4 SCS Deployment Infrastructure on Cloud Infrastructures

We defined the *deployment process* as a set of inter-related *activities* to make a piece of software ready to use. To get an overview of what this means in practice we explain briefly a basic scenario of the deployment of a distributed component-based application, managed by a *deployment infrastructure*, on a *cloud infrastructure*. Our scenario considers a distributed component-based application, an executed deployment infrastructure , and a cloud infrastructure. The deployment process uses planning and installation activities. A description file describes the computational resources requirements for the application. Also, the deployment infrastructure is aware within the availability of the cloud infrastructure. The planning activity loads the description file and send a request to the cloud infrastructure. The installation activity waits for access to the cloud infrastructure and deploys the application on it. This example hides many challenges and research opportunities of both the deployment process and cloud infrastructures.

This chapter describes how we extended the SCS Deployment Infrastructure to support the deployment on cloud infrastructures. Section 4.1 describes the challenges faced in the development of this work. Section 4.2 describes the *elements* developed to enable the deployment on cloud infrastructures. Section 4.3 explains the configurations of the SCS Deployment Infrastructure on the cloud, and our proposed architecture.

4.1 Challenges

As described in subsection 2.2.1, a generic deployment process of distributed component-based applications can be divided into ten activities. We start this section within an overview of how deployment activities could be extended to use cloud infrastructures. Figure 4.1 shows a generic deployment process of distributed component-based applications designed to consume cloud resources. We propose the use of cloud infrastructures for both, the *deployment infrastructure* and the *target environment*. The packaging activity could start up or reuse a virtual machine instance to run the packager service. Publishing, installing, updating, and retiring activities make use of a repository service, where component installation packages are stored. This component repository service could execute in its own virtual machine instance. The installing activity uses a cloud infrastructure as the target environment, and an installation configuration can range from a high-level description to a very customized and detailed configuration.

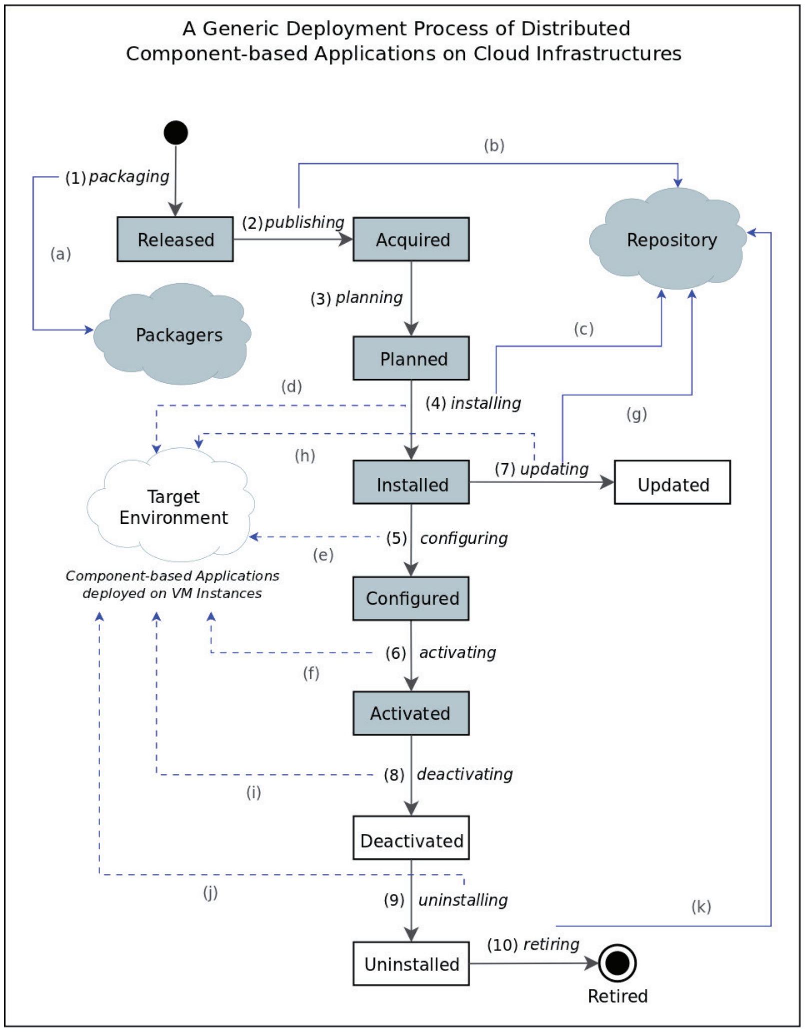

Figure 4.1: A Generic Deployment Process of distributed Component-based Applications on Cloud Infrastructures

The configuring and activating activities need to access to the target environment to make the application ready to use. Deactivating and uninstalling activities make the application unavailable and remove all components from the target environment, respectively. The retiring activity retires the components from the Repository Service.

We use the SCS Deployment Infrastructure to deploy distributed component-based applications. Thus, our first task is to extend the planning activity to support the use of cloud infrastructures as a target environment. In addition to extending planning activity, we could also execute the packaging and publishing activities in the same infrastructure. This means deploying the *Deployment Infrastructure* on a cloud infrastructure, thereby obtaining an experimental Platform as a Service(PaaS). We describe in the next subsections the specific challenges to extend our *SCS Deployment Infrastructure* for distributed component-based applications on the cloud.

4.1.1 Target Environment Requirements

Deployment actors need to define a target environment to deploy their applications. Hence, we need to provide a way to specify a custom target environment that takes advantages of cloud resources. To support several cloud infrastructure configurations we need to set at least three parameters: a *cloud deployment model*, a *cloud platform*, and a *cloud initial state*. For example, if a cloud administrator has a private cloud available then he could define a minimal configuration as follows: set a deployment model parameter to private; set a cloud platform parameter to the type of cloud platform installed or utilized; and set parameters related to the cloud platform such as instances types, virtual machine images, security, etc. To specify a cloud-based customized target environment we could reuse previous parameters. Therefore, we need to organize these parameters and propose a flexible method to write groups of parameter/value.

4.1.2 Cloud Infrastructure API

Building a flexible and scalable cloud API is currently a challenge because an official cloud API standard does not yet exist. The development of cloud APIs has been highly discussed because many organizations and consortiums are trying to determine a standard cloud API specification. Therefore, we need to design and develop a cloud API compatible with current cloud platforms and the SCS Deployment Infrastructure.

4.1.3 User Management

User management is an authentication feature and a critical subject in software systems. Cloud Computing needs a mature user management model because features such as SLAs, billing, security, etc., share a complex model and require an efficient user management. We identify this challenge because deployment infrastructure does not consider any rule for managing users. Therefore, we need to propose a user management model for the deployment infrastructure, and map it to user management tools for cloud infrastructures.

4.1.4 Setting up a Cloud Computing Environment

Setting up a cloud computing environment could be a challenge, this depends on the cloud software stack selected and our lefel of skills in technology related to cloud computing. To install and configure a private cloud we need a cloud platform. Today we have a list of open-source cloud platforms widely acceptanced by users. In section 2.1.1 we studied four cloud deployment models. Private and public clouds are basic forms of deployment models; hybrid and community clouds are a combination of private and public cloud. With the intention of building a flexible and scalable cloud platform, we aim to work with both a private and public clouds. Thus, we need to set up at least an operative private cloud and configure the access to a public cloud.

4.2 Cloud Resource Provisioning

In chapter 3 we explained the *Deployment Infrastructure* for SCS components. We highlighted the *DeployManager* service, which is responsible for instantiating and executing the deployment plans, containing references to the applications, target environment, repository service, and packager service. Later, these plans are executed on a local network. We added support to enable the use of cloud resources for the target environment as well as the deployment infrastructure. To enable the deployment on cloud infrastructures, we need access to a *cloud platform* using an appropriate *cloud interface*.

We have two options to start testing the consumption of cloud resources: choose a cloud provider (public cloud) and request a cloud account, or set up our own cloud infrastructure (private cloud). We decided to work with both public and private clouds, these two options give more flexibility for deployment actors to deploy applications. Although it implies more work, we aim to get a combination of flexible and scalable multi-cloud infrastructures using commodity hardware. Therefore, on the one hand, setting up and using a private cloud will enable us to build a scalable cloud infrastructure, where we can add more cloud nodes according the application's requirements. On the other hand, a public cloud allows us to experiment with large-scale infrastructures. We have adopted *OpenStack* as our main cloud platform to set up a private cloud, and *Amazon Web Services* as public cloud.

To consume either public or private clouds we need a cloud interface to manage the computational resources. Current available interfaces include command-line tools, cloud APIs, and web-based management applications. To complete the integration with our deployment infrastructure it was necessary to develop a customized cloud API. This cloud API was utilized to extend the deployment infrastructure services adding support to instantiate or consume virtual machine instances. To provide a flexible way of specifing the cloud resources required for deployment we propose the use of policies. We also apply the same policy approach to set other configurations such as cloud infrastructure, management of users, deployment infrastructure, and applications. Also, we keep the incremental level of details approach like DeployManager service, as explained in subsection 3.3.1.

This section is divided into five subsections. Subsection 4.2.1 explains the main details considered to make our cloud infrastructures available. Subsection 4.2.2 describes our proposed cloud API to instantiate cloud resources from our deployment infrastructure. Subsection 4.2.3 details the API extensions implemented to support the deployment of distributed component-based applications on cloud infrastructures. Subsection 4.2.4 proposes a model for management users inspired by AWS and deployment infrastructure users. Subsection 4.2.5 describes the use of policies to specify: target environment, users, deployment infrastructure, and applications.

4.2.1 A Cloud Computing Environment

Setting up a cloud computing environment implies obtaining an operative cloud infrastructure. A private cloud comprises the installation and configuration of a cloud platform over a virtualization layer. To install and maintain a private cloud environment demands a good level of system administration skills to deal with non-trivial configurations. Moreover, someone holding an account with a public cloud is able to access the management tools, although they will be expected to pay a bill for the service. We detail a summary of the cloud platform, libraries, and tools used to set up our private cloud. Also, we describe the steps required to access to Amazon Web Services.

Setting up a private cloud

We selected OpenStack as our first option for setting up a private cloud. At the time of designing and testing with OpenStack (July 2011), the lastest version release was 2011.2 under the code name *Cactus*. With a previous background knowledge of Ubuntu, we decided to install Ubuntu Server 11.04 for both host and virtual machine instances and use KVM hypervisor as the virtualization layer. Thus, to facilitate the installation and configuration of

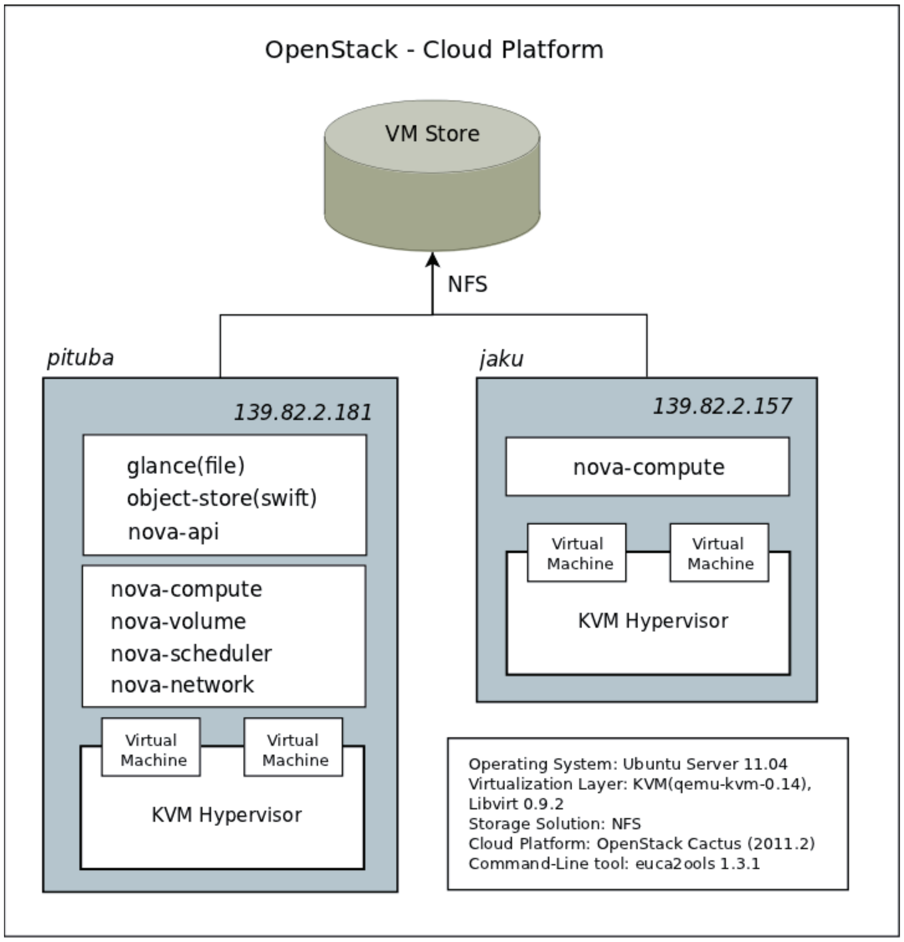

Figure 4.2: OpenStack Cloud Platform

OpenStack we use the Ubuntu repositories. Figure 4.2 shows the OpenStack services configured in two physical machines, with the following hardware configurations. *Pituba* server: a Intel Core 2 Quad CPU 2.40 GHz, 2GB RAM, Disk 1x150GB, and NIC 1x1Gb. *Jaku* server: a Intel Core 2 Duo CPU 2.00GHz, 2GB RAM, Disk 1x100GB, and NIC 1x1GbE. Pituba acts as a server instantiating Nova (compute), Swift (storage), and Glance (image) services, while *jaku* is a nova-compute node. To verify the correct functioning of our OpenStack installation we used *euca2ools* to manage virtual machine instances. It meant making tests of runninging, rebooting, and terminating virtual machine instances. First, we executed this test only into the server *pituba*, i.e. without any connection to *jaku*, and the second test included both *pituba* and *jaku* servers. These scenarios allow us to test a single and dual node configuration. Single configuration means that only one server runs all OpenStack services, whereas with dual configuration a second server will be added to a single configuration [Pepple11]. Our first test was successfully completed, thus we could access the virtual machine instances. However, in the second test we found a problem when we tried to access to the virtual machine instances located in the *jaku* host, because according to the documentation, we should have had for each host two network interfaces [OpenStackDoc11]. Finally, we also made early experiments setting up a private cloud using Eucalyptus 1.6 and Xen hypervisor 3.3, we rely on both as good options to build cloud infrastructures, with a great cloud software stack.

Accessing a public cloud

Currently, the most popular public cloud provider is Amazon Web Services(AWS). We decided to use AWS to access to a public cloud infrastructure. In recent years, AWS has increased its locations and services, becoming the main reference for many cloud standards. To start using AWS, we need an AWS account and to link it to any credit card. Then, after logging-in with our AWS account, we can access to the AWS Management Console. This is a web-based application, which provides a graphical interface to interact with its services. We use the command-line tools *ec2-api-tools* and *euca2ools* for managing virtual machine instances. To start running instances we need to export our private key and certificate using the variables EC2_PRIVATE_KEY and EC2_CERT, respectively.

4.2.2 Cloud API

We need a cloud API with capabilities to manage at least Amazon EC2 and OpenStack Compute. However, although the cloud APIs listed in subsection 2.1.4 currently (2012) have support for them, we do not find any cloud API close to our cloud API requirements, i.e., offering a flexible way to define a group of variables like the policies approach and functioning with an incremental level of details.

Specifically, OpenStack inherited several language-specific API bindings from its original code contributor, Rackspace. For example, OpenStack Object Storage has the following bindings: PHP, Python, Java, C#/.NET, and Ruby. On the other hand, AWS provides Software Development Kit(SDK) for Android, iOS, Java, .NET, and PHP. We require a Lua-based cloud API due to our deployment infrastructure, because SCS components were implemented using a combination of Lua libraries such as OiL, LOOP classes, and LuaRocks. We designed a two-layer cloud API to meet with the requirements to achieve a compatible cloud API for our deployment infrastructure. Figure 4.3 shows an overview of the cloud API designed to provide an API with an incremental level of details, and supporting a flexible way to specify parameter/value variables using policies. Our cloud API is composed of EucaEngine and the Cloud

Deployment Engine(CDE). We focus our effort on the building of EucaEngine layer as generically as possible, because this layer allow us to map its functions for CDE, creating a set of wrapper functions. CDE uses policies to specify values for the target environment, users, deployment infrastructure, and the applications. In the next subsections we describe each cloud API layer.

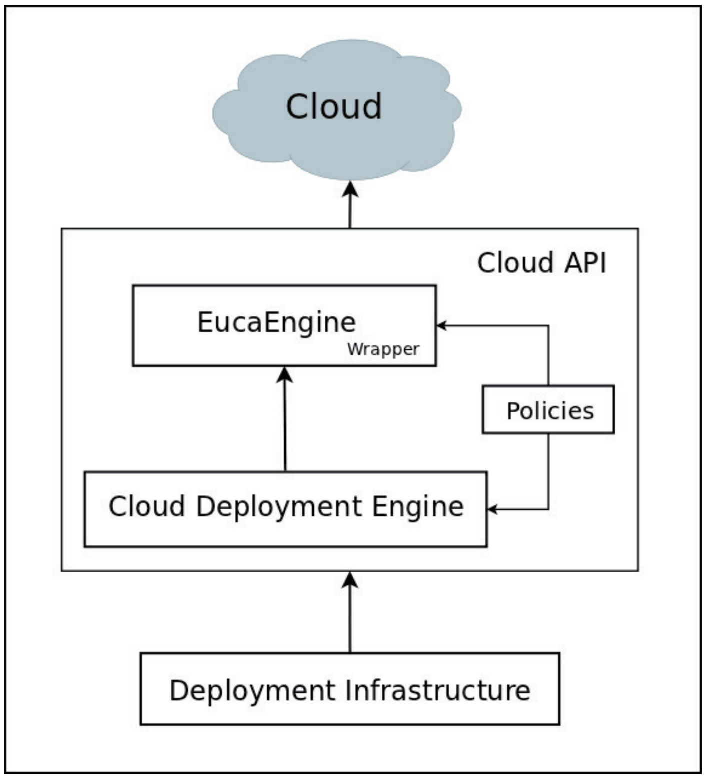

Figure 4.3: Cloud API - Architecture

EucaEngine

EucaEngine is a python-based wrapper library inspired in euca2ools, focused on providing compute services. EucaEngine was built considering three points: (i) an incremental level of details. EucaEngine users can instantiate virtual machines passing as parameters from only the number of instances required to a customized a set of parameters using policies. (ii) to support the use of default and customized policies. EucaEngine takes advantages of default policies for accessing cloud infrastructures. (iii) and thanks to the euca2ools design, it support Eucalyptus, OpenStack, and AWS.

Figure 4.4 displays the main class *Instance*. Our prototype is focused on the provisioning of virtual machine instances, other classes implemented are Image, AvailabilityZone, and Volume. We provide this API with intuitive function names, for example, run_instances, get_instances, reboot_instances, and terminate_instances. These functions allow users to launch the different stages of virtual machines: scheduling, networking, launching, running, terminating, and shut down.

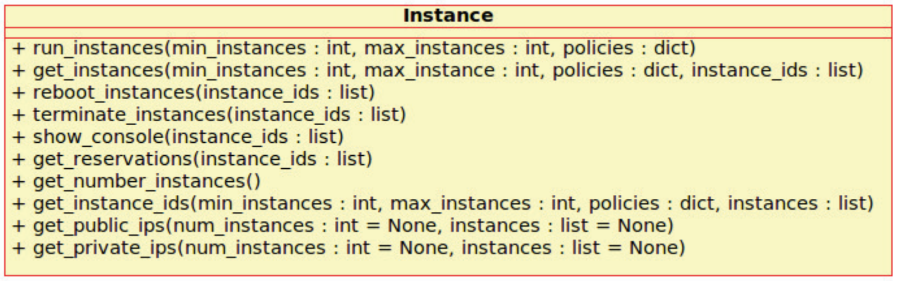
Instance
+ run_instances(min_instances : int, max_instances : int, policies : dict)
+ get_instances(min_instances : int, max_instance : int, policies : dict, instance_ids : list)
+ reboot_instances(instance_ids : list)
+ terminate_instances(instance_ids : list)
+ show_console(instance_ids : list)
+ get_reservations(instance_ids : list)
+ get_number_instances()
+ get_instance_ids(min_instances : int, max_instances : int, policies : dict, instances : list)
+ get_public_ips(num_instances : int = None, instances : list = None)
+ get_private_ips(num_instances : int = None, instances : list = None)

Figure 4.4: EucaEngine - Instance Class

Cloud Deployment Engine

We built the EucaEngine layer to provide a lightweight wrapper to support the management of virtual machine instances. However, due to the DeployManager service being implemented in Lua as a CORBA object, we need a Lua implementation of EucaEngine. We have developed the Cloud Infrastructure Service (CIS) to provide a CORBA intermediate wrapper to map all operations of EucaEngine and then we implemented the CloudEngine as a Lua client proxy. Thus, Cloud Deployment Engine consist of the Cloud Infrastructure Service (CIS) and the CloudEngine.

To implement CORBA objects we use omniORBpy, a free CORBA ORB for C++ and Python. Code 4.1 shows a partial segment of the *Cloud Infrastructure IDL* mapping the data related to the instances and its functions. Line 9 specifies the struct *InstanceData*, it saves the instance's metadata. Line 24 defines the struct *InstanceInfo* including the *instance_id* and a struct *InstanceData*. Line 33 and 38 show the operations *get_instances_num* and *get_instances_policies*, respectively. The interfaces specified in OMG IDL are detailed in appendix A.1.

CloudEngine provides a set of functions to initialize, obtain, and update a pool of virtual machine instances. This was implemented using a Lua class and has the following parameters: instances, policies, target_environment, and the cloud_deployment_model. The *cloud_deployment_model* parameter specifies the type of deployment cloud: public or private. Instance is an OiL/Lua-based proxy client class designed to map the instance operations from Cloud Infrastructure Service(CIS). Also, it maintains the incremental level of details, and supports the use of policies. Figure 4.5 shows the available functions

Listing 4.1: Instance IDL(partial)

```
#ifndef CLOUD_INFRASTRUCTURE_IDL
#define CLOUD_INFRASTRUCTURE_IDL

#include "hashtable.idl"

module CloudInfrastructure{

  /* Instance */
  struct InstanceData{
    string image_id;
    string public_dns_name;
    string private_dns_name;
    string state;
    string key_name;
    string ami_launch_index;
    string product_codes;
    string instance_type;
    string launch_time;
    string placement;
    string kernel;
    string ramdisk;
  };

  struct InstanceInfo{
    string instance_id;
    InstanceData instance_data;
  };
  typedef sequence<InstanceInfo> InstanceInfoSeq;

  interface Instance{

    // Get a num of instances
    InstanceInfoSeq get_instances_num(
      in unsigned short min_instances,
      in unsigned short max_instances);

    // Get a num of instaces according policies
    InstanceInfoSeq get_instances_policies(
       in unsigned short min_instances,
       in unsigned short max_instances,
       in HashObjectSeq policies);

    // [...]
  }
};
#endif
```

of CloudEngine in order to extend the SCS Deployment Infrastructure, and support cloud infrastructures as target environment.

4.2.3 Deployment Infrastructure: API Extension

The deployment infrastructure service, *DeployManager* is responsible for managing the planning activity. Unlike the previous version of the deployment infrastructure, where the target environment is described from a pre-defined list of physical machines specified using description files, we load a pool of virtual machine instances from a cloud infrastructure. Thus, we have a pool of virtual machine instances ready to use. We can also instantiate new instances creating our own pool of instances.

DeployManager has a new variable named *all_instances*, which saves all

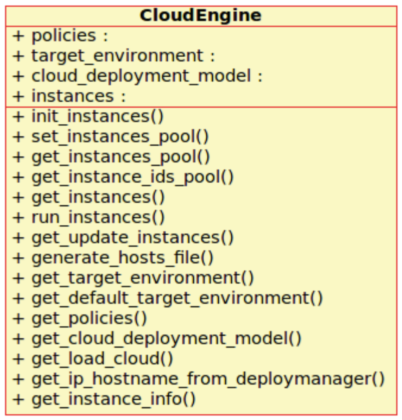

Figure 4.5: Cloud Engine

metadata retrieved by the function *get_instances_pool* from CloudEngine. This function returns all available metadata from the current pool of instances. However, these virtual machine instances can be shut down or maybe new virtual machine instances can be started. Thus, we need to update this information from time to time. We implement the function *scan_instances()* setting a time of 20 seconds. Finally, we create the function *get_updated_instances* that updates the data stored in *all_instances*, that is, whether a virtual machine instance was shut down then we need to put a label “shutdown” on this *instance_info*, and if a new virtual machine instance is started then we need to add its metadata to the *all_instances*. To maintain the compatibility with target environments composed by physical machines, we introduce the variable “machine”, this is either a host (physical machine) or instances (virtual machine instance). Figure 4.6 shows the new functions added to the deployment infrastructure.

4.2.4 User Management Model

The user management for cloud computing requires an efficient and flexible design, regardless of the type of cloud service model, because they share common cloud services such as billing, security, SLA, etc. Hence, the user management of cloud infrastructures is a topic that should be carefully implemented by cloud providers, and that cloud consumers need to adapt to execute their software systems and manage multiple users. For example, AWS offers the *AWS Identity and Access Management(IAM)* [AmazonWSIAM] to manage users, and enable a secure control access, while OpenStack has the Identity Service [OpenStackIdentity12]. IAM provides tools to manage users, roles, permissions, and credentials. IAM allows us to create users with the same capabilities as AWS accounts, but the billing will be charged to a root AWS

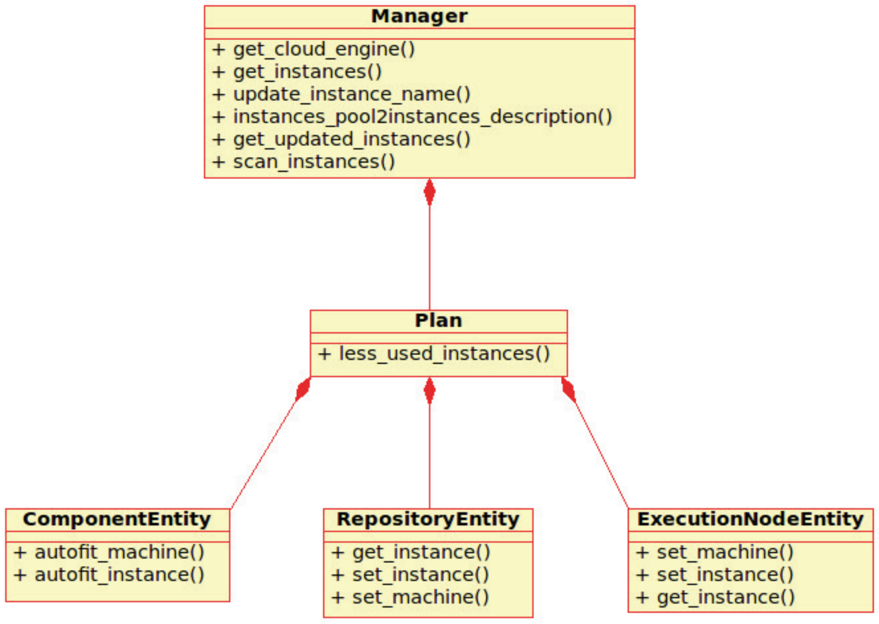

Figure 4.6: DeployManager Extensions

account.

Group	**Function**
Deployment Actor	Users with permission to deploy applications on available cloud infrastructures.
CloudAdmin	Cloud account staff to manage the policies related to the cloud infrastructure.
CloudUser	A restricted cloud account that allows specific permission policies to manage virtual machines instances: start, stop, suspend, terminate, etc.

Table 4.1: Type of Users for Deployment Infrastructure on Cloud Infrastructures

We propose a *user management model* composed of three groups to manage the application deployment on cloud infrastructures. The *Deployment Actor* groups together all users intended to make application deployments. The *CloudAdmin* and *CloudUser* groups accounts have access to cloud infrastructures. Additionally, we need to associate a *CloudAdmin* user with a “cloud account” because this person will be responsible for paying the bill for the cloud resources consumed. Table 4.1 shows the functions of each group.

Figure 4.7 displays the use case diagram, it explains the user interactions. To configure a minimal scenario we need a *cloud admin*, a *cloud user*, and a *deployment actor*. For example, a deployment actor receives an application to be deployed, writes the policies, and executes the deployment. Then, he requires a set of permissions to deploy this application on the cloud. The *cloud*

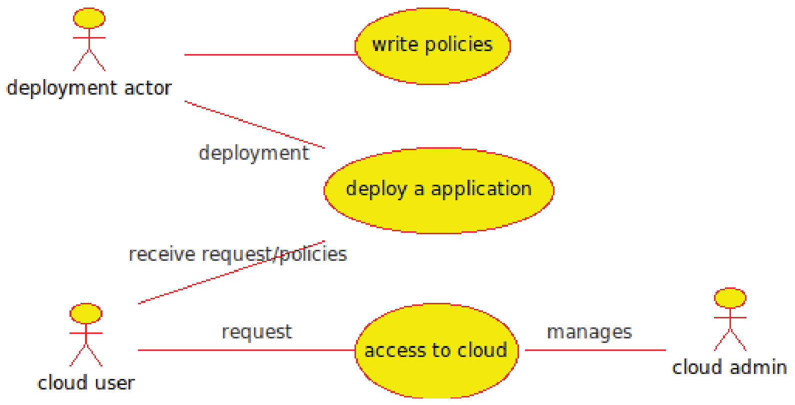

Figure 4.7: User Management

user receives the policies, and uses them to request the cloud-based target environment. The *cloud admin* manages a combination of permissions and keys for the *deployment actor* to access the virtual machine instances. Also, *cloud admin* needs to be associated with a cloud account. Finally, the *deployment actor* could deploy the application on the cloud infrastructure.

4.2.5 Policies

Policy-based management allows users to describe a set of rules to abstract the behaviour of a system from its functionality. A policy represents a relationship between subjects and targets, where the relationships are expressed in a series of actions [Marriot96]. In addition to the management of users, we propose the use of policies to manage the specification of target environments, the deployment infrastructure, and the applications to deploy. Due to the deployment infrastructure which represents our platform to support the deployment of distributed component-based applications, we prefer denominate it simply as *platform*. Table 4.2 displays the type of information that each group of policies stores.

Policy Group	**Description**
Users	Stores all user's information.
Target Environment	Defines the default cloud-based target environment configuration.
Platform	Defines the target environment for the Deployment Infrastructure.
Application	Contains the applications to deploy on a cloud infrastructure specified.

Table 4.2: Policies required to specify the deployment of applications.

We specify policies using Lua tables with the following structure: the table's name represents the name of a new policy, and each key/value pair represents a parameter/value, which the parameter could reference a previous defined table. A parameter can also represent a policy.

Code 4.2 shows an example of the default policy used to request virtual machine instances from Amazon EC2. We can use the *get_instances(1, 1)* function to get a set of virtual machine instances' metadata, it uses the default *getting* policy because there was not any policy as parameter. In the line 2, *ec2_resource* policy defines the memory, vcpus, and storage. From these values the instance type required for the application is calculated. In line 9, *ec2_instances* policy manages the virtual machine instances. It defines the *getting* policy, responsible for specifying the parameters utilized to get the metadata of the virtual machine instances. This policy is composed of four parameters: (i) *resource* defines a reference to *ec2_resource* policy, (ii) *percent* defines a value of percent into the interval [0-1], where "[0]" means get the instances with the exact type of resources defined in *ec2_resource*, a value in the interval "[0.1;0.9]" details the tolerance percentage, and "[1]" means ignore the filter policy, and then returns all virtual machine instances, (iii) *filter_policy* means a type of policy to filter the number of virtual machine instances returned, (iv) *choose_policy* defines the policy to get the number of virtual machines required.

Listing 4.2: An example of default policy to get instances for Amazon EC2

```
1  --- Amazon EC2 - Default Configuration Policy
2  ec2_resource = {
3
4    memory  = '0.6', -- RAM(GB)
5    vcpus   = '2',   -- EC2 Compute Units
6    storage = '8',   -- GiB
7  }
8
9  ec2_instances = {
10
11   getting = {
12
13     resource      = ec2_resource,    -- default resource(t1.micro)
14     filter_policy = 'range_percent', -- range_percent
15     choose_policy = 'random',        -- random
16     percent       = '1',             -- 0:exact, [0.1;0.9]:range, 1:default
17   },
18 }
```

4.3 Configuring SCS Deployment Infrastructure on the Cloud

In the section 4.2, we developed the *elements* required to extend the SCS Deployment Infrastructure to support the deployment of distributed component-based applications on cloud infrastructures. In this section, we explain the integration between the Deployment Infrastructure and the components developed: a cloud API, a set of extensions for the DeployManager service, a user management model, a policy approach, and a cloud computing environment. To simplify we use *Deployment Infrastructure* referred to the SCS Deployment Infrastructure with support of cloud infrastructures from here on.

We have to ensure a suitable cloud infrastructure, this means having the capabilities to consume cloud resources from a private or a public cloud. The cloud infrastructure is managed by a *cloud admin*, and this person is responsible for checking and updating the default cloud-infrastructure policies. The *deployment actor* defines the cloud deployment model specified into the *deployment_model* policy. We use this policy to start the *CIS*, enabling the connection between *CloudEngine* and *EucaEngine*. Then, the *deployment actor* is ready to start the DeployManager service. It involves the first interaction with the cloud infrastructure, which loads the *starting* policy. For example DeployManager could load or not the metadata of all running virtual machine instances, saving them for future deployment of applications. Finally, the *deployment actor* is ready to initialize the repository, packager, and execute the deployment scripts to deploy one or more applications, also policies are used to specify the target environment for both the deployment infrastructure and the applications.

This section explains how we integrated the cloud infrastructures with the Deployment Infrastructure. Subsection 4.3.1 describes an overview of the architecture proposed, and details how the DeployManager service manages the cloud infrastructures. Subsection 4.3.2 describes two possible configurations of the Deployment Infrastructure. The resources utilized to set the target environment are described in subsection 4.3.3. Subsection 4.3.4 discusses the main aspects considered in the building of our prototype and its limitations.

4.3.1 Architectural Overview

Figure 4.8 displays an overview of our architecture proposed, it is composed of two clouds, one cloud hosts the Deployment Infrastructure and the other will host the applications. Although it seems like the model requires two different clouds, we propose working with the same cloud to facilitate the implementation. The figure summarizes the use of *policies* by the *DeployManager extended* to request *cloud resources*. Therefore, a basic scenario for the deployment of a distributed component-based application begins with the definition of one *cloud admin* for managing the cloud infrastructure and one *deployment actor* for managing the deployment. The *cloud admin* assesses the cloud infrastructure making it available for the deployment actor to specify the application and its target environment. Then, the *deployment actor* initializes the deployment infrastructure services according to their defined policies, and finally executes the deployment of the application. Hence, we describe the deployment of distributed component-based applications in three main steps as follows: obtaining access to a cloud infrastructure; deployment of the deployment infrastructure like PaaS; defining the application and its target environment using IaaS.

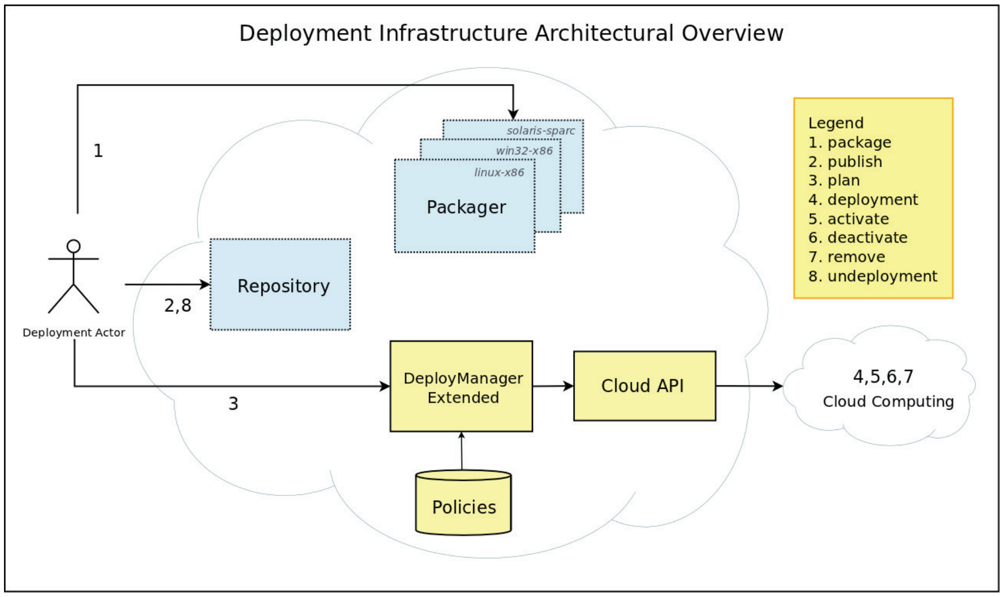

Figure 4.8: Deployment Infrastructure Architectural Overview

In the first step, we need to choose a cloud infrastructure either a private or public cloud, and update their default policies. We require at least one *cloud admin* account and one *deployment actor* registered. The *cloud admin* activates the permissions in order to allow the deployment actor to consume cloud resources, it implies giving capabilities to the *deployment actor* as a *cloud user*. Then, the *deployment actor* is ready to start the deployment infrastructure services.

The second step executes the deployment infrastructure services, the first service initialized is the DeployManager. Code 4.3 shows an example of the policy used to initialize the cloud infrastructure. Line 3 specifies the *cloud platform* policy, it lists the cloud platforms supported. OpenStack and AWS are our private and public clouds, respectively. Line 9 defines the *deployment model* policy. Line 15 defines the *starting* policy, which specifies four parameters. The first two parameters are the *cloud_private_policies* and *cloud_public_policies*, both reference to private and public default policies. The other two parameters are *load_cloud_private* and *load_cloud_public*, they specify the adopted policy just at the time when the DeployManager service has started. The *getting* parameter means the possibility to reuse current virtual machine instances, and *starting* means that it will only support the creation of new virtual machine instances.

Listing 4.3: Cloud Infrastructure Policy

```
cloud_infrastructure = {

  cloud_platform = {

    cloud_private = 'openstack',     -- openstack
    cloud_public = 'ec2',            -- ec2
  },

  deployment_model = {

    policy = 'private',              -- private, public
  },

  -- Initialization
  starting = {

    cloud_private_policies = 'openstack_instances',
    cloud_public_policies  = 'ec2_instances',
    load_cloud_private     = 'getting',
    load_cloud_public      = 'getting',
  },
}

target_environment = {

  policy = 'cloud_infrastructure',   -- local-network
}
```

The code 4.4 describes the policies related to the deployment infrastructure services. It builds the deployment infrastructure as a Platform as a Service, thus each deployment infrastructure service contains the hosting and policy parameters. Hosting means the type of host where the service will be executed, e.g. virtual machine instance or physical machine. The policy defines if the machine is reused or needs to be instantiated.

The third step comprises the specification of the applications ready to deploy. To define one application to deploy, it needs to have three policies as parameters related to the *application*, *deployment actor*, and its *target environment*. Code 4.5 displays an example of the *applications_to_deploy* policy. Line 2 describes the *hello_world_app* policy, it contains the application's metadata such as name, version, and author. Line 10 defines the *deployment_actor* policy describing the user data of the *deployment actor* and his access keys for the cloud infrastructure. Line 34 specifies the *target_environment_cloud* policy utilized for the deployment. Line 40 *applications_to_deploy* defines one or more applications to deploy. The target environment policy is explained in section 4.3.3.

Listing 4.4: Deployment Infrastructure - Platform Policy

```
platform = {

        deploy_manager = {
                hosting = 'instance',
                policy = 'getting',
        },

        repository = {
                hosting = 'instance',
                policy  = 'running',
        },

        packaging = {
                hosting = 'instance',
                policy  = 'running',
        },

        deployment = {
                hosting = 'instance',
                policy  = 'getting',
        },
}
```

The *cloud admin* is responsible for specifying the default cloud infrastructures policies. The *deployment actors* customize policies, also they need to be registered into the cloud user group.

The proposed architecture describes the main aspects considered to obtain a simple and flexible support for the deployment of distributed component-based applications. However, this architecture should consider that many deployment actors could deploy many distributed applications, instantiating their own Deployment Infrastructure services, but requesting cloud resources to the same cloud infrastructure. This means the CIS server needs to deal with multi-deployment actors. In the next section, we detail two possible configurations.

4.3.2 Deployment Infrastructure: PaaS

The subsection 4.3.1 describes an overview of our proposed architecture, we also highlight the necessary steps to deploy one application following a basic configuration. However, more realistic scenarios e.g. test environments or even production environments imply more advanced configurations. In addition to these configurations, due to its multi-tenant feature the deployment infrastructure needs to support simultaneously the deployment of many distributed component-based applications by many deployment actors. We discuss two possible options of deployment, a minimum supported configuration and a multi-deployment configuration.

Minimum Supported Configuration

Figure 4.9 shows the minimal configuration of the Deployment Infrastructure managed by one *deployment actor*, and composed of one instance of *DeployManager*, *Repository*, and *Packager*. DeployManager service is deployed into a virtual machine instance, and needs to be continually running for saving and updating the cloud infrastructure metadata, then we can obtain the current state of the virtual machine instances, e.g. a list of virtual machines currently running. The repository and packager services were designed with dashed boxes because they can be instantiated on-demand, or simply they can reuse any virtual machines previously instantiated. These services constitute the first effort in obtaining our Deployment Infrastructure as Platform as a Service. On the other hand, to access cloud infrastructure metadata deployment actors require a service to act as the entrance to the cloud. This service is the CIS server, responsible for the connection between the PaaS and the target environment. Therefore, the minimal configuration comprises the coordination of one deployment actor, one deployment infrastructure, and one CIS server to enable the provision of cloud resources.

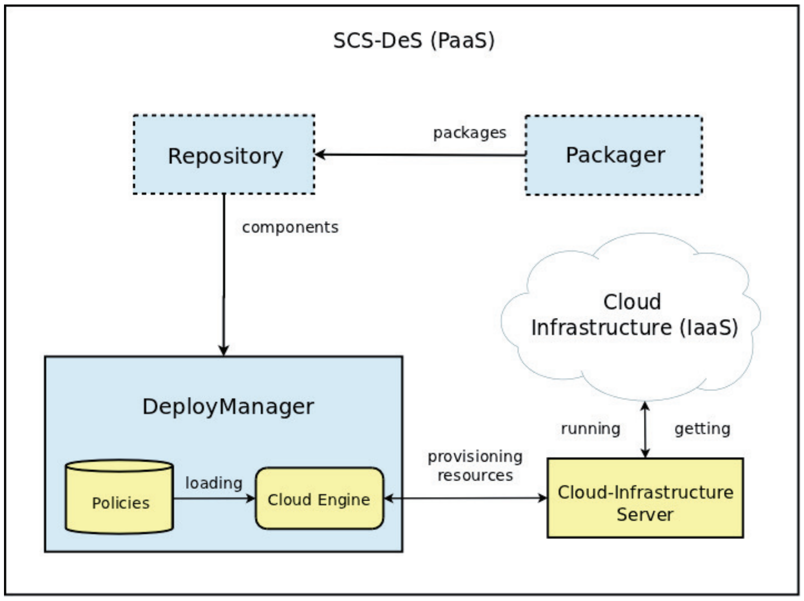

Figure 4.9: Deployment Infrastructure on Cloud Infrastructures: Minimal Architecture

A Multi-Deployment Configuration

A Multi-Deployment Configuration means that many deployment actors working with their own deployment infrastructures are sharing the same CIS server to enable the provision of cloud resources. Figure 4.10 displays a

Multi-Deployment configuration, composed of two *DeployManager* services, two *repositories*, one *packager*, and one *CIS server*. In this configuration, each DeployManager service corresponds to one *deployment actor*, and all DeployManagers share the same CIS server and policies. Ideally, two or more *deployment actors* use their own deployment infrastructure, but share the same cloud infrastructure; therefore they need to connect to the same *CIS server*. We increased the vision of the SCS Deployment Infrastructure because many deployment actors are able to deploy a number of applications using the same target environment.

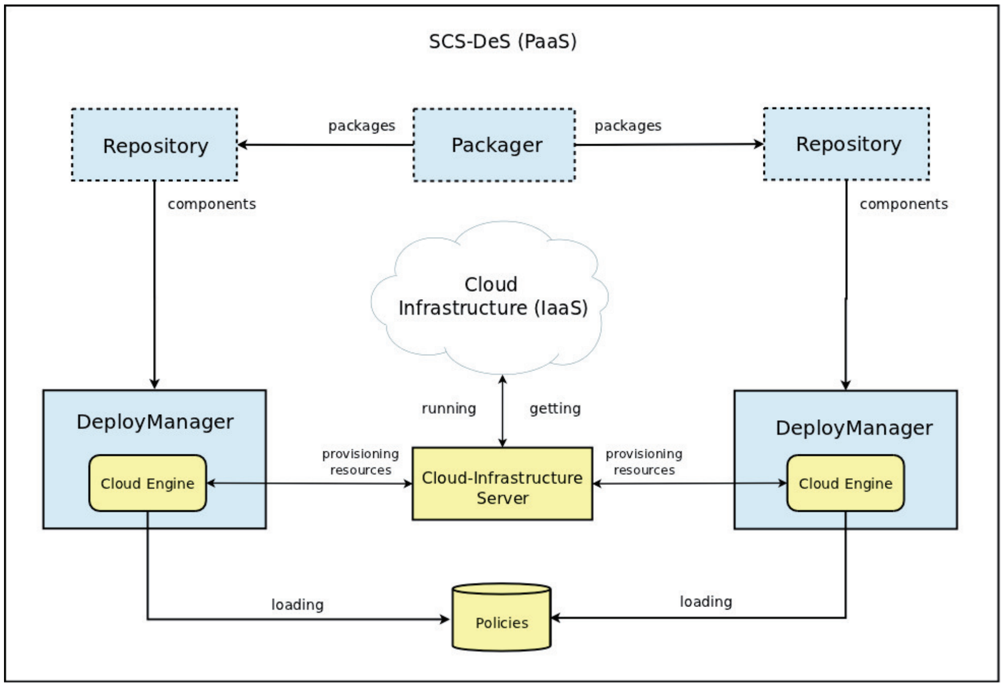

Figure 4.10: A Multi-Deployment Architectural Configuration

4.3.3 Deployment Infrastructure: IaaS

Once the deployment infrastructure services are executed, we can begin executing the scripts to deploy our applications. There are usually two scripts, the first script packages and publishes the components into a repository, and the second script completes the deployment of the application. These scripts load the *target environment* policy to decide the type of policy that will be used to install the application's components. Code 4.6 displays an example of the target environment policy. Line 4 specifies the *getting* policy, line 13 shows the *running* policy, both are mapped to *get_instances* and *run_instances* functions, respectively. Thus, they obtain access to virtual machine instances by either reusing or instantiating new virtual machines. The parameters of the *getting* policy are the same as those we reviewed in subsection 4.2.5. The *running* policy groups six parameters utilized to instantiate new virtual machine instances. The *resource* parameter defines a resource specification(vcpu,ram,disk) as the requirements to execute new instances, *instance_type* parameter is the default type of instance, if this parameter is defined then the previous resource policy will be invalidated because the type of instance implies a pre-defined resource specification managed by the cloud platform. But if *instance_type* is not defined, then the *policy* and *percent* parameters are utilized to find a compatible *instance_type*. The *image* parameter is the image_id utilized as the base image to start virtual machine instances, and the *languages* parameter defines a list of pre-installed programming languages and development kits required for the base image. We highlight the use of policies to specify the target environment, it allow us to define a set of options to specify a customized cloud-based target environment. Also, we found advantages in using *getting* policies because we could reduce the billing by deploying test applications.

4.3.4 Prototype and Limitations

The prototype was built over the Deployment Infrastructure detailed in chapter 3. First, we rely on a cloud infrastructure built with Eucalyptus and Xen hypervisor obtaining a good performance, but we also experiment using OpenStack as our cloud platform. We found greater capabilities working with OpenStack because it supports a single node installation, thus we decided to use OpenStack and KVM as our virtualization layer. Second, the implementation of the cloud API involved extra work because we do not find any implementation of cloud API based on Lua programming language. Hence, we need to implement our own Cloud API to support a Lua interface. Third, we use this cloud API to develop a series of extensions for the DeployManager service. Fourth, we propose a simple model for user management, we test it with the Identify and Access Management(IAM) module from Amazon Web Services. Finally, we utilized Lua tables to write policies taking advantage of Lua's ease of definition and reuse.

Despite all our efforts to build a low coupling *components* developed in 4.2, our prototype has some limitations. Our cloud API was focused on providing compute resources, which could be interesting for adding some storage services to the deployment infrastructure or as part of the target environment. Currently, our policy syntax only supports Lua tables and does not provide any form to define actions. Therefore, to support new actions it is necessary to be hard-coded. For example, when *deployment actors* require the writing if new actions from a combination of policies, these actions need to be programmed and tested. A solution could be to write Lua functions specifying these actions, while a policy approach should offer a simple method of defining actions.

Listing 4.5: Application Policy

```
--- Demo Application
hello_world_app = {

  name = "HelloWorld", version = "0.1", author = user_data.user_a,
  deployment_script = "/home/usera/projects/scs-deploysystem/src/lua"..
                      "/scs/demos/deployer/helloworld-deployment.lua"
}

-- Deployment Actor
deployment_actor = {

  -- User Data
  user_data = user_a,

  --- AWS Credentials
  ec2_creds = {

    EC2_HOME              = update,
    EC2_PRIVATE_KEY       = update,
    EC2_CERT              = update,
    EC2_ACCESS_KEY        = update,
    EC2_SECRET_KEY        = update,
    EC2_URL               = update,
    AWS_ACCESS_KEY_ID = optional,
    AWS_SECRET_ACCESS_KEY = optional,
    key_pair              = root,
  },

  --- OpenStack Credentials
  openstack_creds = {},
}

--- Target Environments
target_environment = {

  policy = 'cloud_infrastructure',
}

--- Applications to Deploy
applications_to_deploy = {

  --- Application 1
  application_1 = {

    application        = hello_world_app,
    deployment_actor   = deployment_actor,
    target_environment = target_environment,
  },
}
```

Listing 4.6: Target Environment Policy

```
--- Amazon EC2 - Default Configuration Policy

-- default resource values(type_instance: t1.micro)
ec2_resource = {

  memory  = '0.6', -- RAM(GB)
  vcpus   = '2',   -- EC2 Compute Units
  storage = '0',   --
}

-- default image value(ubuntu server 11.04, x86_64)
ec2_image = {

  default_ami  = 'ami-1b814f72',
}

-- default programming languages
default_languages = {

  lua = '5.1',
  java = '1.6',
  oil  = '0.4',
}

-- Amazon EC2
ec2_instances = {

  getting = {

    resource = ec2_resource,          -- default resource(t1.micro)
    --
    filter_policy = 'range_percent',  -- range_percent
    choose_policy = 'random',         -- random
    percent       = '1',              -- 0:exact,[0.1;0.9]:range,1:default
  },

  running = {

    resource = ec2_resource,          -- default resource
    --
    instance_type = 't1.micro',       -- invalidate resource policy
    policy        = 'range_percent',  -- range_percent
    percent       = '1',              -- 0:exact,[0.1;0.9]:range,1:default
    image         = ec2_image,        -- default image
    languages     = default_languages, -- default languages
  },
}
```