose upgrade did not complete within the time limit indicated as max-completion-time. This is measured from the time of the time stamp of the upgrade request with the corresponding change set. These undo units and the associated change sets are also marked as failed.

While updating the RG with respect to an undo operation, the actions-to-execute attributes of all the affected resources (excluding the failed resources) in the failed undo unit are adjusted so that they will be taken to the undo version indicated for the resources. These undo actions are organized into the first execution level of the resources so that they will be executed first. Since there might be upgrade actions associated with other change sets in the actions-to-execute attributes of these resources, which were not completed yet, they need to be adjusted as well. For this, the upgrade actions of other execution levels of the resources are re-evaluated with respect to the potentially new source and target versions as well as the upgrade actions are updated based on the component descriptions in the catalog. Isolated-only resources which are at the undo version are released from isolation, otherwise an attempt is made to take them to the undo version. If this attempt fails, they are marked as failed resources.

As mentioned earlier, new upgrade requests are added to the upgrade request model and then to the RG. New upgrade requests may be targeting resources that are part of pending change requests. Such new upgrade request may also result in new incompatibilities. To identify these, we use a graph similar to the RG: The *New Request Graph (NRG)*. It is created only from the new upgrade requests without considering any ongoing upgrades. We extract from the component descriptions the upgrade actions for the new change sets and organize them into execution levels as required. Next, we identify any newly introduced incompatibility and create the corresponding new upgrade units in the NRG. We use this NRG to update the RG as follows: With respect to the actions-to-execute attributes of resources already in the RG, we create and append a new execution level for each execution level in the NRG. The newly added execution levels are associated with the upgrade units identified in the NRG.

5.2.2 Step 2 - Grouping the IaaS Resources for Upgrade

Some dependency requirements between resources necessitate that they are upgraded at the same time in a single iteration. To facilitate the coordination of the upgrade of these resources, we coarsen the RG, into the CG, as indicated in Flowchart 5.2. In the CG each vertex represents a resource group, i.e. an individual resource or a group of resources of the RG to be upgraded at the same time. Here we provide more details on the creation/update of the CG:

Dependency based edge contraction: During the upgrade of a container its contained resource(s) experience an outage in addition to the outage during their own upgrade. Likewise, during the upgrade of constituent resources, their composite resource experiences an outage. To reduce the outage time, resources with container/contained and resources with composition dependencies should be upgraded at the same time in a single iteration. Thus, we contract the edges representing such dependencies in the RG to merge the vertices representing these resources into a single vertex of the CG. A vertex in the CG, representing a resource group of the RG, will have the same dependencies to other resources as the resources of the merged vertices of the RG except for the container/contained and the composition dependencies. Figure 5.6 shows the CG corresponding to the RG given in Figure 5.2, for the illustrative example in

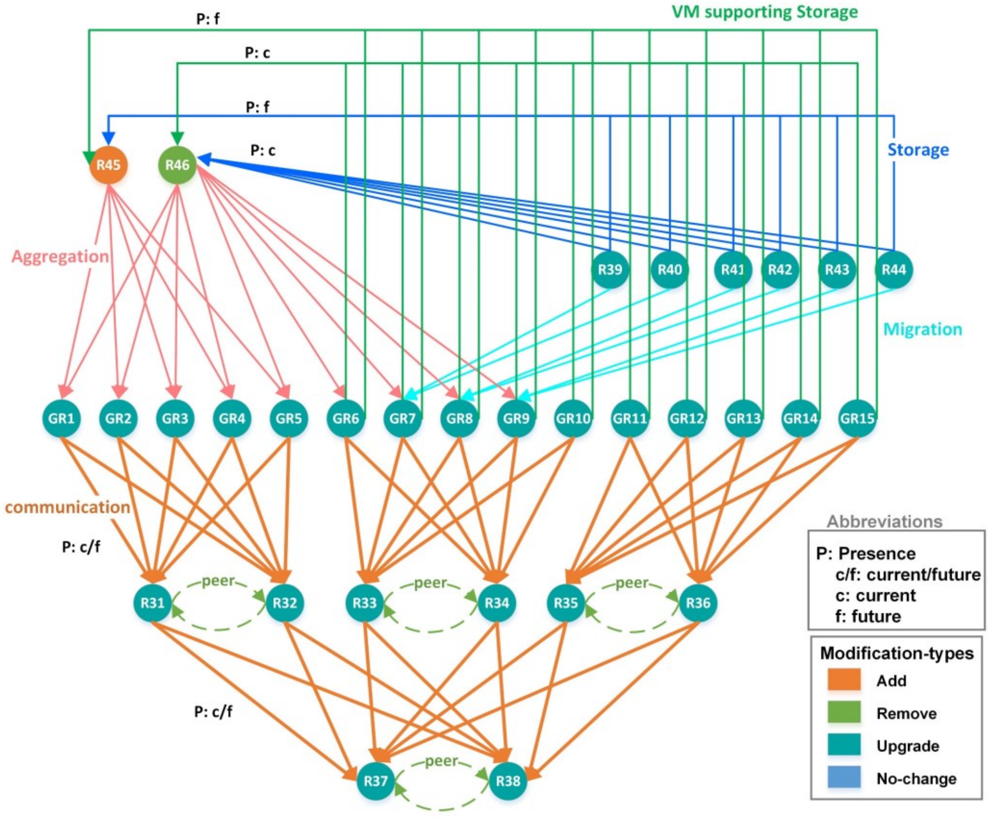


Figure 5.6. Upgrade control graph for the illustrative example

Figure 5.1. An edge contraction of this type was applied to the vertices of the RG representing the resources R1, R16, R47, R48, R49, and R50 to coarsen them into vertex GR1 of the CG. Note that in Figure 5.6, the upgrade related parameters of the CG are not shown.

Upgrade method based vertex contraction: Some upgrade methods avoid incompatibilities by upgrading resources at the same time in a single iteration. We perform vertex contraction for such resources based on the associated upgrade methods of the first execution-level in their actions-to-execute attribute. In case of a vertex contraction, the resulting vertex of the CG will have the union of all dependencies that the resources of the group had in the RG. For example, the vertices representing the resources of an upgrade unit to be upgraded using the split mode upgrade method, will be contracted according to the sub-partitioning of the upgrade unit for the split mode. This allows the proper coordination of the upgrade of the resources without introducing incompatibilities.

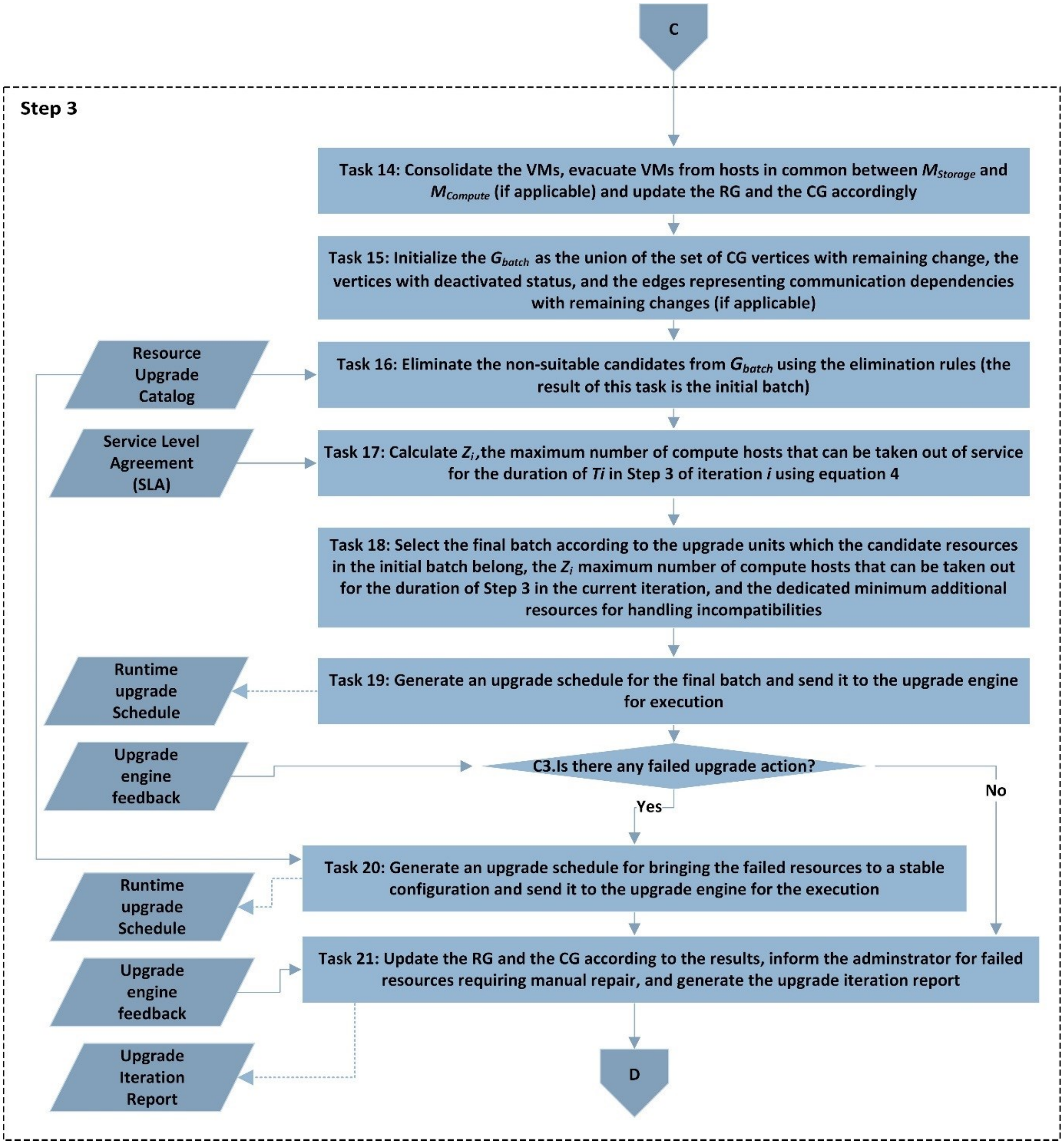
In subsequent iterations, the CG is also updated to maintain consistency with the RG.

5.2.3 Step 3 - Selecting the Batch of IaaS Resources for Upgrade

In this step, the batch of IaaS resources to be upgraded in the current iteration is selected considering both the existing dependencies and the SLA constraints, and applied on the IaaS resources. The tasks for selecting the batch of IaaS resources are indicated from task 14 to 21, within Flowchart 5.3. Since VMs represent the service the IaaS cloud system provides, they are handled separately in Step 4 by considering different criteria.

In this step, first if applicable, the VMs are consolidated on the compute hosts as much as possible to free up some hosts. In particular, if VM supporting infrastructure resources need to be upgraded in an incompatible way, we try to evacuate the VMs from the physical hosts in common between the sets of $M_{storage}$ and $M_{compute}$, to accommodate as much as possible the PPU method. Note that during VM consolidation, we have to respect the availability constraint, inferred from the anti-affinity grouping, by migrating only the allowed number (e.g. one) of VMs at a time from each anti-affinity group. After consolidation, the RG and the CG have to be updated accordingly.

To handle the dependencies during the upgrade, using the CG we need to identify the resource groups that can be upgraded in the current iteration without violating any of their dependencies


C
Step 3
Task 14: Consolidate the VMs, evacuate VMs from hosts in common between $M_{Storage}$ and $M_{Compute}$ (if applicable) and update the RG and the CG accordingly
Task 15: Initialize the G_{batch} as the union of the set of CG vertices with remaining change, the vertices with deactivated status, and the edges representing communication dependencies with remaining changes (if applicable)
Resource Upgrade Catalog
Task 16: Eliminate the non-suitable candidates from G_{batch} using the elimination rules (the result of this task is the initial batch)
Service Level Agreement (SLA)
Task 17: Calculate Z_i,the maximum number of compute hosts that can be taken out of service for the duration of Ti in Step 3 of iteration i using equation 4
Task 18: Select the final batch according to the upgrade units which the candidate resources in the initial batch belong, the Z_i maximum number of compute hosts that can be taken out for the duration of Step 3 in the current iteration, and the dedicated minimum additional resources for handling incompatibilities
Runtime upgrade Schedule
Task 19: Generate an upgrade schedule for the final batch and send it to the upgrade engine for execution
Upgrade engine feedback
C3.Is there any failed upgrade action?
Yes
No
Task 20: Generate an upgrade schedule for bringing the failed resources to a stable configuration and send it to the upgrade engine for the execution
Runtime upgrade Schedule
Upgrade engine feedback
Task 21: Update the RG and the CG according to the results, inform the adminstrator for failed resources requiring manual repair, and generate the upgrade iteration report
Upgrade Iteration Report
D

Flowchart 5.3. Selecting the batch of IaaS resources for upgrade

(G_{batch}). To do so in a systematic way, we first initialize G_{batch} as the union of the set of CG vertices with remaining changes (i.e. modification-type of “Upgrade”, “Add”, “Remove”) and the set of CG vertices with deactivated status (i.e. need to be activated). The G_{batch} will also include the edges representing communication dependencies with remaining changes. Note that as mentioned earlier the communication dependency is realized by link (virtual or physical) resource, which can be upgraded as well.

Next, we eliminate from G_{batch} the vertices, which cannot be upgraded in the current iteration due to some dependencies. To do so we have defined a set of rules, referred to as *elimination rules*. The elimination rules identify the non-suitable candidates in G_{batch} based on the modification-type of the resources, the upgrade method associated with the upgrade unit of the first execution level in the actions-to-execute attribute of the resources, the characteristics of the dependencies of the resources (i.e. incompatibilityFactor and presence), the activation-status of the resources, and the availability of additional resources required as prerequisite for the related upgrades. Note that more than one elimination rule may be applicable to a resource in the G_{batch}, however the resources will be eliminated according to the first applicable rule, as we are applying them sequentially according to their number.

The detailed descriptions of the elimination rules are given in Appendix II. The goal of each elimination rule is as follow:

Elimination rule 1 guarantees keeping the current VM service available by avoiding selection of in-use physical hosts (hosting VMs) and VMs for the upgrade.

Elimination rule 2 guarantees the satisfaction of dependency requirements before removing a resource from the system.

Elimination rule 3 guarantees the satisfaction of dependency requirements before adding a resource to the system.

Elimination rule 4 guarantees the enforcement of compatibility requirements of sponsorship dependencies between resources.

Elimination rule 5 guarantees the correct order of upgrading resources with respect to the upgrade method associated with the upgrade unit of the first execution level in the actions-to-execute attribute of the resources.

Elimination rule 6 guarantees the availability of services provided by peer resources.

***Elimination rule** 7* guarantees the satisfaction of the resource requirements of the *PPU* method used for upgrading a VM supporting infrastructure resource when it cannot be upgraded in place without impacting its services.

As mentioned earlier, the communication dependencies are realized by link resources in the system and they may need to be upgraded as well. Since upgrading a dependency impacts the dependent resource, we evaluate the dependency requirements for the upgrade of communication dependencies (i.e. link resource) as upgrade of its dependent resource. Thus, a communication dependency can stay in the G_{batch} only if its dependent resource can potentially stay in the G_{batch} according to our defined elimination rules, unless in case of having peer link resources.

After applying all of the elimination rules, the vertices remaining in the G_{batch} represent the resource groups that can potentially be upgraded in this iteration (aka initial batch). However, this selection does not consider yet the dynamicity of the IaaS cloud; i.e. SLA violations may still occur if all these resource groups are upgraded in the current iteration. Namely, only a certain number of compute hosts can be taken out of service considering potential failovers and scale-out requests during the iteration. Thus, with these considerations we select a final batch of resource groups from the initial batch.

We estimate the potential scale-out requests in each iteration based on the time required to upgrade and recover from possible failures (by reverting the upgrade) for the initial batch, in which the resources are upgraded in parallel. In each iteration different resources may be upgraded, hence in each iteration we need to consider the resources in the G_{batch} and take the maximum of their required time to upgrade and recover from possible failures (T_i). Note that the required time to upgrade and recover from failures for each resource in the initial batch can be identified based on the estimated time required for installation/removal of the infrastructure component descriptions provided by the vendors and collected in the upgrade resource catalog. Using this the maximum scaling adjustment requests per tenant (S_i) during the upgrade of G_{batch} in iteration i is calculated according to (3).

$$S_i = max(s_n * \left\lceil \frac{T_i}{c_n} \right\rceil) \tag{3}$$

Where s_n is the scaling adjustment per cooldown period c_n of the n^th^ tenant. Since tenants may have different scaling adjustment and cooldown time values, we take the maximum scaling adjustment among them as S_i and by that we handle the worst case scenario. This calculation is valid for a single iteration only and it is recalculated for each iteration since in each iteration different resources may remain in the G_{batch}, and also tenants may be added and/or removed.

We calculate the maximum number of compute hosts that can be taken out of service (Z_i) for the duration of T_i in each iteration using (4).

$$\begin{aligned} Z_i = & \left|M_{computeForOldVM} - M_{usedComputeForOldVM}\right| \\ & -Scaling\ Re\ s\ v_{forOldVM} - Failover\ Re\ s\ ev_{forOldVM} \end{aligned} \tag{4}$$

Where $|M_{computeForOldVM} - M_{usedComputeForOldVM}|$ is the number of compute hosts that are not in use and are eligible to provide compute services for tenants with VMs of the old version (i.e. compatible with the old configuration of VM supporting infrastructure resources or old hypervisor). $FailoverResev_{forOldVM}$ is the number of compute hosts reserved for failover for VMs of the old version. This number is equal to the number of host failures to be tolerated during an iteration (F), when there are VMs of the old version on hosts belonging to $M_{ComputeForOldVM}$ (i.e. $M_{usedComputeForOldVM}$ is not zero); otherwise F will be zero. F can be calculated based on the hosts' failure rate and a probability function as in [64] which estimates the required failover reservations for period T_i. $ScalingResv_{forOldVM}$ is the number of compute hosts for scaling reservation of tenants with VMs of the old version and it is calculated using (5).

$$Scaling\ Re\ s\ v_{forOldVM} = S_i * \left\lceil \frac{A_i}{K} \right\rceil \tag{5}$$

Where A_i indicates the number of tenants with VMs of the old version only and who have not reached their max_n, the maximum number of VMs, therefore may scale out on hosts compatible with the old version of the VMs.

Whenever $M_{usedComputeForOldVM}$, the set of compute hosts in use with the old version is empty, the maximum number of compute hosts that can be taken out of service in the iteration becomes equal to the set of hosts belonging to $M_{computeForOldVM}$.

Note that if there are no incompatibilities related to the upgrade of VM supporting infrastructure resources or hypervisors, the compute hosts of IaaS cloud system are not partitioned into old and new partitions. In this case the above calculations are applied to all compute hosts (as opposed to those hosting old VMs) and all VMs as there is no need to consider the compatibility of VMs and compute hosts. Without incompatible partitions there is no need for Step 4.

To select the final batch of resource groups from the initial batch G_{batch}, we distinguish resource groups that can be returned to service after their upgrade, from those that have to be kept deactivated due to potential incompatibilities. We select the resource groups from the initial batch that can be taken out of service and immediately be returned to the service after their upgrade such that their total number of affected compute hosts for the duration of T_i is not more than Z_i. As mentioned in Section 5.1.4, the resource groups belonging to upgrade units with possible incompatibilities may require to remain deactivated after their upgrade, until they can be safely returned to service without causing incompatibilities. During their upgrade extra additional resources are required to prevent SLA violations. Since we aim to minimize the amount of required additional resources in our approach, we only upgrade limited amount of these resource groups in each iteration according to the minimum additional resources dedicated for handling incompatibilities.

Note that based on the booking strategy of cloud providers, sometimes extra resources might be available in the system which can be taken out of service for longer than the duration of T_i, without impacting the SLA commitments. However, this is not the case when the cloud providers commit to provide more VMs than the actual capacity of the system, referred to as overbooking [65][66]. In this thesis, we assume that the cloud provider's booking strategy is such that the IaaS cloud system at least can carry out all the SLA commitments without considering the upgrade process. In addition, we assume the cloud provider dedicates minimum additional resources for the upgrade techniques handling incompatibilities. As mentioned in Section 5.1.4.2, this minimum required additional resources is calculated according to the maximum degradation (in terms of compute hosts) per upgrade unit in the system. Note that in case of having incompatibilities during the upgrade of VM supporting infrastructure and hypervisors, the failover reservation for tenants with VMs of the new version have to be considered in this calculation as well.

The upgrade coordinator selects such a final batch and generates the corresponding upgrade schedule. This upgrade schedule includes the upgrade actions of the first execution-level of the actions-to-execute attribute of each resource group in G_{batch}. The generated schedule is sent to the upgrade engine for execution. After execution, the upgrade engine sends back to the upgrade coordinator the results.

Note that applying some of the upgrade methods may require additional prerequisite actions. If a resource group in the final batch belongs to an upgrade unit with such an associated upgrade method, the upgrade coordinator includes in the upgrade schedule the prerequisite actions before the upgrade actions of that resource and wrap up actions after them. For example, as prerequisite actions for upgrading some physical hosts in an upgrade unit, the upgrade coordinator might need to include in the upgrade schedule before their upgrade actions to evacuate VMs from those physical hosts. As wrap-up actions it might need to include in the upgrade schedule the actions to bring the VMs back to the upgraded physical hosts.

If the upgrade actions of a resource in the final batch were executed successfully, the first execution-level is removed from its actions-to-execute attribute. The modification-type of the resource is adjusted according to the upgrade actions of the new first execution-level of the actions-to-execute attribute.

For a resource with a failed upgrade action, the counter of failed attempts is incremented, but the actions-to-execute attribute remains unchanged. As mentioned earlier, to bring the resource back to a stable configuration, a new upgrade schedule is created from the undo actions of the completed upgrade actions within the failed attempt to revert their effect. This upgrade schedule is given to the upgrade engine right away for execution. If this operation fails as well, the resource is isolated and marked as a failed. Note that the actions for isolating the resources are indicated as post condition in case of failures in the newly generated upgrade schedule.

Finally, the upgrade request model, the RG and the CG are updated according to the results of this step.

5.2.4 Step 4 - Selecting the Batch of VMs for Migration

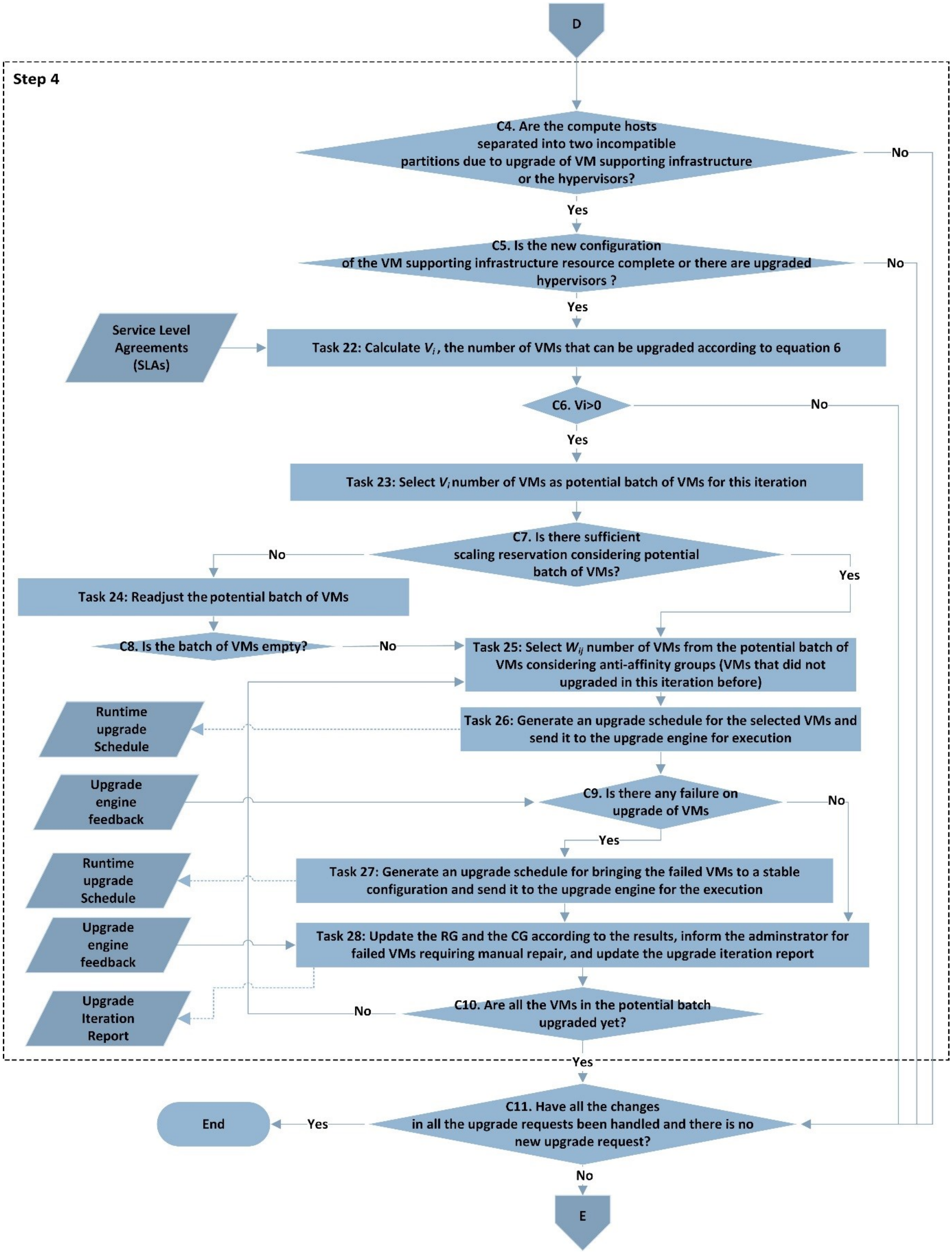
This step is only necessary when the compute hosts are separated into two incompatible partitions due to the upgrade of the VM supporting infrastructure and/or the hypervisors hosting VMs and therefore the VMs need to be migrated (and potentially upgraded) between them. For example, when the PPU method is used to handle the incompatibilities of the VM supporting infrastructure resource.

Before VMs of the old version can be upgraded and migrated to the hosts compatible with the new VM version, the new configuration of the VM supporting infrastructure resource has to be completed. If the new configuration is not ready the VM migration/upgrade is delayed to a subsequent iteration, when it is re-evaluated. In case of incompatibilities due to hypervisor upgrade, this step can be started after a successful upgrade of at least one hypervisor. The tasks for selecting the batch of VMs for migration/upgrade are indicated from task 22 to 28, within Flowchart 5.4.

We calculate the number of VMs (V_i) that can be migrated and if necessary upgraded in the current iteration *i* using equation (6).

$$\begin{aligned} V_i = (&\left|M_{computeForNewVM} - M_{usedComputeForNewVM}\right| \\ &- Scaling\ Re\ s\ v_{forNewVM} - Failover\ Re\ s\ ev_{forNewVM}) * K' \end{aligned} \tag{6}$$

Where $M_{computeForNewVM}$ is the set of hosts that are eligible to provide compute services for tenants with VMs of the new version, $M_{usedComputeForNewVM}$ is the set of in-use hosts that are eligible


D
Step 4
C4. Are the compute hosts separeted into two incompatible partitions due to upgrade of VM supporting infrastructure or the hypervisors?
No
Yes
C5. Is the new configuration of the VM supporting infrastructure resource complete or there are upgraded hypervisors ?
No
Yes
Service Level Agreements (SLAs)
Task 22: Calculate V_i , the number of VMs that can be upgraded according to equation 6
C6. Vi>0
No
Yes
Task 23: Select V_i number of VMs as potential batch of VMs for this iteration
C7. Is there sufficient scaling reservation considering potential batch of VMs?
No
Yes
Task 24: Readjust the potential batch of VMs
C8. Is the batch of VMs empty?
No
Task 25: Select W_{ij} number of VMs from the potential batch of VMs considering anti-affinity groups (VMs that did not upgraded in this iteration before)
Runtime upgrade Schedule
Task 26: Generate an upgrade schedule for the selected VMs and send it to the upgrade engine for execution
Upgrade engine feedback
C9. Is there any failure on upgrade of VMs
No
Yes
Runtime upgrade Schedule
Task 27: Generate an upgrade schedule for bringing the failed VMs to a stable configuration and send it to the upgrade engine for the execution
Upgrade engine feedback
Task 28: Update the RG and the CG according to the results, inform the adminstrator for failed VMs requiring manual repair, and update the upgrade iteration report
Upgrade Iteration Report
No
C10. Are all the VMs in the potential batch upgraded yet?
Yes
End
Yes
C11. Have all the changes in all the upgrade requests been handled and there is no new upgrade request?
No
E

Flowchart 5.4. Selecting the batch of VMs for migration to provide compute services for tenants with VMs of the new version, $FailoverResev_{forNewVM}$ is the number of hosts reserved for any failover for upgraded (new) VMs. $FailoverResev_{forNewVM}$ is calculated similarly to the failover reservation for tenants with VMs of the old version, i.e. F as mentioned in Step 3, but for the period of time required for upgrading V_i number of VMs. Note that since this failover reservation for the new partition is only required for handling incompatibilities during the upgrade of VM supporting infrastructure and/or the hypervisors hosting VMs, it may not be considered by the cloud provider while selling SLAs to the customer. Therefore, additional resources might be required temporarily for accommodating failover reservation for tenants with VMs of the new version. $ScalingResv_{forNewVM}$ is the number of hosts reserved for scaling for the tenants with upgraded (new) VMs, and *K'* is the new host capacity in terms of VMs after the upgrade. Here, $ScalingResv_{forNewVM}$ is calculated similar to (5) for the tenants with VMs of the new version who have not reached their max_n (their maximum number of VMs). They may only scale out on hosts compatible with VMs of the new version. Note that a new scaling adjustment per tenant have to be calculated similar to (3), while considering the time required to migrate/upgrade and if necessary to recover from possible failures for V_i number of VMs potentially through multiple sub-iterations as discussed below.

Considering the application level redundancy, we can typically migrate (and upgrade) only one VM per anti-affinity group at a time. Therefore, we may need to upgrade the V_i VMs in several sub-iterations. Thus, the time required to migrate (and upgrade) and recover from possible failure for V_i number of VMs depends on the number of sub-iterations and the time required for a single VM. In each sub-iteration *j*, one VM is selected from each anti-affinity group with VMs of the old version. The batch of sub-iteration *j* will be W_{ij}. In order to speed up the upgrade process, we use two criteria for selecting the anti-affinity groups and their VMs for the upgrade:

1) To free more hosts, anti-affinity groups from the tenants with the highest number of old version VMs are selected.
2) To minimize the number of VM migrations for VM consolidation, VMs of the hosts that have more VMs from the selected anti-affinity groups are selected.

Note that the VMs from selected anti-affinity groups can belong to tenants that did not have upgraded (new) version VMs yet. The number of such tenants were not considered in the scaling reservation calculation, however after migrating (and upgrading) their VMs, they may be required to scale-out on hosts compatible with VMs of the new version. Thus, before performing the migration (and upgrade) of the selected VMs, $ScalingResv_{forNewVM}$ must be re-evaluated to determine if it is sufficient for scaling-out of such tenants as well. The batch of VMs for each sub-iteration may have to be re-adjusted accordingly. This re-adjustment is based on the number of not-in use hosts (compatible with VMs of the new version), the newly calculated scaling reservation, and failover reservation.

After the upgrade coordinator selects the VMs for the migration/upgrade, a schedule is created per sub-iteration and it is provided to the upgrade engine for execution. After the execution of each sub-iteration, the upgrade engine returns the results to the upgrade coordinator. The actions-to-execute attribute of VMs successfully migrated/upgraded is updated by removing the first execution level. For VMs with failed attempts, the failed attempts counter is incremented and a new schedule is generated to bring them back to a stable configuration. If this operation also fails for a VM it is isolated and marked as failed. If the number of migrated/upgraded VMs is less than V_i VMs and there is possibility of migrating/upgrading more VMs, the upgrade proceeds to the next sub-iteration. Otherwise, the upgrade proceeds to the next iteration.

Whenever in Step 3 the final batch of resources (*G_{batch}*) and in Step 4 the batch of VMs (*V_i*) are both empty for an iteration, the upgrade process stops until there are enough resources available to continue (e.g. freed up through scaling in).

5.3 Informal validation

In this section, we provide an informal validation of four main properties of our approach. In our reasoning, we will refer to the tasks and conditions of the flowcharts of our approach, given in Flowchart 5.1 to Flowchart 5.4.

Property 1) *A given change set in an upgrade request will be applied successfully or will be undone, while keeping the system configuration consistent. Failed resources are isolated to keep the system configuration consistent.*

1. If in an iteration, only some of the target resources for the change set get selected in the final batch in task 18 (or task 25 in case of VMs), one of the following cases may happen:

  1.1. If there are enough resources available for potential scaling out and failover reservation, the upgrade process will continue to the next iteration for applying the necessary upgrade actions on the remaining target resources, according to condition *C.11*, until handling all the changes of the change set.

  1.2. If there are not enough resources available for potential scaling out and failover reservation (i.e. the upgrade process is paused), one of the following cases may happen:

    1.2.a. If the max-completion-time is reached, the change set will be undone.

    1.2.b. If the max-completion-time is not reached yet, either:

      1.2.b.i. The system will eventually scale in and resources will be available to continue to the upgrade. The upgrade proceeds according to case 1.1.

1.2.b.ii. The system will not scale in for a long time and the max-completion-time will reach. The change set will be undone according to case 1.2.a.

2. For the target resource selected in the final batch in task 18 (or task 25 in case of VM). All the upgrade actions corresponding to the change in the change set for the resource will be included in the upgrade schedule and will be sent for execution in task 19 (or task 26 in case of VM). After execution one of the following cases may happen:

2.1. If all the upgrade actions of the change set on the target resource succeed, the RG and the CG will be updated in task 21 (or task 28 in case of VM), to remove the successful upgrade actions from the vertex representing the resource. Thus, the change set on the target resource is applied successfully.

2.2. If some of upgrade actions on the target resource fail, another upgrade schedule will be generated, in task 20 (or task 27 in case of VM), to bring back the resource to a consistent configuration.

2.2.a. If all the upgrade actions for the resource in this new schedule succeed, the failed upgrade attempt will be recorded for the target resource in the upgrade iteration report in task 21 (or task 28 in case of VM). The upgrade will proceed to the next iteration.

2.2.b. While executing this schedule, if these upgrade actions fail on the resources, the resources will be isolated and considered failed. In task 21 (or task 28 in case of VM), they will be recorded in the upgrade iteration report and reported to the administrator requiring manual repair. The upgrade proceeds to the next iteration according to case 1.

3. In task 6 of next iteration, one of the following cases may happen to a target resource:

3.1. In task 6, if the number of failed attempts for the resource does not exceed the maximum retry attempt for the resource, the upgrade actions of the change set will remain in the RG for retry operation on the resource. The upgrade process will proceed until selection of the resource in the final batch in task 18 (or task 25 in case of VM) for retry operation.

3.2. In task 6, if the number of failed attempts for the resource exceed the maximum retry attempt for the resource, the resource will be isolated. If the number of isolated-only resources and failed resources for a change in the change set exceed the undo threshold for that change, all the changes of the change set will be undone. In task 6, for the target resources of the change set, the undo upgrade actions will be included in the RG to take the resources to indicated undo version. They will be executed similar to normal upgrade actions in the next iterations when the resource is selected in task 18 (or task 25 in case of VM), until all are applied (i.e. until condition C.11 holds for the change set). Thus, the change set will be undone.

3.3. In task 6, if the upgrade actions for deploying the change set could not be applied within the max-completion-period specified by the administrator, the change set will be undone.

Therefore, a given change set will be applied successfully or will be undone. Failed resources will be isolated to keep the system configuration consistent.

Property 2) *If there is no new upgrade request, all the previously issued upgrade requests will be completed.*

Note that an upgrade request is considered completed, if its change sets have been either successfully applied or undone. The failed resources will be isolated to keep the system consistent.

1. As we discussed earlier in Property 1, a given change set in an upgrade request will be applied successfully or will be undone.

2. A given upgrade request is a collection of change sets. According to 1, some of the change sets in an upgrade request will be applied successfully and some of them will be undone. Thus, a given upgrade request will be completed, since its change sets have been either applied or undone.

3. Since any upgrade request will eventually be completed according to 2, if there is no new upgrade requests, all the previously issued upgrade requests will be completed.

Property 3) *If the tenants scale out with respect to SLAs, and if the probability function for failure estimation gives accurate results, our approach will respect the SLA constraints of elasticity and availability. Note that the SLA violations caused by VM live migration/consolidation is not considered.*

Precondition:

1) The IaaS cloud system is configured such that it can carry out all the existing SLA commitments and can maintain the availability of the services, without considering the upgrade process.
2) There are minimum additional resources in the system, dedicated for handling incompatibilities during the upgrade. This amount equals to the maximum possible service degradation (in terms of compute hosts) per upgrade unit and failover reservation required for tenants with VMs of the new version.

1. In each iteration, the final batch for the upgrade is selected considering the SLA constraints of elasticity and availability in task 18. Only the resource groups that can be tolerated (i.e. while maintaining the availability and respecting SLAs) to be out of service are selected in the final batch. Different criteria is used for selection of resource groups that can be returned to service after their upgrade, than those that are required to remain deactivated for handling incompatibilities:

1.1. Final batch selection for resource groups that can immediately come back to service after getting upgraded in the same iteration: In task 17, the maximum number of compute hosts that can be taken out of service (*Z_i*), for the duration of maximum required time for upgrading the initial batch, is calculated. For this, the number of compute hosts required for accommodating the potential scaling-out and for potential failovers are considered. Accordingly in task 18, the resource groups are selected such that their total number of affected compute hosts is not more than Z_i. Since the system can accommodate the existing SLA commitments and maintain availability of services without considering the upgrade process according to Pre-condition 1, and since scaling-out reservation and failover reservation are taken into account (in task 17) during the upgrade of resource groups, the availability of the system is maintained and the SLAs are respected during the upgrade.

1.1.a. In task 17, the number of compute hosts required for scaling-out reservation is calculated by multiplying the maximum scaling adjustment requests per tenant and the number of compute hosts to accommodate the scaling out of tenants with old version VMs only that who have not reached their maximum number of VMs yet. Note that scaling reservation for the tenants with new version VMs are considered to be on the compute hosts compatible with the new version VMs and is calculated in task 22. Since we calculate the scaling reservation using SLA parameters for the tenants with old version VMs, as long as these tenants scale out with respect to SLAs, we will not need extra compute hosts for their scaling out reservations.

1.1.b. In task 17, the number of compute hosts required for failover reservation for tenants with old version VMs is calculated according to the number of host failures to be tolerated during an iteration. For this the hosts' failure rate and a probability function which estimates the required failover reservations for meeting the requested level of availability is used. As long as this estimation for failover reservation gives accurate results, we will not need extra compute hosts for failover reservations during an iteration.

1.2. Final batch selection for resource groups that may not come back to service after getting upgraded due to potential incompatibilities (i.e. belong to upgrade units with incompatibilities): According to Pre-condition 2, there are minimum additional resources in the system dedicated for handling incompatibilities during the upgrade. In task 18, we select the resource groups belonging to upgrade units with incompatibilities, if we can compensate their upgrade impact (i.e. affected compute hosts) using the dedicated available additional resources. Otherwise, their upgrade will be postponed to the next iterations. Since we only upgrade portion of these resource groups according to the available additional compute hosts, the availability of service will be maintained and SLAs will be respected during their upgrade.

2. In each iteration, if the compute hosts are separated into two incompatible partitions (due to the upgrade of the VM supporting infrastructure and/or the hypervisors hosting VMs), the batch of VMs for migration and if necessary upgrade, is selected considering the SLA constraints of elasticity and availability in task 23 and 25.

2.1. In task 22, the maximum number of VMs that can be taken out of service (V_i) is calculated, for the duration of the time required to migrate and if necessary upgrade V_i number of VMs. For this, the number of compute hosts required for accommodating the potential scaling-out and for potential failovers of upgraded (new) VMs are considered on the compute hosts compatible with the new version VMs. Accordingly in task 23, the V_i number of VMs are selected as potential batch of VMs. Condition C7 evaluates if the scaling reservation ($ScalingResv_{forNewVM}$) is enough for potential batch of VMs. If not, the batch of VMs are readjusted in task 24. Since scaling-out reservation and failover reservation are taken into account (in task 22, 23, and 24) during the upgrade/migration of VMs, the availability of the system is maintained and the SLAs are respected.

2.1.a. In task 22, the number of compute hosts required for scaling-out reservation for the tenants of new version VMs is calculated by multiplying the maximum scaling adjustment requests per tenant and the number of compute hosts to accommodate the scaling out of tenants with new version VMs. Since we calculate the scaling reservation using SLA parameters for the tenants with new version VMs, as long as these tenants scale out with respect to their SLAs, we will not need extra compute hosts for their scaling out reservations.

2.1.b. In task 22, the number of compute hosts required for failover reservation for tenants with new version VMs is calculated according to the number of host failures to be tolerated during upgrade of V_i number of VMs. For this the hosts' failure rate and a probability function which estimates the required failover reservations for meeting the requested level of availability is used. Since we assume this failover reservation for the tenants of new version VMs is considered in the additional resources dedicated for handling incompatibilities, during the upgrade of VMs we will have enough failover reservations.

2.1.c. The final batch of VMs will be selected and upgraded with respecting anti-affinity groups in task 25. Since anti-affinity groups indicate the availability constraints of applications running on the VMs, and since we migrate and upgrade the VMs in task 25 according to these constraints, the availability of application running on the VMs are respected.

3. In task 14, during the VM consolidation, we only take one VM from an anti-affinity group. Thus, we respect the availability of application running on the VMs.

4. When the upgrade process is paused due to the lack of enough resources for potential scaling out and failover reservation, the tenants can scale out to their maximum number of VMs. Since the system can accommodate the existing SLA commitments without considering the upgrade process (according to Pre-condition 1), and in each iteration the SLAs are respected during the upgrade (according to case 1 and case 2), the SLAs will be respected during the paused time as well.

Therefore, since we consider scaling reservation, failover reservation, and available additional resources dedicated for handling incompatibilities for selecting the final batch of upgrade, if the tenants scale out with respect to their SLAs and if the probability function for failure estimation gives accurate results, our approach will maintain the availability of the service.

Property 4) *We use minimum additional resources during the upgrade***.**

1. Additional resources are required to handle incompatibilities while maintaining availability. We identify subsystems where additional resources are required:

1.1. In task 3, we identify the possible incompatibilities along the dependencies due to the upgrade requests using infrastructure component descriptions in the resource upgrade catalog. We capture this information on the RG as IncompatibilityFactor parameter of the edges representing dependencies between resources. In task 5, we identify the upgrade units indicating group of resources that have to be upgraded using an appropriate upgrade method to handle the potential incompatibilities. Based on the appropriate upgrade method for the upgrade units, the additional resources may be required.

1.2. Since there might be new incompatibilities as a result of new upgrade requests, we identify the incompatibilityFactors of the edges representing dependencies in task 9 and the upgrade units in task 11 for the new upgrade requests on a new graph (aka NRG). In task 12, these information is added to the RG.

1.3. In case of having incompatibilities during the upgrade of VM supporting infrastructure and the hypervisors hosting VMs, we need additional resources for failover reservation of the tenants of new version VMs. We also may need additional resources for maintaining in parallel both the old and the new configurations of the VM supporting infrastructure resource during the application of PPU.

2. We only use additional resources as necessary for supporting upgrade of identified subsystems, and we pace their upgrade process to minimize the amount of required additional resources, as follow:

2.1. Upgrading the resources of upgrade units with incompatibilities, requires additional resources. In task 18, for the final batch the resources that belongs to upgrade units with incompatibilities are selected according to the minimum additional resources dedicated for handling incompatibilities. Since we only upgrade portion of these resources in each iteration, we use less additional resources than the amount required for upgrading all of them.

2.2. In our approach for upgrading the infrastructure services supporting VM operations (e.g. storage, controller), we use PPU method, which applies the partial parallel universe locally to a subsystem (e.g. storage or controller subsystem) instead of creating a complete IaaS system as a parallel universe. For supporting this method while applying the elimination rules in task 16, we try to use available resources as much as possible and request for additional resources only if they are necessary.

5.4 Summary

In this chapter, we presented our approach for the upgrade of IaaS cloud systems under SLA constraints such as availability and elasticity. The approach is used by the upgrade coordinator in our proposed upgrade management framework, introduced in Chapter 4.

In this approach, an upgrade is initiated by an upgrade request which is composed of change sets requested by a system administrator indicating the desired changes in IaaS cloud system. In addition to the initial change sets, the proposed approach takes into consideration the new upgrade requests at the beginning of each iteration. The approach identifies the upgrade actions required to upgrade each IaaS resource, the upgrade method appropriate for each subset of resources, and the batch of resources to upgrade in each iteration. Since in each iteration, the batch of resources to upgrade is selected according to the current state of the system with respect to the dependencies and the SLA constraints, the inference between autoscaling and the upgrade process is mitigated. This is the case as long as the tenants scale out with respect to SLAs and the probability function for failure estimation gives accurate results. In case of upgrade failures, localized retry and undo operations are issued according to the failures and undo/retry thresholds indicated by administrator.

The proposed approach is applicable to the upgrade of IaaS resources of any kind. However as mentioned earlier, it has some limitations when it comes to the upgrade of hardware resources and may require administrative assistance for actions such as replacement of a piece of hardware. For example, the time required for such replacement and availability of administrative assistance have to be taken into account for selection of the batch for hardware resources.

This approach verifies the completeness of the change sets within upgrade requests with respect to the dependencies indicated in the infrastructure component descriptions (provided by the vendors). To infer all detailed/missing changes, we expect the infrastructure component descriptions include all the software and hardware dependencies of the infrastructure components. Otherwise, our approach cannot satisfy the dependency requirements and maintain availability during the upgrade. In addition, in this work we assume that the current and the desired configurations are consistent, meaning there are no incompatibilities between resources in the configurations.

In this chapter, we also analysed and proved informally four important properties of our approach. Although this does not represent a formal proof, but the rigorous analysis give more confidence on the correctness of the proposed approach. In the next chapter, we discuss the proof of concepts developed for demonstrating the feasibility of our proposed approach and the framework.

Chapter 6

Proof of Concepts

In this chapter, we present proof of concepts (PoCs) developed for demonstrating the feasibility of our proposed upgrade management framework and approach. The first PoC is an implementation of our proposed upgrade management framework for the upgrade of IaaS compute. The second PoC is the prototype implementation of the upgrade coordinator, shown in Figure 4.2, realizing our proposed approach for the upgrade of all kinds of IaaS resources. We use an upgrade scenario for each PoC for illustration purposes. We also conduct some experimental evaluation to demonstrate how our approach works to respect SLA constraints of availability and elasticity, compared to the traditional rolling upgrade method.

6.1 Proof of Concept for Upgrade of IaaS Compute and its Application in Real Deployment

This PoC has been implemented for the upgrade of IaaS compute and its application in an OpenStack [18] cluster. A virtualized OpenStack cloud platform is considered as the testbed for this PoC. This virtual cluster is deployed using Vagrant [67] and Ansible [51]. Vagrant is a

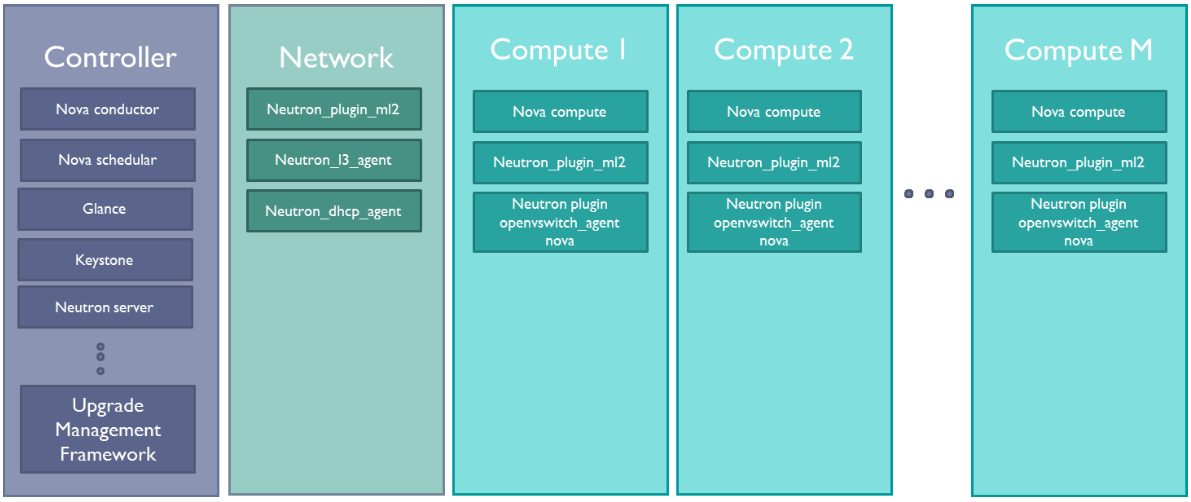


Figure 6.1. Virtualized OpenStack cloud platform in the PoC for IaaS compute

tool for creating a lightweight deployment environment and Ansible is a configuration management tool. In this implementation, Vagrant is used for creating virtual machines (VMs), and Ansible is used to install and configure OpenStack components on those VMs.

In this PoC, Openstack-ansible-galaxy (vagrant-ansible-openstack) [68] repository is reused[3], which contains the code for the deployment of VMs using Vagrant, and Ansible roles for the deployment of the main OpenStack [18] services (Nova, Glance, Horizon, Keystone, and Neutron). In addition to the existing roles in the Openstack-ansible-galaxy, playbooks and roles for deploying other OpenStack services (e.g. Heat, Ceilometer, and Cinder) are added by an intern from Ericsson (Gabriel Hardy) to this PoC. The upgrade management framework is implemented using Go language [69].

Figure 6.1 shows the virtualized OpenStack cloud platform used in this PoC consisting of one controller, one network, and multiple compute nodes. OpenStack components are installed on

[3] The Ansible playbooks of this repository [68] are modified to be compatible with the newer version of Ansible 2.3.

each node. Note that our proposed upgrade management framework is deployed as an upgrade service on the OpenStack controller node in addition to the OpenStack components.

The scope of this PoC is limited to the upgrade of IaaS compute. The orchestrating upgrade of different kinds of resources using upgrade graphs (i.e. RG and CG), and failure cases for IaaS compute upgrades were not considered. However, the required semantics for coordinating the upgrade of IaaS compute (e.g. grouping the upgrade of resources or upgrading not in-use hosts) are considered implicitly in this PoC, without using the graphs.

6.1.1 Architecture of the PoC for Upgrade of IaaS Compute

The overall architecture of this PoC is given in Figure 6.2. The functions in the deployment package is responsible to create the virtual OpenStack cluster, which includes Vagrant Configuration and the Ansible playbooks as shown in Figure 6.3. In this package, Makefile file includes commands to provision the VMs using the configuration indicated in the Vagrantfile file, and to execute the Ansible playbooks for configuring the OpenStack services on those VMs, creating tenants in the OpenStack cluster and deploying the upgrade management framework (upgrade service) code on the controller node.

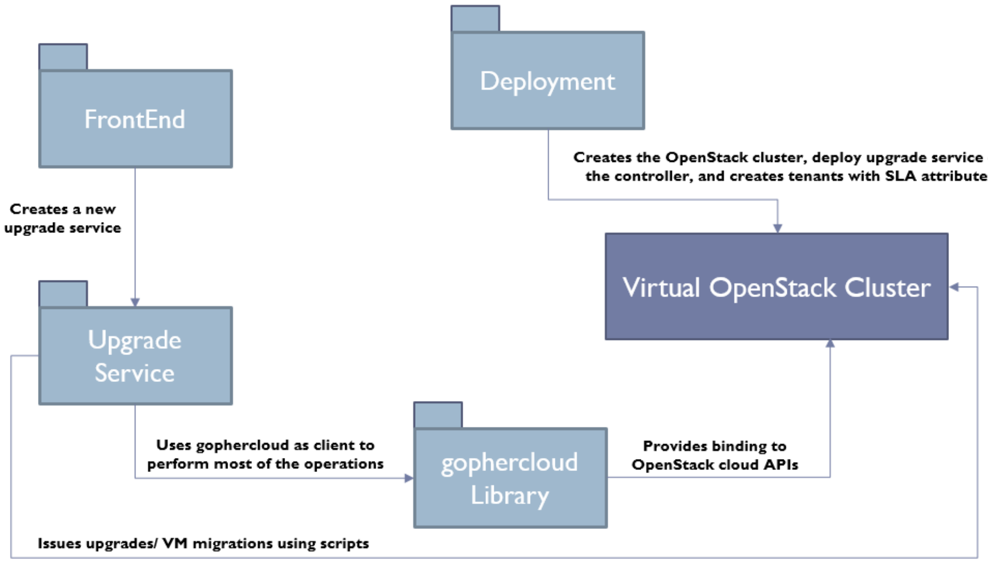

FrontEnd
Deployment
Creates the OpenStack cluster, deploy upgrade service on the controller, and creates tenants with SLA attributes
Creates a new upgrade service
Virtual OpenStack Cluster
Upgrade Service
Uses gophercloud as client to perform most of the operations
gophercloud Library
Provides binding to OpenStack cloud APIs
Issues upgrades/ VM migrations using scripts

Figure 6.2. Overall architecture of the PoC for the upgrade of IaaS compute

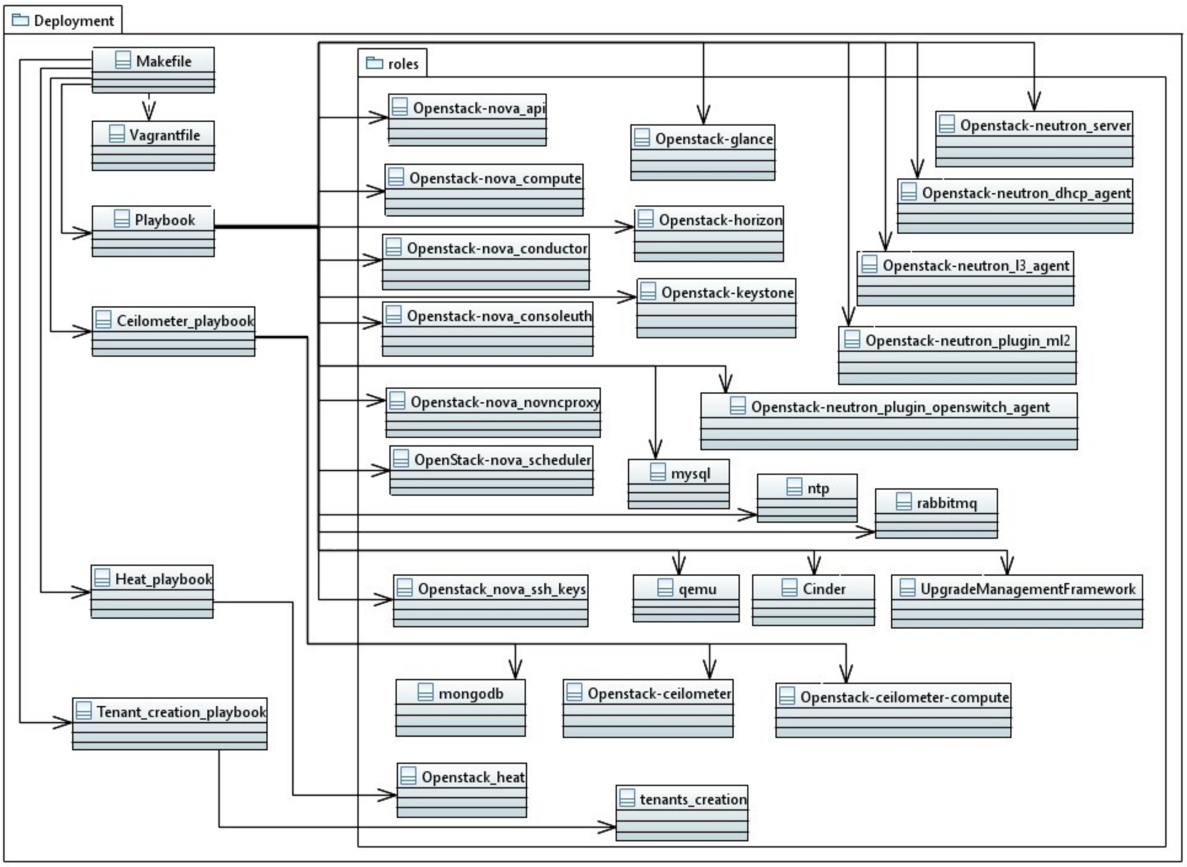


Figure 6.3. Main classes and the Ansible playbooks with related roles in the deployment package

In this architecture, the FrontEnd package is responsible to create and to start the upgrade service . The UpgradeService package is the implementation of subset of our proposed upgrade management framework for the upgrade of IaaS compute. Figure 6.4 shows the main classes and files in the FrontEnd and the UpgradeService packages. Command design pattern is used for implementation of the upgrade service. For the calculation of the maximum scaling adjustment requests per tenant, the batch size for host upgrade, and the batch size for VM upgrade concrete commands are created. Helper class includes all in common operations between the concrete commands.

UpgradeService uses gophercloud [70] Library to have Go binding to OpenStack cloud API. Gophercloud library is an open source Software Development Kit (SDK) which enables Go developer to connect their application written in Go language with OpenStack clouds. In this

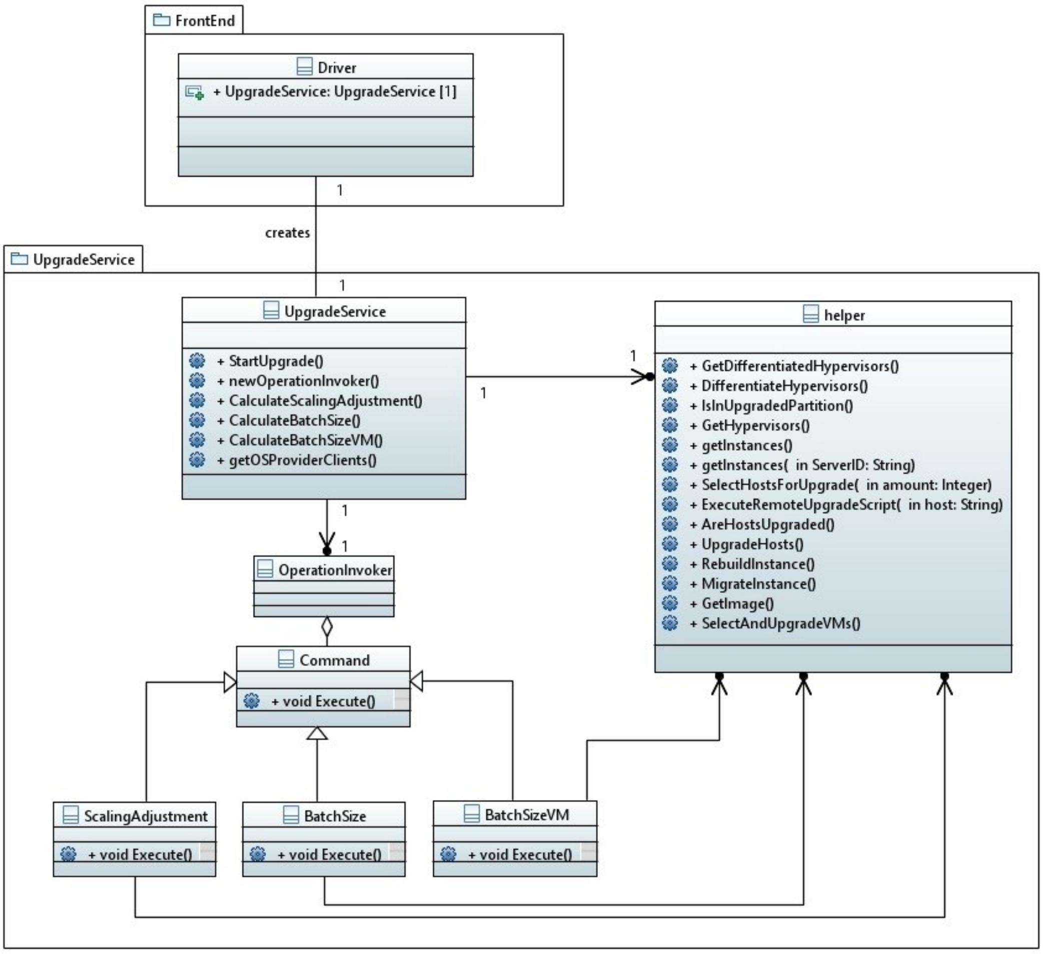


Figure 6.4. Front End and Upgrade Engine packages

PoC, this library is extended to get information about the hypervisor running on the compute nodes. Upgrade service performs most of the operations using this extended gophercloud library. In addition to this library, the upgrade service uses some scripts to perform the upgrades and the VM migrations in the virtual OpenStack cluster.

6.1.2 Illustration Scenario for IaaS Compute Upgrade

As a case study, we aim to upgrade the version of hypervisor (QEMU [71]) and we consider potential incompatibilities during this upgrade. We assume the worst case scenario where the old VMs are incompatible with the new hypervisor and the new VMs are incompatible with the old hypervisor. Thus, live migration of VMs of the old version to upgraded hypervisors is not possible, similarly for VMs of the new version and old version hypervisors. Since the compute hosts will be separated into two incompatible partitions due to the upgrade of the hypervisors hosting VMs, the VMs need to be upgraded between these two partitions. The upgrade of VMs will be by converting their base image before bringing them up on the new version hypervisors.

In our deployment scenario we consider 10 compute nodes ($|M_{compute}|$=10) hosting VMs for four tenants (N_i =4). Each node has capacity for K=3 VMs. For simplicity, we assume that these numbers remain constant throughout the scenario. We also assume that all the VMs of a tenant form a single anti-affinity placement group. In this PoC we are using virtualized OpenStack cluster, thus the compute nodes (i.e. hosts) are virtual servers. Note that due to the limitation of the deployment environment in our lab, we were only able to deploy six compute nodes for our PoC. However, for more clarification we use 10 compute nodes in our deployment scenario.

In the figures used to illustrate the example we use different patterns to show the VMs of different tenants as shown in Figure 6.5. Each of the tenants has an initial, a max_n and a min_n number of VMs. Again for simplicity, we assume that all tenants have the same scaling adjustment and cooldown period configured. The tenants can scale in/out with a scaling adjustment of s_n=1 VM. We consider the time of the upgrade and recover from possible failures of one batch size of host upgrade is equal to cooldown period, which means that S_i is also equal to 1 according to equation (3) in Chapter 5 at Step 3. For simplicity we also assume, the time required to upgrade and recover from possible failures for a selected number of VMs through multiple sub-iterations is equal to cooldown period as well, which means scaling adjustment per tenant is equal to 1 in Step 4 according to Chapter 5.

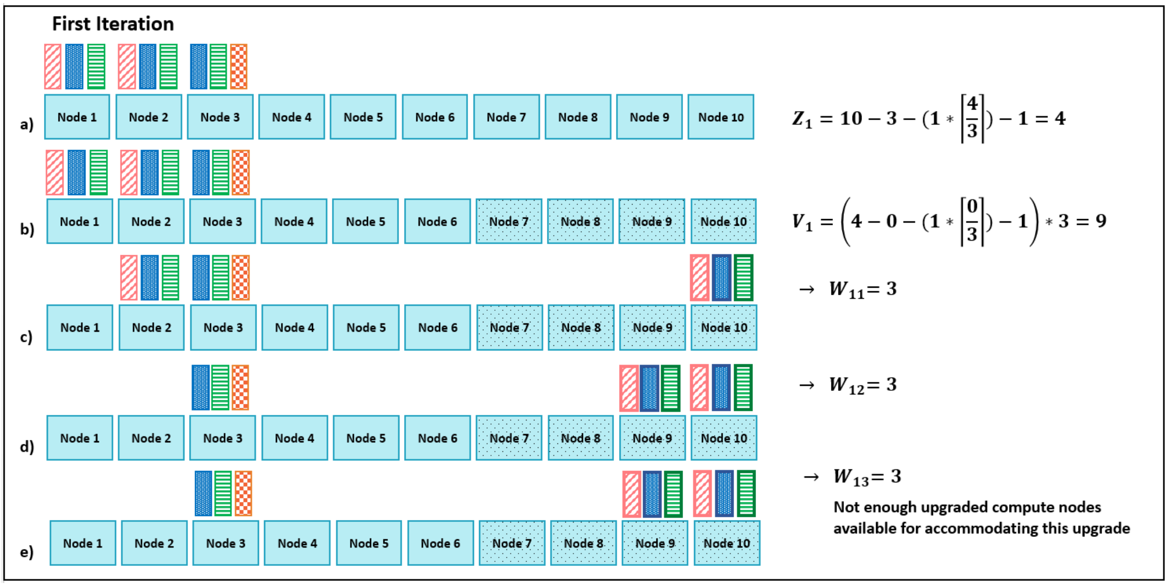
Tenant ID	Tenant	Initial Number of VMs	Min-size	Max-size	New Version
1	🟥	2	2	6	🟥
2	🟦	3	3	7	🟦
3	🟩	3	2	5	🟩
4	🟧	1	1	4	🟧

Figure 6.5. Legend and scaling parameters for the tenants of the example

We assume the scaling adjustment and cooldown period remain unchanged for all tenants during the upgrade, thus S_i is constant in all the iterations. To indicate the scaling in/out operations in the figures, we use down/up arrows at the top of the removed/added VMs, respectively. Note that scaling can happen in both, the old and the upgraded partitions. However, to prevent hindering the upgrade process we consider scaling out reservation for the tenants with VMs of the new version on new version compute nodes, and for the tenants with VMs of the old version only on the old compute nodes.

First Iteration

Figure 6.6 shows the first iteration in the illustration scenario for the upgrade of IaaS compute. The initial number of in-use compute hosts (i.e. nodes) with VMs of the old version



$$Z_1 = 10 - 3 - (1 * \left\lceil \frac{4}{3} \right\rceil) - 1 = 4$$

$$V_1 = \left(4 - 0 - (1 * \left\lceil \frac{0}{3} \right\rceil) - 1\right) * 3 = 9$$

→ $W_{11} = 3$

→ $W_{12} = 3$

→ $W_{13} = 3$

Not enough upgraded compute nodes available for accommodating this upgrade

Figure 6.6. First iteration of the example scenario for IaaS compute

($M_{usedComputeForOldVM}$) is 3. Since none of the tenants have upgraded VMs, for scaling we need to reserve space for all tenants on the old version of the compute nodes (the only partition we have at this point). As shown in Figure 6.6.a, the maximum number of compute hosts that can be taken out of service Z_1 is 4 according to equation (4) in Chapter 5 at Step 3. So, as shown in Figure 6.6.b, we select 4 nodes which are not in use and upgrade their hypervisors. Note that the upgraded nodes are shown with dotted pattern in the figures for the illustration example.

Since the compute nodes are separated into two incompatible partitions due to the upgrade of hypervisors hosting VMs, therefore the VMs need to be migrated and potentially upgraded between them, according to Chapter 5 at Step 4. The number of nodes eligible to provide compute services for tenants with VMs of the new version is now 4 ($M_{computeForNewVM}$ =4). However, the number of in-use nodes that are eligible to provide compute services for tenants with VMs of the new version is still 0 ($M_{usedComputeForNewVM}$ = 0). In this iteration there is no tenant with upgraded VMs yet, therefore the scaling reservation is zero in the initial calculation for the tenants with upgraded (new) VMs. The number of VMs that can be potentially upgraded V_1 is 9. We select all 9 VMs (running on node 1, 2, and 3) as potential batch of VMs for this iteration. At a time we cannot upgrade more than one VM from a single anti-affinity group (i.e. here tenant), hence in the first sub-iteration we select one VM from each anti-affinity group (i.e. here tenant). In the first sub-iteration, initially W_{11} is 4; all the VMs running on "node 1" and one VM from tenant 4 running on "node 3" are selected considering our selection criteria mentioned in Chapter 5. We re-evaluate the selected batch for this sub-iteration, to determine if the previously calculated scaling reservation is enough for scaling-out of four selected tenants. Two compute nodes are required to accommodate the potential scaling out of four tenants, which is more than previously calculated scaling reservation based on current state of the system. Upgrading the selected four VMs in the first sub-iteration is not possible, considering required nodes for potential scaling-out (two nodes), failover (one node) reservations, and the total not in-use nodes (four nodes) eligible to provide compute services for tenants with VMs of the new version. Thus, we re-adjust the batch of sub-iteration by removing the VM of one of the tenants. The batch size for VM upgrade W_{11}, after re-adjustment will be 3 and considering our selection criteria the VM from tenant 4 running on “node 3” will be removed from selected batch. Remaining will be the VMs running on “node 1”. All the VMs of “node 1” will be upgraded and placed on one of the empty upgraded nodes (herein “node 10” as shown in Figure 6.6.c).

Since V_1 is 9, and we only upgraded 3 out of 9 VMs, the possibility of upgrading more VMs will be re-evaluated. Note that after the adjustment of batch for the first sub-iteration, only three VMs from three tenants get upgraded. The number of tenants with upgraded VMs is changed to three. Therefore, the required nodes for potential scaling-out will change to one node. So, the upgrade proceeds to second sub-iteration. In the second sub-iteration the batch size W_{12} is 3 and we select three more VMs each from different anti-affinity groups. Based on our criteria three VMs of “node 2” are selected, as shown in Figure 6.6.c. Again, we have to re-evaluate if we can accommodate their upgrade. Since we have enough nodes to accommodate their upgrade, all the VMs of “node 2” can be upgraded and placed on one of the empty upgraded nodes (herein “node 9” as shown in Figure 6.6.d).

In the third sub-iteration although the batch size of W_{13} is 3, the upgrade of the remaining VMs of “node 3” , shown in Figure 6.6.e, cannot be carried out due to a lack of sufficient nodes for scaling and failover reservation on the upgraded compute nodes during their upgrade. Thus, the step 4 of iteration 1 is completed and the upgrade proceeds to the next iteration.

Second Iteration

In the second iteration, the maximum number of compute hosts that can be taken out of service Z_2 is calculated 3, as shown in Figure 6.7.a. Notice that at this point three out of four tenants have upgraded VMs, therefore we have to consider scaling reservation only for the remaining tenant with no new version VM (i.e. tenant 4) in the old partition. So, 3 not used compute nodes are selected and upgraded in this iteration, as shown in Figure 6.7.b. This changes the number of nodes eligible to provide compute services for tenants with VMs of the new version $M_{computeForNewVM}$ to 7. Assume that at this point we have scaling out request for each of the tenants. The tenants with new version VMs are scaled with new versions on nodes running the new version of the hypervisor, and the tenant with only old version VMs (i.e. tenant 4) is scaled with old version on nodes running the old version of hypervisor. This changes the number of in-use nodes that are eligible to provide compute services for tenants with VMs of the new version $M_{usedComputeForNewVM}$ to 3. Accordingly, the number of VMs that can be potentially upgraded V_2 is 6. Since there are only 4 VMs with old version, all of them are selected as potential batch of VMs. These VMs have to be upgraded in several iterations to respect the anti-affinity

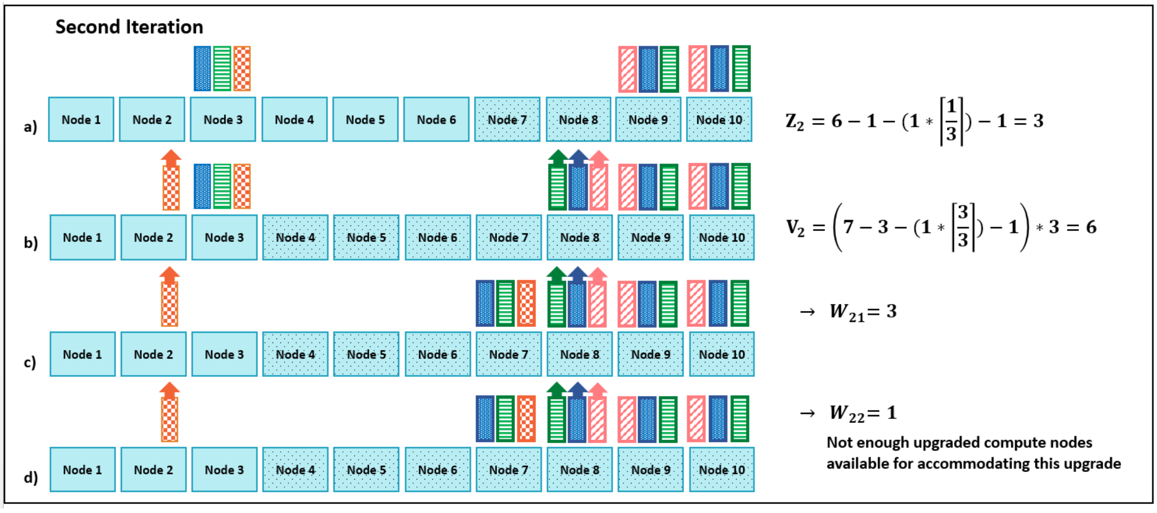

Second Iteration
a)
b)
c)
d)
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
Node 8
Node 9
Node 10
$Z_2 = 6 - 1 - (1 * \left\lceil \frac{1}{3} \right\rceil) - 1 = 3$
$V_2 = \left(7 - 3 - (1 * \left\lceil \frac{3}{3} \right\rceil) - 1\right) * 3 = 6$
$\rightarrow W_{21} = 3$
$\rightarrow W_{22} = 1$
Not enough upgraded compute nodes available for accommodating this upgrade

Figure 6.7. Second iteration of the example scenario for IaaS compute

group constraints. Similar to the previous iteration, the batch for sub-iterations have to be re-evaluated according to the required scaling reservations.

In the first sub-iteration the batch size for VM upgrade W_{21} is 3, and based on our two selection criteria, “node 3” is selected. All the VMs of “node 3” can be upgraded and placed on one of the empty upgraded nodes (herein “node 7” as shown in Figure 6.7.c). In the second sub-iteration although the batch size for VM upgrade W_{22} is 1, the upgrade of the remaining VM of “node2”, shown in Figure 6.7.d, cannot be carried out. This is because, two nodes are required for potential scaling-out reservation and one node is required for failover reservation for tenants with VMs of the new version. Since there are only three not in-use nodes eligible to provide compute services for tenants with VMs of the new version, there are not enough upgraded compute nodes to support the upgrade of remaining VM. Thus, the upgrade proceeds to the next iteration.

Third Iteration

In the next iteration as it is shown in Figure 6.8, the maximum number of compute hosts that can be taken out of service Z3 is calculated 1. So, 1 not in use compute node (i.e. “node 3”) is selected and upgraded in this iteration. This changes the number of nodes eligible to provide compute services for tenants with VMs of the new version $M_{computeForNewVM}$ to 8. Let us assume

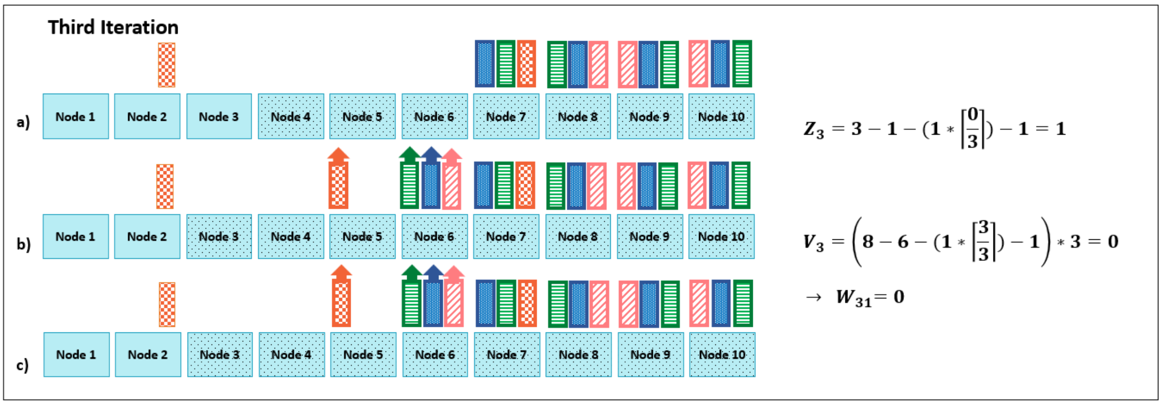


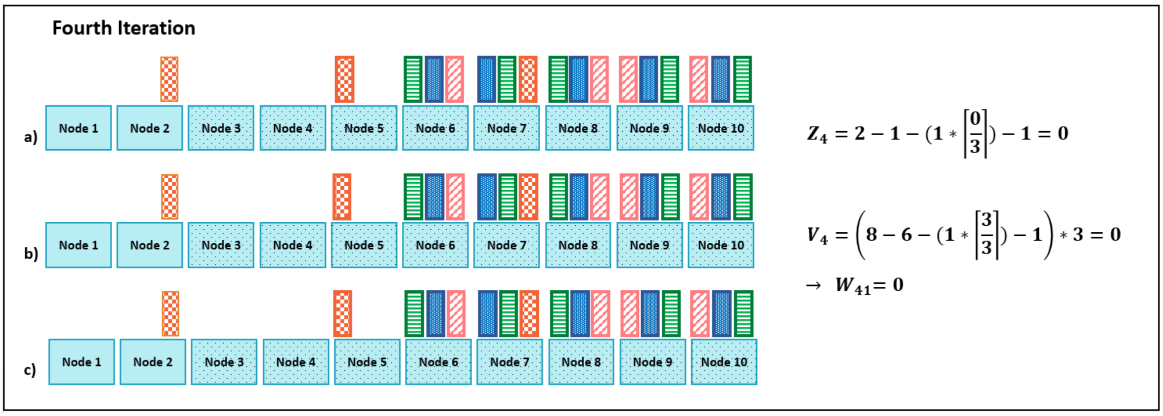
Figure 6.8. Third iteration of the example scenario for IaaS compute

at this point we have scaling out request for each of the tenants. Since, all the tenants have new versions of VMs, this time they are scaled with new versions on nodes running the new version of the hypervisor. This changes the number of in-use nodes that are eligible to provide compute services for tenants with VMs of the new version $M_{usedComputeForNewVM}$ to 6. Accordingly, the number of VMs that can be potentially upgraded V_3 is zero, as shown in Figure 6.8.b. Notice that the number of tenants with VMs of the new version who have not reached their maximum number of VMs and can potentially scale out on upgraded compute nodes is 3. The tenant 3 (with green pattern VMs) is already reached its maximum number of VMs, which is 5. Thus, we do not need to consider scaling reservation for this tenant (i.e. tenant 3).

Fourth Iteration

In the Fourth iteration as it is shown in Figure 6.9, the calculations of the maximum number of compute hosts that can be taken out of service Z_4 and the number of VMs that can be potentially upgraded V_4 are both zero. The upgrade process stops here until scaling in requests free up enough hosts (physical resources) to continue.

Note that the tenants may scale out to their maximum number of VMs while the upgrade process is paused, as shown in Figure 6.10. Since we assume the IaaS cloud system is configured such that it can carry out all the existing SLA commitments and can maintain the availability of the services without considering the upgrade process, all the tenants can safely scale out



$$Z_4 = 2 - 1 - (1 * \left\lceil \frac{0}{3} \right\rceil) - 1 = 0$$

$$V_4 = \left(8 - 6 - (1 * \left\lceil \frac{3}{3} \right\rceil) - 1\right) * 3 = 0$$

$$\rightarrow \ W_{41} = 0$$

Figure 6.9. Fourth iteration of the example scenario for IaaS compute

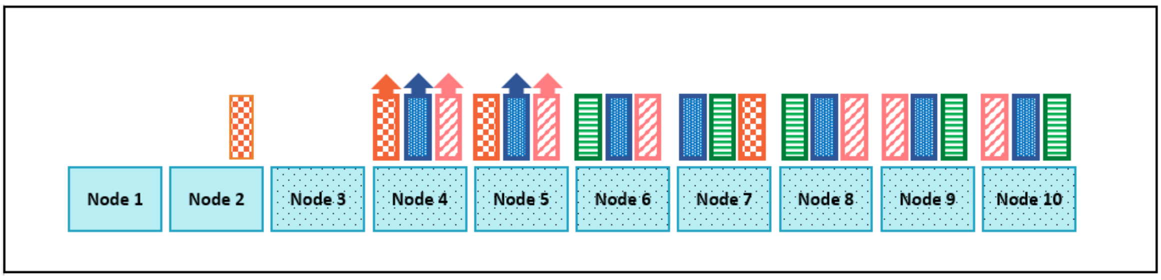


Figure 6.10. Maximum scaling out of all tenants in the example scenario during paused upgrade process

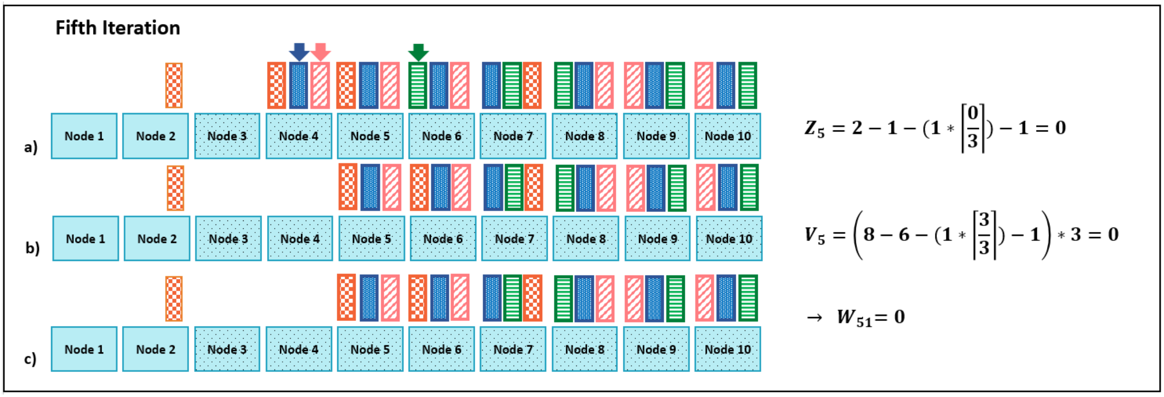

Fifth Iteration
a)
b)
c)
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
Node 8
Node 9
Node 10
$Z_5 = 2 - 1 - (1 * \left\lceil \frac{0}{3} \right\rceil) - 1 = 0$
$V_5 = \left(8 - 6 - (1 * \left\lceil \frac{3}{3} \right\rceil) - 1\right) * 3 = 0$
$\rightarrow W_{51} = 0$

Figure 6.11. Fifth iteration of the example scenario for IaaS compute

during this pause time. Note that in this example, we assumed one out of ten compute nodes is considered for the failover reservation required for tenants with VMs of the new version.

Fifth Iteration

Now assume that scaling in requests arrive from three tenants (i.e. tenant 1, tenant 2, and tenant 3). Thus, in the fifth iteration we examine to determine if the upgrade process can resume (Figure 6.11.a). Since the tenants requesting a scaling in operation do not have any old version VMs, the scaling in will remove VMs of the new version for each of the requesting tenants, as shown in Figure 6.11.a. Note that if a tenant requesting a scaling in operation has any old version VM, for that specific tenant the scaling in removes an old version VM. As a result of this scaling in operation, “node 4” will be freed up after VM consolidation. The calculations for determining the batch sizes are recalculated. However, both result in zero again and the upgrade process cannot continue. Notice that this is due to required scaling out reservation for those tenants which scaled in, so they may need to scale out to their maximum number of VMs.

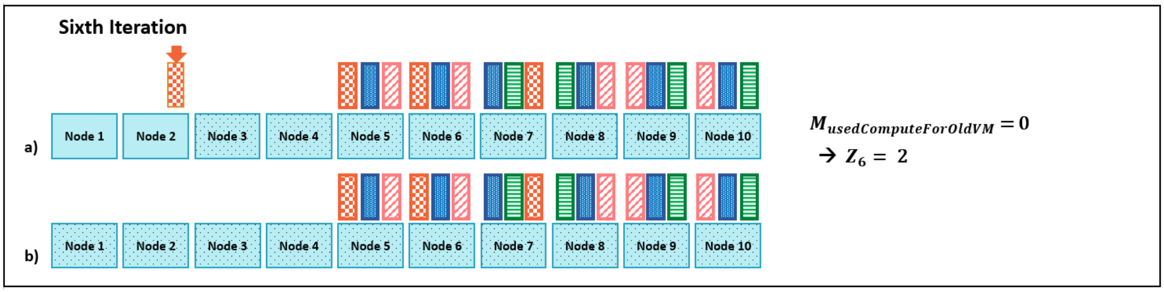


Figure 6.12. Sixth iteration of the example scenario for IaaS compute

Sixth Iteration

Assume we get one more scaling in request from one of the tenants (i.e. tenant 4) and this frees up one of the hosts remaining with the old version as shown in Figure 6.12. This triggers the sixth iteration and since number of in-use compute nodes with VMs of the old version becomes zero, the maximum number of compute hosts that can be taken out of service Z_6 becomes equal to the set of compute nodes with the old version ($M_{computeForOldVM}$), which is 2. Therefore, the upgrade process resumes and all remaining hosts are upgraded and finally the upgrade process completes, as shown in Figure 6.12.

6.1.3 Experimental Evaluation

We performed experiments to demonstrate how our approach works to respect SLA constraints of availability and elasticity, compared to the traditional rolling upgrade method with different fixed batch sizes. In our evaluation scenario we considered 10 compute hosts hosting VMs for four tenants, as presented in Section 6.1.2. We evaluated two different case studies when the tenants; a) have their initial number of VMs as shown in Figure 6.13.a, and b) after some scaling in/out as shown in Figure 6.13.b. The scaling parameters for both cases are indicated in Figure 6.5. In both cases we are assuming the VMs are consolidated. As there is a challenge of incompatibilities associated with the rolling upgrade method, we assumed there are no possible incompatibilities during the upgrade of hypervisors in our evaluation scenario, to have a fair

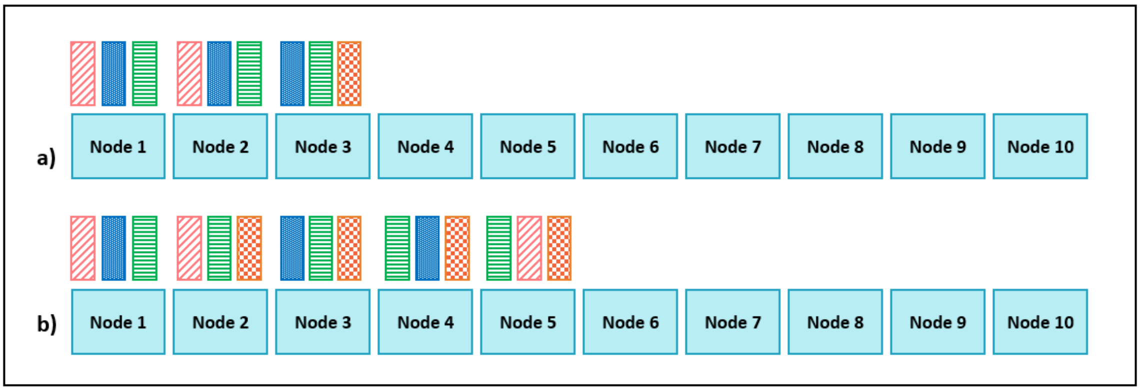


Figure 6.13. Two different cases for our evaluation scenario

comparison of our approach and the rolling upgrade method. Thus, the VMs can be migrated between old and new version of the hypervisors, with no need to be upgraded.

As mentioned earlier, due to the limitation of the deployment environment in our lab, we were only able to deploy six compute nodes for our PoC. The required measurements for our evaluation have been obtained from real deployment considering six compute nodes. The upgrade scenario for the six nodes was executed ten times, and for each measurement the average was considered. According to our measurements, upgrading the version of QEMU hypervisor (i.e. executing the compiled binaries of the new version) takes on average 41 seconds. Live migrating a VM between old and new versions of hypervisors takes on average 23 seconds. The introduced outage during live migration of a VM (with the tiny flavor) in OpenStack takes less than 0.6 seconds according to [72]. Performing the necessary calculations of each iteration of upgrade in our approach takes on average 0.23 seconds. These measurements have been used in our calculations for the evaluation scenario with 10 compute nodes.

To evaluate the availability at the application and the VM level, we calculated and compared: the total duration of the upgrade, the average outage time at the application level for each tenant, and the average outage time of each VM during the upgrade. With respect to SLA violations, we calculated and compared: the number of SLA violations per tenant, the duration of SLA violations in each breach, the total duration of SLA violation per tenant, and the applicable penalties. Note that the selection of the nodes in each batch of upgrade, the distribution of the VMs running on the selected nodes, and the order of their upgrade can result in different outage and SLA violations. Therefore, we performed our assessments considering different batch selections and considered the average result.

Figure 6.14 shows the comparison of the total duration of the upgrade when the tenants have their initial number of VMs, i.e. case study (a) as shown in Figure 6.13.a. The results show that the duration of the upgrade using our approach is shorter than using the rolling upgrade method with fixed batch size of 1, 2, and 3, while it is comparable with the duration of upgrade using batch size of 4. Note that we assume in the rolling upgrade method with a fixed batch size, the VMs of the selected nodes in the batch are migrated to other nodes, prior to their upgrade. Depending on the selection of in-use or not in-use nodes in the batch and the upgrade order of batches, the VMs may have to be migrated once, twice, or thrice during applying the rolling

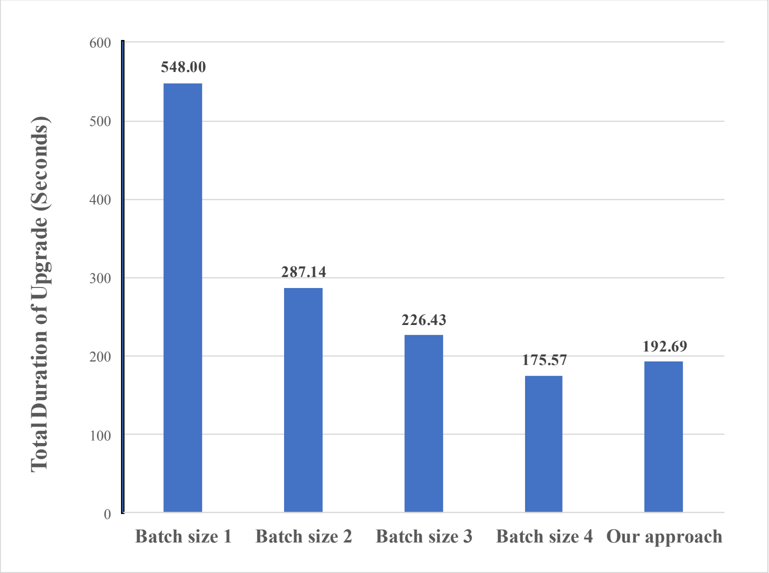


Figure 6.14. Comparison of the total duration of upgrade using different upgrade methods for case study (a)

upgrade method for this case study. For example, when the in-use nodes are upgraded prior to the upgrade of not in-use nodes and their VMs are live migrated to the old version nodes, the VMs will be migrated three times. This impacts the total duration of upgrade, as well as the outage time at the application and the VM level.

Considering the application level redundancy, upgrading more than one node hosting VMs from a single anti-affinity group (i.e. tenant in this example scenario) introduces outage at the application level. This is because of the outage experienced by the VMs of the same anti-affinity group. Since the VMs are evacuated (using live migration) from the nodes that are being upgraded, the duration of introduced outage at the application level will depend on the duration of the outage of the VMs during live migration (i.e. 0.6 seconds). This is valid as long as there are enough available nodes to host the evacuated VMs. Note that this is not the case for the tenants that are not configured HA at the application level or have only one VM (e.g. tenant 4).

Figure 6.15 shows the comparison of the average outage at the application level for case study (a), for impacted tenants and per tenant. Note that by the average outage time per tenant we mean the average outage time considering all tenants, whether they are impacted or not impacted. In the rolling upgrade method with a batch size of one, similar to our approach, the tenants do not experience any outage at the application level, except for tenant 4. This is because tenant 4 has initially one VM, meaning it is not configured HA at the application level. Since we assume in the rolling upgrade method, the nodes are selected according to the size of the batch (regardless of their usage state), each VM may have to be migrated between one and three times. Accordingly, the outage at the application level experienced by tenant 4 may be 0.6, 1.2, or 1.8 seconds using rolling upgrade method with batch size of one. Since in our approach we upgrade not in-use nodes before in-use nodes, tenant 4 will only experience an outage of 0.6 seconds at the application level. Note that if a similar rule is followed while applying

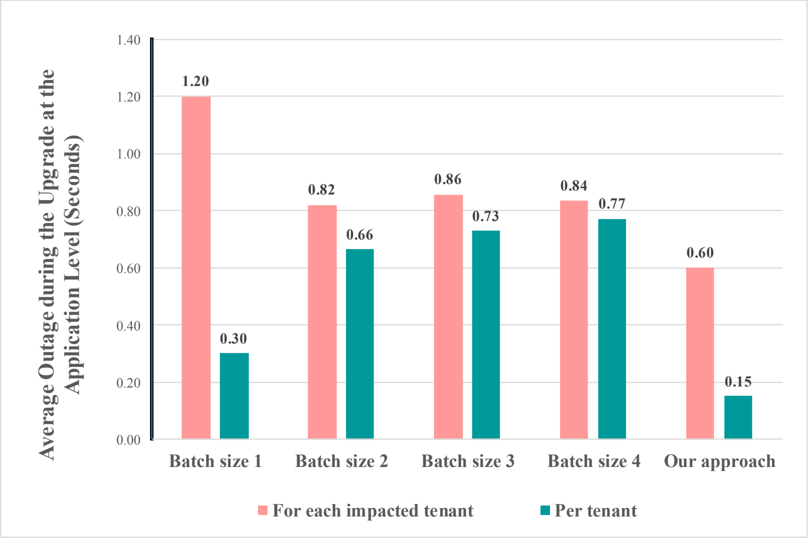


Figure 6.15. Comparison of the average outage at the application level for case study (a)

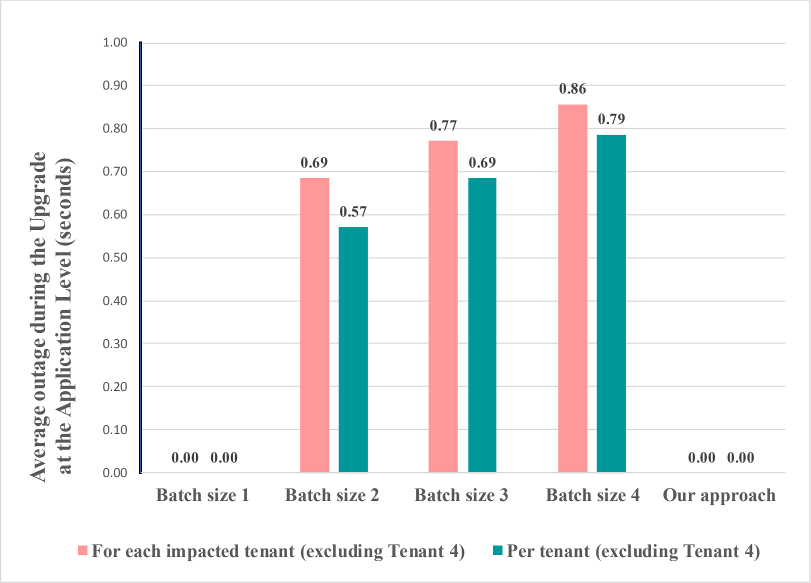


Figure 6.16. Comparison of the average outage at the application level for case study (a) excluding tenant 4

the rolling upgrade method with batch size one, the resulting outage at the application level will also be 0.6 seconds, similar to our approach. Note that if tenant 4 had more than one VM, it will not experience any outage for either using our approach or the rolling upgrade method with batch size one. For this reason, we calculated the average outage at the application level for case study (a) excluding tenant 4, as shown in Figure 6.16. As it was expected, the introduced outage at the application level for our approach and the rolling upgrade method with batch size of one is equal to zero (excluding tenant 4).

According to our results, as the batch size in the rolling upgrade method increases as shown in Figure 6.15 and Figure 6.16, the number of impacted tenants and the average outage time at the application level increases as well. This is due to increasing the probability of the selection of the VMs from the same anti-affinity in a single batch. Note that although in our approach the batch size can be more than one node, however we upgrade only one VM from a single anti-affinity group at a time, which prevents introducing the outage at the application level. Therefore, for the tenants that are configured HA in the application level, our approach does not introduce any outage at the application level.

Figure 6.17 shows the comparison of the total duration of upgrade for case study (b), after the tenants have some scaling in/out as shown in Figure 6.13.b. The results indicate that the duration of the upgrade using our approach is less than the rolling upgrade method with fixed batch size of one. However, it is more than the duration of the upgrade using rolling upgrade with batch size of two or three. The comparison of the average outage time at the application level for case study (b) for impacted tenants and per tenant, are shown in Figure 6.18. Our approach and the rolling upgrade method with batch size of one demonstrate similar results, with no

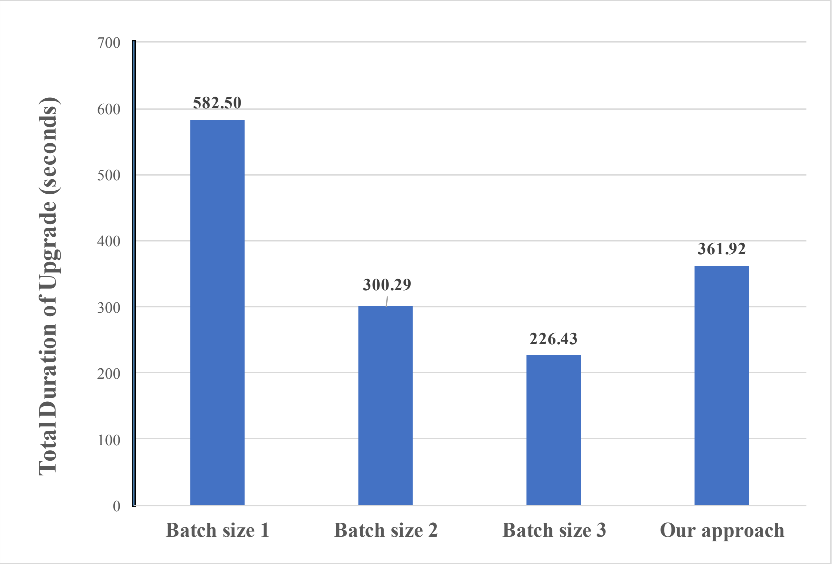


Figure 6.17. Comparison of the total duration of upgrade using different upgrade methods for case study (b)

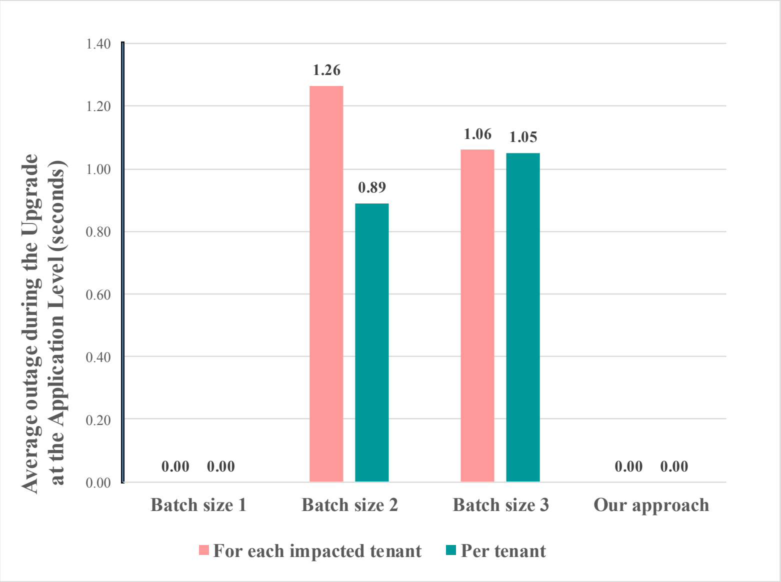
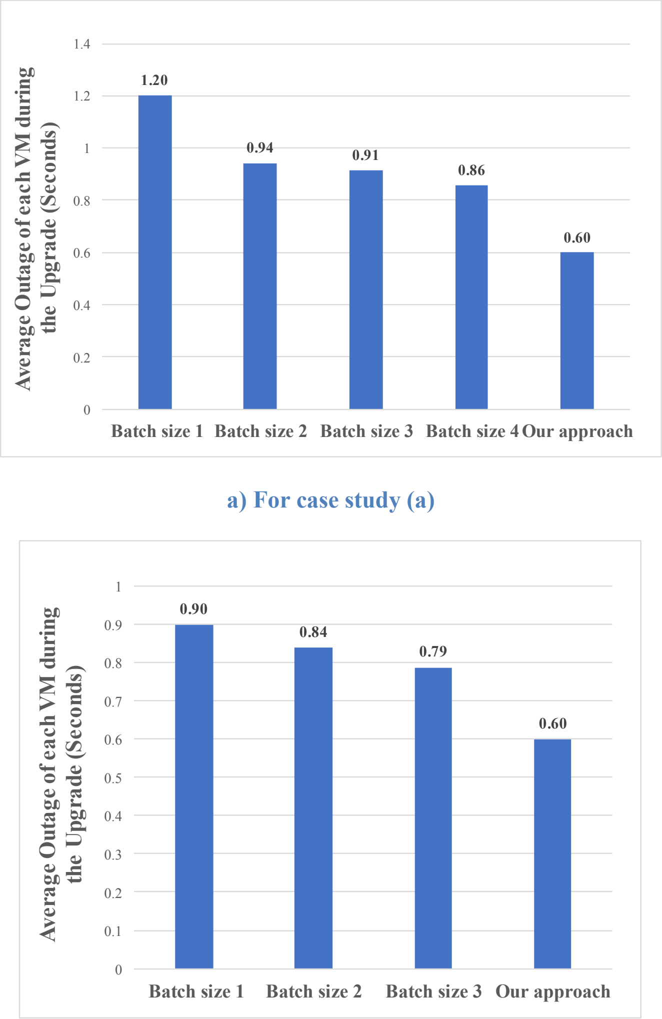


Figure 6.18. Comparison of the average outage at the application level for case study (b)

outage at the application level. While the rolling upgrade method with batch size two or three, introduce outage at the application level. Again in our calculations we considered multiple possible batch selections, as well as different upgrade ordering of the batches, and we used the average results.

While using the rolling upgrade method or using our approach, for both cases of (a) and (b), each VM experiences an outage during its migration. Figure 6.19 presents the average outage time of each VM during the upgrade for case study (a) and (b). As the results indicate each VM experiences less outage using our approach, compared to the rolling upgrade method. Based on the upgrade order of in-use or not in-use nodes, each VM may be migrated more than once during the upgrade, which will impact the total outage time that the VM experiences.

During the upgrade whenever VMs of the tenants experience an outage (during live migration), SLA violations will occur. This is because the current number of VMs for the tenants drops below the required number of VMs to accommodate their current workload. Table 6.1 and Table 6.2 demonstrate the comparison of SLA violations during the upgrade for case (a) and case (b), respectively. In our evaluation we calculated the number of times SLA violations



a) For case study (a)

b) For case study (b)

Figure 6.19. Comparison of the outage for each VM during the upgrade

occur for each tenant and we considered the average as the number of SLA violations per tenant. The number of impacted VMs per tenant in each SLA violation indicates how many VMs are impacted. For the average total duration of SLA violations per tenant, we calculated the total time of SLA violations for each tenant and considered their average.

For each SLA violations, penalties are applied. The penalties can be formulated in different ways by different cloud providers. In our evaluation, we measured two different types of applicable penalties of *delay-dependent penalty* and *proportional penalty* as described in [73]. In delay-dependent penalty, the penalty is only proportional to the occurred delay in providing the required capacity and it is calculated by multiplying the SLA violation duration to an agreed

Table 6.1. SLA violation related measurement results for all possible batch selections for case study (a)

Method /Approach	Total Duration of Upgrade (Seconds)	Number of SLA Violations per Tenant		Number of Impacted VMs per Tenant in each SLA Violation		Average Total Duration of SLA violation per tenant (seconds)	Applicable delay dependent penalty per tenant	Applicable proportional penalty per tenant
		Min	Max	Min	Max			
Batch Size 1	548.00	2	6	1	1	2.25	2.25 q	2.25 q’
Batch Size 2	287.14	2	4	1	2	1.69	1.69 q	2.26 q’
Batch Size 3	226.43	1	3	1	2	1.35	1.35 q	1.99 q’
Batch Size 4	175.57	1	2	1	2	1.24	1.24 q	1.93 q’
Our Approach	192.69	1	3	1	1	1.35	1.35 q	1.35 q’

Table 6.2. SLA violation related measurement results for multiple possible batch selections for case study (b)

Method /Approach	Total Duration of Upgrade (Seconds)	Number of SLA Violations per Tenant		Number of Impacted VMs per Tenant in each SLA Violation		Average Total Duration of SLA violation per tenant (seconds)	Applicable delay dependent penalty per tenant	Applicable proportional penalty per tenant
		Min	Max	Min	Max			
Batch Size 1	582.50	5	8	1	1	3.38	3.38 q	3.38 q’
Batch Size 2	300.29	3	4	1	2	2.20	2.20 q	3.11 q’
Batch Size 3	226.43	2	3	1	3	1.53	1.53 q	2.96 q’
Our Approach	361.92	3	5	1	1	2.25	2.25 q	2.25 q’

penalty rate of *q* (per unit time). Proportional penalty is a form of delay-dependent penalty, where the penalty is additionally proportional to the difference between a user’s provisioned capacity and the current allocation. It is calculated by multiplying an agreed penalty rate of *q’* (per unit capacity per unit time), the duration of SLA violations, and the difference in the expected and provisioned capacity.

As it was expected, the results reported in Table 6.1 and Table 6.2 show that by increasing the batch size in the rolling upgrade method, the duration of SLA violations per tenant decreases, as well as the duration of the upgrade. However, the number of impacted VMs, per SLA violation, increases. The applicable proportional penalties for our approach are less than the rolling upgrade method, as our approach prioritizes the upgrade of not in-use nodes which reduces the number of VM migrations during the upgrade.

Note that when the tenants are scaled out to their maximum number of the VMs, the upgrade process will be paused in our approach until scaling in happens. Whereas the rolling upgrade method will continue regardless of the state of the system. This will causes more SLA violations and increase in the applicable penalties, compared to the case studies in our evaluation (as presented in Table 6.1 and Table 6.2).

Overall consideration of our evaluation demonstrates that our approach works better to respect the SLA constraints of availability and elasticity, compared to the rolling upgrade method with fixed batch sizes.

6.2 Prototype for the Upgrade Coordinator

6.2.1 Prototype Architecture and Assumptions

We implemented a prototype for the upgrade coordinator based on our proposed approach, presented in Chapter 5, for the upgrade of all kinds of IaaS resources. It is implemented in Java and it uses JGraphT [74] java library for implementing and manipulating RG and CG graphs used throughout our approach. In this implementation, to demonstrate the progress of the upgrade, we simulated the behaviour of the upgrade engine, which is responsible for applying the schedules generated by the upgrade coordinator in a real system.

Figure 6.20 shows the overall architecture of our prototype for the upgrade coordinator and the interaction between its modules. The two main modules in this prototype are as follow:

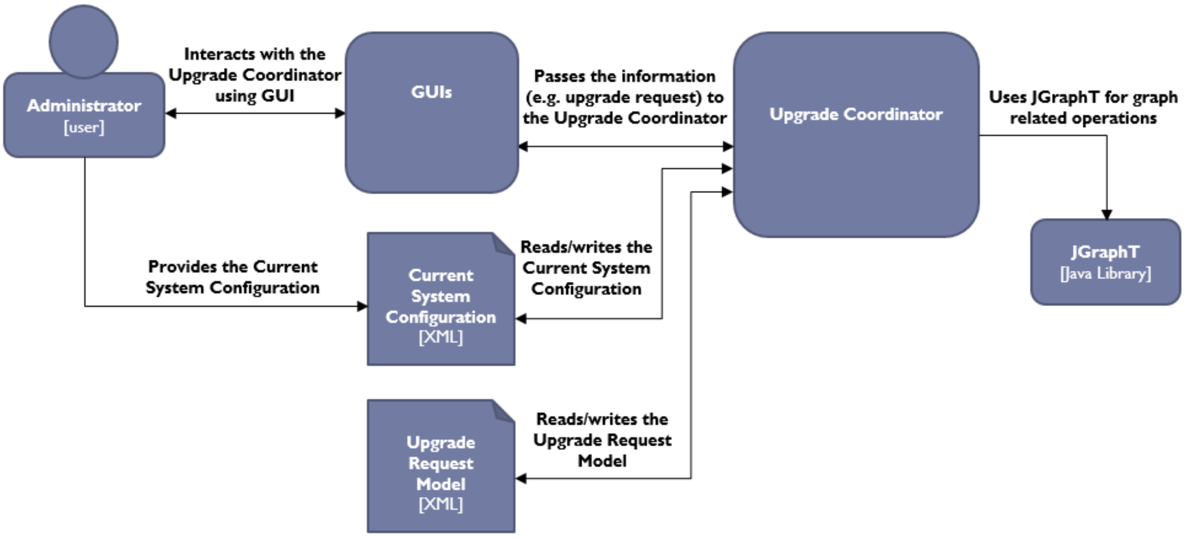

Administrator
[user]
Interacts with the Upgrade Coordinator using GUI
GUIs
Passes the information (e.g. upgrade request) to the Upgrade Coordinator
Upgrade Coordinator
Uses JGraphT for graph related operations
JGraphT
[Java Library]
Provides the Current System Configuration
Current System Configuration
[XML]
Reads/writes the Current System Configuration
Upgrade Request Model
[XML]
Reads/writes the Upgrade Request Model

Figure 6.20. Overall architecture of upgrade coordinator prototype

Graphical User Interfaces (GUIs): They are used by the system administrator to 1) create the initial system configuration, 2) add new tenants with their SLAs, 3) add new upgrade requests, and 4) submit the simulation input as feedback for the execution of the upgrade actions of the schedule generated by the upgrade coordinator. The upgrade actions for the upgrade schedule generated by the upgrade coordinator is also shown to the administrator using GUI. The administrator may choose to use XML file to import the current configuration instead of using a GUI, and later modify the resources and dependencies using the GUI before starting the upgrade process. Once the upgrade process starts, all of the changes in the configuration (except the failures, scaling in/out, and live migration) have to be requested as upgrade requests. Note that in the real system, the initial configuration will be collected automatically from the system.

Upgrade Coordinator: This module represents the implementation of our proposed upgrade approach presented in Chapter 5. This module uses JGraphT library to create, update, and traverse the graphs (e.g. the RG and the CG). The upgrade actions for the upgrade schedules are generated by this module and passed to the GUI module for display to the administrator. To demonstrate the progress of the upgrade, this module is additionally responsible to apply the simulated input received from the administrator as the feedback for the execution of identified upgrade actions. After starting the upgrade process, the current configuration will be kept updated according to the progress results of the upgrade process. As mentioned in Chapter 4 and Chapter 5, in the real system the upgrade engine is responsible for applying the upgrade actions to the system. To track the process of applying the upgrade actions, the upgrade request model is stored as XML document and is kept updated by the upgrade coordinator during the upgrade.

In this prototype, for simplicity, we assume the administrator indicates all the required changes (including complementary changes) for a change set when specifying an upgrade request. We also assume that each change indicates the addition, removal, or upgrade of some resources, each requiring a single upgrade action on a resource. We also assume that the administrator provides the estimated time required for upgrade and recovering from possible failures for each change. In the real deployment this information is extracted from the infrastructure component descriptors.

6.2.2 Case Study for Illustration

As a case study, we use similar example scenario presented in Section 5.1.1. As mentioned earlier, there are 15 hosts in this example as shown in Figure 6.21. Nine of these hosts participate in creation of a VMware Virtual Storage Area Network (VSAN) [63] in the system ($|M_{Storage}|$=9), while 10 of the hosts provide compute services ($|M_{compute}|$=10). Each host has at least one CPU, memory and NIC. In this example we assume that each host in $M_{compute}$ has a capacity to serve two VMs (K=2) and this capacity remains unchanged after upgrade. In addition to these resources, there are dedicated network resources: switches and routers shown at the bottom of the figure. The example assumes four tenants (N_i =4) each with their scaling policy. Each of the tenants has an initial, a max_n and a min_n number of VMs. Again for simplicity, we assume that all the tenants have the same scaling adjustment and cooldown period. The VMs

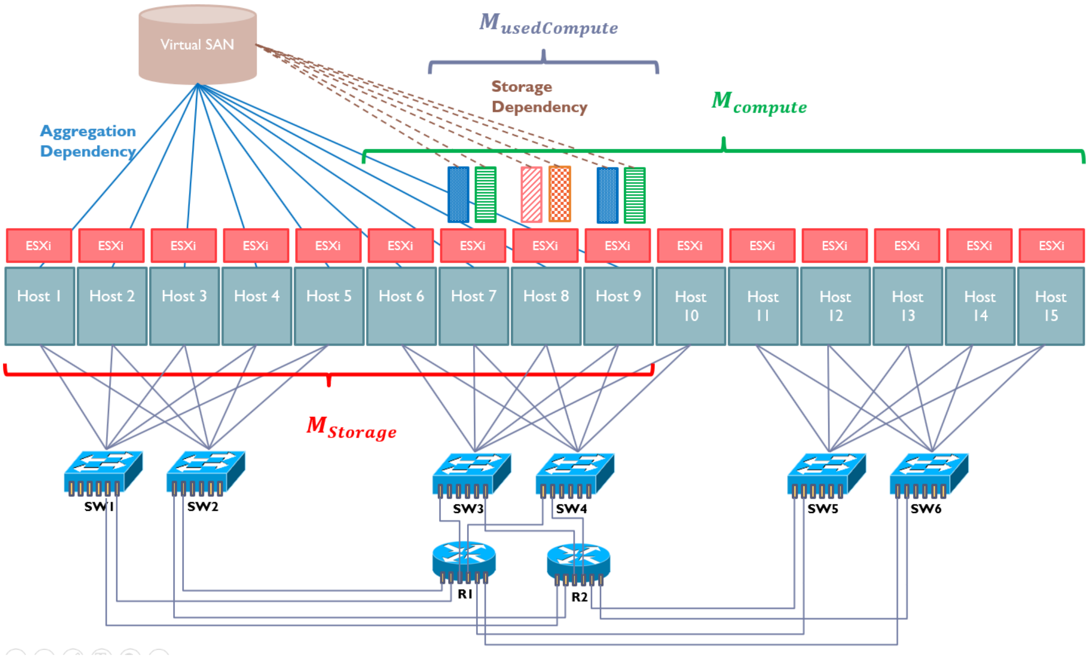


Figure 6.21. The illustrative example scenario for IaaS cloud upgrade approach

of each tenant can scale in/out with a scaling adjustment of s_n=1 VM. Similar to the example scenario in Section 6.1.2 different patterns are used to show the VMs of different tenants as shown in Figure 6.5. In this example, we also assume that all the VMs of a tenant form a single anti-affinity placement group.

The administrator issues an upgrade request with two change sets: (1) upgrade the virtual shared storage from VSAN to Ceph [50]; and (2) upgrade the networking infrastructure from IPv4 to IPv6 considering dual stack. Note that the VSAN and Ceph are incompatible with each other, thus the upgrade process have to prevent introducing possible incompatibilities while applying change set 1.

As mentioned earlier in this prototype we assume the administrator indicates all the required changes for a change set as an input. Upgrading VSAN to Ceph requires detaching hosts from VSAN cluster, upgrading hypervisors on the hosts from ESXi to hypervisors supported by Ceph (e.g. QEMU or Xen), and configuring Ceph components (e.g. OSD, monitoring, and client daemons) on hosts. Upgrading network infrastructure from IPv4 to IPv6, requires upgrading all switches and routers to IPv6 and upgrading all hosts by configuring to IPv6. We assume the minimum required number of storage hosts for configuring Ceph is five, while it is three for VSAN. In addition, for simplicity we assume these number of storage hosts can handle the data of VMs for existing tenants. Note that the detailed requirements of VSAN, Ceph, IPv4, and IPv6 products are out of the scope of this example, and the aforementioned change sets may require additional changes to those that are presented in this example scenario. Thus, the change sets will be indicated as presented in Table 6.3. Here for simplicity, we use V1 and V2 as two different versions of the resources, and we consider Ceph monitor and Ceph OSD as Ceph-storage component to be configured on storage hosts. Note that in the real deployment, based on the requirements of Ceph [50] it is recommended to have Ceph OSD and Ceph monitors installed on separate nodes.

The administrator also specifies additional parameters with respect to retry and undo operations for the change sets and the changes as presented in Table 6.4. In this example we assume the estimated time required for upgrade and recovering from possible failures for each change and the estimated required time to upgrade and to recover from possible failures for the selected

Table 6.3. The change sets and their changes of the upgrade request for example scenario

Change sets	Changes
Change set 1	Change 1: Upgrade Hypervisors (on Host1 to Host15) from V1-H to V2-H
	Change 2: Add Ceph-client component to Compute Hosts (Host6 to Host15)
	Change 3: Add Ceph-storage component to Storage Hosts (Host1 to Host5)
Change set 2	Change 1: Upgrade Switches (SW1 to SW6) from V1-SW to V2- SW
	Change 2: Upgrade Routers (R1 and R2) from V1-R to V2-R
	Change 3: Upgrade hosts (Host1 to Host15) from V1-H to V2-H

Table 6.4. Additional information provided by the administrator for example scenario

Change sets	Changes	max-retry threshold (for set)	max-completion-period (for set)	undo-threshold (for change)	undo version (for change)
Change set 1	Change 1	3	3600 seconds	15	V1-H
	Change 2			10	-
	Change 3			5	-
Change set 2	Change 1	2	3600 seconds	6	V1-SW
	Change 2			2	V1-R
	Change 3			15	V1-H

number of VMs through multiple sub-iterations are each equal to cooldown period for the tenants. However, in the implementation estimated required time for the upgrade and the cooldown periods may be different. Note that the max-completion-time(s) presented in Table 6.4 for the change sets are only given for the sake of example, and they do not reflect the time required for changes in the real deployment. In the real deployment the complementary changes, the configuration requirements and the time required for the upgrade and recovery from possible failures will be inferred from infrastructure components of the products provided by the vendor.

In this illustrative example, we present the three following scenarios: 1) Successful changes for both change sets, 2) failed upgrade actions for change set 2 which triggers retry and undo operations, and 3) new upgrade requests while there are ongoing upgrades.

6.2.2.1 *Successful changes for both change sets*

First Iteration

As the first step of the first iteration, the RG is created as shown in Figure 6.22. Note that different colors are used for demonstrating different modification-types and dependency types. In this graph, vertices of R1 to R15 represent the hypervisors running on host1 to host15 represented by vertices R16 to R30. This hosting relation (i.e. container/contained dependency) is represented by the edges between the vertices e.g. R1 and R16. VMs are represented by R39 to R44 running on hypervisors represented by R7 to R9, and the migration dependency between VMs and hypervisors are represented with edges between the vertices e.g. R39 and R7. For readability of this graph, the constitute resources (i.e. CPU, memory, NIC) of only two of the hosts are represented, e.g. vertices R47 to R50 represent the constitute resources of host1 represented by R16. The composition dependencies represented by the edges between vertices e.g.

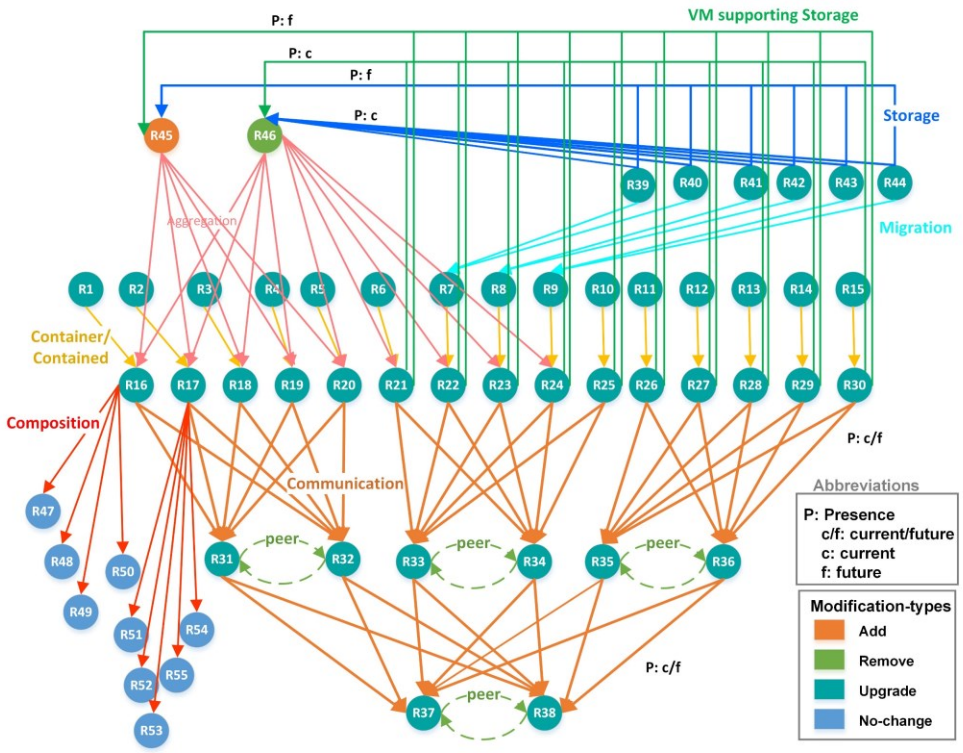


Figure 6.22. The RG of the illustrative scenario in the first iteration after step 1

R47 and R16. In this RG, vertices of R31 to R36 represent the switch SW1 to SW6, and vertices of R37 and R38 represent the routers R1 and R2, respectively. The existing communication links which are realization of communication dependencies in the system, are represented by edges with this dependency type between vertices e.g. R16 and R31.

Since existing virtual shared storage (i.e. VSAN) is a storage infrastructure resource supporting the VM operations, and cannot be upgraded to Ceph in place due to incompatibilities between these two products, PPU method has to be used for its upgrade. In the RG two vertices of R46 and R45 are created to represent the old (i.e. VSAN) and the new (i.e. Ceph) configuration of the VM supporting infrastructure, respectively. Note that in the current configuration, storage hosts R16 to R24 are aggregated into the virtual shared storage of R46, while in the future configuration R16 to R20 will be aggregated into R45 based on change set 1. Accordingly, the presence attribute of edges representing aggregation dependencies between virtual shared storages and their constituent resources is "Current" for VSAN, while it is "Future" for Ceph. Note that the VM supporting storage dependencies are represented by edges between vertex representing storage infrastructure resources (e.g. R45 for VSAN) and vertices representing the compute hosts (e.g. R21 to R30). In addition, the current VMs are using VSAN as the storage, thus they have storage dependencies towards VSAN. This storage dependencies are represented with edges between vertices representing VMs (i.e. R39 to R44) and VSAN (i.e. R46) with presence attribute of "Current". The similar dependencies are depicted between VMs and Ceph in the future configuration by edges with presence attribute of "Future".

Modification-type of vertices are set based on whether the resources are to be upgraded, added, or removed according to the requested change sets. For example, the modification-type for compute hosts represented by R21 to R30 will be "Upgrade", since the addition of the Ceph-client component and the configuring of the host to IPv6 is upgrading the compute hosts in the system. Since PPU has to be used for the upgrade of storage infrastructure resource (i.e.VSAN to Ceph), the modification-type of vertices representing them, R46 and R465, is set respectively to remove and to add, while setting their related-resource attribute indicating this relation between them.

Since we assumed the administrator indicates all the required changes for a change set, thus there are no additional complementary changes for these changes. Each change set is assigned to a unique undo unit which includes all target resources of the change set. The undo units for our illustrative scenario are shown in Figure 6.23. Note that hosts represented by R16 to R30 are in common between two undo units.

The actions-to-execute attribute of each vertex representing a resource will be set according to the upgrade actions required for changes requested on that resource. For example, the actions-to-execute attribute of vertex R30 representing Host15 includes upgrade actions for both change 2 in change set 1 (i.e. adding Ceph-client component) and change 3 in change set 2 (i.e.

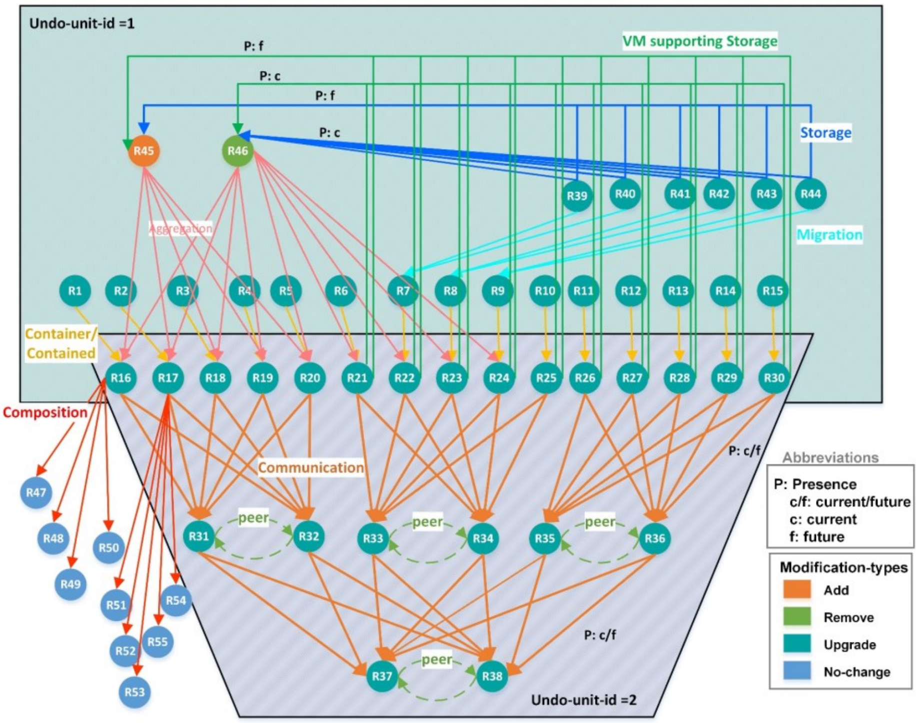


Figure 6.23. The RG and the identified undo units in the illustrative scenario

upgrading to V2-H for IPv6 configuration). Since the upgrade actions of both of these changes can be applied in a single iteration, they can be organized in a single execution level. For simplicity, in this implementation we assume that each change has one corresponding upgrade action.

In this example the only possible incompatibility may be introduced during the upgrade of the storage infrastructure supporting VM operations. This incompatibility is handled in a global way throughout our approach using the PPU method. Thus, we exclude VMs and VM supporting infrastructure resources while identifying the upgrade units. Since in this scenario there are no other possible incompatibilities, the resources without incompatibilities along their dependencies are in separate upgrade units; and the rolling upgrade method is selected as their appropriate upgrade method.

According to the step 2 as described in Chapter 5, we perform dependency based contraction for hosts (represented by R16 to R30) and hypervisors (represented by R1 to R15) with container/contained dependencies, and for hosts and constitute resources (i.e. CPU, Memory, and NIC) of hosts (e.g. represented by R47 to R54) with composition dependencies. Since the rolling upgrade method is the associated upgrade method with the identified upgrade units, here we do not need to perform the upgrade method based vertex contraction. Note that the PPU method is applied globally, and there is no need for applying vertex contraction for resources being upgraded with the PPU method. The resulting CG graph is shown in Figure 6.24.

At the step 3, first the VMs from physical hosts in common between the sets of $M_{Storage}$ and $M_{Compute}$, i.e. host6 to host9, are evacuated and consolidated while respecting the availability constraint inferred from the anti-affinity grouping. The RG and CG are updated accordingly as shown in Figure 6.25 and Figure 6.26, respectively. Next, the G_{batch} is initialized. Subsequently, the elimination rules is applied to eliminate the non-suitable candidates from the G_{batch}:

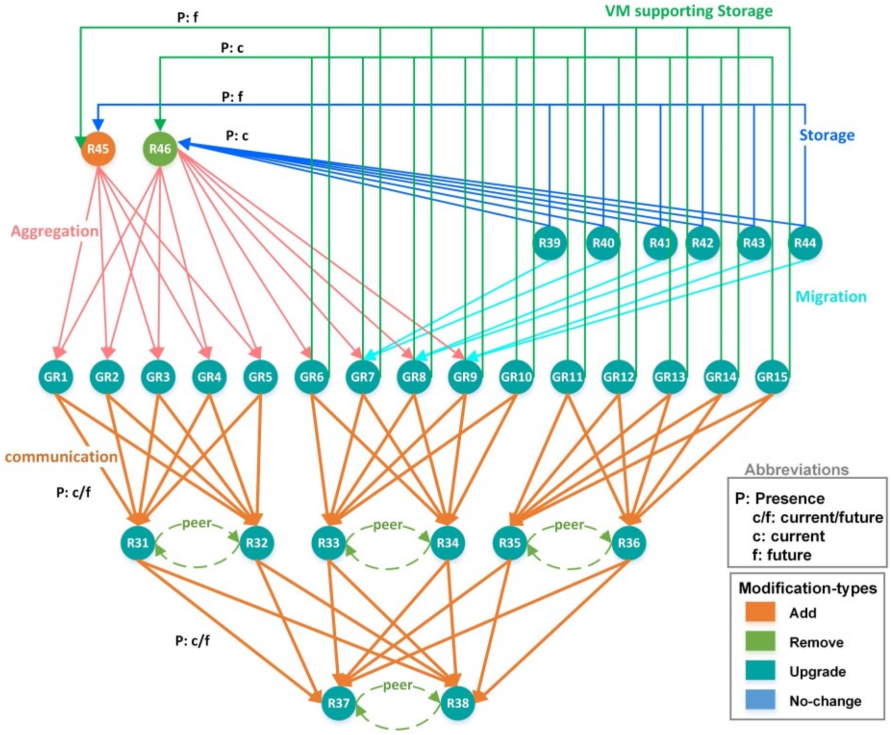


Figure 6.24. The CG of the illustrative scenario in the first iteration after step 2

- Elimination rule 1 removes CG vertices representing in-use physical hosts (i.e. GR13, GR14, and GR15) and VMs (i.e. R39, R40, R41, R42, R43, and R44) involved in migration dependency.
- Elimination rule 2 removes CG vertex of R46 representing the old configuration of the shared storage (VSAN), as there are other resources depend on this resource other than VM supporting infrastructure dependency and its related resource (R45) has modification-type of "Add".
- Elimination rule 3 removes the CG vertex of R45 representing the new configuration of shared storage (Ceph), as this resource is an aggregate resource and its required number of constituent resources (i.e. five storage hosts) are not ready yet to satisfy the requirements of Ceph configuration.

- Elimination rule 4 removes dependent CG vertices of GR1 to GR12, and R31 to R36 from the G_{batch} according to case 1.a. of this rule, as explained in Appendix II.

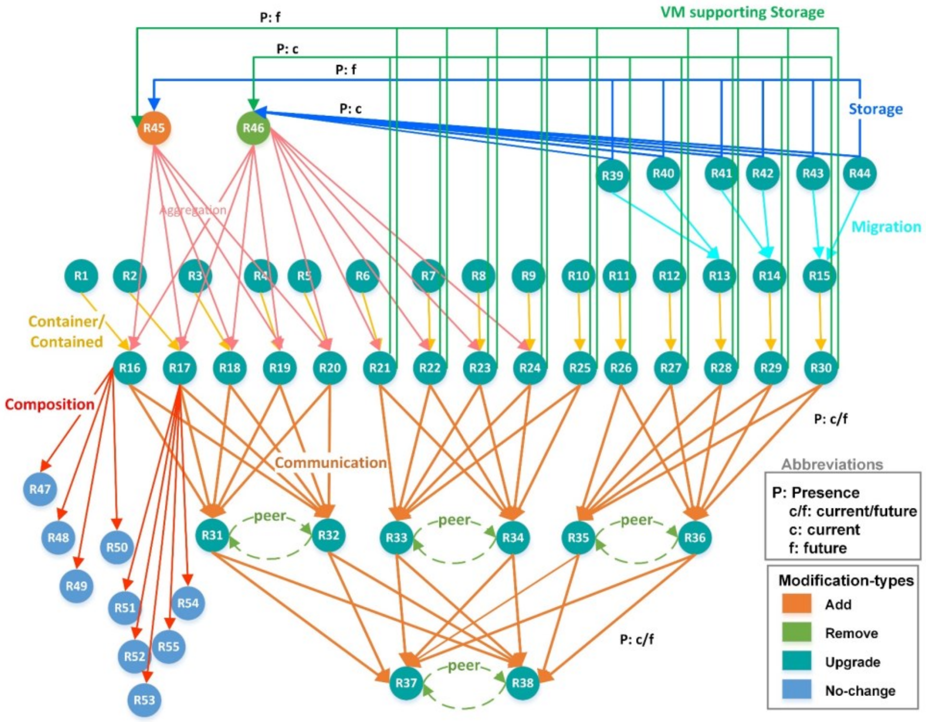


Figure 6.25. The RG of the illustrative scenario after VM consolidation in step 3 of the first iteration

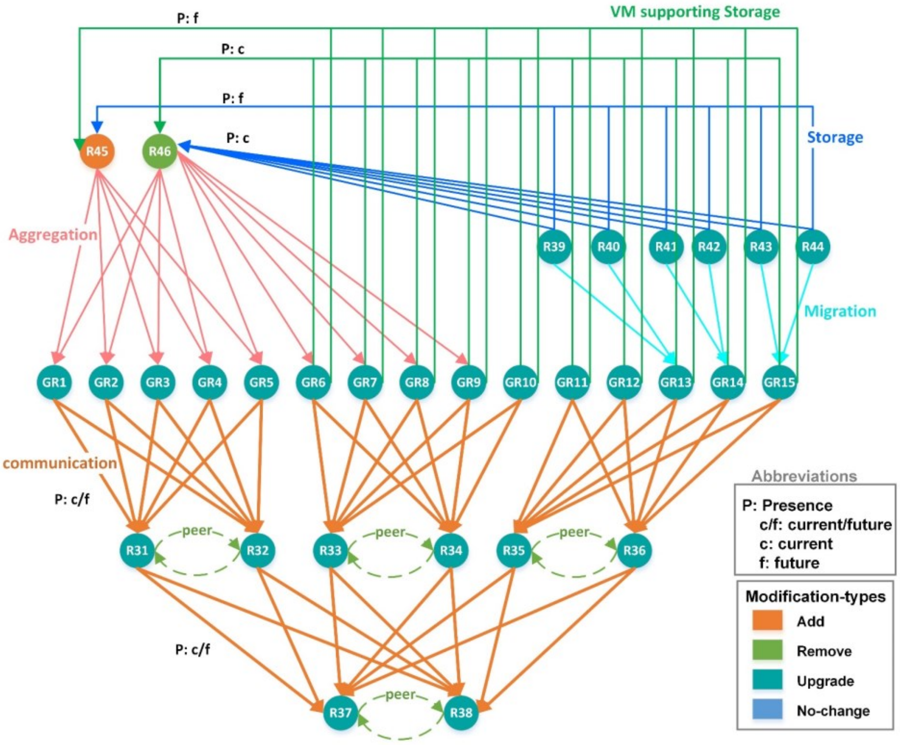


Figure 6.26. The CG of the illustrative scenario after VM consolidation in step 3 of the first iteration

- Elimination rule 5 does not remove any CG vertices from the G_{batch}, as the associated upgrade method of the identified upgrade units is the rolling upgrade method and there is no upgrade unit with possible incompatibilities in this example which needs ordering.
- Elimination rule 6 removes one of the peer CG vertex of R37 and R38 to guarantee the availability of services provided by peer resources. Let us assume R38 is removed from the G_{batch}.
- According to elimination rule 7 there are enough resources for upgrading virtual shared storage. We assumed the minimum number of storage hosts required for VSAN configuration ($MinHostReqConf_{oldStorage}$) is three and the minimum number of storage hosts required for Ceph configuration ($MinHostReqConf_{newStorage}$) is five. We also assumed that these storage hosts provide minimum number of required storage hosts for storing data of all VMs. Hence, $MinHostReqCapacity_{oldStorage}$ is three and $MinHostReqCapacity_{newStorage}$ is five. The number of storage hosts that are not in use as compute hosts ($|M_{Storage}\text{-}M_{usedCompute}|$) is 9. According to Equation (7) in Appendix II, we have:

  $$9 \geq 3 + 5$$

  This means the current system has enough storage hosts to support the upgrade of virtual shared storage, thus this elimination rule will not remove the resources related to the upgrade of virtual shared storage from the G_{batch}. Since in this scenario, we assume there are no resource failures, the result of this evaluation will be the same for all the iterations.

After applying the elimination rules, the remaining CG vertex in the G_{batch} is R37. In this example we assume the estimated required time to upgrade and to recover from possible failures for each change is equal to the cooldown period for the tenants, and scaling adjustment of each tenant is equal to one (s_n=1). Thus, S_i is also equal to 1 according to equation (3) in Chapter 5 in Section 5.2.3. Since none of the tenants have upgraded VMs yet, the number of tenants who may scale out on hosts compatible with the old version of the VMs (*A₁*) is 4. The number of compute hosts for scaling reservation of tenants with VMs of the old version for this iteration is calculated based on equation (5) in Chapter 5 in Section 5.2.3, as follow:

$$Scaling\ Re\ s\ v_{forOldVM} = 1 * \left\lceil \frac{4}{2} \right\rceil = 2$$

Accordingly, the maximum number of compute hosts that can be taken out of service in the first iteration (Z_1) is calculated based on equation (4) in Chapter 5 in Section 5.2.3, as follow:

$$Z_1 = 7 - 2 - 1 = 4$$

The number of affected compute hosts during the upgrade of initial batch (i.e. R37) is zero and less than 4, thus the final batch will include R37 as well. The upgrade action of the first execution-level of the actions-to-execute attribute of R37 will be presented as the schedule of this iteration. Assuming the upgrade action for the change is executed successfully (i.e. simulated input as feedback is true) on the R37, the first execution-level is removed from its actions-to-execute attribute of this resource. Since there is no further remaining change on R37, the modification-type of the resource changes to "No-change". The upgrade request model, RG and CG are updated according to the results of this step. The updated RG and CG are shown in Figure 6.27 and Figure 6.28, respectively. The step 4 in the first iteration is not necessary since the compute hosts are not separated into two incompatible partitions yet.

Second Iteration

Note that in this scenario we assume that all of the changes for both change sets are successful and there is no new upgrade requests issued by the administrator. Thus, the RG and CG will be remained unchanged through step 1 and step 2 after their last update in each previous iteration. Thus, in the following iterations we explain only the step 3 and step 4.

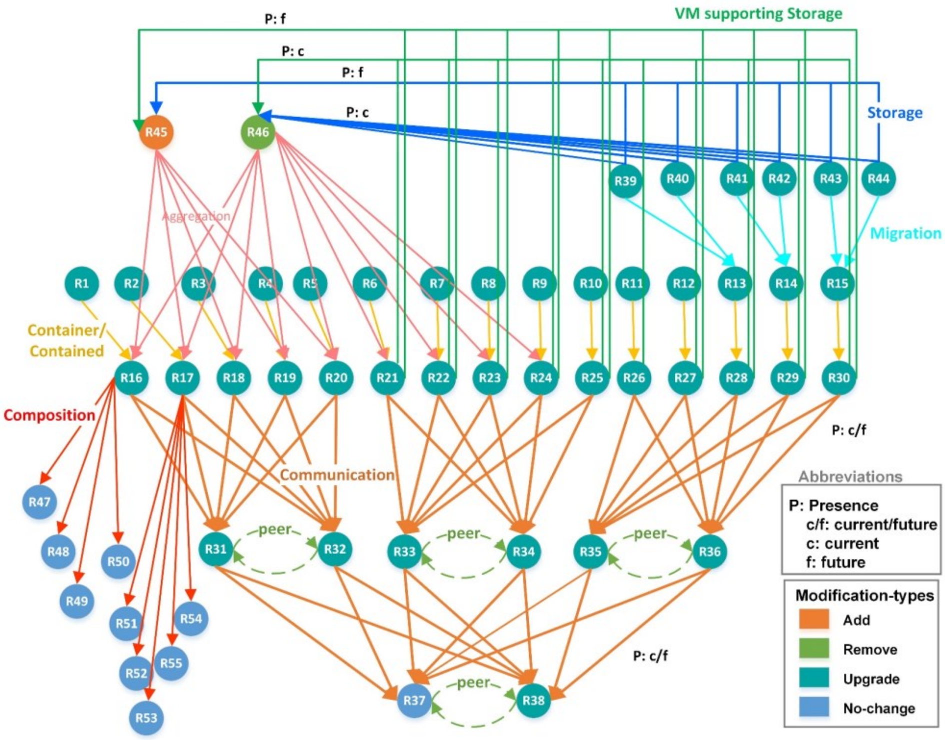


Figure 6.27. The RG of the illustrative scenario after successful upgrade of the first iteration in step 3

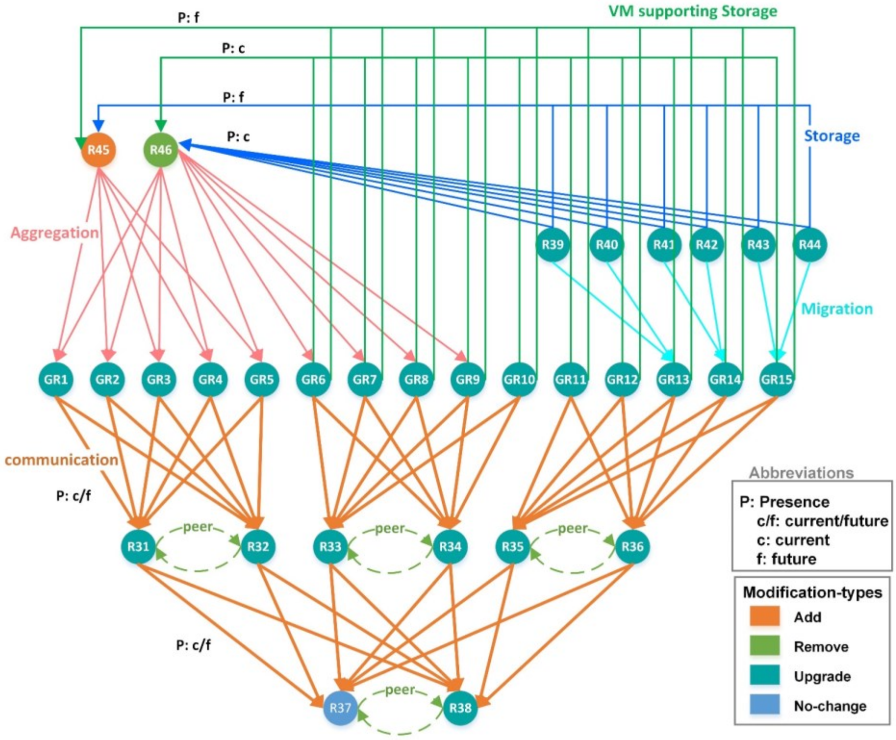


Figure 6.28. The CG of the illustrative scenario after successful upgrade of the first iteration in step 3

The G_{batch} is initialized including all the CG vertices shown in Figure 6.28 except R37 which is already upgraded. The CG vertices representing hosts, VMs, virtual share storages, and switches are eliminated from the G_{batch} by elimination rule 1, 2, 3, and 4. However, elimination rule 6 will not eliminate R38 from the initial batch since its peer resource R37 is in service after its upgrade. The maximum number of compute hosts that can be taken out of service in the second iteration will be 4 similar to the first iteration. The number of affected compute hosts during the upgrade of R38 is zero, therefore this resource can be upgraded in this iteration while respecting the dependencies and the SLA constraints. The upgrade request model, RG and CG are updated according to the results of this step. Again, the step 4 is skipped as the compute hosts are not upgraded yet.

Third Iteration

Similar to previous iterations the CG vertices representing hosts, VMs, and virtual share storages are eliminated from the G_{batch} by elimination rule 1, 2, 3, and 4. Note that in this iteration, elimination rule 4 does not remove CG vertices representing switches (R31 to R36) from the G_{batch} since their sponsors (R37 and R38) have been upgraded already. However, one switch out of each peer switches will be eliminated according to elimination rule 6 to protect the availability of services provided by peer switches. Let us assume R31, R33, and R35 remains in the initial batch. Similar to the first iteration, the maximum number of compute hosts that can be taken out of service in the third iteration (Z_3) is 4. The number of affected compute hosts during the upgrade of the initial batch is zero, so the final batch includes all the same resources as initial batch. After their upgrade, the upgrade request model, RG and CG are updated.

Fourth Iteration

Similar to the third iteration, in the fourth iteration the R32, R34 and R36 are upgraded. The updated RG and CG at the end of this iteration are shown in Figure 6.29 and Figure 6.30, respectively.

Fifth Iteration

Elimination rule 1 removes in-use physical hosts (i.e. GR13, GR14, and GR15) and VMs (i.e. R39, R40, R41, R42, R43, and R44) from the G_{batch}. Elimination rule 2 removes R46 as there are dependent resources on this resource, with dependencies other than VM supporting infrastructure dependency. Elimination rule 3 removes R45 since the required number of constituent resources (i.e. five storage hosts) for the resource represented by R45 (Ceph) are not ready yet. According to elimination rule 4, all the remaining compute hosts (GR6 to GR12) in the G_{batch} are eliminated, since their sponsor virtual shared storage (R45) in the new configuration is not

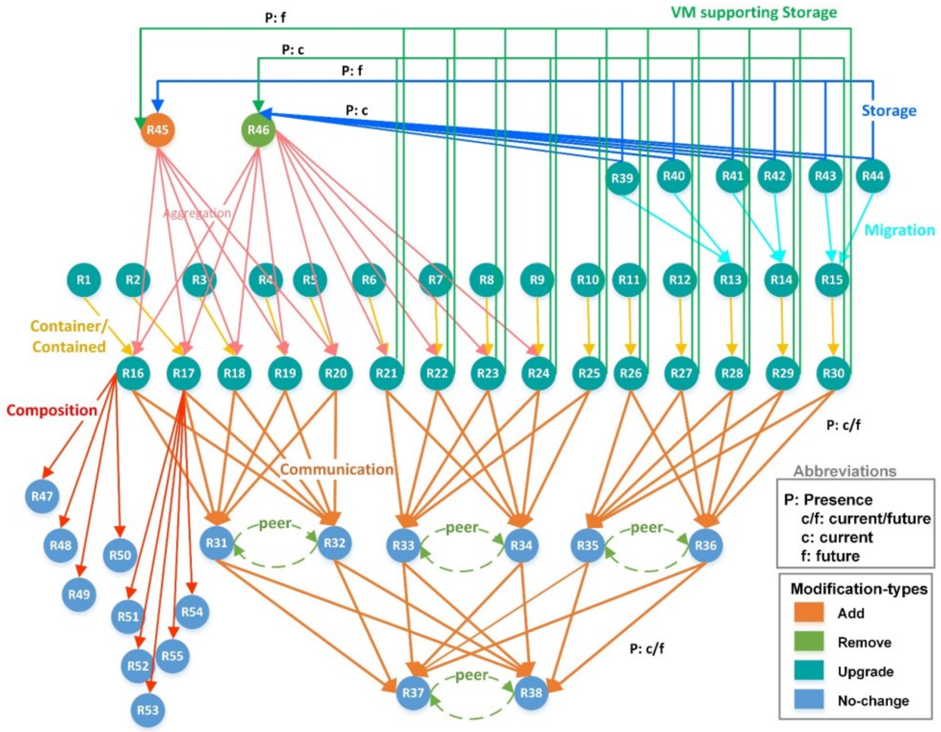


Figure 6.29. The RG of the illustrative scenario after successful upgrade of the fourth iteration

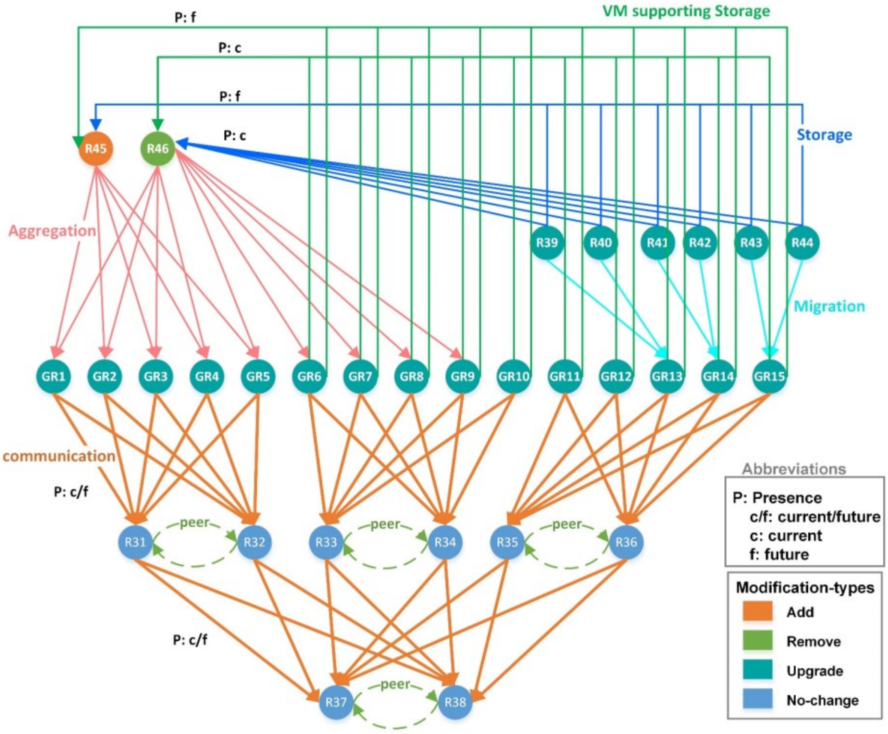


Figure 6.30. The CG of the illustrative scenario after successful upgrade of the fourth iteration

added yet (according to case 3.b in elimination rule 4 described in Appendix II). According to elimination rule 6 maximum of one constituent resource of aggregation dependency can stay in the G_{batch} to guarantee the availability of services given by peer resources, since this does not violate the minimum resource requirement of the aggregate resource with respect to its configuration and possibly the data stored. Note that the exception case for elimination rule 6 is not valid either, as the two conditions for the exception are not true, e.g. there are dependencies towards the aggregate resource with modification-type of "Remove" (R46) with are not from VM supporting infrastructure dependency type. Thus, only one of CG vertices of GR1 to GR5 can stay in the batch according to elimination rule 6. Let us assume GR1 remains in the batch, and GR2 to GR5 are eliminated. According to elimination rule 7 we still have enough resources to satisfy the PPU method, so the resources related to upgrade of VM supporting infrastructure resource can remain in the G_{batch}. Similar to previous iterations, the maximum number of compute hosts that can be taken out of service in fifth iteration (Z_5) is 4. The number of affected compute hosts during the upgrade of the remaining resource group (GR1) in the initial batch is zero and it can be selected for the final batch. The upgrade request model, RG and CG are updated upon providing the successful simulated feedback. Note that as a result of this upgrade, the aggregation dependency between R46 (old configuration of virtual shared storage, i.e. VSAN) and GR1 is going to be removed and the aggregation dependency between R45 (i.e. new configuration of the virtual shared storage, i.e. Ceph) and GR1 will be stablished in the current configuration. This means the presence attribute of the edge representing this dependency will in the RG changes from the "future" to "current/future". Again step 4 is not applicable since none of the compute hosts are upgraded yet.

Sixth, Seventh, Eighth, and Ninth Iterations

Similar to fifth iteration in the sixth, seventh, eighth, and ninth iterations only one of CG vertices of GR2 to GR5 is going to be upgraded due to elimination rule 6 and the constraint for keeping maximum one constituent resource of aggregation dependency in the initial batch. The RG and CG after ninth iteration are presented in Figure 6.31 and Figure 6.32, respectively.

Tenth Iteration

Similar to previous iterations GR13 to GR15 and R39 to R44, are eliminated from the G_{batch} according to elimination rule 1, while R46 is eliminated according to elimination rule 2. However, since the required number of constituent resources (five storage hosts) for the Ceph configuration (represented by R45) is ready, the elimination rule 3 will not remove R45 from the initial batch. Elimination rule 4 eliminates GR6 to GR12 according to elimination rule 4 since their sponsor virtual shared storage (R45) is not added yet. Considering the calculated 4 compute hosts as the maximum number of compute hosts that can be taken out of service in tenth iteration (Z_{10}), R45 will remain in the final batch. Thus, the configuration of the Ceph virtual shared storage will be completed in this iteration. This means the modification-type of R45 in the updated RG and CG changes to “No-change”.

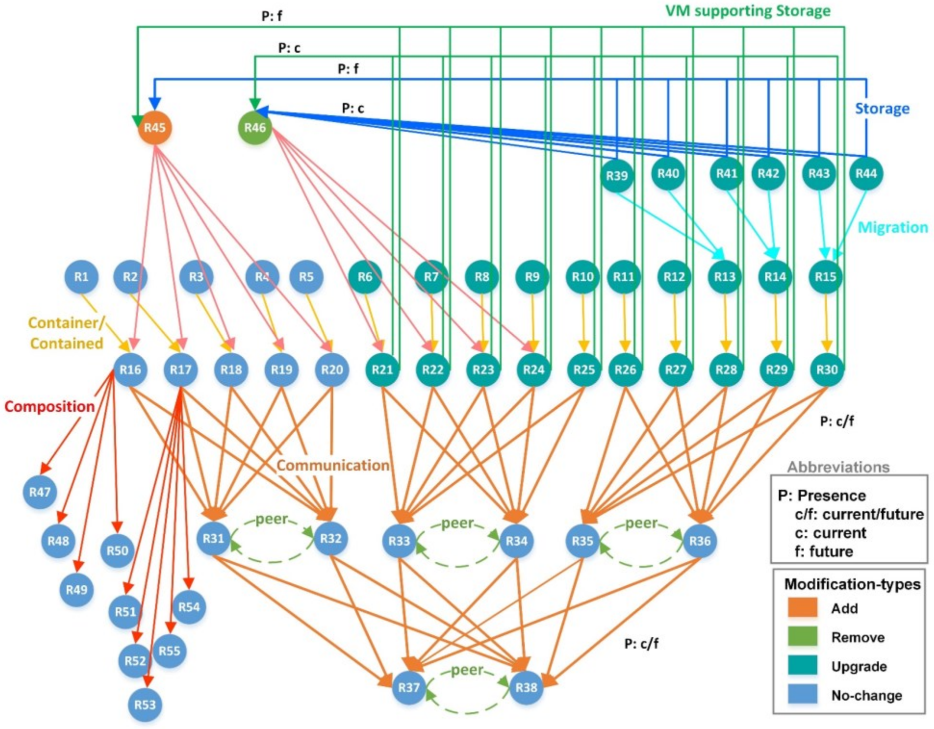


Figure 6.31. The RG of the illustrative scenario after successful upgrade of the ninth iteration in step 3

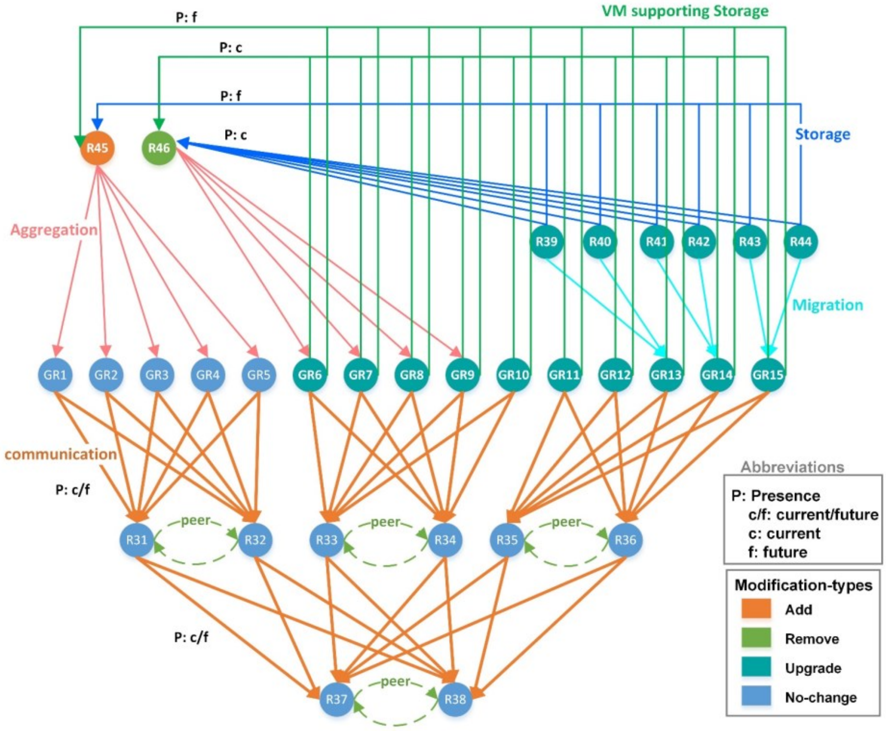


Figure 6.32. The CG of the illustrative scenario after successful upgrade of the ninth iteration in step 3

Eleventh Iteration

According to elimination rule 1 and 2, GR13 to GR15, R39 to R44, and R46 are removed. However, as R45 representing Ceph configuration is ready the elimination rule 4 does not remove GR6 to GR12 from G_{batch}. According to Elimination rule 6 only one of CG vertices of GR6 to GR9 can stay in the batch. Let us assume GR6 will remain in the G_{batch}. Therefore, the initial batch will include GR6, GR10, GR11, and GR12. The maximum number of compute hosts that can be taken out of service in eleventh iteration (Z_{11}) is 4, which is equal to the number of affected compute hosts during the upgrade of initial batch. Thus, the final batch can include GR6, GR10, GR11, and GR12. Considering the successful simulated feedback, the RG and the CG are updates as shown in Figure 6.33 and Figure 6.34. Note that as a consequence of the upgrades in this iteration, the edges representing the VM supporting storage dependency of the upgraded compute hosts towards the VSAN virtual shared storage represented by R46

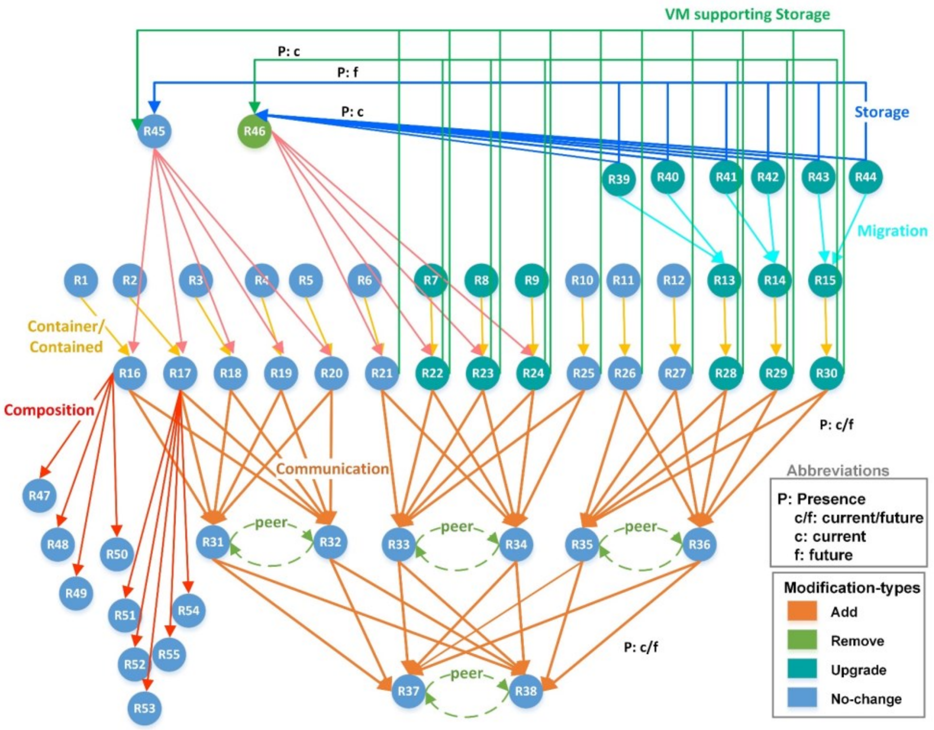


Figure 6.33. The RG of the illustrative scenario after successful upgrade of the eleventh iteration in step 3

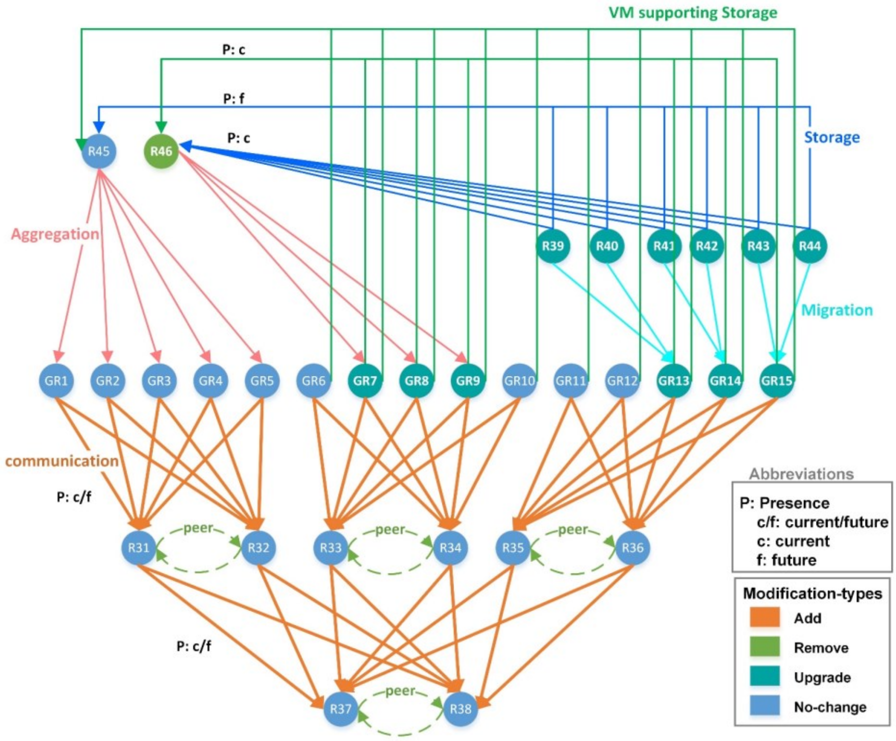


Figure 6.34. The CG of the illustrative scenario after successful upgrade of the eleventh iteration in step 3

is removed, and the presence of edge representing the VM supporting storage dependency of these compute hosts towards R45 are changed to “current/future”.

In this iteration after completing step 3, the compute hosts are separated into two incompatible partitions due to upgrade of virtual shared storage, therefore step 4 is performed. Note that the new configuration of the virtual shared storage (Ceph) is already completed. The set of compute hosts eligible to provide compute services for tenants with VMs of the new version ($M_{computeForNewVM}$) is 4, and none of them are in-use ($M_{usedComputeForNewVM} = 0$). Considering single host failure at a time and ability to recover before the next host failure, we reserve one compute hosts for any failover for upgraded VMs ($FailoverResev_{forNewVM} = 1$). Since in this iteration there is no tenant with upgraded VMs yet, therefore the scaling reservation for the tenants with upgraded (new) VMs ($ScalingResv_{forNewVM}$) is zero in the initial calculation. As mentioned earlier we assumed the host capacity in terms of VMs remains unchanged after the upgrade (K'=2).

The number of VMs that can potentially be migrated and if necessary upgraded in the current iteration V_{11} is calculated according to equation (6) in Chapter 5 in Section 5.2.4, as follow:

$$V_{11} = (4 - 0 - 1) * 2 = 6$$

We select all 6 VMs (represented by R39 to R44) as potential batch of VMs for this iteration. Considering the application level redundancy and anti-affinity group constraint, the number of VMs in the first sub-iteration $W_{11,\,1}$ is 4. We select one VM from each anti-affinity group (here tenant). Let us assume VMs represented by R39, R40, R41 and R42 are selected for the first sub-iteration. Before performing the VM migration/upgrade, we re-evaluate the scaling reservation to determine whether it is sufficient for scaling-out of selected tenants in this sub-iteration on the upgraded compute hosts. In one hand, two compute hosts are required for scaling reservation of all the selected tenants and one compute host is reserved for possible failover for upgraded VMs. In the other hand, two compute hosts are required for hosting the VMs of the selected tenants and in total there is only four compute hosts eligible to provide compute services for tenants with VMs of the new version. Thus, the upgrade of VMs from the selected four tenants cannot be carried out. The batch of VMs for this sub-iteration have to be re-adjusted to 2 VMs. Let us assume VMs represented by R39 and R40 are selected in this sub-iteration. Considering the successful simulated feedback, the RG and the CG will be updated after step 4 as shown in Figure 6.35 and Figure 6.36. Note that as a consequence of upgrading and migrating VMs to the upgraded compute hosts compatible with the new shared storage, the storage dependency of the upgraded VMs towards the VSAN represented by R46 is removed and the presence of the edge representing the storage dependency of these VMs towards R45 are changed to “current/future”.

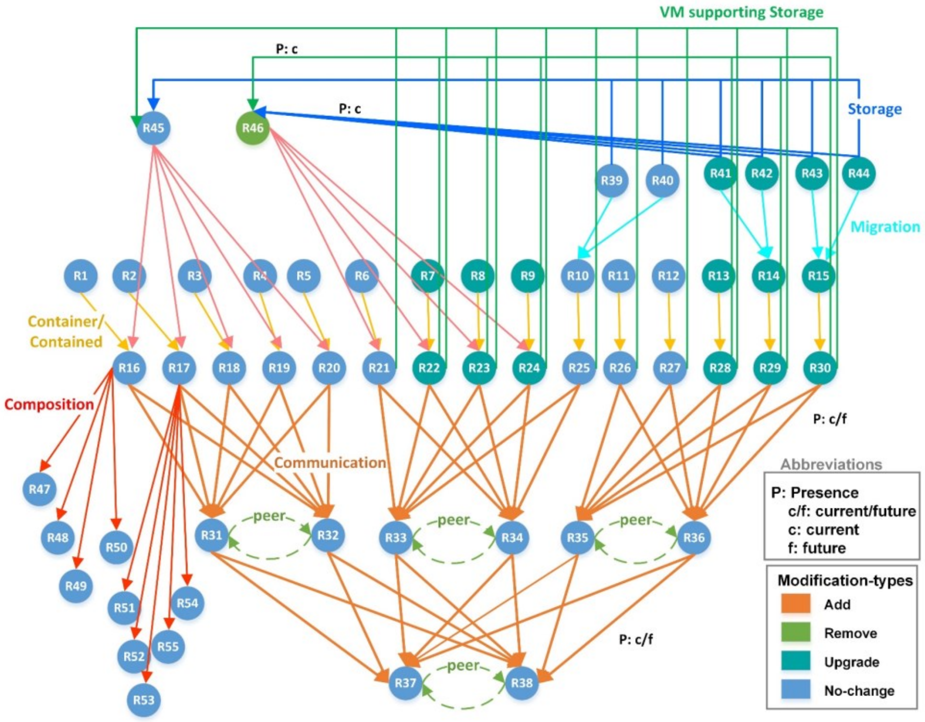


Figure 6.35. The RG of the illustrative scenario after successful upgrade of the eleventh iteration in step 4

Twelfth Iteration

Elimination rule 1 removes in-use compute hosts (GR14 and GR15) and VMs (R41 to R44) from the G_{batch}. Elimination rule 2 eliminates R46 as there are other dependencies (i.e. storage dependencies) than VM supporting infrastructure dependency towards it. Elimination rule 6 removes all group resources constituent (GR7 to GR9) of the old version of virtual shared storage (R46), as taking out any of them will result in violating the minimum resource requirement of the aggregate resource (R46), which is 3. Thus, the initial batch will only include GR13. Since two of the tenants have upgraded VMs, the number of tenants who may scale out on hosts compatible with the old version of the VMs (A_{12}) is 2. S_{12} is still equal to 1. The number of compute hosts for scaling reservation of tenants with VMs of the old version for this iteration is 1 based on equation (5) in Chapter 5 in Section 5.2.3, as follow:

$$Scaling\ Re\,s\,v_{forOldVM} = 1 * \left\lceil \frac{2}{2} \right\rceil = 1$$

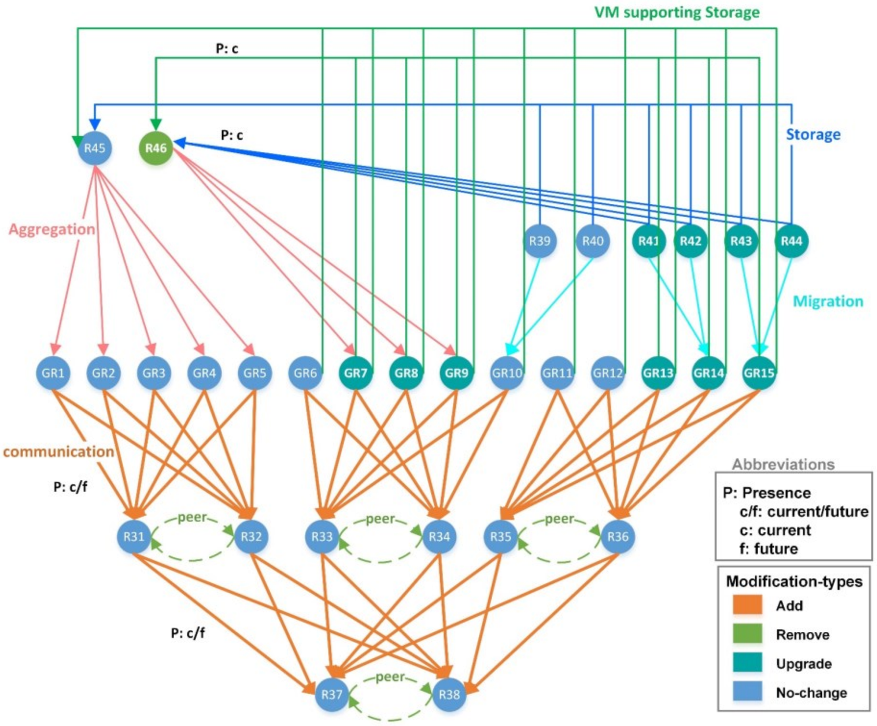


Figure 6.36. The CG of the illustrative scenario after successful upgrade of the eleventh iteration in step 4

Now $|M_{ComputeForOldVM}$ - $M_{usedComputeForOldVM}|$ the number of compute hosts that are not in use and are eligible to provide compute services for tenants with VMs of the old version is 4. The maximum number of compute hosts that can be taken out of service in the twelfth iteration (Z_{12}) is calculated based on equation (4) in Chapter 5 in Section 5.2.3, as follow:

$$Z_{12} = 4 - 1 - 1 = 2$$

The number of affected compute hosts during the upgrade of initial batch is 1, which is less than Z_{12}. Thus, the GR13 will be selected in the final batch. Considering the successful simulated feedback, the modification-type of the GR13 in the CG and its corresponding resources in the RG will be updated as "No-change". The upgrade will proceed to step 4.

The set of compute hosts eligible to provide compute services for tenants with VMs of the new version is 5, and one of them is in-use ($|M_{computeForNewVM}$ -$M_{usedComputeForNewVM}| = 4$). Since in this iteration there is two tenants with upgraded VMs, the scaling reservation for the tenants with upgraded VMs (*ScalingResv_{forNewVM}*) is 1. Accordingly, the number of VMs that can potentially be migrated and if necessary upgraded in twelfth iteration V_{12} is calculated 4, as follow:

$$V_{12} = (4 - 1 - 1) * 2 = 4$$

We initially select 4 VMs represented by R41 to R44 as potential batch of VMs. Since each of the selected VMs are from different tenants, we can potentially upgrade them in one sub-iteration while respecting anti-affinity constraint. We need two compute hosts for scaling-out reservation of the selected tenants and one compute host for possible failover for upgraded VMs. Considering the capacity of compute hosts, the remaining one compute host (out of four not in-use ones) is not enough to accommodate the upgrade of 4 VMs. The batch of VMs for this iteration is re-adjusted to 2 VMs. Let us assume VMs represented by R41 and R42 are selected in this iteration for the migration and upgrade. Considering the successful simulated feedback, the RG and the CG are updated after step 4 as shown in Figure 6.37 and Figure 6.38.

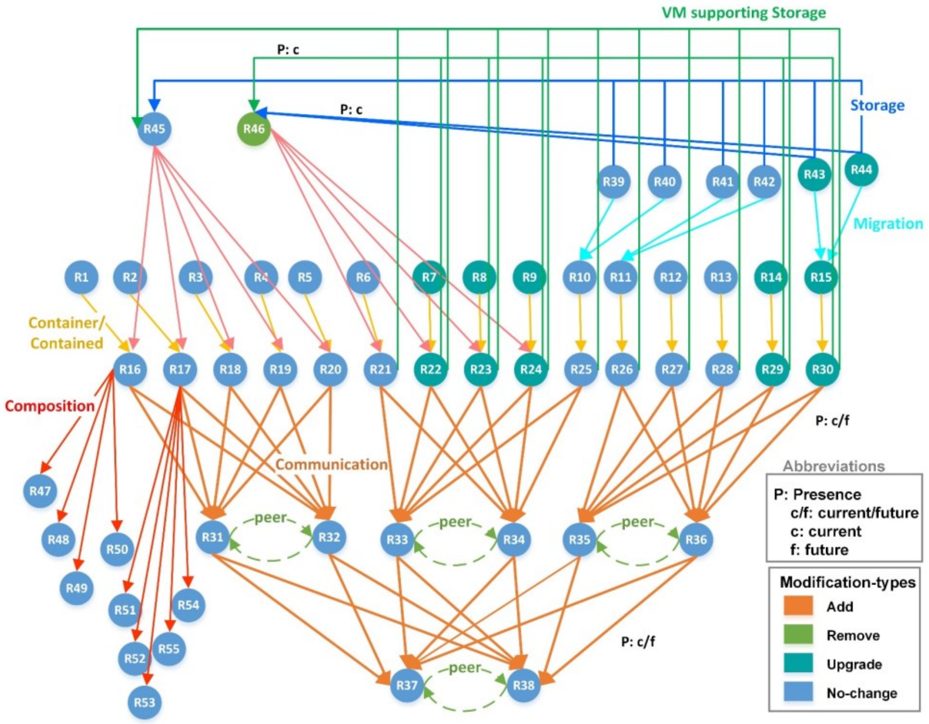


Figure 6.37. The RG of the illustrative scenario after successful upgrade of the twelfth iteration in step 4

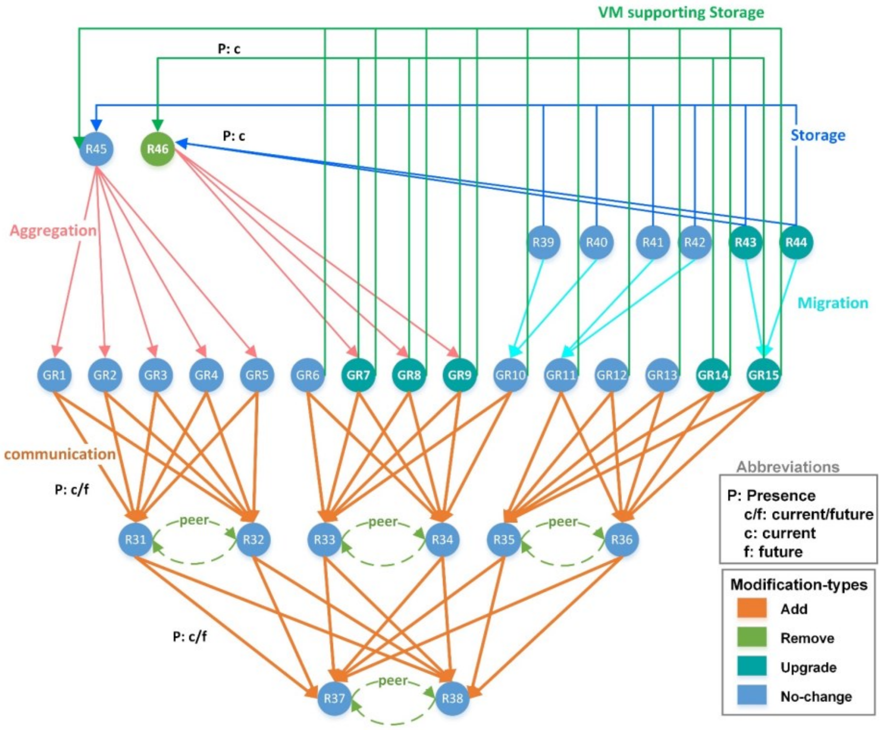


Figure 6.38. The CG of the illustrative scenario after successful upgrade of the twelfth iteration in step 4

Thirteenth Iteration

After applying the elimination rules, GR14 stays in the initial batch. Since all the tenants have upgraded VMs, therefore we do not need any compute hosts for scaling reservation on the old version compute hosts ($ScalingResv_{forOldVM}$ =0). However, still the failover reservation has to be considered for old version VMs. The maximum number of compute hosts that can be taken out of service in the thirteenth iteration (Z_{13}) is 3 based on equation (4) in Chapter 5 in Section 5.2.3, as follow:

$$Z_{13} = 4 - 0 - 1 = 3$$

The number of affected compute hosts during the upgrade of the initial batch (GR14) is 1, which is less than Z_{13}. Thus, the GR14 will be selected and upgraded as the final batch. The upgrade proceeds to step 4.

Considering the newly upgraded compute host, $|M_{computeForNewVM} - M_{usedComputeForNewVM}|$ is 4. All four tenants have upgraded VMs, thus the scaling reservation for the tenants with upgraded VMs ($ScalingResv_{forNewVM}$) is 2 and the number of hosts reserved for failover for upgraded VMs ($FailoverResev_{forNewVM}$) is 1. Accordingly V_{13} is 2, as follow:

$$V_{13} = (4 - 2 - 1) * 2 = 2$$

The remaining two VMs represented by R43 and R44 are selected as potential batch of VMs. Since the scaling-out reservation is considered for all the VMs in the upgraded VMs, the upgrade of potential batch can be carried out safely with respect to possible future scaling-out requests. After feeding the successful simulated feedback for VM upgrades, the RG and the CG are updated. The updated CG after thirteenth iteration is shown in Figure 6.39.

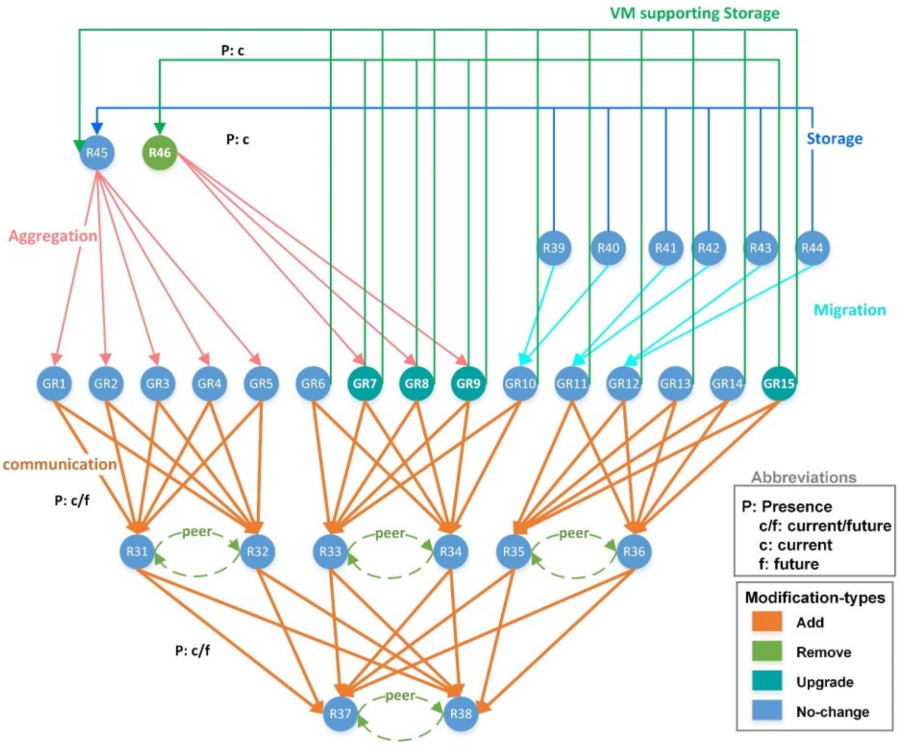


Figure 6.39. The CG of the illustrative scenario after successful upgrade of the thirteenth iteration in step 4

Fourteenth Iteration

In the fourteenth iteration, after applying the elimination rules GR15, GR7, GR8, GR9 and R46 will stay in the initial batch. In this iteration, elimination rule 2 does not remove R46 from the G_{batch}, since the only dependency towards R46 is the VM supporting storage dependency and its related resource (R45) has modification-type of "No-change". Note that all the VMs are already migrated to the compute hosts compatible with the new version of virtual shared storage (R45), so there is no storage dependency towards the old version of virtual shared storage (R46). This means R46 can be safely removed. Elimination rule 6 does not remove the constituent resource of R46 as well, since the two conditions for the exception case are true; there is no dependency except VM supporting storage towards R46 and the upgrade of the related resource of R46 is already completed. Thus, the constituent resources of the old version of virtual shared storage (R46) can be upgraded all at the same time. At this point, all the tenants have upgraded VMs and there is no old version VMs running on the compute hosts compatible with the old version virtual shared storage. Hence, there is no need to have compute hosts for scaling reservation or failover reservation for the old VMs ($ScalingResv_{forOldVM}$ =0 and $FailoverResev_{forOldVM}$ = 0). The maximum number of compute hosts that can be taken out of service in the fourteenth iteration (Z_{14}) is 4. The number of affected compute hosts during the upgrade of initial batch is 4, which is equal to Z_{14}. Thus, the final batch includes GR15, GR7, GR8, GR9 and R46. Assuming successful upgrade actions of the final batch, R46 will be removed from the system, while the compute hosts are upgraded. The updated RG and CG are presented in Figure 6.40 and Figure 6.41, respectively. Since all the changes in the upgrade request have been completed and there is no new upgrade request, the upgrade process terminates.

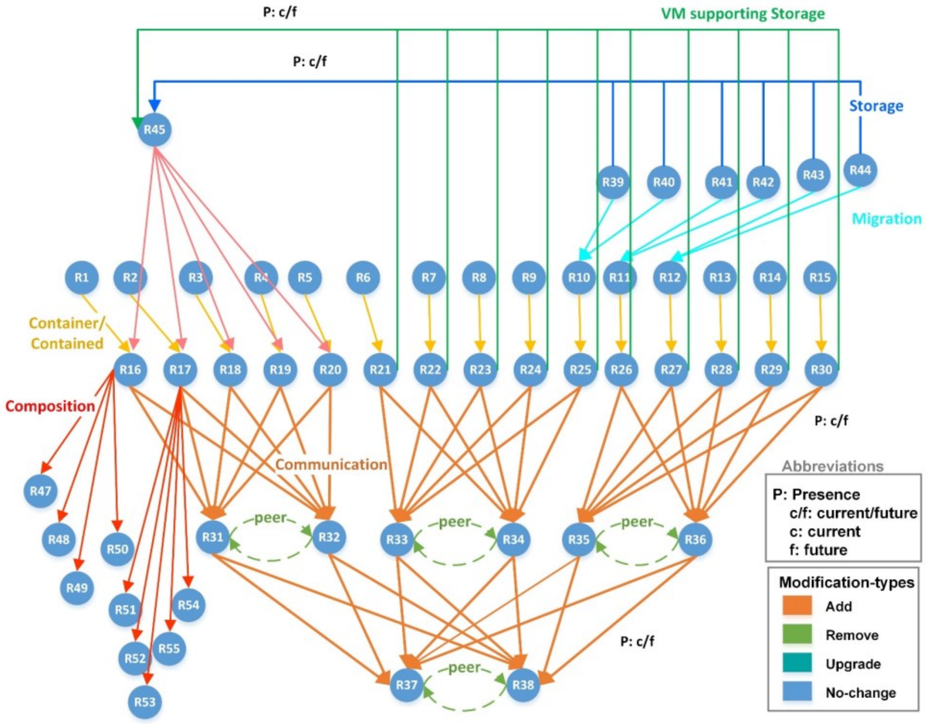


Figure 6.40. The RG of the illustrative scenario after successful upgrade of the fourteenth iteration in step 4

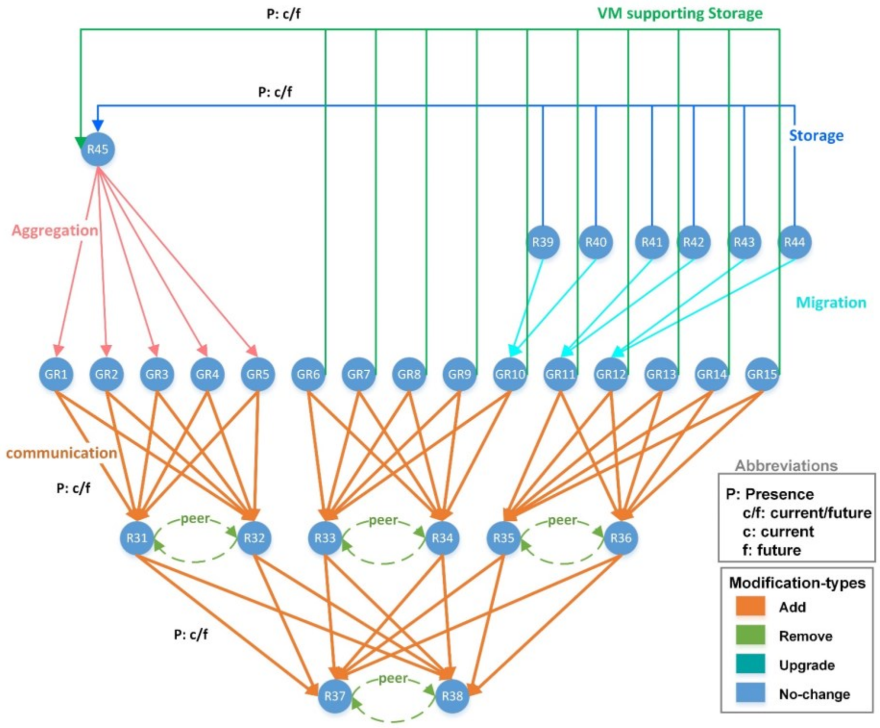


Figure 6.41. The CG of the illustrative scenario after successful upgrade of the fourteenth iteration in step 4

6.2.2.2 *Failed upgrade actions for change set 2 triggering retry and undo operation*

In this scenario we consider failed upgrade actions for the change set 2 which triggers recovery operations of retry and undo for this change set.

Third Iteration

Let us assume the upgrade actions for change set 2 fails in the third iteration on one of the switches represented by R31. This means the failed upgrade action feedback for resource represented by R31 will be provided to the upgrade coordinator. Note that in the real system, the upgrade coordinator will generate an upgrade schedule to bring back the resource to a stable configuration. For simplicity in this prototype implementation, we assume the resource with the failed upgrade actions is still in a stable configuration, thus it is not required to generate a new schedule for resource level undo. Upon receiving the feedback, the upgrade resource model, the RG and the CG will be updated. The first execution-level from the actions-to-execute attribute of resources with successful upgrade actions (R33 and R35) is removed, while it remains unchanged for the resource with failed upgrade action (R31). The counter of failed attempt on R31 is incremented. Note that the information regarding the failed upgrade action will be recorded in the upgrade iteration report. The updated RG and CG after completing third iteration with failed upgrade action on R31 are presented in Figure 6.42 and Figure 6.43, respectively. The step 4 will not be applicable as the compute hosts are not separated into two incompatible partitions yet.

Fourth Iteration

In the fourth iteration in step 1, the RG is updated to identify the necessary retry or undo operations for change set with failed upgrade actions. For this, the upgrade iteration report of the previous iteration is used. The number of failed upgrade attempt on R31 is 1 and the maximum allowed number of upgrade attempts on each resource for change set 2 is 2 (max-retry threshold = 2), as indicated in Table 6.3. Thus, a retry operations is allowed on R31. In step 2, the CG

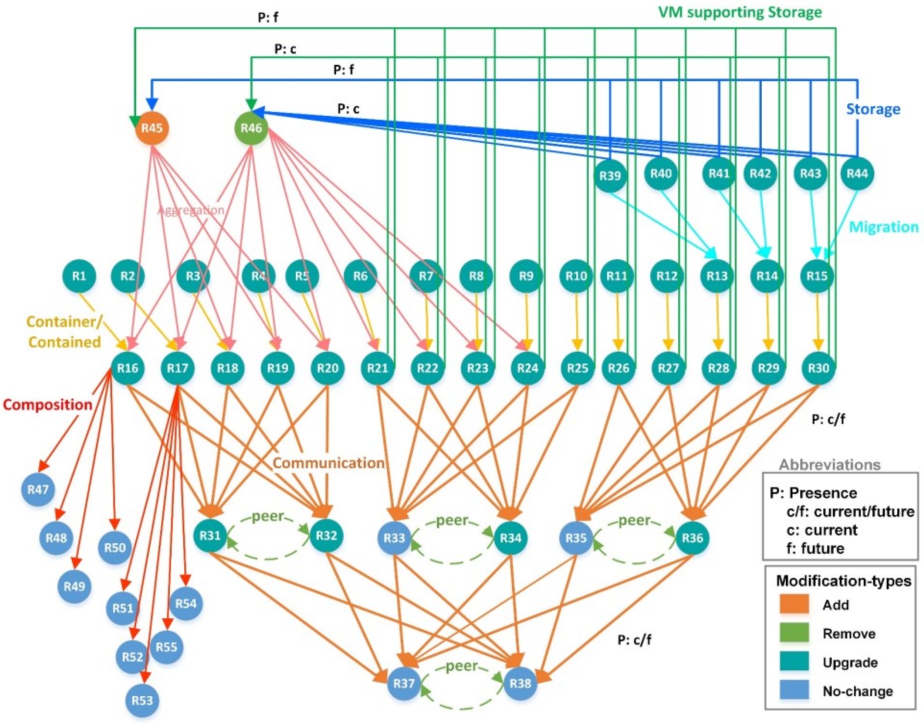


Figure 6.42. The RG of the illustrative scenario with failed upgrade action after third iteration in step 3

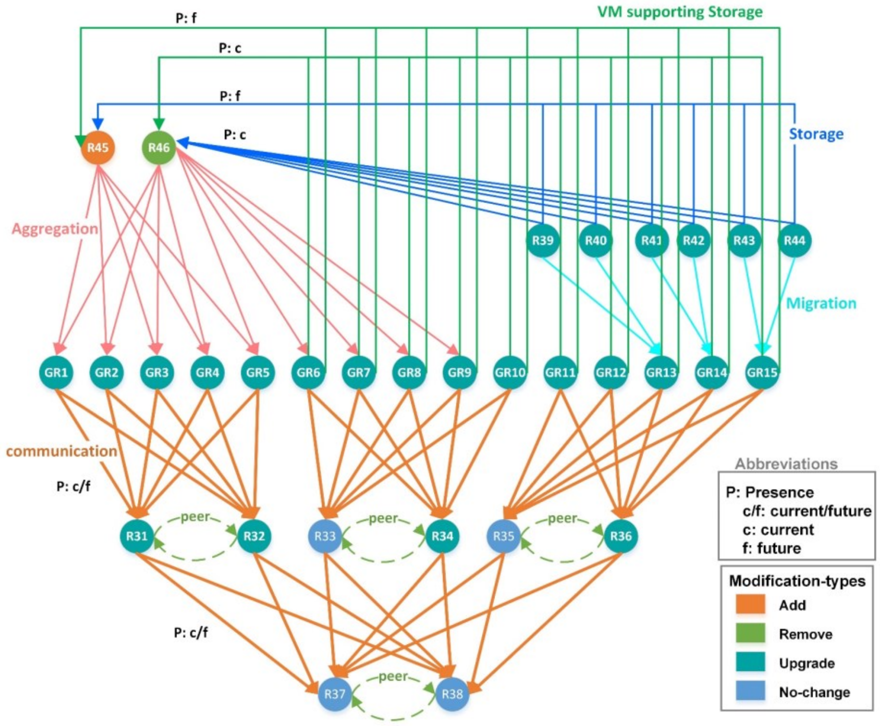


Figure 6.43. The CG of the illustrative scenario with failed upgrade action after the third iteration in step 3

will be updated accordingly as well.

After applying the elimination rules R31, R34, and R36 remains in the initial batch. The maximum number of compute hosts that can be taken out of service in the fourth iteration (Z_4) will be 4. Since the number of affected compute hosts during the upgrade of initial batch is zero (less than Z_4), the final batch includes R31, R34, and R36 as well. Let us assume in this iteration, the upgrade actions on R31 fails once more time, while the upgrade actions on R34 and R36 completes successfully. Similar to previous iteration, while updating the RG and the CG, the first execution-level from the actions-to-execute attribute of resources with successful upgrade actions (R34 and R36) is removed, while it remains unchanged for the resource with failed upgrade action (R31). The counter of failed attempt on R31 is incremented, which means the number of failed upgrade attempt on R31 reaches 2. Step 4 is not applicable in this iteration. The updated RG and CG after this iteration will be as shown in Figure 6.44 and Figure 6.45.

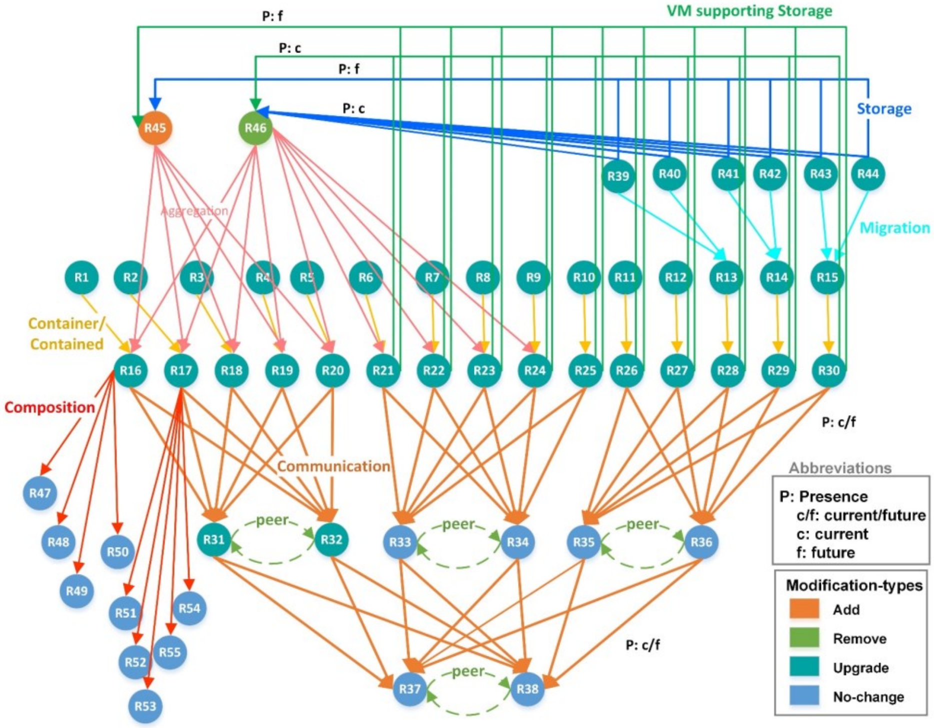


Figure 6.44. The RG of the illustrative scenario with failed upgrade action after fourth iteration in step 3

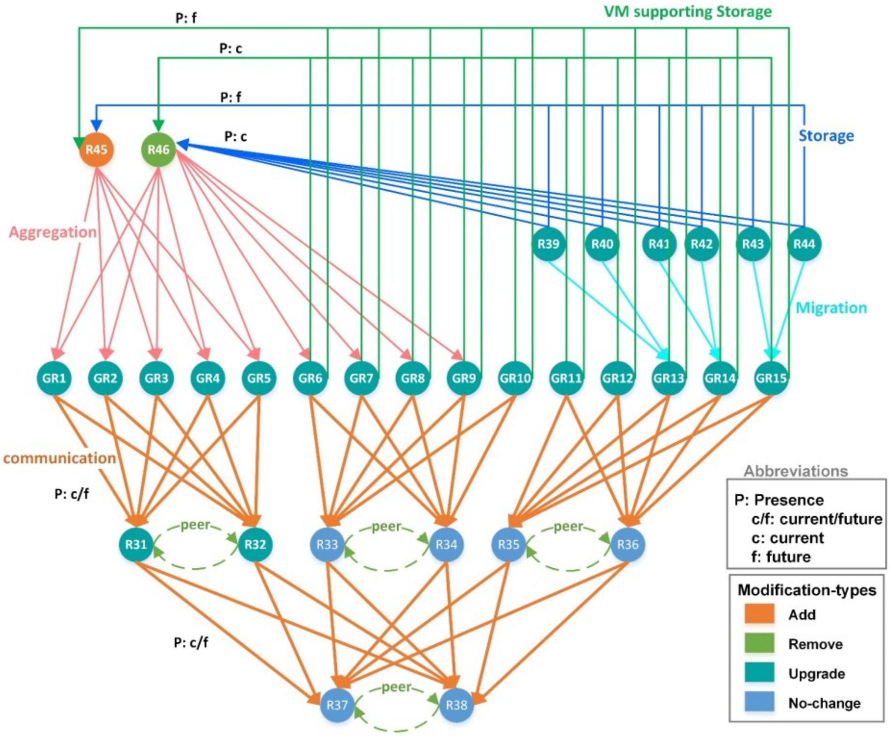


Figure 6.45. The CG of the illustrative scenario with failed upgrade action after the fourth iteration in step 3

Fifth Iteration

Similar to the previous iteration the RG have to be updated to evaluate the necessity of retry or undo operations. Since the number of failed attempts on R31 is reached to the max-retry threshold for the change set 2, the retry operation cannot be initiated on R31.

Thus, the switch represented by R31 is isolated. As a result of isolating this resource, the number of operational switches will change to 5, which is less than 6 minimum required number of operational switches indicated as undo-threshold for the change 1 of the change set 2. This means the undo operation for change set 2 is triggered. All the changes which are already applied to the resources of undo unit 2 (associated with change set 2) have to be undone. The undo unit 2 and its associated change set 2 is marked as failed.

As shown in Figure 6.23, the undo unit 2 includes the routers, switches and all the hosts. The actions-to-execute attributes of the resources belonging to undo unit 2 will be adjusted. For the resources that the changes of the change set 2 is already applied (i.e. R33 to R38), this adjustment is including the undo actions in the first execution-level of the actions-to-execute attribute of the resources to take them to the undo version. For the others that the changes of change set 2 is not applied yet (i.e. R16 to R32), the upgrade actions for the change set 2 will be removed from their actions-to-execute attribute. The modification-type of the resources will be updated according to the remaining changes to be applied to the resources. Note that R31 and R32 are already at the undo version, thus after removing the upgrade actions for the change set 2, there is no more remaining change to be applied on them (modification-type is ”No-change”). R31 will be released from isolation. The updated RG after this step is presented in Figure 6.46. In step 2, the CG will be updated as well. The updated CG after this step is presented in Figure 6.47.

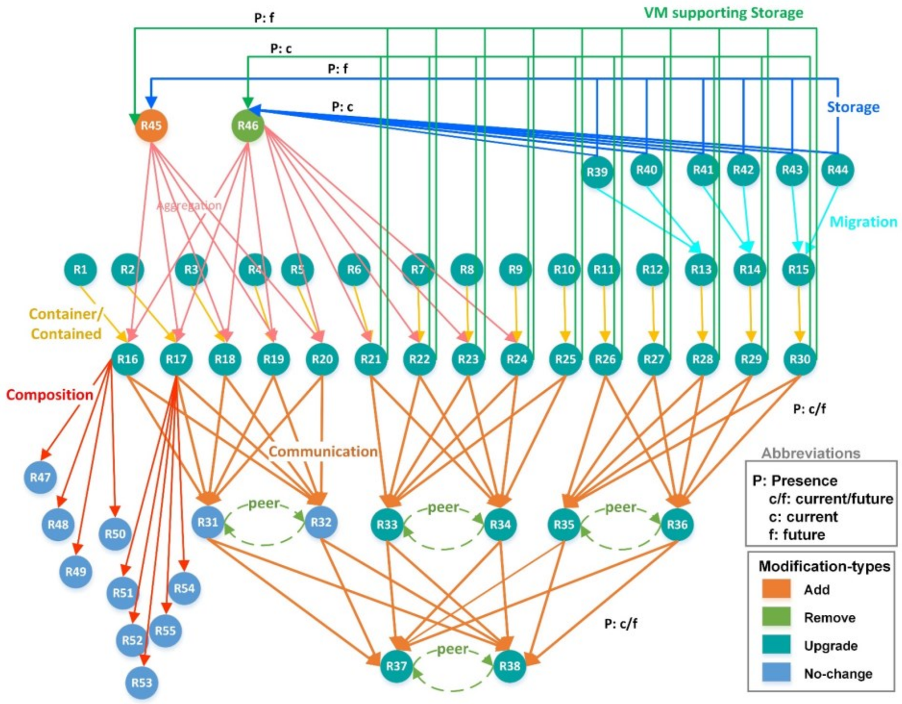


Figure 6.46. The RG of the illustrative scenario with failed upgrade action after fifth iteration in step 1

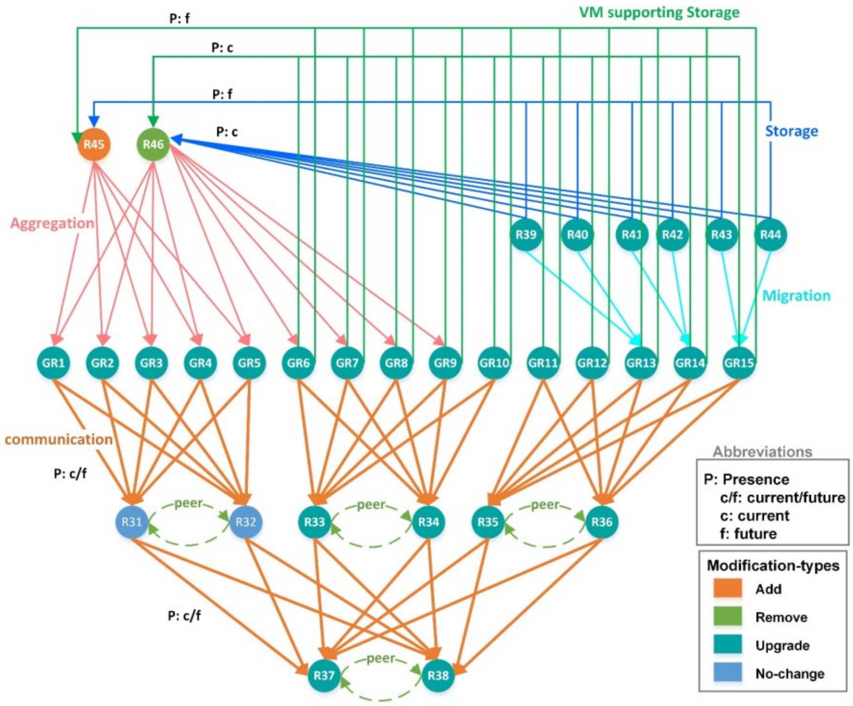


Figure 6.47. The CG of the illustrative scenario with failed upgrade action after fifth iteration in step 2

The initial batch includes one of the routers represented with R37 and R38, similar to the first iteration in the successful scenario described in Section 6.2.1.1. Note that due to the initiated undo operation for change set 2, there are changes (e.g. for undoing the change 2 of the change set 2) to be applied on previously upgraded resources. Elimination rule 4 enforces to upgrade R37 and R38 before other resources. While elimination rule 6 eliminates one of these peer resources to protect the availability of services provided by peers. We assume R37 remains in the initial batch. After calculation of the maximum number of compute hosts that can be taken out of service in fifth iteration (Z_5) and considering zero number of affected compute hosts during the upgrade of the initial batch, R37 is selected for the final batch. Thus, the change set 2 is undone on R37.

Sixth, Seventh, and Eighth Iterations

Similarly, in the sixth iteration R38 will be undone. In the seventh iteration, not only one out of peer switches with modification-type of "Upgrade" is going to be selected in the final batch, but also similar to the fifth iteration in the successful scenario described in Section 6.2.1.1, one of the constituent resource (GR1 to GR5) of R46 remains in the final batch as there is no change to be applied on their sponsor switches (R31 and R32). Let us assume the selected resources in this iteration are R33, R35, and GR1. In the eighth iteration R34, R36 and GR2 are upgraded.

Remaining Iterations

From here after, the selection of resources for the upgrade will continue similar to sixth iteration of the successful scenario (described in Section 6.2.1.1) until the end of the upgrade process.

6.2.2.3 *New upgrade requests during ongoing upgrades*

In this scenario we consider receiving a new upgrade request while the previously issued one is still in progress. Let us assume the administrator issues a new upgrade request consisting a change set to upgrade the routers to a new version (V3-R) while the second iteration of the successful scenario (described in Section 6.2.1.1) is in progress. Since the administrator aims to only upgrade the routers, the new change set (change set 3) has one change as presented in Table 6.5. We assume the administrator also specifies max-retry, max-completion-period, undo-threshold and undo version for the change set and its change as presented in Table 6.6. For simplicity, we assume the estimated time required for upgrade and recovering from possible failures for the change is equal to cooldown period for the tenants. Note that the max-completion-time for the change set is given for the sake of example and it does not reflect the time required for upgrading a router in the real deployment.

The upgrade coordinator takes them into account the new upgrade requests at the beginning of the next iteration. Hence, the change set 3 will be taken into account at the begging of the third iteration.

Table 6.5. Change set of the new upgrade request

Change sets	Changes
Change set 3	Change 1: Upgrade Routers (R1 and R2) from V2-R to V3-R

Table 6.6. Additional information provided by the administrator for the change set 3

Change sets	Changes	max-retry threshold (for set)	max-completion-period (for set)	undo-threshold (for change)	undo version (for change)
Change set 3	Change 1	3	1800 seconds	2	V2-R

Third Iteration

The new upgrade request is first added to the upgrade request model and a new undo unit (undo unit 3) is assigned for its change set. A new request graph (NRG) is created for the new upgrade request without considering any ongoing upgrades for capturing the new incompatibilities that may arise due to the new upgrade request. In this example, we assume the change set 3 will not introduce any additional incompatibilities to the system. Therefore, the upgrade units identified on the NRG will be similar as the ones in the RG. The actions-to-execute attributes of the routers represented by R37 and R38 include the upgrade actions for the change 1 in change set 3. The RG will be updated for these resources. Note that since the upgrade of R37 and R38 for the change set 2 was already completed, the actions-to-execute attributes of them were empty at the beginning of step 1. Now, after updating the RG, the actions-to-execute attributes of R37 and R38 will have one execution-level including the upgrade actions for the change set 3. The modification-type of these resources will be updated to “Upgrade”. The updated RG after this step is presented in Figure 6.48. In the step 2, the CG will be updated accordingly.

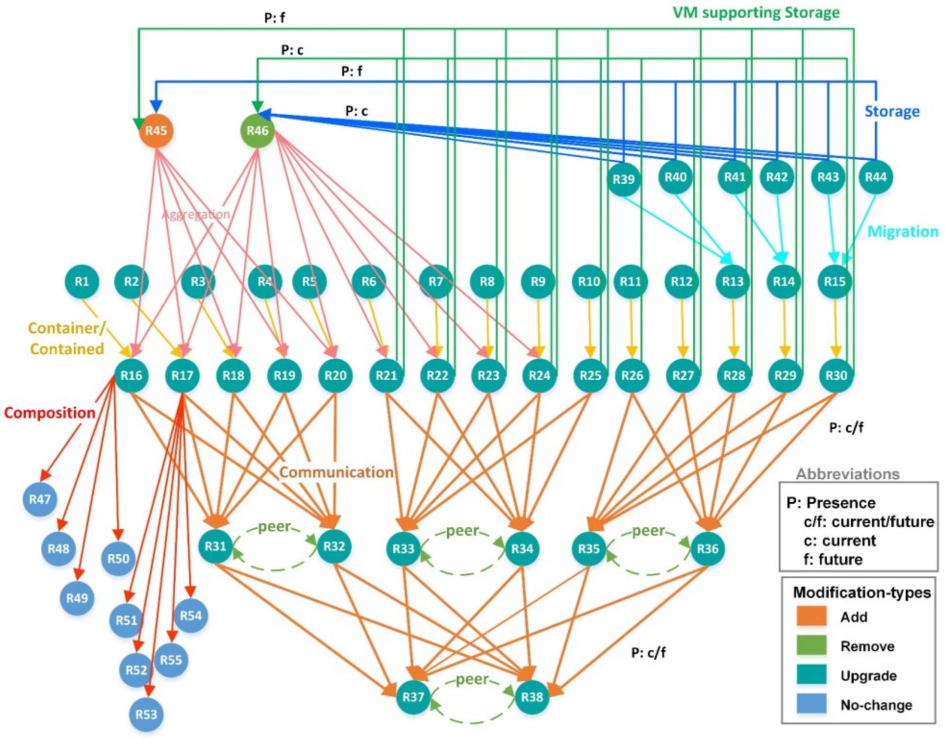


Figure 6.48. The RG of the illustrative scenario with new upgrade request in third iteration after step 1

In step 3 similar to the first iteration of this scenario, R37 will be upgraded. However, contrary to the first iteration which upgrades R37 to V2-R, in this iteration R37 will be upgraded from V2-R to V3-R.

Remaining Iterations

Hereafter, the selection of resources in the final batch in the remaining iterations will continue similar to the second iteration of the successful scenario (described in Section 6.2.1.1) until the end, till completion of all upgrade requests.

6.3 Summary

In this chapter, we presented the proof of concept developed for upgrading the IaaS compute and its application in a virtualized OpenStack cluster. This PoC is designed and partially developed within Ericsson to demonstrate the feasibility of our proposed framework in a real deployment. This PoC implementation is able to upgrade the IaaS compute nodes in a real system under SLA constraints for availability and elasticity.

In addition, in this chapter we presented the prototype implementation of our proposed upgrade approach applicable to upgrade of different kinds of IaaS resources. In this implementation the behaviour of the upgrade engine, responsible for applying the generated runtime upgrade schedules in the real system, is simulated to show the progress of the upgrade. Using this prototype implementation in each iteration, the upgrade actions for the runtime upgrade schedule are determined while considering SLA constraints of elasticity and availability. An illustrative example with different scenarios was used to demonstrate the handling of different challenges of upgrade in the cloud (e.g. dependencies, dynamicity and failure handling) and also the continuous delivery feature of our proposed approach.

Using the proof of concepts and the case studies presented as illustrative examples, we demonstrated that our upgrade management framework can upgrade the IaaS cloud system under SLA constraints of availability and elasticity while addressing the identified challenges. We also performed some experiments that show our approach does not introduce any outage at the application level for the tenants that are configured HA (i.e. have more than one VM), and it causes less SLA violations, compared to the rolling upgrade method with fixed batch sizes. Although the introduced outage at the application level for the rolling upgrade with batch size of one is similar to our approach, however the duration of upgrade using our approach is less than duration of upgrade with fixed batch size of one.

In order to realize the prototype implementation in a real deployment, the configuration information of different kinds of IaaS resources has to be gathered from the real system automatically. In addition, the identified upgrade actions for the schedule in each iteration have to be organized into a specific format (e.g. playbooks, recipes) based on the configuration management tool chosen as the upgrade engine.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented an upgrade management framework for automating the upgrade of IaaS cloud systems, under SLA constraints of availability and elasticity. Our upgrade management framework has two components, the upgrade coordinator to coordinate the upgrade process, and the upgrade engine to execute the necessary upgrade actions on the infrastructure resources. For the coordination of the upgrade process, we proposed an upgrade approach, which determines and schedules the necessary upgrade methods and actions appropriate for the upgrade requests in an iterative manner. In this approach, applicable to all kind of IaaS resources, the entire process is orchestrated to minimize the service disruption of the IaaS cloud system during the upgrade. We also evaluated several configuration management tools (e.g. Ansible, Salt, and Chef) as potential upgrade engines. As mentioned in Chapter 4, the upgrade engine can be any other engine capable of running upgrade actions on the IaaS resources.

An upgrade is initiated by an upgrade request which is composed of change sets requested by a system administrator indicating the desired changes in the IaaS cloud system. In addition to the initial change sets, our approach allows for new upgrade requests to be issued and taken into account during the upgrade process, which makes it suitable for continuous delivery. The upgrade actions required to upgrade each IaaS resource, the upgrade method appropriate for each subset of resources, and the batch of resources to upgrade in each iteration are determined automatically and applied in an iterative upgrade process.

The approach tackles in an integrated manner the challenges posed by the dependencies and the possible incompatibilities along dependencies, by the dynamicity of IaaS cloud systems, by potential upgrade failures, and by the amount of used extra resources.

To minimize the service disruption during the upgrade of different kinds of IaaS resources, existing dependencies must be respected. In this thesis, we have defined infrastructure resource information models for the purpose of the upgrade, and we have characterised infrastructure resource dependencies. A set of rules – called elimination rules – have been used to order the upgrade of different resources with respect to their dependency requirements. In addition, the upgrade method templates have been defined to specify the appropriate upgrade methods to subsystems for handling the potential incompatibilities along the resource dependencies. To minimize the duration of the upgrade, the resources that can be upgraded simultaneously (i.e. the batch of resources) are identified, and the appropriate upgrade methods are selected to upgrade the selected resources.

Since in each iteration, the batch of resources for the upgrade is selected according to the current state of the system with respect to the dependencies and the SLA constraints, the interferences between autoscaling and the upgrade process is mitigated. Furthermore, since the upgrade process is regulated based on the current state of the system, cloud providers can perform the upgrades gradually according to the state of the system, and they do not need to designate a maintenance window for performing the upgrades. In case of upgrade failures, localized retry and undo operations are issued automatically according to the failures and undo/retry thresholds indicated by the administrator. This feature provides the capability to undo a failed change set, while the upgrade proceeds with other change sets.

In our approach, to minimize the amount of additional resources used during the upgrade, we identify the subsystem where additional resources are required, and we only use the minimum amount as necessary. For example, instead of bringing up a complete IaaS system as a parallel universe, we use this method locally to upgrade the infrastructure resources supporting VM operations.

To have more confidence on the correctness of our approach, we provided an informal validation and a rigorous analysis of four important properties of our approach. The feasibility of our upgrade management framework in a real deployment is demonstrated by developing a proof of concept for upgrading the IaaS compute and its application in a virtualized OpenStack cluster. Furthermore, we presented the prototype implementation of our upgrade approach for all kinds of IaaS resources. We used an illustrative example with different upgrade scenarios to show that the upgrade of IaaS resources proceeds as expected in our prototype implementation, under SLA constraints of availability and elasticity.

We conducted some experiments to show how our approach works to respect the SLA constrains of availability and elasticity. The measurement results demonstrate that our approach does not introduce any outage at the application level for the tenants that are configure HA in the application level. Also, the results indicate that our approach avoids the outage at the application level and reduces SLA violations during the upgrade, compared to the rolling upgrade with fixed batch sizes.

7.2 Future work

In this section, we briefly discuss potential future research.

In the proposed approach, the upgrade of IaaS resources for the requested change sets are carried out in an iterative process according the current state of the system with respect to the dependencies and the SLA constraints. In our approach, we do not prioritize change sets based on the urgency of the upgrades and their required completion time. The final batch for the upgrade is selected in each iteration, regardless of the urgencies of the change sets. We assumed any subset of IaaS resources can be chosen from the initial batch, as long as their number of affected compute hosts are less than the maximum number of compute hosts that can be taken out of service in an iteration. In addition, if a change set requested by the administrator cannot be finalized within the maximum time allotted to complete all the changes of change sets, the undo operation is triggered for that change set. As a future work, heuristics can be considered to prioritize the selection of IaaS resources for the upgrade according to the maximum completion time of change sets. Urgent upgrades may impact the whole system if not addressed within a fixed time window. For such upgrades, the upgrade process may need to proceed even when there is a shortage of resources to guarantee scaling. Thus, SLA violation penalties may apply. A future work can target the optimization problem of such upgrades with multi objectives of minimizing the penalties of an IaaS provider for not meeting scaling requests according to the SLAs and the costs associated with delaying the urgent upgrades.

As mentioned in Chapter 5, the potential scaling-out requests are calculated based on the scaling policies indicated in the SLAs. Some cloud providers may not use a reactive rule-based autoscaling mechanism based on scaling policy parameters as presented in this thesis (e.g. cooldown time, scaling adjustment).

Although the approach presented in this thesis targets the upgrade of IaaS cloud systems, the principles of our approach can be reused for Software as a Service (SaaS) and Platform as a Service (PaaS) cloud as well, which can minimize the service disruption and SLA violation of these systems during the upgrade.

As mentioned in Chapter 6, a proof of concept has been developed for upgrading the IaaS compute and its application in a virtualized OpenStack cluster. In addition, a prototype implementation of our proposed approach is presented for the upgrade of all kinds of IaaS resources. From the realization perspective, a future work can involve putting our upgrade management framework and approach at work for the upgrade of all kind of IaaS resources and validating them in practice. This will require automatic collection of configuration information from an IaaS cloud system according to the infrastructure resource information models presented in Chapter 3. For example, in case of OpenStack cloud system this information has to be collected from metadata of different OpenStack services (e.g. Nova, Cinder, and Controller).

Bibliography

[1] National Institute of Standards and Technology, “NIST Cloud Computing Standards Roadmap,” NIST Special Publication 500 - 291, 2013.

[2] M. Toeroe and F. Tam, *Service availability principles and practice*. John Wiley and Sons Ltd publication, 2012.

[3] M. Nabi, M. Toeroe, and F. Khendek, “Availability in the cloud: State of the art,” *J. Netw. Comput. Appl.*, vol. 60, pp. 54–67, 2016.

[4] A. Undheim, A. Chilwan, and P. Heegaard, “Differentiated availability in cloud computing SLAs,” in *2011 IEEE/ACM 12th International Conference on Grid Computing*, 2011, pp. 129–136.

[5] M. Nabi, F. Khendek, and M. Toeroe, “Upgrade of the IaaS cloud: Issues and potential solutions in the context of high-Availability,” in *26th IEEE International Symposium on Software Reliability Engineering, Industry track*, 2015, pp. 21–24.

[6] N. Roy, A. Dubey, and A. Gokhale, “Efficient autoscaling in the cloud using predictive models for workload forecasting,” in *2011 IEEE 4th International Conference on Cloud Computing (CLOUD)*, 2011, pp. 500–507.

[7] Amazon Web Services, “Amazon EC2 Auto Scaling User Guide,” 2018. [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-dg.pdf. [Accessed: 05-Jul-2018].

[8] F. Paraiso, P. Merle, and L. Seinturier, “Managing elasticity across multiple cloud providers,” in *2013 International workshop on Multi-cloud applications and federated clouds - MultiCloud ’13*, 2013, pp. 53–60.

[9] Amazon Web Services, “UpdatePolicy Attribute,” 2019. [Online]. Available: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-updatepolicy.html. [Accessed: 05-Aug-2019].

[10] Amazon Web Services, “AWS::AutoScaling::ScheduledAction,” 2019. [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-as-scheduledaction.html. [Accessed: 23-Aug-2019].

[11] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud Computing and Grid Computing 360-Degree Compared,” in *2008 Grid Computing Environments Workshop*, 2008, pp. 1–10.

[12] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” pp. 7–18, 2010.

[13] Amazon, “Amazon EC2,” 2018. [Online]. Available: http://aws.amazon.com/ec2/. [Accessed: 30-Jul-2018].

[14] “Google App Engine,” 2018. [Online]. Available: https://cloud.google.com/appengine/.

[Accessed: 30-Jul-2018].

[15] "Salesforce," 2018. [Online]. Available: https://www.salesforce.com/. [Accessed: 30-Jul-2018].

[16] H. Alipour, Y. Liu, and A. Hamou-Lhadj, "Analyzing Auto-scaling Issues in Cloud Environments," *Proc. 24th Annu. Int. Conf. Comput. Sci. Softw. Eng. IBM Corp.*, pp. 75–89, 2014.

[17] F. L. Ferraris *et al.*, "Evaluating the auto scaling performance of flexiscale and amazon EC2 clouds," *Proc. - 14th Int. Symp. Symb. Numer. Algorithms Sci. Comput. SYNASC 2012*, pp. 423–429, 2012.

[18] "OpenStack." [Online]. Available: http://www.openstack.org/. [Accessed: 05-Aug-2019].

[19] OpenStack, "Heat documentation." [Online]. Available: http://docs.openstack.org/developer/heat/. [Accessed: 01-May-2019].

[20] H. Khazaei, M. Jelena, V. B.Misic, and N. Beigi Mohammadi, "Availability Analysis of Cloud Computing Centers," in *Communication Software, Service and Multimeda Symposium*, 2012, pp. 1981–1986.

[21] F. Longo, R. Ghosh, V. K. Naik, and K. S. Trivedi, "A Scalable Availability Model for Infrastructure-as-a-Service Cloud," in *IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, 2011, p. pp.335,346.

[22] M. Mihailescu, A. Rodriguez, and C. Amza, "Enhancing application robustness in infrastructure-as-a-service clouds," *Proc. Int. Conf. Dependable Syst. Networks*, pp. 146–151, 2011.

[23] Q. Zhang, M. F. Zhani, M. Jabri, and R. Boutaba, "Venice: Reliable virtual data center embedding in clouds," *IEEE INFOCOM 2014 - IEEE Conf. Comput. Commun.*, pp. 289–297, 2014.

[24] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible, "Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-Aware Virtual Machine Placement," in *IEEE International Conference on Services Computing*, 2011, pp. 72–79.

[25] A. Jahanbanifar, F. Khendek, and M. Toeroe, "Providing Hardware Redundancy for Highly Available Services in Virtualized Environments," *8th IEEE Int. Conf. Softw. Secur. Reliab.*, no. Vmm, pp. 40–47, 2014.

[26] Distributed-Management-Task-Force (DMTF), "Open Virtualization Format Specification," 2013. [Online]. Available: https://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.0.pdf. [Accessed: 10-Dec-2018].

[27] E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Comput.*, vol. 5, no. 4, pp. 46–55, 2001.

[28] T. Dumitras, P. Narasimhan, and E. Tilevich, "To Upgrade or Not to Upgrade Impact of

Online Upgrades across Multiple Administrative Domains," *ACM Int. Conf. Object oriented Program. Syst. Lang. Appl. (OOPSLA '10)*, pp. 865--876, 2010.

[29] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about It? Toward dependable, online upgrades in enterprise system," in *10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)*, 2009, vol. 5896 LNCS, pp. 349–372.

[30] T. Dumitras, "Cloud Software Upgrades : Challenges and Opportunities," in *2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA '11)*, 2011, pp. 1–10.

[31] T. Das, E. T. Roush, and P. Nandana, "Quantum Leap Cluster Upgrade," in *Proceedings of the 2nd Bangalore Annual Compute Conference (COMPUTE '09)*, 2009, pp. 2–5.

[32] X. Ouyang, B. Ding, and H. Wang, "Delayed switch: Cloud service upgrade with low availability and capacity loss," in *2014 IEEE 5th International Conference on Software Engineering and Service Science (ICSESS)*, 2014, pp. 1158–1161.

[33] T. Dumitras, "Dependable, Online Upgrades in Enterprise Systems," *24th ACM SIGPLAN Conf. Companion Object Oriented Program. Syst. Lang. Appl. (OOPSLA '09)*, pp. 835–836, 2009.

[34] T. Dumitra and P. Narasimhan, "Toward Upgrades-as-a-Service in Distributed Systems," in *10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)*, 2009.

[35] B. Calder *et al.*, "Windows Azure Storage : A Highly Available Cloud Storage Service with Strong Consistency," in *23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, vol. 20, pp. 143–157.

[36] Amazon Web Services, "AWS Elastic Beanstalk Developer Guide API Version 2010-12-01," 2010. [Online]. Available: http://awsdocs.s3.amazonaws.com/ElasticBeanstalk/latest/awseb-dg.pdf. [Accessed: 05-Aug-2019].

[37] D. Sun, D. Guimarans, A. Fekete, V. Gramoli, and L. Zhu, "Multi-objective Optimisation for Rolling Upgrade Allowing for Failures in Clouds," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, 2015, pp. 68–73.

[38] V. Gramoli, L. Bass, A. Fekete, and D. W. Sun, "Rollup: Non-Disruptive Rolling Upgrade with Fast Consensus-Based Dynamic Reconfigurations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, pp. 2711–2724, 2016.

[39] D. Sun *et al.*, "Quantifying failure risk of version switch for rolling upgrade on clouds," *2014 IEEE Fourth Int. Conf. Big Data Cloud Comput.*, pp. 175–182, 2014.

[40] D. Sun, A. Fekete, V. Gramoli, G. Li, X. Xu, and L. Zhu, "R2C: Robust Rolling-Upgrade in Clouds," *IEEE Trans. Dependable Secur. Comput.*, pp. 1–1, 2016.

[41] K. Liu, D. Zou, and H. Jin, "UaaS: Software Update as a Service for the IaaS Cloud," *Proc. - 2015 IEEE Int. Conf. Serv. Comput. SCC 2015*, pp. 483–490, 2015.

[42] Distributed-Management-Task-Force(DMTF), “Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol: An Interface for Managing Cloud Infrastructure.” .

[43] Open-Grid-Forum, “Open Cloud Computing Interface - OCCI.” [Online]. Available: http://occi-wg.org/. [Accessed: 01-May-2015].

[44] “Cloud Application Management for Platforms Version 1.1.” [Online]. Available: http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.

[45] OASIS, “Topology and Orchestration Specification for Cloud Applications (TOSCA).” [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf.

[46] R. Jain and S. Paul, “Network Virtualization and Software Defined Networking for Cloud Computing : A Survey,” *IEEE Commun. Mag.*, no. November, pp. 24–31, 2013.

[47] Intel, “PCI-SIG Single Root I / O Virtualization (SR-IOV) Support in Intel ® Virtualization Technology for Connectivity,” 2008.

[48] H. M. Tseng, H. L. Lee, J. W. Hu, T. L. Liu, J. G. Chang, and W. C. Huang, “Network virtualization with cloud virtual switch,” *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, pp. 998–1003, 2011.

[49] “ESXi: Bare Metal Hypervisor,” 2018. [Online]. Available: https://www.vmware.com/ca/products/esxi-and-esx.html. [Accessed: 01-Oct-2018].

[50] “Ceph.” [Online]. Available: https://ceph.com/. [Accessed: 05-Jan-2017].

[51] “Ansible.” [Online]. Available: http://www.ansible.com/home. [Accessed: 20-Aug-2019].

[52] “OpenSAF - The Open Service Availability Framework.” [Online]. Available: http://opensaf.sourceforge.net/documentation.html. [Accessed: 20-Aug-2019].

[53] P. Heidari, M. Hormati, M. Toeroe, Y. Al Ahmad, and F. Khendek, “Integrating OpenSAF High Availability Solution with OpenStack,” in *Services (SERVICES), 2015 IEEE World Congress on*, 2015, pp. 229–236.

[54] “Puppet labs.” [Online]. Available: https://puppetlabs.com/?_ga=1.122891208.2105885589.1429055377. [Accessed: 01-May-2018].

[55] “Ruby.” [Online]. Available: https://www.ruby-lang.org/en/. [Accessed: 01-May-2019].

[56] “Chef.” [Online]. Available: https://www.chef.io/chef/. [Accessed: 01-May-2018].

[57] “Salt.” [Online]. Available: http://docs.saltstack.com/en/latest/. [Accessed: 01-May-2018].

[58] “Python.” [Online]. Available: https://www.python.org/. [Accessed: 01-May-2019].

[59] “Mistral.” [Online]. Available: https://docs.openstack.org/mistral/latest/. [Accessed: 01-Jun-2019].

[60] "TaskFlow." [Online]. Available: https://wiki.openstack.org/wiki/TaskFlow. [Accessed: 01-May-2018].

[61] "The Official YAML Web Site." [Online]. Available: http://yaml.org/. [Accessed: 01-May-2019].

[62] M. Nabi, M. Toeroe, and F. Khendek, "Rolling upgrade with dynamic batch size for Iaas cloud," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016, pp. 497–504.

[63] "VMware vSAN." [Online]. Available: https://docs.vmware.com/en/VMware-vSAN/index.html. [Accessed: 05-Jan-2018].

[64] H. Pham, "System Reliability Concepts," *Syst. Softw. Reliab.*, pp. 9–75, 2006.

[65] L. Tomás and J. Tordsson, "Improving cloud infrastructure utilization through overbooking," *Proc. 2013 ACM Cloud Auton. Comput. Conf. - CAC '13*, p. 1, 2013.

[66] L. Tomas and J. Tordsson, "An autonomic approach to risk-aware data center overbooking," *IEEE Trans. Cloud Comput.*, vol. 2, no. 3, pp. 292–305, 2014.

[67] "Vagrant." [Online]. Available: https://www.vagrantup.com/. [Accessed: 01-Oct-2018].

[68] "vagrant-ansible-openstack." [Online]. Available: https://github.com/dguerri/vagrant-ansible-openstack.

[69] "The Go Programming Language," 2018. [Online]. Available: https://golang.org/. [Accessed: 10-Oct-2018].

[70] "gophercloud: The OpenStack SDK for Go," 2018. [Online]. Available: http://gophercloud.io/. [Accessed: 10-Oct-2018].

[71] "QEMU." [Online]. Available: https://www.qemu.org/. [Accessed: 20-Aug-2019].

[72] A. T. Foundjem, "Towards Improving the Reliability of Live Migration Operations in OpenStack Clouds.," Ecole Polytechnique de Montreal, 2017.

[73] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, "SLA-based virtual machine management for heterogeneous workloads in a cloud datacenter," *J. Netw. Comput. Appl.*, vol. 45, pp. 108–120, 2014.

[74] "JGraphT a Java library of graph theory data structures and algorithms." [Online]. Available: https://jgrapht.org/. [Accessed: 05-Dec-2018].

Appendix I

Table A.1 Parameters used in the proposed approach

Symbols	**Description**	**Symbols**	**Description**
K, K'	Host capacity in terms of VMs (before and after a hypervisor upgrade)	$M_{network}$	Set of hosts dedicated to networking services
N_i	Number of tenants in iteration *i*	$M_{controller}$	Set of hosts dedicated to controller services
min_n	Minimum number of VMs for tenant *n*	$M_{computeForOldVM}$	Set of compute hosts capable of hosting VMs of the old version
max_n	Maximum number of VMs for tenant *n*	$M_{computeForNewVM}$	Set of compute hosts capable of hosting VMs of the new version
c_n	Cooldown time for tenant *n*	$M_{usedCompute}$	Set of in-use compute hosts
s_n	Scaling adjustment in terms of VMs per cooldown time for tenant n	$M_{usedComputeForOldVM}$	Set of in-use compute hosts with VMs of the old version
S_i	Maximum scaling adjustement requests per tenant that may occur during iteration *i*	$M_{usedComputeForNewVM}$	Set of in-use compute hosts with VMs of the new version
T_i	The time required to upgrade and to recover from potential failures of the batch of iteration *i*	$ScalingResv_{forOldVM}$	Number of compute hosts reserved for scaling of VMs of the old version
F	The number of compute host failures to be tolerated during an iteration	$ScalingResv_{forNewVM}$	Number of compute hosts reserved for scaling of VMs of the new version
A_i	Number of tenants who might scale out on hosts compatible with the old VM version in iteration *i*	$FailoverResev_{forOldVM}$	Number of compute hosts reserved for failover of VMs of the old version
Z_i	The maximum number of compute hosts that can be taken out of service in iteration *i*	$FailoverResev_{forNewVM}$	Number of compute hosts reserved for failover of VMs of the new version
V_i	The total number of VMs to be upgraded in iteration *i*	$MinHostReqConf_{oldStorage}$	Minimum required number of storage hosts for the old configuration of the virtual storage
W_{ij}	The batch size in terms of VMs where each VM belongs to a different anti-affinity group in the main iteration *i* and sub-iteration *j*	$MinHostReqConf_{newStorage}$	Minimum required number of storage hosts for the new configuration of the virtual storage
$M_{Storage}$	Set of hosts eligible to participate in the creation of virtual storage (storage hosts)	$MinHostReqCap_{oldStorage}$	Minimum required number of storage hosts for data of VMs of the old version
$M_{compute}$	Set of hosts eligible to provide compute services (compute hosts)	$MinHostReqCap_{newStorage}$	Minimum required number of storage hosts for data of VMs of the new version

Appendix II – Elimination Rules

Elimination rule 1: this elimination rule guarantees keeping the current VM service available by avoiding selection of in-use physical hosts (hosting VMs) and VMs for the upgrade. As mentioned earlier, since VMs represent the service the IaaS cloud system provides, they are upgraded separately in Step 4, as described in Chapter 5, by considering different criteria. This rule removes from G_{batch} all the resources involved in migration dependency.

Elimination rule 2: This elimination rule guarantees the satisfaction of dependency requirements before removing a resource from the system. It applies to the resources in the G_{batch} with modification-type of "Remove", which means they will be removed from the configuration after completion of the upgrade process.

- If the resource is not a VM supporting infrastructure resource: the resource can stay in the G_{batch} only if there isn't any resource with any modification-type dependent on this resource. This means there isn't any sponsorship dependency towards the resource which is going to be removed.
- If the resource is a VM supporting infrastructure resource: the resource can remain in G_{batch} only if following conditions are valid: 1) there are only VM supporting infrastructure dependencies towards the resource which is going to be removed, and 2) its related resource (if any) has modification-type of "No-change", which means it is already added to the system.

Elimination rule 3: This elimination rule guarantees the satisfaction of dependency requirements before adding a resource to the system. It applies to the resources in the G_{batch} with modification-type of “Add”, which means they are going to be added to the configuration after completion of the upgrade process.

- A resource with “Add” modification-type can stay in G_{batch}, if at least one sponsor of the resource from each sponsorship dependency (except aggregation dependency) is got modified or added in previous iterations. This can be determined based on the modification-type of the sponsor resources, meaning if the modification-type of at least one sponsor is “No-change”. This will guarantee the existence of at least one ready sponsor to satisfy the requirements of the dependent resource before adding to the system.
- An aggregate resource with “Add” modification-type can stay in G_{batch}, if at least required number of its sponsors (constituent resources) is ready to satisfy the requirements of the aggregated resource. This information is extracted from the infrastructure component descriptions of the product to be installed as aggregate resource in the system. If the number of constituent resources with the modification-type of “No-change” is equal to the minimum number of constituent resources for an aggregate resource, the aggregate resource can stay in the G_{batch}; otherwise, it will be eliminated.

Elimination rule 4: This elimination rule guarantees the enforcement of compatibility requirements of sponsorship dependencies between resources. It will be applied if either the dependent resource or the sponsor resource in a sponsorship dependency is in the G_{batch}. Note that the incompatibilityFactor of the dependency between the resources is “false”, which means they belong to different upgrade units. In the description of this elimination rule, the activation-status of the resource is “true”, unless it is specified otherwise. Based on the presence parameter of the dependencies following cases have to be considered:

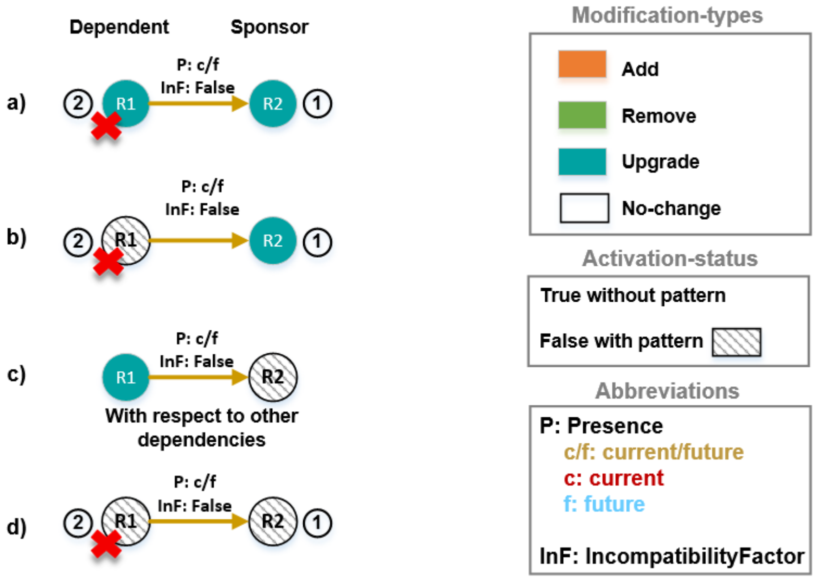


Figure A.1. Elimination rule 4 - case 1

- ***Case 1- if the presence of the sponsorship dependency is "current/ future":*** the order of the upgrade will be dependent on the modification-type and activation-status of the resources. Figure A.1 shows different possible situations of this case and the order of the upgrade of resources. Note that the resource indicated with number 2 in Figure A.1 will be eliminated from G_{batch}.

  a) If both dependent and sponsor resources have modification-type of "Upgrade", the sponsor has to be upgraded before the dependent, as shown in Figure A.1-a. So, elimination rule 4 will remove R1 (dependent resource) from G_{batch}.

  b) If the sponsor resource has modification-type of "Upgrade", and the dependent resource has modification-type of "No-Change" and activation-status of "false", the elimination rule 4 will remove R1 (dependent resource) from G_{batch}, as shown in Figure A.1-b. This guarantees having a compatible sponsor before activating the dependent resource.

c) If the sponsor resource has modification-type of “No-change” and activation-status of “false”, and the dependent resource has modification-type of “Upgrade”, the upgrade order of them has to be decided with respect to other dependencies of the resources. This case is shown in Figure A.1-c.

d) If both dependent and sponsor resources have modification-type of “No-change” and activation-status of “false”, elimination rule 4 will eliminate the dependent resource (R1), as shown in Figure A.1-d. This will guarantee existence of a compatible sponsor before activating the dependent resource.

- ***Case 2- if the presence of the sponsorship dependency is “current”:*** the order of the upgrade will depend on the modification-type of resources. Figure A.2 shows different possible situations of this case and the order of the upgrade of the resources. The resource with the second order will be eliminated from G_{batch}.

a) If both dependent and sponsor resources have modification-type “Upgrade”, the dependent has to be upgraded before sponsor, as shown in Figure A.2-a. Note that due to upgrade of R1 the dependency of R1 towards R2 will be removed, thus the

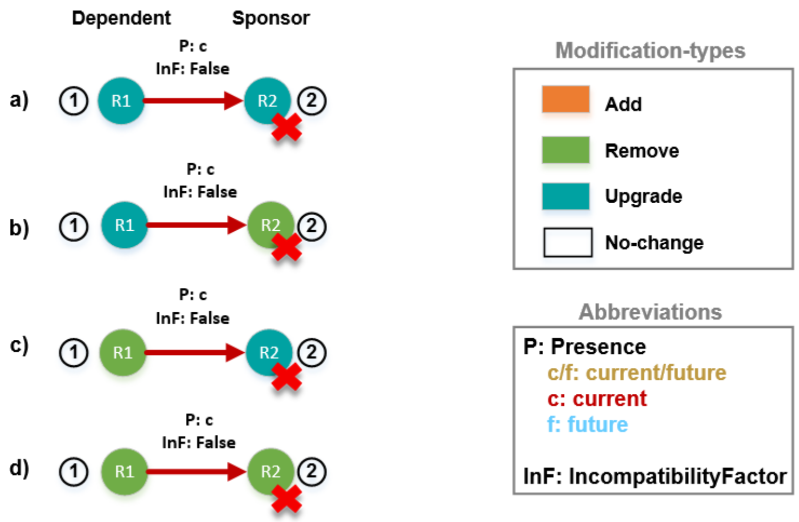


Figure A.2. Elimination rule 4 - case 2

dependent has to be upgraded before sponsor resource. The elimination rule 4 will remove R2 (sponsor resource) from G_{batch}.

b) If the sponsor resource has modification-type "Remove" and the dependent resource have modification-type "Upgrade", the dependent resource (R1) has to be upgraded before removing the sponsor resource (R2), as shown in Figure A.2-b. As a result of upgrading the dependent resource (R1), the dependency will be removed.

c) If the sponsor resource has modification-type "Upgrade" and the dependent resource has modification-type "Remove", except in case of aggregation dependency, the dependent resource (R1) has to be removed before upgrading the sponsor one (R2), as shown in Figure A.2-c. Note that in case of aggregation dependency, if the aggregate resource is a VM supporting infrastructure resource which is getting upgraded using PPU method, the old configuration of the aggregate resource can only be removed once all the VMs have been migrated from the old hosts compatible with the old configuration to the new one.

d) If both dependent and sponsor resources have modification-type "Remove", the dependent resource (R1) has to be removed before sponsor (R2), as shown in Figure A.2-d.

- ***Case 3- if the presence of the sponsorship dependency is "future":*** the order of the upgrade will be dependent on the modification-type of the resources. Figure A.3 shows different possible situations of this case and the order of the upgrade of the resources. The resource indicated with number 2 in Figure A.3 will be eliminated from G_{batch}.

  a) If both dependent and sponsor resources have modification-type of "Upgrade", the sponsor resource (R2) has to be upgraded before the dependent resource (R1), as shown in Figure A.3-a. This ensures having compatible version of sponsor ready, before upgrading the dependent one.

b) If the sponsor resource has modification-type of "Add" and the dependent resource has modification-type "Upgrade", the sponsor resource (R2) has to be added before upgrading the dependent resource (R1) due to sponsorship dependency requirements, as shown in Figure A.3-b.

c) If the sponsor resource has modification-type "Upgrade" and the dependent resource has modification-type "Add", except in case of aggregation dependency, the sponsor resource (R2) has to be upgraded before adding the dependent one (R1), as shown in Figure A.3-c. Note that in case of aggregation dependency, if the minimum requirement of dependent resource (the aggregate) in terms of constituent resources is satisfied, the dependent resource can be added before finishing upgrade of all sponsor resources (constituent resources).

d) If both dependent and sponsor resources have modification-type "Add", again the sponsor resource has to be added before the dependent one to ensure compatibility requirements of the dependency, as shown in Figure A.3-d.

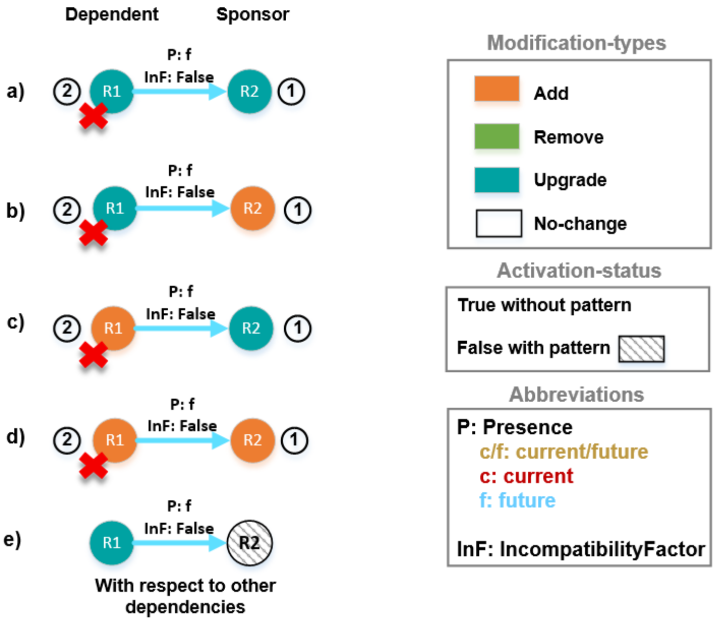


Figure A.3. Elimination rule 4 - case 3

e) If the sponsor resource has modification-type “No-Change” with activation-status of “false”, and the dependent resource has modification-type “Upgrade”, the upgrade order of them has to be decided with respect to other dependencies of the resources. This case is shown in Figure A.3-e.

Elimination rule 5: This elimination rule guarantees the correct order of upgrading resources with respect to the upgrade method associated with the upgrade unit of the first execution level in the actions-to-execute attribute of the resources. It applies to the resources of upgrade units which their associated upgrade method is split mode, modified split mode, or modified split mode with multiple constituent resources. In another terms, there are potential incompatibilities during the upgrade of resources of the upgrade unit.

- Based on the associated upgrade method and the upgrade status of the resources in an upgrade unit, the resources that cannot be upgraded or activated yet will be removed from G_{batch}. For example, let us assume the upgrade method of an upgrade unit is split mode method and none of the resources of the upgrade unit has been upgraded yet. As mentioned in Section 5.1.4, the resources of such an upgrade unit will be divided into two partitions, to be upgraded one at a time. This elimination rule will remove the resources of the second partition of the upgrade unit as non-suitable candidates. Thus, the resources of the first partition will remain as candidate resources in G_{batch}. However, if the resources of the first partition are already upgraded and deactivated, this elimination rule will allow resources of both partitions to remain in G_{batch}. The resources of the first partition will potentially be activated, and the resources of the second partition will potentially be upgraded in this iteration. Note that as we mentioned earlier, the deactivation of the second partition is a prerequisite for activation of first partition. This will be indicated during the generation of the upgrade schedule, according to the associated upgrade method of the upgrade unit.

- The resources with new upgrades will be removed from G_{batch}, if the upgrades of their previously associated upgrade units still in progress on other resources of the upgrade unit. This will ensure completing the upgrade of all the resources of an upgrade unit with the associated upgrade method, before applying upgrade actions of the new upgrade units on the resources. As mentioned in Chapter 4, after a successful execution of all the upgrade actions of the first execution-level for a resource, the execution-level (with all its upgrade actions) is removed from the list of execution-levels of the actions-to-execute of the resource. Thus, first execution level in the actions-to-execute attribute of a resource may be associated with a new upgrade unit due to new upgrade request. This elimination rule ensures that all the resources of an upgrade unit are upgraded with respect to previously issued upgrade requests, before applying the new ones.

Elimination rule 6: This elimination rule guarantees the availability of services provided by peer resources (with modification-type of "Remove" or "Upgrade"). It applies to resources in the G_{batch} that are peers or their dependent (direct or indirect) resources are peers. Note that the presence of the edges representing peer dependencies and symmetrical dependencies are "current/ future" or "current", meaning the peer dependency exist in the current configuration between resources and their dependents. Note that there isn't any limitation for peer resources with modification-type of "Add".

- Only one active resource out of peer paths (i.e. peer resource and its dependent resources) can stay in the G_{batch}. Note that the resources with deactivated status (i.e. need to be activated) will be evaluated according to elimination rule 5, which considers the upgrade method associated with the upgrade unit the resource belongs.

- Only maximum of one constituent resource of aggregation dependency with aggregate resource can stay in the G_{batch}, as long as the minimum resource requirement of the aggregate resource with respect to its configuration and possibly the data stored will not be violated, otherwise all of constituent resource of the aggregation dependency will be eliminated. The exception is when the aggregate resource is a VM supporting infrastructure resource (with modification-type of "Remove") and following conditions are valid: 1) there is no dependency towards the aggregate resource except VM supporting infrastructure dependency and 2) the related resource (if any) of the aggregate resource has modification-type of "No-change". In this case, if the aggregate resource with modification-type of "Remove" is in the G_{batch}, all its constituent resources can stay in the batch as well. This happens when removing the old version of infrastructure supporting resource while using the PPU method. Note that as mentioned in Chapter 3, constituent resources are peers, so this elimination rule will guarantee the availability of services (i.e. aggregate resource) provided by constitute resources.

Elimination rule 7: This elimination rule guarantees the satisfaction of the resource requirements of the *PPU* method used for upgrading a VM supporting infrastructure resource when it cannot be upgraded in place without impacting its services. As mentioned in Chapter 4, to maintain in parallel both the old and the new configurations of the VM supporting infrastructure resource additional resources may be required. If these cannot be provided using available resources, the administrator is asked to provide additional resources. Until these resource requirements are not satisfied, all the resources with changes related to the upgrade of the VM supporting infrastructure resource are eliminated from G_{batch}. Note that the resources contributing to the old configurations of the VM supporting infrastructure can remain in the G_{batch} only if they can be taken out of service without impacting the requirements of old configuration, otherwise additional resources will be required for their upgrade.

In our illustrative example given in Figure 5.1, the PPU method is used to upgrade the VM supporting virtual shared storage from VSAN to Ceph as the new and the old versions of the virtual shared storage are incompatible. To keep the continuity of the VM supporting service (e.g. VM live migration and failover) during the upgrade, the old configuration of the virtual shared storage (i.e. VSAN) has to remain operational until the new configuration (i.e Ceph) is ready for use. In addition, the compute hosts hosting the VMs need to be partitioned into those compute hosts compatible with the old version of the virtual shared storage (old partition) and those compute hosts compatible with the new version of the shared storage (new partition). To complete this upgrade, data conversion is also necessary and it is performed as the VMs are migrated from the old partition to the new. Once all the VMs have been migrated as well as completing the related data migration, the old configuration of the virtual shared storage can be safely removed.

To guarantee the continuity of VM services during the upgrade of the shared storage, we need to meet the minimum resource requirements for both the old and the new virtual shared storages with respect to their configurations and the data stored. For this reason, we need to have enough physical storage hosts to keep the old configuration of the storage alive while bringing up the configuration of the new. We evaluate whether the current system has enough storage hosts using equation (7).

$$\left|M_{storage} - M_{usedCompute}\right| \geq max(MinHost\,Re\,q\,Conf_{oldStorage}, MinHost\,Re\,q\,Cap_{oldStorage}) + \\ + max(MinHost\,Re\,q\,Conf_{newStorage}, MinHost\,Re\,q\,Cap_{newStorage}) \quad (7)$$

$|M_{Storage}\text{-}M_{usedCompute}|$ represents the number of storage hosts that are not in use as compute hosts. This number should be equal to or greater than the minimum number of hosts required to support both the old and the new storage configurations during the upgrade. If equation (7) is satisfied, the resources with upgrade actions related to the undo unit associated with virtual storage upgrade remain in G_{batch}. Otherwise, applying the elimination rule will remove these resources from G_{batch} as non-suitable candidates. Since the same check is performed in each subsequent iteration, whenever the additional number of storage hosts becomes available to fulfill this requirement, these resources will remain in the G_{batch} as suitable candidates. Note that as the upgrade proceeds the number of available resources may change due to failures or scaling operations on compute hosts, but also if additional hosts are provided. Thus, in any iteration when equation (7) is not satisfied, this elimination rule will remove from G_{batch} the resources related to the upgrade of VM supporting infrastructure resource (i.e. their upgrade will be paused) until the required resources will become available (again).

As mentioned earlier, the communication dependencies are realized by link resources in the system and they may need to be upgraded as well. Since upgrading a dependency impacts the dependent resource, we evaluate the dependency requirements for the upgrade of communication dependencies (i.e. link resource) as upgrade of its dependent resource. Thus, a communication dependency can stay in the G_{batch} only if its dependent resource can potentially stay in the G_{batch} according to our defined elimination rules. Note that the exception is in case of having peer link resources (i.e. more than one communication dependencies between two resources). In this case, even though the dependent resource cannot stay in the batch, one out of the peer links can stay in the G_{batch} at a time, regardless of upgrade limitation of their dependent resources.