seaclouds

AGILITY AFTER DEPLOYMENT

Modelling Planning Controlling

SeaClouds Project

D3.1 Discovery, Design and Orchestration Functionalities: First Specification

Project Acronym	SeaClouds
Project Title	Seamless adaptive multi-cloud management of service-based applications
Call identifier	FP7-ICT-2012-10
Grant agreement no.	Collaborative Project
Start Date	1st October 2013
Ending Date	31st March 2016

Work Package	WP3, SeaClouds Design-Time modelling and orchestration
Deliverable Code	D3.1
Deliverable Title	Discovery, Design and Orchestration Functionalities: First Specification
Nature	Report
Dissemination Level	Public
Due Date:	M12
Submission Date:	10th October 2014
Version:	1.0
Status	Final
Author(s):	Michela Fazzolari, PengWei Wang (UPI); Francesco D'Andria (ATOS); Elisabetta Di Nitto, Raffaela Mirandola, Diego Perez (Polimi); Javier Cubo (UMA)
Reviewer(s)	Francesco D'Andria (ATOS); Andrea Turli (Cloudsoft)

Dissemination Level

Project co-funded by the European Commission within the Seventh Framework Programme		
PU	Public	X
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Table of Contents

List of figures

List of tables

Executive Summary

This deliverable presents the first specification of the discovery, design and orchestration functionalities of the SeaClouds platform, which includes the SeaClouds discovery functionality, the SeaClouds formalism to specify properties and requirements, the SeaClouds application topology model and the SeaClouds planning policies.

The deliverable describes an application meta-model to specify properties and requirements of an application, a TOSCA model used to describe the application topology, an abstract model to represent capabilities advertised by cloud providers, and also a possible mapping from the proposed models into TOSCA representations. A preliminary description of the SeaClouds orchestration policies is presented, which includes a matchmaking process and an optimization process.

Finally, an initial prototype design and description for the planner component is provided.

1. Introduction

The main objective of the SeaClouds platform is to allow a user to automatically deploy her applications on multiple-clouds and to manage the application itself during the entire lifecycle. This document is going to focus on the first step, thus analysing at design time which are the inputs needed by the platform to decide where and how to deploy a user application, by guaranteeing at the same time the satisfaction of user requirements.

In the next two sections, the inputs needed by the platform are analysed. In section 2, the user input is taken into account and a meta-model is proposed to represent all the information needed to describe a user application. In this model, concepts like application module, application topology, user requirements, etc., are described. The last part of this section presents a possible mapping of the proposed meta-model into TOSCA concepts, in order to promote the standardization of cloud service descriptions. It must be highlighted that the meta-model is quite general and can be reused in case the TOSCA standard should turn out unsuccessful.

Section 3 deals with the discovery of capabilities and services offered by cloud providers. To this end, a cloud meta-model is proposed to provide a uniform description of these capabilities and services. As in Section 2, we also propose a possible mapping of the cloud meta-model into TOSCA concepts.

In Section 4, we give a first specification of the planner component, which is in charge of analysing the user input and finding a suitable allocation of the application modules on multiple cloud services, by guaranteeing that user requirements are satisfied. The planner service is composed of two consecutive steps, a matchmaking process and an optimization process. The output produced by the planner component includes all the information needed to deploy a user application on multiple clouds.

Finally, in Section 5, a description of a first prototype of the planner component is given, which implements the concepts described so far.

In order to enhance the readability of this deliverable, some additional but still important concepts and analysis, like the background on TOSCA, analysis of common services offered by cloud providers, and existing resource allocation strategies in cloud computing, have been moved to a set of appendices, which can be found at the end of the document.

1.1 Glossary of Acronyms

Acronym	Definition
SaaS	Software-as-a-Service
PaaS	Platform-as-a-Service
IaaS	Infrastructure-as-a-Service
QoS	Quality of Service
SLA	Service Level Agreement
TOSCA	Topology and Orchestration Specification for Cloud Applications
CAMP	Cloud Application Management for Platforms
GUI	Graphical User Interface
API	Application Programming Interface

APP	Application
DB	Database
WP	Work Package
QoB	Quality of Business
DAM	Deployable Application Model
PaaS	Platform-as-a-Service
XML	Extensible Markup Language
HTTP	HyperText Transfer Protocol
SSL	Secure Sockets Layer
SLO	Service Level Objective
KPI	Key Performance Indicator
PHP	Hypertext Preprocessor
YAML	YAML Ain't a Markup Language
ADP	Abstract Deployment Plan
VM	Virtual Machine
UML	Unified Modeling Language
CSAR	Cloud Service ARchive
USB	Universal Serial Bus
RAID	Redundant Array of Independent Disks
SAN	Storage Area Network
FAT	File Allocation Table
NTFS	New Technology File System
DBMS	Database Management System
OS	Operating System

Table 1. Glossary of acronyms

2. Specification of Application Properties and Requirements

This section describes the initial input of the SeaClouds platform given by a user. To this end, an application meta-model is proposed to represent all the related information.

2.1 Application Model

The main purpose of the application model is to keep a track of the constituents of a multi-cloud application during its life-cycle through the SeaClouds platform. Figure 1 shows the application model lifecycle of the platform. The initial input for SeaClouds is an abstract application, which is described through an *Abstract Application Model*. This model contains a definition of all Modules of the application and of the requirements these modules pose on the lower level Modules or Services they are based on.

The task of the SeaClouds Planner is to transform the Abstract Application Model into a *Deployable Application Model*. In this last model all leaves application Modules are either mapped into some deployable Artifacts or they are associated to some Concrete Services that can offer the functionality required by the Modules. In other terms, the Deployable Application Model contains the information needed by the SeaClouds Deployer to deploy, configure and execute the application on some clouds (this will be more detailed in Deliverable D4.1 related to the runtime phase).

During execution, the *Live Application Model* is the one that keeps track of the status of all application's Modules and that is used for supporting the dynamic evolution of the application (also described in Deliverable D4.1). Figure 1 shows a UML activity diagram that describes the transitions from one Application Model to the other.

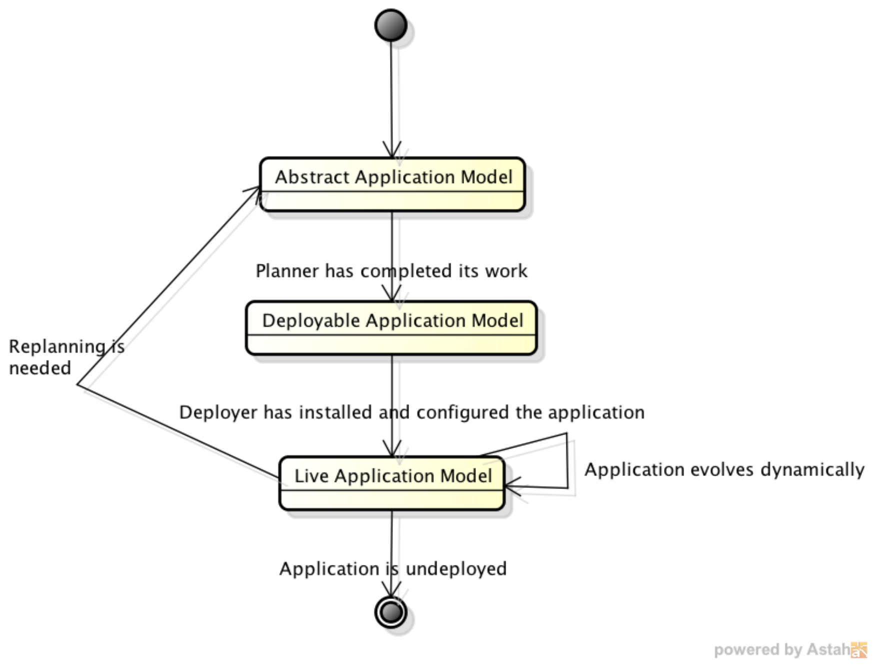
Abstract Application Model
Planner has completed its work
Deployable Application Model
Replanning is needed
Deployer has installed and configured the application
Live Application Model
Application evolves dynamically
Application is undeployed
powered by Astah

Figure 1. Application Model lifecycle

Abstract, Deployable and Live Application Models are all defined according to the metamodel defined in the next section.

2.2 Application Metamodel

Figure 2 shows the main constituents of the Application Metamodel. An example of instantiation of such metamodel is presented in Section 2.3.

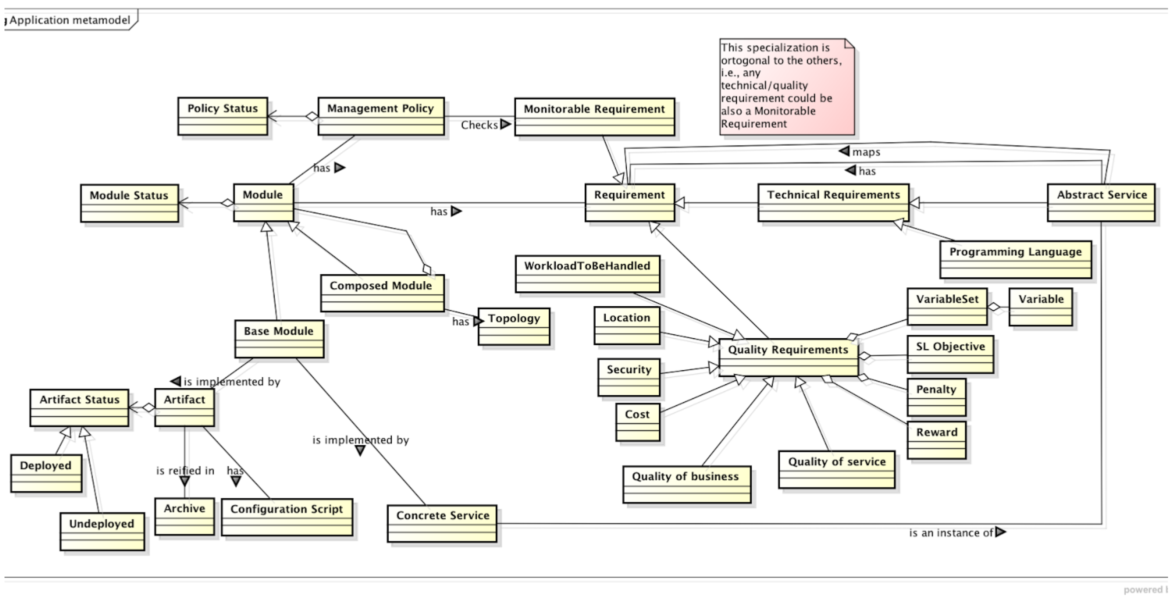

Figure 2. Application metamodel

The central element of the metamodel is the *Module*. An application can be constituted by a simple module (the *Base Module*) or, more frequently, by a composition of various modules. This second possibility is represented by the concept of *Composed Module* that can be constituted by various Modules, either Base ones or Composed. Thus, a Composed Module has always associated a *Topology* that describes the way the constituent Modules are connected together. A detailed definition of the Topology description is given in Section 2.4.

A Base Module can either be implemented by some *Artifact* (more details on Artifacts are given in Section 2.2.1) or by some *Concrete Service*. In the first case, the configuration, deployment, runtime management of the Artifact is under the responsibility of the SeaClouds Deployer. In the second case, the responsibility of the Deployer is to create the right communication channels between the Module and the Concrete Service and the actual interaction between the two is in charge of the application itself.

A Concrete Service is typically a specialization of an *Abstract Service*. Abstract Services are made available to the SeaClouds Planner subcomponent called Matchmaker. This is in charge, during the planning phase, of mapping *Requirements* defined for some Module(s) into Abstract Services.

Requirements can be of various kinds and are categorized as follows. They can be either *Technology Requirements* (e.g., the usage of a specific *Programming Language* or of a specific Abstract Service) or they can be *Quality Requirements*. A detailed list of the Quality Requirements we consider is provided in Section 2.2.2. Orthogonally to this classification, Requirements are Monitorable Requirements when it is possible to build proper sensors that allow to acquire data concerning their fulfillment.

At runtime, a Module has associated a *Module Status*, which includes any runtime information associated with the Module itself. Moreover, the Module can have associated a *Management Policy*. This defines the actions to be executed when some condition happens. Considered conditions can predicate on *Monitorable Requirements* or on other characteristics of the multicloud application.

As discussed in Section 2.1, the Application Metamodel is used in the three different models described in Figure 2.

The Abstract Application Model typically contains Modules and Requirements. The Base Modules may be associated to Artifacts or to some Concrete Services, but this is not mandatory at this level. Other information is filled in at the Deployable and Live Models level. More specifically, in the Deployable Model all Base Modules are associated to specific Artifacts or Concrete Services. Moreover, the entire set of Requirements to be considered is finalized and Management Policies are defined. Some of these data may vary at runtime, as a result of Management Policies. For instance, the Concrete Service associated in the Deployable Application Model to some Base Module may change dynamically.

The Module Status and the Policy Status are data specific of the runtime and vary depending on the execution.

In general, the Live Application Model is stored within the Deployer. This last one can be invoked by the Planner when this is required to replan the application. This can be done following the Deployer strategy defined in Deliverable D4.1 (Section 2) through the query interface offered by the Deployer as explained in Deliverable D4.2 about the API, Section 3.3.

2.2.1 Artifact

In SeaClouds, an Artifact is anything that can be installed on and/or configured in the Cloud. The Artifact has associated a *Configuration Script* that describes the way it should be configured to work as required by the application that is exploiting it. Finally, the Artifact has an *Artifact Status* that can be *deployed* when it is ready for execution or *Undeployed* in the opposite case.

2.2.2 Quality Requirements

Quality Requirements specify criteria that can be used to judge the operation of a system, rather than a specific behavior. This should be contrasted with Technology Requirements that define specific constraints on behavior or functions.

Terms for Quality Requirement are, "quality attributes", "quality goals", "quality of service requirements" and "non-behavioral requirements". Example of specialization of Quality Requirements are listed below.

Location
The concept Location describes where a Module will be located on top of some hardware infrastructure providing the cloud resource. A Location can be classified into Region or SubRegion. A Region is a macro area corresponding to a continent, while a SubRegion corresponds to a subcontinent or a geographic area within a continent, so a SubRegion is always contained in a unique Region.

Security
The concept Security describes the security requirements of the Modules belong to the Application. The isolation of the instance where the Module runs or the use of the SSL protocol are example of Security requirements.

Cost
For instance for PaaS, the concept Cost describes application requirements in terms of cost. Together with the concept QoS will help the Planner to arrange the optimum/sub-optimum concrete plan to maximize the relation QoS of the App / Cost of the App.

The cost is expressed as Gold, Silver, Bronze + a fixed price; the fluctuation of the final Module cost is between the <u>**fixed cost and the fixed cost + fixed cost * X;**</u>
X = 100 % (Gold)
X = 50 % (Silver)
X = 20 % (Bronze)

Example:
a) Cost Policy = Gold; fixed price 100 = Fluctuation of the cost Module is between 100 and 200; Maximize and economy of the Module at runtime.

b) Cost Policy = Silver; fixed price 150 = Fluctuation of the cost Module is between 150 and 225 and Maximize the performance of the Module at runtime.

The Range Cost is described by a Lower_bound and an Upper_bound;

WorkloadToBeHandled
WorkloadToBeHandled models the characteristics of the workload, in terms of types and number of requests per seconds over a certain period of time that the application should be able to fulfill.

Quality of Service

Quality of Service (QoS) refers to a level of service that is satisfactory for the actor consuming the cloud resource. In the context of SeaClouds the application providers and final users.

QoS terms are associated to metrics and a Service Level Objective (SLO) and they are used to make a first matchmaking analysis of the best-fit cloud service and configure "a smart" monitoring service that will assess the terms at runtime. Typical SLO could be: [ResponseTime < 100 ms; MemoryConsumed > 100 Mb; ServiceAvailability > 90%].

The unfulfillment of the QoS terms could generate a scaling-up action (of the cloud resources) or even the activation of a migration procedure. For instance, [if the ServiceAvailability > 90% scaleup].

Quality of Business

The Quality of Business (QoB), unlike the QoS, has direct business implications on the economy of the application running on the cloud.

QoB terms are associated to metrics and a service level objective and they are used to make a first matchmaking analysis of the best-fit cloud service and to generate SLA agreements that will be evaluated at runtime. The unfulfillment of the QoB terms could generate rewards like discounts or actions with business implication.

An example of QoB may be : [if in 30 minutes three violation of the SLO ResponseTime < 100 ms have been raised, it will be generated a discount of the 30% on the price of the service].

VariableSet (Metrics definition)

VariableSet (from WS-Agreement specification) contains the expressions that refer to aspects of the service(s) subject to the guarantee. For instance, metrics for availability and response time must refer to named concepts (availability, response time) and must be declared as named variables that can be used in assertions. The semantics of those variables must be defined to interpret the condition expression.

Example:

```
 <wsag:VariableSet>
     <wsag:Variable wsag:Name="Availability" wsag:Metric="./resources/metricXML:Percentage">
             <wsag:Location>\\Availability</wsag:Location>
     </wsag:Variable>
     <wsag:Variable wsag:Name="InitCost" wsag:Metric="./resources/metricXML:Cost">
             <wsag:Location>\\InitCost</wsag:Location>
     </wsag:Variable>
     <wsag:Variable wsag:Name=”bandwidth”  wsag:Metric=”job:networkBandwidth”>
             <wsag:Location>//JobDescription/Resources/IndividualNetworkBandwidth/Exact
             </wsag:Location>
     </wsag:Variable>
```

```
</wsag:VariableSet>
```

Service Level Objective

Service Level Objective (SLO) defines an objective that must be met in order to provide a service with a particular service level or with a particular quality of service (QoS) or Quality of Business (QoB). At the service provisioning time, the actual QoS/QoB properties are derived from the service monitoring system. This SLO essentially defines how the agreed QoS/QoB properties are related to the actual QoS/QoB properties.

It defines a logical expression that can be assessed in order to determine the fulfillment of a guarantee.

The Service Level Objective element is expressed as an assertion over service attributes and/or external factors such as date, time following the GRAAP WS-Agreement Specification [23, 24]. However, most often a Service Level Objective is expressed as a target for a Key Performance Indicator (KPI) such as average response time, completion time, availability, etc. Hence, the core specification provides a simple and/or complex expression structure for specifying a target for any domain specific KPIs.

```
<wsag:ServiceLevelObjective>
<wsag:KPITarget>
        <wsag:KPIName>xs:string</wsag:KPIName>
        <wsag:Target>xs:any</wsag:Target>
</wsag:KPITarget>
<wsag:CustomServiceLevel> … </wsag:CustomServiceLevel>
</ServiceLevelObjective>
```

- /wsag:ServiceLevelObjective: specifies a service level objective in a guarantee term, and contains an element either of type wsag:KPITarget or wsag:CustomServiceLevel.
- /wsag:ServiceLevelObjective/wsag:KPITarget: defines service level objective as an expression of a target of a key performance indicator associated with the service.
- /wsag:KPITarget/wsag:KPIName: This name of a key performance indicator associated with the service.
- /wsag:KPITarget/wsag:Target: This element defines the target value for a KPI.
- /wsag:ServiceLevelObjective/wsag:CustomServiceLevel: is of type xs:anyType and can be customized by using a domain specific expression or assertion language.

Penalty and Reward

The ***Penalty*** and ***Reward*** concepts come from WS-Agreement specification. In fact, the business values of a guarantee term may include the relative importance of meeting the Service Level Objective.

In that case ***penalties*** can be defined when the Service Level Objective is not meet. Example:

Guarantee1:

SLO: qos:MTBF=150 time:minutes,
Qualifying Condition: numRequests < 1000,
Penalty: 5 USD, Importance 8

On the other hand, a bonus could be paid by the Customer when certain SLA levels are reached within a period of time. An example of an SLA ***reward*** is the example of a cake to be brought to the service meeting by the concerning party. The customer brings a cake when levels are met. So the rewards are not concerned money or punishment, but about (improving, rating) performance of the service.

2.3 Instantiation of the Application Metamodel into the NURO Case Study

In order to show how the metamodel works, we have considered the NURO case study and we have defined the corresponding application model (see Figure 3).

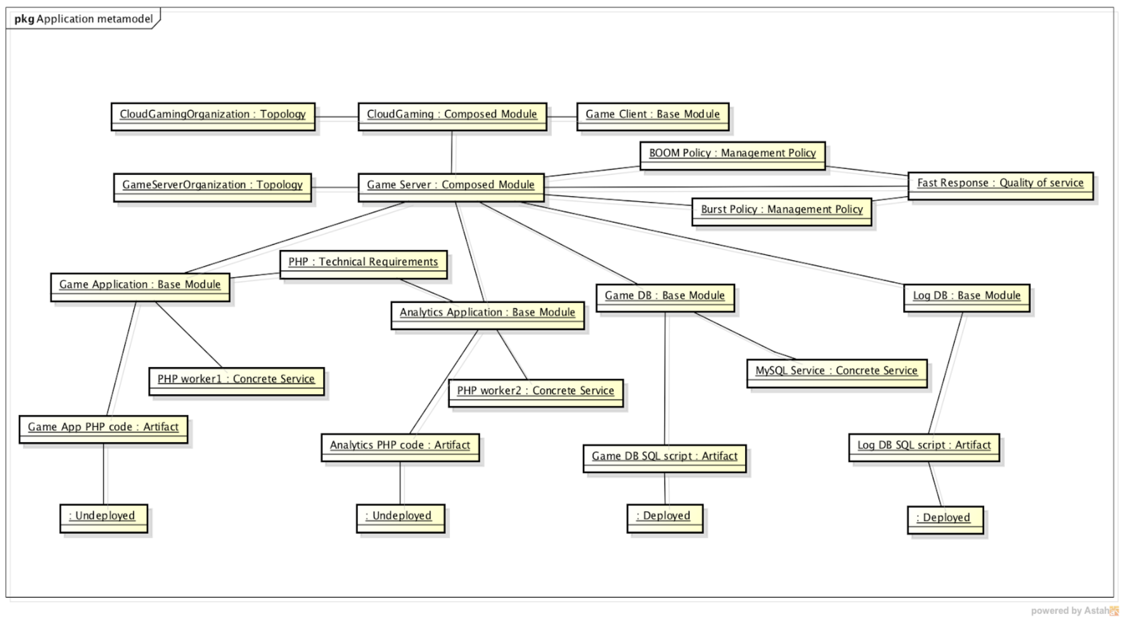
pkg Application metamodel
CloudGamingOrganization : Topology
CloudGaming : Composed Module
Game Client : Base Module
BOOM Policy : Management Policy
GameServerOrganization : Topology
Game Server : Composed Module
Fast Response : Quality of service
Burst Policy : Management Policy
PHP : Technical Requirements
Game Application : Base Module
Analytics Application : Base Module
Game DB : Base Module
Log DB : Base Module
PHP worker1 : Concrete Service
MySQL Service : Concrete Service
PHP worker2 : Concrete Service
Game App PHP code : Artifact
Analytics PHP code : Artifact
Game DB SQL script : Artifact
Log DB SQL script : Artifact
: Undeployed
: Undeployed
: Deployed
: Deployed
powered by Astah

Figure 3. Application Model of the NURO case study

As described in Figure 3, the model contains a first level Composed Module called CloudGaming. This represents the whole case study. In turn, it is composed of one or more Game Clients (these are not further detailed in this description as they are not relevant for the deployment of the case study on the cloud) and of a Game Server. This is composed of the following Base Modules: Game Application and Analytics Application that are PHP components, exploiting PHP workers offered as services by some PaaS, and Game DB and Log DB both exploiting the same MySQL service for their execution.

While in this example the database service is already populated with the data associated with Game DB and Log DB (the corresponding Artifacts are in the Deployed

state), the PHP code associated to the Game and to the Analytics Applications is still to be deployed.
Both Game Application and Analytics Application have the PHP language as Technology Requirement.

The whole Game Server has associated a Quality of Service requirement, that is, Fast Response, and two Management Policies called BOOM and Burst, respectively, that will be activated as soon as the monitoring system realizes that the Fast Response requirement is violated.

2.4 Topology Model

In SeaClouds, we employ the OASIS standard TOSCA (Topology and Orchestration Specification for Cloud Applications) [19] to represent the topology of cloud application, which indeed provides a powerful modelling language to describe the structure of an application as a typed topology graph. A brief and compact introduction about TOSCA can be found in Annexes (A. Background on TOSCA). In addition, some of the following descriptions, examples and XML syntax are taken from the TOSCA specification [19].

In TOSCA, a Service Template describes the structure of a cloud application by means of a Topology Template, and it defines the manageability behavior of the cloud application in the form of Plans. A Topology Template (also referred to as topology model) defines the structure of a cloud application, and its model can be found in Figure 18 in Annex A. Therefore, we will use the Topology Template in TOSCA to represent the Topology in our application metamodel described in Figure 2.

A Topology Template in TOSCA consists of a set of Node Templates and Relationship Templates that together define the topology model of a service as a directed graph. A node in this graph is represented by a Node Template, which specifies the occurrence of a Node Type as a component of a service. A Node Type defines the properties of such a component (via Node Type Properties) and the operations (via Interfaces) available to manipulate the component. Node Types are defined separately for reuse purposes, and a Node Template references a Node Type and adds usage constraints, such as how many times the component can occur.

A Relationship Template in this model specifies the occurrence of a relationship between nodes in a Topology Template. Each Relationship Template refers to a Relationship Type that defines the semantics and any properties of the relationship. Relationship Types are defined separately for reuse purposes. The Relationship Template indicates the elements it connects and the direction of the relationship by defining one source and one target element (in nested SourceElement and TargetElement elements).

The Module in our SeaClouds' application metamodel is mapped to the Node Template in this Topology Template in TOSCA, while the relationships between Modules in a

Composed Module in the application metamodel are mapped to the Relationship Templates in TOSCA.

All elements needed to define a TOSCA Service Template, including Node Type definitions, Relationship Type definitions, as well as Service Templates themselves, are provided in TOSCA Definitions documents. The following pseudo schema defines the XML syntax of a Definitions document, which explains the overall structure of a TOSCA Definitions document.

```
<Definitions id="xs:ID"
             name="xs:string"?
             targetNamespace="xs:anyURI">

  <Extensions>
    <Extension namespace="xs:anyURI"
               mustUnderstand="yes|no"?/> +
  </Extensions> ?

  <Import namespace="xs:anyURI"?
          location="xs:anyURI"?
          importType="xs:anyURI"/> *

  <Types>
    <xs:schema .../> *
  </Types> ?

  (
    <ServiceTemplate> ... </ServiceTemplate>
  |
    <NodeType> ... </NodeType>
  |
    <NodeTypeImplementation> ... </NodeTypeImplementation>
  |
    <RelationshipType> ... </RelationshipType>
  |
    <RelationshipTypeImplementation> ...
         </RelationshipTypeImplementation>
  |
    <RequirementType> ... </RequirementType>
  |
    <CapabilityType> ... </CapabilityType>
  |
    <ArtifactType> ... </ArtifactType>
  |
    <ArtifactTemplate> ... </ArtifactTemplate>
  |
    <PolicyType> ... </PolicyType>
  |
    <PolicyTemplate> ... </PolicyTemplate>
  ) +

</Definitions>
```

In addition, this document (including also the following pseudo schemas) uses the following syntax to define the serialization of resources:

- Characters are appended to items to indicate cardinality:
    - "?" (0 or 1)
    - "*" (0 or more)
    - "+" (1 or more)
- Vertical bars, "|", denote choice. For example, "a|b" means a choice between "a" and "b".
- Parentheses, "(" and ")", are used to indicate the scope of the operators "?", "*", "+" and "|".
- Ellipses (i.e., "...") indicate points of extensibility. Note that the lack of an ellipses does not mean no extensibility point exists, rather it is just not explicitly called out - usually for the sake of brevity.

From the above pseudo schema, we can know that a TOSCA Definitions document must define at least one of the elements: *ServiceTemplate*, *NodeType, NodeTypeImplementation, RelationshipType, RelationshipTypeImplementation, RequirementType, CapabilityType, ArtifactType, ArtifactTemplate, PolicyType*, or *PolicyTemplate*, but it can define any number of those elements in an arbitrary order.

Then, with respect to Service Template itself, the following pseudo schema defines its XML syntax, which specifies how a Service Template is defined.

```
<ServiceTemplate id="xs:ID"
                 name="xs:string"?
                 targetNamespace="xs:anyURI"
                 substitutableNodeType="xs:QName"?>

  <Tags>
    <Tag name="xs:string" value="xs:string"/> +
  </Tags> ?

  <BoundaryDefinitions>
    <Properties>
      XML fragment
      <PropertyMappings>
        <PropertyMapping serviceTemplatePropertyRef="xs:string"
                         targetObjectRef="xs:IDREF"
                         targetPropertyRef="xs:string"/> +
        </PropertyMappings/> ?
    </Properties> ?

    <PropertyConstraints>
      <PropertyConstraint property="xs:string"
                          constraintType="xs:anyURI"> +
        constraint ?
      </PropertyConstraint>
    </PropertyConstraints> ?

    <Requirements>
      <Requirement name="xs:string"? ref="xs:IDREF"/> +
    </Requirements> ?

    <Capabilities>
      <Capability name="xs:string"? ref="xs:IDREF"/> +
    </Capabilities> ?
    <Policies>
      <Policy name="xs:string"? policyType="xs:QName"
              policyRef="xs:QName"?>
        policy specific content ?
      </Policy> +
    </Policies> ?

    <Interfaces>
      <Interface name="xs:NCName">
        <Operation name="xs:NCName">
          (
            <NodeOperation nodeRef="xs:IDREF"
                           interfaceName="xs:anyURI"
                           operationName="xs:NCName"/>
          |
            <RelationshipOperation relationshipRef="xs:IDREF"
                                   interfaceName="xs:anyURI"
                                   operationName="xs:NCName"/>
          |
            <Plan planRef="xs:IDREF"/>
          )
        </Operation> +
      </Interface> +
    </Interfaces> ?

  </BoundaryDefinitions> ?

  <TopologyTemplate>
    (
      <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
                    minInstances="xs:integer"?
                    maxInstances="xs:integer | xs:string"?>
        <Properties>
          XML fragment
        </Properties> ?

        <PropertyConstraints>
          <PropertyConstraint property="xs:string"
                              constraintType="xs:anyURI">
            constraint ?
          </PropertyConstraint> +
        </PropertyConstraints> ?

        <Requirements>
          <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
            <Properties>
              XML fragment
            <Properties> ?
            <PropertyConstraints>
              <PropertyConstraint property="xs:string"
                                  constraintType="xs:anyURI"> +
                constraint ?
              </PropertyConstraint>
            </PropertyConstraints> ?
          </Requirement>
        </Requirements> ?
        <Capabilities>
          <Capability id="xs:ID" name="xs:string" type="xs:QName"> +
            <Properties>
              XML fragment
            <Properties> ?
            <PropertyConstraints>
              <PropertyConstraint property="xs:string"
                                  constraintType="xs:anyURI">
                constraint ?
              </PropertyConstraint> +
            </PropertyConstraints> ?
            </Capability>
        </Capabilities> ?

        <Policies>
          <Policy name="xs:string"? policyType="xs:QName"
                  policyRef="xs:QName"?>
            policy specific content ?
          </Policy> +
        </Policies> ?

        <DeploymentArtifacts>
          <DeploymentArtifact name="xs:string"
artifactType="xs:QName"
                           artifactRef="xs:QName"?>
             artifact specific content ?
          </DeploymentArtifact> +
        </DeploymentArtifacts> ?
      </NodeTemplate>
    |
      <RelationshipTemplate id="xs:ID" name="xs:string"?
                            type="xs:QName">
        <Properties>
          XML fragment
        </Properties> ?

        <PropertyConstraints>
          <PropertyConstraint property="xs:string"
                              constraintType="xs:anyURI">
            constraint ?
         </PropertyConstraint> +
        </PropertyConstraints> ?

        <SourceElement ref="xs:IDREF"/>
        <TargetElement ref="xs:IDREF"/>

        <RelationshipConstraints>
          <RelationshipConstraint constraintType="xs:anyURI">
            constraint ?
          </RelationshipConstraint> +
        </RelationshipConstraints> ?

      </RelationshipTemplate>
    ) +
  </TopologyTemplate>

  <Plans>
    <Plan id="xs:ID"
```

```
             name="xs:string"?
             planType="xs:anyURI"
             planLanguage="xs:anyURI">

        <Precondition expressionLanguage="xs:anyURI">
          condition
        </Precondition> ?

        <InputParameters>
          <InputParameter name="xs:string" type="xs:string"
                          required="yes|no"?/> +
        </InputParameters> ?

        <OutputParameters>
          <OutputParameter name="xs:string" type="xs:string"
                           required="yes|no"?/> +
        </OutputParameters> ?
       (
         <PlanModel>
             actual plan
         </PlanModel>
        |
         <PlanModelReference reference="xs:anyURI"/>
       )

     </Plan> +
   </Plans> ?

</ServiceTemplate>
```

In Service Template definition, the element of *TopologyTemplate* specifies the overall structure of the cloud application, i.e., the components it consists of, and the relations between those components. The components of a service are referred to as Node Templates, the relations between the components are referred to as Relationship Templates.

For our SeaClouds platform, which are the application modules to be deployed and their relationships are our first concern from user inputs. Obviously, they are corresponding to the above-mentioned Node Templates and Relationship Templates in Topology Template. Therefore, we introduce more about these two elements here.

- *NodeTemplate*: This element specifies a kind of a component making up the cloud application. The QName value of the attribute *type* refers to the Node Type providing the type of the Node Template. The *Properties* element specifies initial values for one or more of the Node Type Properties, and the initial values are specified by providing an instance document of the XML schema of the corresponding Node Type Properties. The *Requirements* element contains a list of requirements for the Node Templates, according to the list of requirement definitions of the Node Type, while the *Capabilities* element contains a list of capabilities for the Node Template, according to the list of capability definitions of the Node Type.

- *RelationshipTemplate*: This element specifies a kind of relationship between the components of the cloud application. For each specified Relationship Template the source element and target element must be specified in the Topology Template. The QName value of the *type* property refers to the corresponding Relationship Type. The *Properties* element specifies initial values for one or more of the Relationship Type Properties of the Relationship Type providing the property definitions in the concrete context of the Relationship Template. The *SourceElement* specifies the origin of the relationship represented by the current Relationship Template, and the *ref* attribute of this element references a Node Template or a Requirement of a Node Template within the same Service Template document that is the source of the Relationship Template. The *TargetElement* specifies the target of the relationship, and the ref attribute references a Node Template or a Capability of a Node Template within the same Service Template document that is the target of the Relationship Template.

In addition, the *BoundaryDefinitions* element specifies the properties the Service Template exposes beyond its boundaries, i.e. properties that can be observed outside from the Service Template. More precisely, the (optional) *BoundaryDefinitions* element allows to specify the properties, requirements and operations of internal components which are externally visible, and also describe non-functional behaviors that the whole application declare to expose through *Policies*. The latter also concerns user inputs (as we will see in the next section).

2.5 Representation of Properties and Requirements into TOSCA

In addition to the user inputs described in the previous section, namely, the application modules to be deployed and their relationships, the desired QoS properties and technology requirements for individual application modules, and the QoS properties for the whole application are also our concerns. How to express them in TOSCA is an important step for our SeaClouds platform. Thus, in this section we will describe in detail how to map these inputs from application model into TOSCA.

2.5.1 Technology Requirements

With respect to the technology requirements of each module inputted by the user, they can be mapped into *Properties* of corresponding Node Template in TOSCA Definitions. In this mapping process, we first need to define the structure of such properties in related Node Type via *Properties Definition*, i.e. the names, data types and allowed values the properties defined in Node Templates using a Node Type or instances of such Node Templates can have.

Based on the predefined XML schema for module properties, we can complete this specification via *Property Definition* in Node Type. The following shows a snippet of the XML syntax of such definition.

```
<PropertiesDefinition element="xs:QName"? type="xs:QName"?/>
```

Here, the PropertiesDefinition element has one but not both of the following properties:

- element: This attribute provides the QName of an XML element defining the structure of the Node Type Properties.
- type: This attribute provides the QName of an XML (complex) type defining the structure of the Node Type Properties.

Through this way, the structure of observable properties of Node Type can be specified by means of the predefined XML schema. The following shows a snippet of an example Node Type "Server", which uses a predefined XML element ComputeProperties to specify the structure of its properties.

```
<NodeType name="Server">
    <documentation>A basic cloud compute resource</documentation>
    <DerivedFrom typeRef="ns1:RootNodeType"/>
    <PropertiesDefinition element="ns1:ComputeProperties"/>
</NodeType>
```

After defining the structure of properties in Node Type through *Properties Definition*, the Node Template of this Node Type will specify initial values for one or more of the Node Type Properties, via *Properties* in the Node Template. The following shows the snippet of such XML syntax.

```
<NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
                    minInstances="xs:integer"?
                    maxInstances="xs:integer | xs:string"?>
        <Properties>
          XML fragment
        </Properties> ?
</NodeTemplate>
```

The initial values are specified by providing an instance document of the XML schema of the corresponding Node Type Properties. This instance document considers the inheritance structure deduced by the *DerivedFrom* property of the Node Type referenced by the *type* attribute of the Node Template.

The instance document of the XML schema might not validate against the existence constraints of the corresponding schema: not all Node Type properties might have an initial value assigned, i.e. mandatory elements or attributes might be missing in the instance provided by the *Properties* element. Once the defined Node Template has been instantiated, any XML representation of the Node Type properties MUST validate according to the associated XML schema definition.

The following shows a snippet of an example Node Template "VmMySQL", which is based on the "Server" Node Type described above. We can see that this template includes specific property settings that the application developer has provided that describes settings to be applied to that server.

```
<NodeTemplate id="VmMySQL" name="VM for MySQL" type="ns1:Server">
   <Properties>
      <ns1:ComputeProperties>
         <num_cpus>2</num_cpus>
         <mem_size>4096</mem_size>
         <disk_size>10</disk_size>
         <os_arch>x86_64</os_arch>
         <os_type>RHEL</os_type>
         <os_version>6.5</os_version>
      </ns1:IaaSCapabilityProperties>
   </Properties>
</NodeTemplate>
```

TOSCA assumes the existence of a base set of Node Types (e.g., a 'Compute' node), and other types for creating TOSCA Service Templates. It is envisioned that many additional Node Types for building service templates will be created by communities. With respect to these basic Node Types (including Compute, SoftwareComponent, DBMS, Database, ObjectStorage, and BlockStorage), we define the corresponding preliminary XML schemas that are used in their Property Definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.org/SeaClouds/TOSCA"
    xmlns="http://www.example.org/SeaClouds/TOSCA">
 <xs:complexType name="tComputeProperties">
  <xs:sequence>
   <xs:element name="num_cpus" type="xs:positiveInteger"/>
   <xs:element name="mem_size" type="xs:positiveInteger"/>
   <xs:element name="disk_size" type="xs:positiveInteger"/>
   <xs:element name="os_arch" type="xs:string"/>
   <xs:element name="os_type" type="xs:string"/>
   <xs:element name="os_distribution" type="xs:string"/>
   <xs:element name="os_version" type="xs:string"/>
   <xs:element name="ip_address" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element type="tComputeProperties" name="ComputeProperties"/>

 <xs:complexType name="tSoftwareComponentProperties">
  <xs:sequence>
   <xs:element name="version" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element type="tSoftwareComponentProperties"

name="SoftwareComponentProperties"/>

 <xs:complexType name="tDBMSProperties">
  <xs:sequence>
   <xs:element name="dbms_root_password" type="xs:string"/>
   <xs:element name="dbms_port" type="xs:integer"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element type="tDBMSProperties" name="DBMSProperties"/>

 <xs:complexType name="tDatabaseProperties">
  <xs:sequence>
   <xs:element name="db_user" type="xs:string"/>
   <xs:element name="db_password" type="xs:string"/>
   <xs:element name="db_port" type="xs:integer"/>
   <xs:element name="db_name" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element type="tDatabaseProperties" name="DatabaseProperties"/>

 <xs:complexType name="tObjectStorgeProperties">
  <xs:sequence>
   <xs:element name="store_name" type="xs:string"/>
   <xs:element name="store_size" type="xs:integer"/>
   <xs:element name="store_maxsize" type="xs:integer"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element type="tObjectStorgeProperties"
name="ObjectStorgeProperties"/>

 <xs:complexType name="tBlockStorgeProperties">
  <xs:sequence>
   <xs:element name="store_mount_path" type="xs:string"/>
   <xs:element name="store_fs_type" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element type="tBlockStorgeProperties"
name="BlockStorgeProperties"/>

</xs:schema>
```

2.5.2 Quality Requirements

TOSCA employs *Policies* to describe non-functional behavior and/or QoS that an application and its components can declare to expose. A policy has an abstract Policy Type definition and is instantiated by defining a Policy Template. The Policy Type describes the structure and required parameters of a policy, while the Policy Template is used to define a specific policy instance. Then, Service Templates and Node Templates can declare their non-functional features by referring the Policy Templates describing them.

A Policy Type is a reusable entity that describes a kind of non-functional behavior or a kind of QoS that a Node Type can declare to expose. It defines the structure of observable properties via *PropertiesDefinition*, i.e. the names, data types and allowed values the properties defined in a corresponding Policy Template can have. It can inherit properties from another Policy Type by means of the *DerivedFrom* element. A Policy Type declares the set of Node Types it specifies non-functional behavior for via the *AppliesTo* element. Note that being “applicable to” does not enforce implementation. Whether or not an instance of a Node Type to which a Policy Type is applicable will show the specified non-functional behavior, is determined by a Node Template of the corresponding Node Type.

The following pseudo schema defines the XML syntax of Policy Types.

```
<PolicyType name="xs:NCName"
            policyLanguage="xs:anyURI"?
            abstract="yes|no"?
            final="yes|no"?
            targetNamespace="xs:anyURI"?>
  <Tags>
    <Tag name="xs:string" value="xs:string"/> +
  </Tags> ?

  <DerivedFrom typeRef="xs:QName"/> ?

  <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

  <AppliesTo>
    <NodeTypeReference typeRef="xs:QName"/> +
  </AppliesTo> ?

  policy type specific content ?

</PolicyType>
```

The attribute *policyLanguage* specifies the language used to be specify the details of the Policy Type. These details can be defined as policy type specific content of the PolicyType element. For this attribute, we can select whatever language we want, such as the WS-Agreement used in our Application Metamodel in Section 4.2, or even we can use YAML or XML.

The *PropertiesDefinition* element specifies the structure of the observable properties of the Policy Type by means of XML schema. It has one but not both of the following properties:

- *element*: This attribute provides the QName of an XML element defining the structure of the Policy Type Properties.
- *type*: This attribute provides the QName of an XML (complex) type defining the structure of the Policy Type Properties.

The following shows a snippet of an example Policy Type “HighAvailability”, in which the "HAProperties" element defines the properties of the Policy Type, and it is defined as an XML element.

```
  <PolicyType name="HighAvailability">
    <PropertiesDefinition element="spp:HAProperties"/>
  </PolicyType>
```

Correspondingly, a Policy Template represents a particular non-functional behavior or QoS that can be referenced by a Node Template. It refers to a specific Policy Type that defines the structure of observable properties (metadata) of the non-functional behavior, and then typically defines values for those properties inside the *Properties* element.

The following pseudo schema defines the XML syntax of Policy Template.

```
<PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">

  <Properties>
    XML fragment
  </Properties> ?

  <PropertyConstraints>
    <PropertyConstraint property="xs:string"
                        constraintType="xs:anyURI"> +
      constraint ?
    </PropertyConstraint>
  </PropertyConstraints> ?

  policy type specific content ?

</PolicyTemplate>
```

The *Properties* element specifies the invariant properties of the Policy Template, i.e. those properties that will be commonly used across different contexts in which the Policy Template is used. The initial values are specified by providing an instance document of the XML schema of the corresponding Policy Type Properties. This instance document considers the inheritance structure deduced by the *DerivedFrom* property of the Policy Type referenced by the type attribute of the Policy Template.

The following shows a snippet of an example Policy Template "MyHAPolicy", which is of type "HighAvailability" described above. This Policy Template provides values for the properties defined by the Properties Definition of the “HighAvailability” Policy Type. The AvailabilityClass property is set to “4”. The value of the HeartbeatFrequency is “250”, measured in “msec”.

```
<PolicyTemplate id="MyHAPolicy"
                name="My High Availability Policy"
                type="bpt:HighAvailability">
    <Properties>
      <HAProperties>
        <AvailabilityClass>4</AvailabilityClass>
        <HeartbeatFrequency measuredIn="msec">
           250
        </HeartbeatFrequency>
      </HAProperties>
    </Properties>
</PolicyTemplate>
```

With the Policy Types and Policy Templates are defined, then we can specify the policies for Node Templates (or for Service Template via *BoundaryDefinitions*) in their definitions. The following pseudo schema shows the snippet of such XML syntax.

```
<NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
             minInstances="xs:integer"?
             maxInstances="xs:integer | xs:string"?>
  <Policies>
    <Policy name="xs:string"? policyType="xs:QName"
            policyRef="xs:QName"?>
      policy specific content ?
    </Policy> +
  </Policies> ?

</NodeTemplate>
```

The *Policies* element specifies policies associated with the Node Template. In this element, the *policyType* attribute specifies the type of this Policy. The QName value of this attribute should correspond to the QName of a *PolicyType* defined in the same Definitions document or in an imported document.

The QName value of the *policyRef* attribute references a Policy Template that is associated to the Node Template. This Policy Template can be defined in the same TOSCA Definitions document, or it can be defined in a separate document that is imported into the current Definitions document. The type of Policy Template referenced by the *policyRef* attribute must be the same type or a sub-type of the type specified in the *policyType* attribute.

Note that this policyRef attribute is optional, which means that we do not need to reference a Policy Template. If no Policy Template is referenced, the policy specific content of the *Policy* element alone is assumed to represent sufficient policy specific information in the context of the Node Template. In this part, we can use any language that we like to describe the QoS and/or non-functional properties (we just need to specify this language in the *policyLanguage* attribute of the corresponding Policy Type). Therefore, through these ways, we can easily represent the Quality Requirements defined in our Application Metamodel in TOSCA.

Moreover, since Policy Templates can provide invariant information about a non-functional behavior (i.e. information that is context independent, such as the availability class of an availability policy), we can use the *Policy* element defined in a Node Template to provide variant information (i.e. information that is context specific, such as a specific heartbeat frequency for checking availability of a component) in the policy specific body of the *Policy* element.

In addition, since the Module in the Application metamodel can have associated a *Management Policy*, which defines the actions to be executed when some condition happens. This management policy can be described either in Brooklyn [15] language or in some abstraction that maps into it. Therefore, we can also use Policies in the manners described above to represent such management policy in TOSCA.

2.6 Nuro (early) Case: Topology Specification in TOSCA

In order to illustrate the topology specification using TOSCA, we use the early version of the Nuro Case, which consists of two modules: PHP and Database.

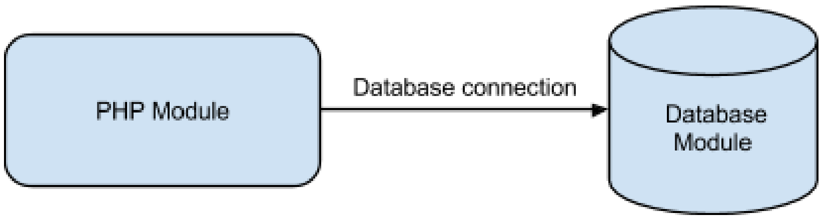

Figure 4. Nuro case requirement

The Nuro systems are based on PHP and MySQL, thus the needed modules for this case are PHP module and Database (MySQL) module. We use the Winery tool [21], developed in the context of the open source OpenTOSCA environment, to generate the topology model of this case. The tool allows the representation of the application's modules through forms and the composition of the topology in a graphical way by means of the drag-and-drop technique. Figure 5 presents the topology for the Nuro Case using Winery.

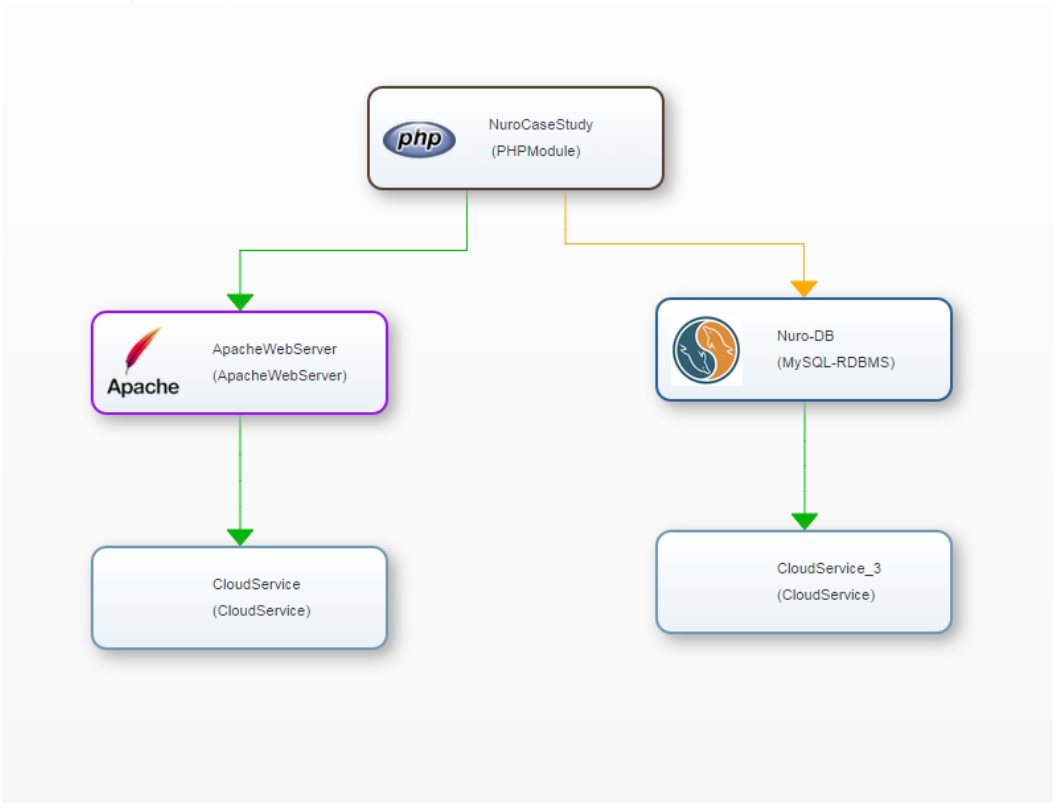

Figure 5. TOSCA-compliant topology specification for Nuro Case in Winery

3. Discovery of Capabilities and Services Featured by Cloud Providers

Together with the description of the user input, the planner component needs to know the possible services available from cloud providers. Multiple cloud resources can be offered by cloud providers at different levels of abstraction. Therefore, the aim of this section is to provide a cloud meta-model to represent and generalize cloud services (infrastructures and platforms), by following an approach similar to the one presented in Section 2 to model a cloud application.

To this end, a study of the common services featured by some of the topmost cloud providers has been performed and it has been summarized in Annex B. This study has highlighted that different providers offer services with certain configurations, cost profiles and policies. Nevertheless, one of the objective of the SeaClouds platform is to offer an automatic way to compare these services, in order to select the one that best fit user's requirements. Therefore, a general model to describe cloud services is needed.

3.1 Definition of a General Cloud Profile Model

The cloud profile model is a generalization used to represent in a common way the services offered by different cloud providers. The model proposed in this sub-section has been defined by taking into account those services that are common for several cloud providers.

In order to ease the matchmaking process, the services described in the cloud profile model should be similar to the services required by the user to run her application modules. Moreover, since PaaS and IaaS providers offer services at different levels (infrastructure and platform), the proposed model should reflect this distinction.

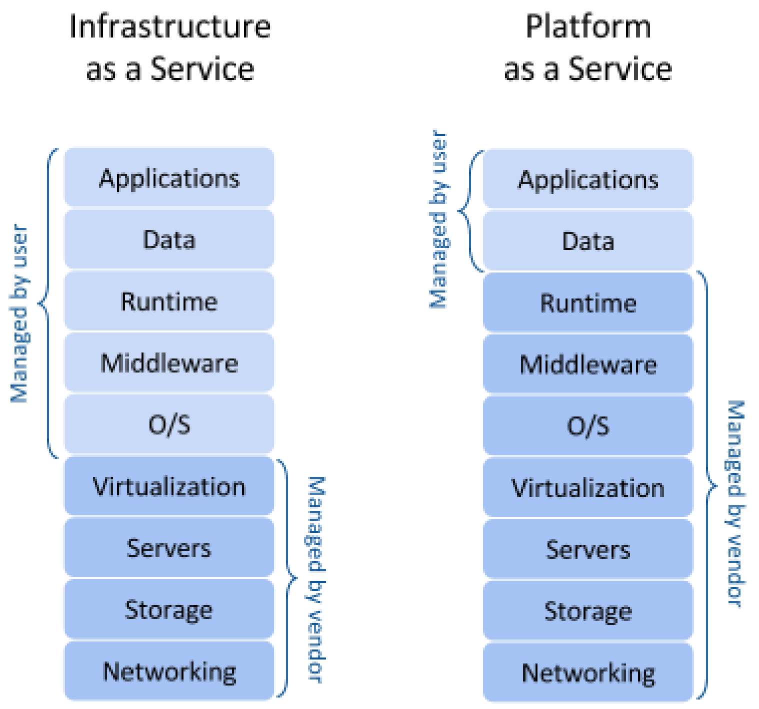

Figure 6. Cloud service models

Generally speaking, IaaS providers deliver an infrastructure mainly formed by computing, storage and networking resources, while PaaS providers deliver computational resources through a platform. In the latter case the user does not care about the management of the underlying layers of hardware and software, but this advantage is counterweighted by a loss in flexibility.

In the following we present the description of the cloud profile model, proposed to provide a general representation of cloud providers and the services they offer. This model is enough general to allow cloud services to be represented independently from a specific cloud provider. This model has been derived by the generalization of the cloud providers services analyzed in Annex B and it represents a simplification of the Cloud Provider Independent Model (CPIM) described in the MODACloud project [16].

A generic **CloudProvider** is usually an entity that offers several **CloudServices**, which can be classified as **Iaas-, PaaS-** and **SaaS-Services.** The proposed model is going to focus on the first two types of services: **IaaS-Service** and **PaaS-Service**. An IaaS-Service is a CloudService composed of one or more **CloudResources**, while a PaaS-Service is composed of one or more **CloudPlatforms**. A cloud application provided by a user can be deployed on CloudPlatforms or run directly on CloudResources (in this case the cloud application provided will include any middleware needed by application itself to run). CloudPlatforms and CloudResources can be generalized by the concept of **CloudElement**, which can be characterized by a **CostProfile** (used to express the pricing model of that service). A CloudService can also have **ScalingPolicies**, which are composed by **ScalingRules**. Each scaling rule describe the metric of interest, the threshold value and the activation rule. Usually, scaling policies are defined on one or more **ResourcePools**, on which CloudPlatforms or user applications can run. The entities described so far are represented in the model in Figure 7.

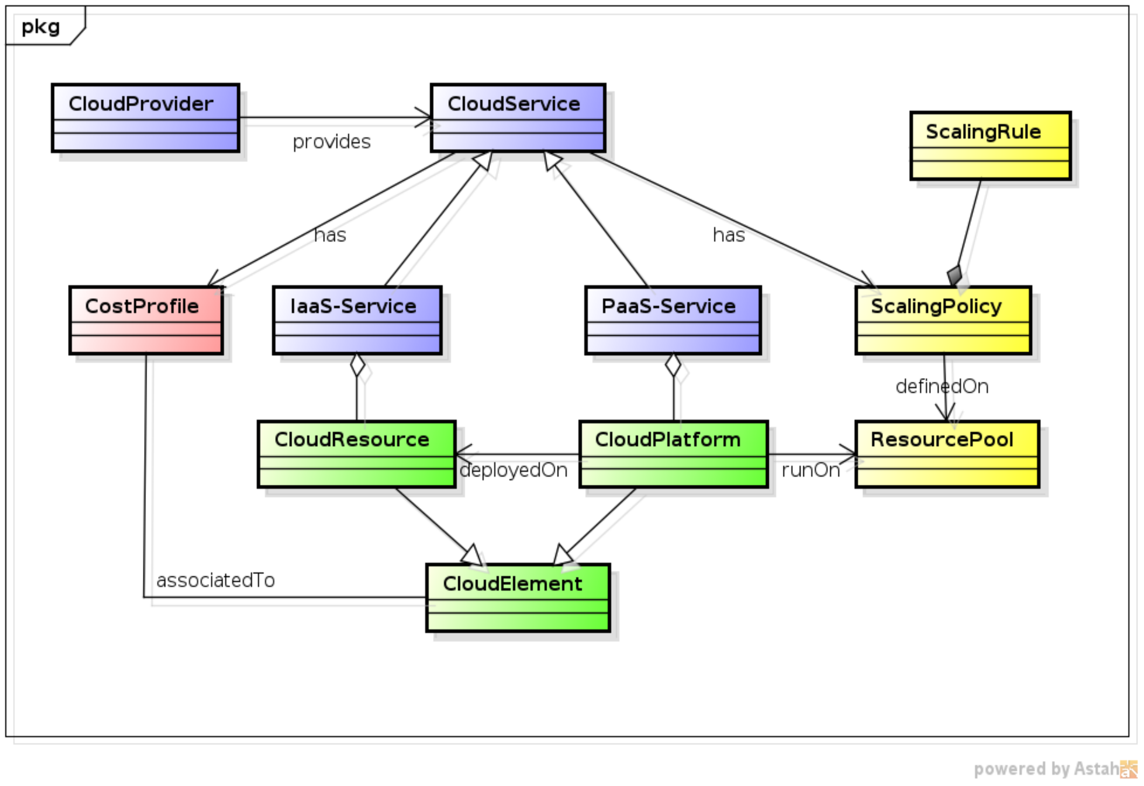

Figure 7. General cloud profile model

In the following we will further detail this model, and in particular we will refer to services offered by IaaS and PaaS providers.

3.1.1 A Model for IaaS Services

In the previous paragraph, we have defined a **CloudResource** as a particular kind of **IaaS-Service**. A CloudResource is the minimal resource unit provided by an IaaS-Service and can be classified into **Compute** and **Storage**. A Compute unit represents a general computational resource, like a Virtual Machine. A Storage unit is a resource that can store structured or unstructured data. As identified in the previous sections, there are three types of storage: **BlockStorage**, **FileStorage** and **ObjectStorage**. BlockStorage is organized into unstructured blocks, FileStorage provides access to a file system, while ObjectStorage provides access to whole objects. For some CloudResources it is possible to define a **Location**, which specifies where the infrastructure provided by the cloud resource is located.

As said before a CloudResource is a specific CloudElement and a CloudElement can be connected to one or more CloudElements through point-to-point **Connections** in order to create a virtual network of CloudElements (topology). A **ResourcePool** is a set of Compute CloudResources and it is associated to an **AllocationProfile**, which is a set of **Allocations** specifying how the number of allocated instances within the Resource Pool changes in a certain reference period. In this way, the model is able to consider scaling policies.

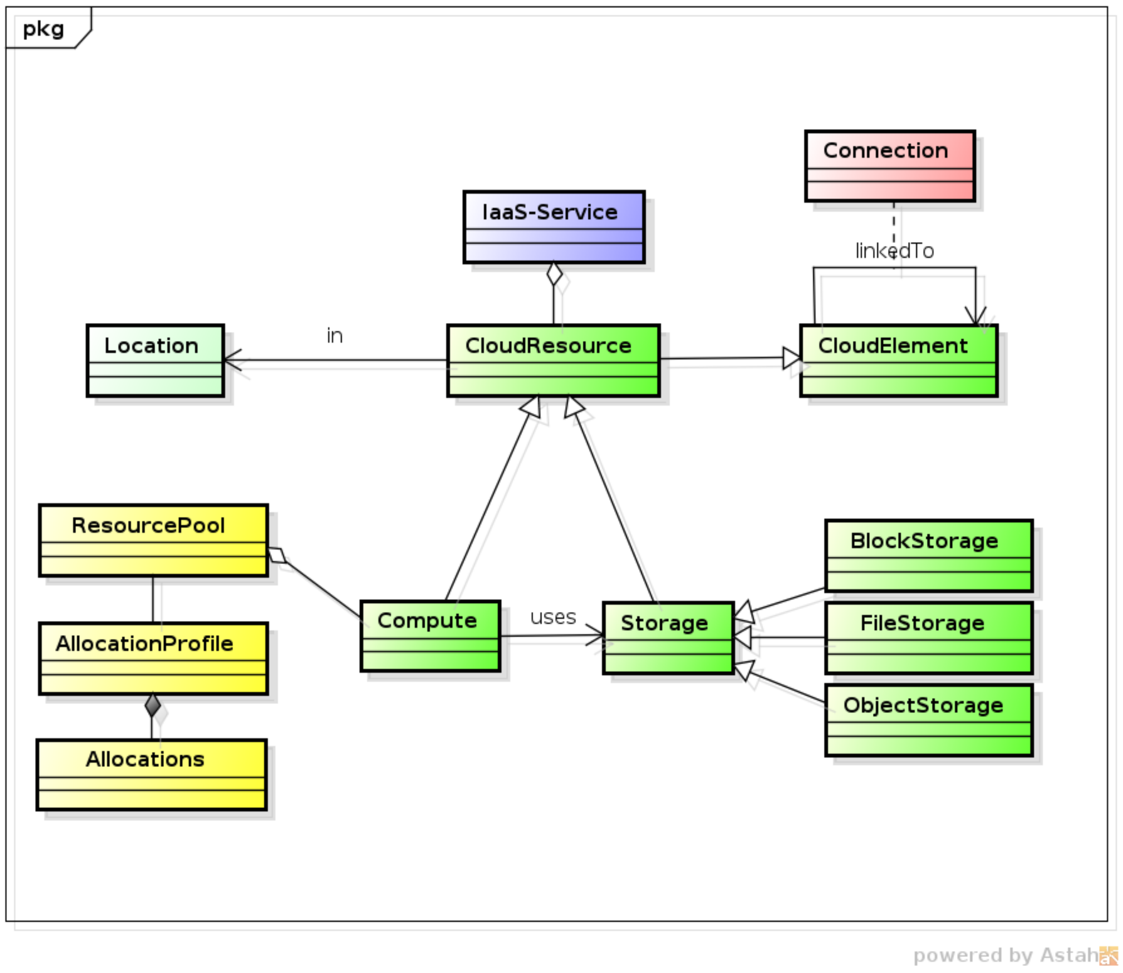

Figure 8. Cloud profile model focused on IaaS services

3.1.2 *Model for PaaS Services*

A **CloudPlatform** is a software framework exposing a defined API provided to a user to develop custom applications and services and it also provides an execution environment for them. A CloudPlatform runs on at least one CloudResource. There are several types of cloud platform services. **WebServer**, **Firewall**, **RuntimeContainer**, **LoadBalancer**, **MessageQueue**, **Database** are the services most commonly offered by PaaS providers. A Database platform can store structured or semi-structured data and can be specialized into **RelationalDB** and **NoSQLDB**. A RelationalDB is based on the relational model, while NoSQLDB is based on a distributed architecture, with data having no relational structure. A CloudPlatform is a CloudElement itself, therefore it can be linked to other CloudElements.

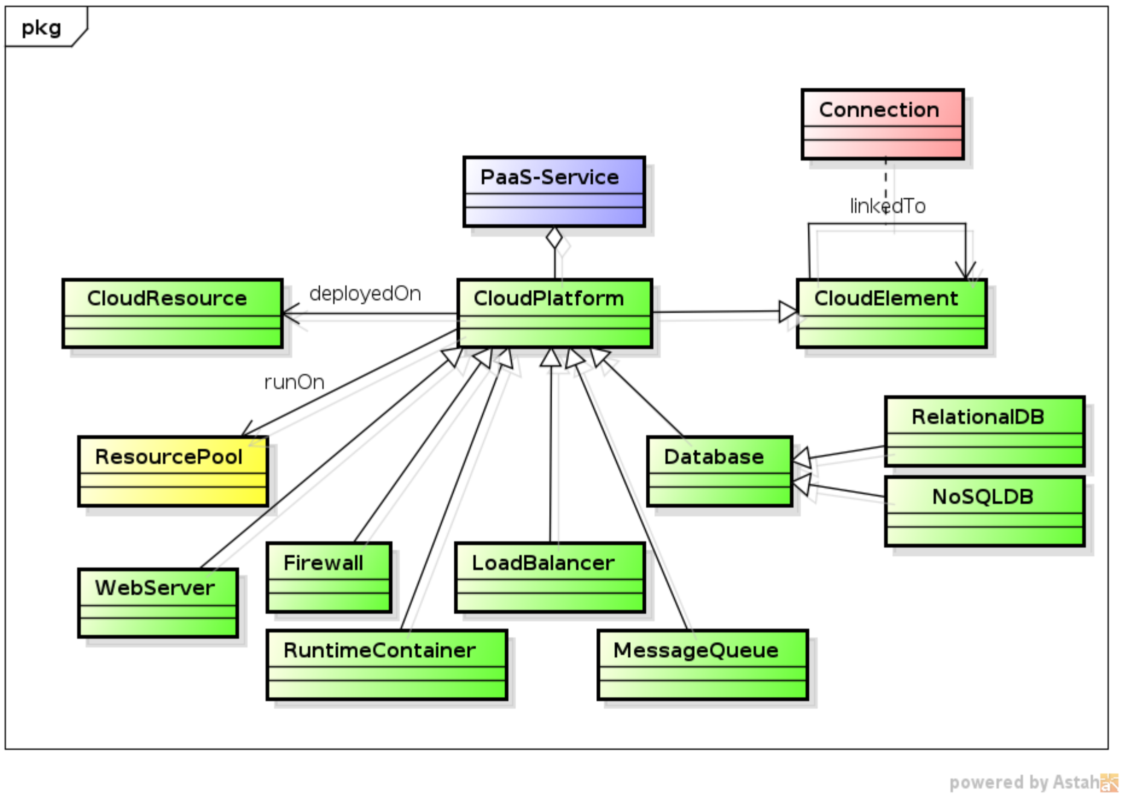

Figure 9. Cloud profile model focused on PaaS services

The overall model, including PaaS and IaaS services, is depicted in Figure 10.

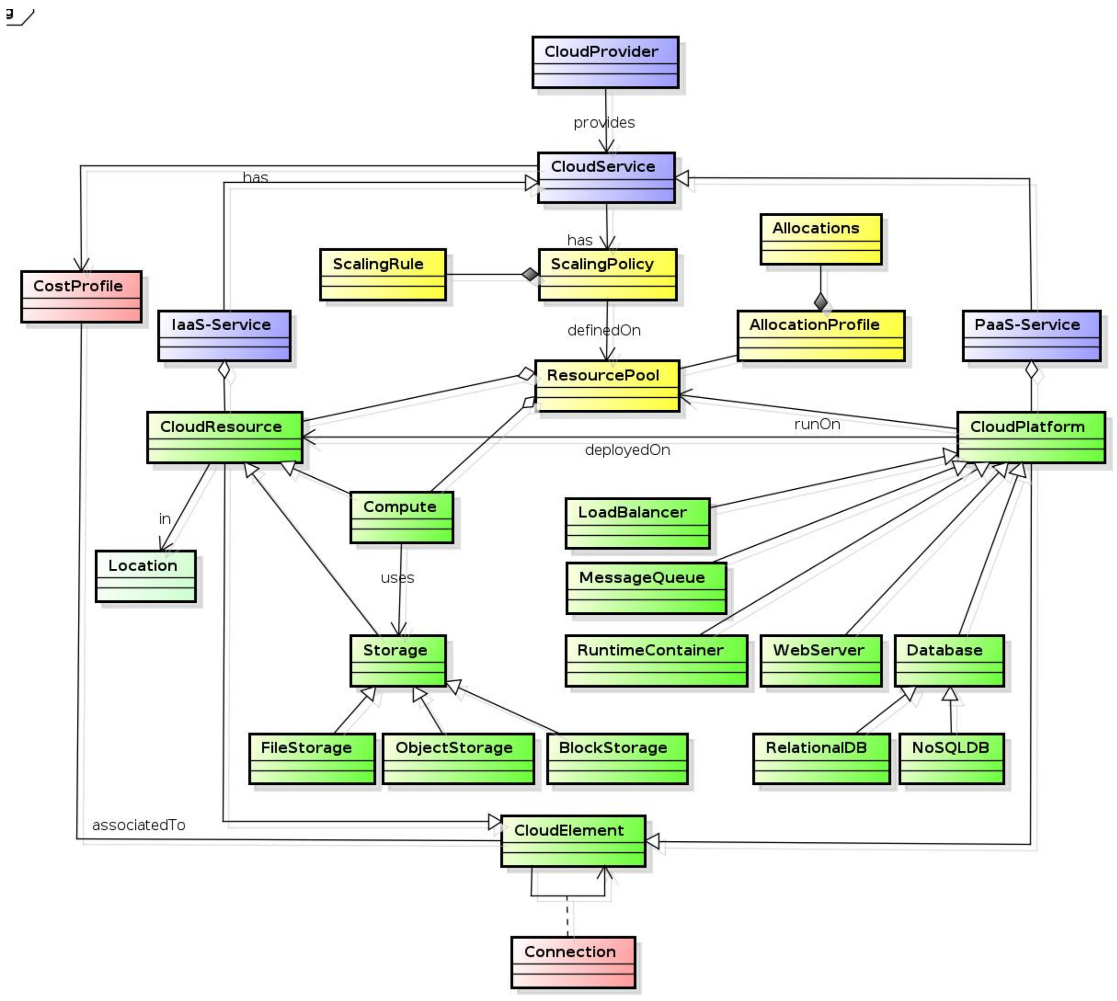

Figure 10. Overall cloud profile model

3.2 Definition of the Cloud Profile Model in TOSCA

Amazon and Rackspace are converging towards the use of a template to describe services orchestration. Amazon proposes AWS Cloud Formation Template [17] as a way to create a collection of AWS resources and provision them to form a stack that will be used to run the user application. In a similar way, Rackspace uses the OpenStack Heat Orchestration Template_to describe the infrastructure for a cloud application in a human readable text file.

Generally speaking, each template offers means to map the services required by a user application to the resources offered by a cloud provider. To this end, each provider defines a set of resources types that can be used in the corresponding template. For example Amazon defines a resource of "AWS::EC2::Instance" for an amazon EC2 computing instance. Similarly, OpenStack defines its resource types, and more specialized resources are added to support RackSpace cloud.

As highlighted in the previous section, some of the resources belonging to different providers can be abstracted by representing them with a common high-level definition and then specialized for each provider. One of the objectives of the TOSCA standard is precisely to provide a meta-model to describe cloud services in a uniform way, in order to promote application portability among different cloud providers.

In this sub-section, we describe how the concepts of TOSCA (see Annex A) can be used to describe the concepts introduced in the Cloud Profile Model, by focusing in particular on the structure of a service and therefore on the Topology Template. In practice, a cloud service can be represented in TOSCA by a Service Template including no Plans. All elements needed to define a TOSCA Service Template - such as Node Type definitions, Relationship Type definitions, etc. - as well as Service Templates themselves are provided in one or more TOSCA *Definitions* documents, which can contain only element definitions of building blocks, or complete models of cloud services (definitions plus instantiations).

The most general block in the proposed cloud profile model is the CloudElement that can be seen as a general representation of any service in the cloud. This concept can be easily mapped to the concept of *NodeType* in TOSCA. A Node Type is a reusable entity that defines the type of one or more Node Templates. For example any CloudResource or CloudPlatform can be described by a NodeType. In the following example, we show the "VirtualMachine" NodeType by referencing an externally defined set of standardized properties, and by declaring that it provides the capability to "host" an operating system (or node).

```
<NodeType name="VirtualMachine">
<documentation> A basic cloud compute resource</documentation>
<DerivedFrom typeRef="tns:RootNodeType"/>
  <PropertiesDefinition element="tns:VirtualMachineProperties"/>
  <CapabilityDefinitions>
    <CapabilityDefinitioncapabilityType="tns:OSContainerCapability"
    lowerBound="0" name="os" upperBound="1"/>
  ...
  </CapabilityDefinitions>
...
</NodeType>
```

Where the externally provided "VirtualMachineProperties" complex type would be defined as follows:

```
<xs:complexType name="VirtualMachineProperties">
    <xs:sequence>
      <xs:element default="1" name="NumCpus">
        <xs:annotation>
          <xs:documentation>Number of CPUs</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:enumeration value="1"/>
            <xs:enumeration value="2"/>
            <xs:enumeration value="4"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="Memory" type="xs:int">
        <xs:annotation>
          <xs:documentation>Memory size (in MB)</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Disk" type="xs:int">
        <xs:annotation>
          <xs:documentation>Disk size (in GB)</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

This complex type defines the properties that can be used for the “VirtualMachine” NodeType. For example the number of CPUs can be defined and its value can assume 1, 2 or 4. Moreover, it is possible to specify the memory size in MB and the disk size in GB.

The same approach can be applied to any kind service. The following example describes “WebServer” and “ApacheWebServer” NodeTypes:

```
  <NodeType name="WebServer">
    <documentation>Web Server</documentation>
    <DerivedFrom typeRef="tns:RootNodeType"/>
    <RequirementDefinitions>
      <RequirementDefinition lowerBound="1" name="container"
    requirementType="tns:SoftwareContainerRequirement"
upperBound="1"/>
    </RequirementDefinitions>
    <CapabilityDefinitions>
      <CapabilityDefinition
        capabilityType="tns:WebApplicationContainerCapability"
        lowerBound="0" name="webapps" upperBound="unbounded"/>
    </CapabilityDefinitions>
  </NodeType>
```

```
  <NodeType name="ApacheWebServer">
    <documentation>Apache Web Server</documentation>
    <DerivedFrom typeRef="ns1:WebServer"/>
    <PropertiesDefinition element="tns:ApacheWebServerProperties"/>
    <CapabilityDefinitions>
      <CapabilityDefinition
        capabilityType="tns:ApacheWebApplicationContainerCapability"
        lowerBound="0" name="webapps" upperBound="unbounded"/>
      <CapabilityDefinition
        capabilityType="tns:ApacheModuleContainerCapability"
        lowerBound="0" name="modules" upperBound="unbounded"/>
    </CapabilityDefinitions>
    <Interfaces>
      <Interface name="http://www.example.com/interfaces/lifecycle">
```

```
        <Operation name="install"/>
        <Operation name="configure"/>
        <Operation name="start"/>
        <Operation name="stop"/>
        <Operation name="uninstall"/>
      </Interface>
    </Interfaces>
  </NodeType>
```

3.2.1 Abstract and Concrete Cloud Services

In Section 2.2 the difference between Abstract and Concrete Services has been introduced. From the previous examples it is possible to understand in practice this difference. An Abstract Service is a general description of a service needed by a module to run, for example a web application can require to be hosted on a "WebServer", without specifying exactly which kind of web server is needed, thus leaving to the matchmaking process the possibility to retrieve all the possible "WebServer" services available. On the other hand, a Concrete Service can be considered as a *specialization* of the correspondent Abstract Service. In the previous example this is exactly represented by the "ApacheWebServer" Node Type. To this end, TOSCA provides the concept of inheritance among Node Types: a Node Type can inherit properties from another Node Type by means of the DerivedFrom element. Moreover, the Node Types might be declared as abstract, meaning that they cannot be instantiated. The purpose of such abstract Node Types is to provide common properties and behavior for re-use in specialized, derived Node Types. Node Types might also be declared as final, meaning that they cannot be derived by other Node Types.

3.2.2 Services Instantiation

As explained so far, Node Types are reusable entities. They can be actually instantiated by means of Node Templates. In this way, the same Node Type can be instantiated by different Node Templates corresponding to different services featured by different cloud providers. The following example shows how to instantiate a Node Template of type "ApacheWebServer".

```
  <NodeTemplate id="ApacheWebServer" name="Apache Web Server"
                type="ns2:ApacheWebServer">
    <Properties>
      <ns2:ApacheWebServerProperties>
        <httpdport>80</httpdport>
      </ns2:ApacheWebServerProperties>
    </Properties>
  </NodeTemplate>
```

3.2.3 Turning Service Templates into Composable

Often, not only individual types can be reused but complete topologies are meaningful in many situations. For example, in most cases an entire stack offered by a PaaS provider can be reused. TOSCA supports to model a corresponding service template and turn this service template into a substitutable for a node type. This is achieved by means of the BoundaryDefinitions element: this element contains nested elements that can refer to the constituencies (like node templates etc.) of the service template and "export" corresponding definitions "to the boundary" of the service template. This way, the service template "looks like a node type" and can be used as such in another service template. This approach can be used for example to export the ensemble of services normally offered as a single service by a PaaS provider. For instance, a service template composed of a Container Node Type (that is the basic compute unit offered by a PaaS provider) plus WebServer and a LoadBalancer NodeTypes can be turned into a composable.

3.2.4 Cost Profile, Location and Scaling Policies

Sometimes services can be associated to concepts that describes their non-functional behaviors and/or quality of service. To express these concepts TOSCA provides Policy Types, which are reusable entities that describes non-functional behaviors or QoS properties that a Node Type can declare to expose. For example, a Policy Type can be defined to express high availability for specific Node Types (e.g. a Node Type for an application server). As well as Node Types, Policy Types can be instantiated by means of Policy Template.

In addition, TOSCA provides a way to describe logical groupings (parts) of an application that should be deployed, configured and managed together as "tiers". This same grouping concept can be associated to scaling policies that apply to each tier so that they can be independently scaled by cloud service providers to accommodate variations in consumer demand. To this end, a Node Type "Tier" and the corresponding Node Template can be used. For example, a Tier node can group both an "ApacheWebServer" Node Template and a "VirtualMachine" Node Template and define a scaling policy for the whole group. In this way both components (i.e. the compute nodes and the software on-top of each compute node) will be scaled as a unit.

```
<NodeType name="Tier">
    <documentation>Tier</documentation>
    <DerivedFrom typeRef="tns:RootNodeType"/>
    ...
</NodeType>

<NodeTemplate id="WebTier" name="Web Tier"
                type="ns1:Tier">
</NodeTemplate>
```

4. Planner Service

The Planner component is in charge of providing an Abstract Deployment Plan (ADP) that defines where each application module will be deployed. Given a set of modules with their requirements, the topology of the application and a set of cloud resources, the Planner will generate an ADP that meet the requirements specified by the user. The Abstract Deployment Plan includes the concrete services associated with each Base module and the policies to manage the scaling mechanism of each module. The ADP is described in detail in section 4.2.3.

The generation of the Abstract Deployment Plan can be performed in two steps:
1) Matchmaking: this first step aims to identify the cloud resources that are suitable to allocate each module. To this end, the user can be allowed to specify its requirements as hard or soft requirements. Hard requirements must be satisfied and will be used during the matchmaking to discard all the cloud services that do not match them, whereas both hard and soft requirements can be used in the subsequent step.
2) Optimization: once a set of suitable cloud services have been identified for each modules, an optimization process can be performed. The space of solutions is composed of the set of Abstract Deployment Plans that can be obtained by allocating each module on the different cloud resources (mapping).

4.1 Matchmaking

The matchmaking activity of the Planner module is in charge of identifying, for each module, a set of cloud resources that satisfy the technology requirements specified by the user. This first step aims to reduce the search space, by selecting only those services that can be used to construct candidate solutions for the optimization step. In fact, the matchmaking step does not take into account QoS requirements and possible relationships among modules.

4.1.1 Inputs and Output

The first input of the matchmaking component is a set of service descriptions provided by the user. For each module the user defines a service description, which describes the capabilities needed to host that module.

The second input of the matchmaking component is a set of concrete service descriptions offered by cloud providers. The ensemble of these services is stored in a repository generated and updated by the discoverer component. Since both IaaS and PaaS providers can offer their services, this repository contains services at IaaS and PaaS level. Therefore, the user can specify either a service description at PaaS level or at IaaS level. In the second case, the user will be responsible to provide and install the platform services needed by her application to be run.

Given these inputs, the matchmaking component matches, for each module, the service description provided by the user with the service descriptions available in the repository and provides, for each module, a list of feasible services. This information is then passed to the optimization component.

4.1.2 Matchmaking Process

The matchmaking process is performed on each application module separately. As can be seen from the Application meta-model described in section 2.2, a module is associated to a set of requirements that are mapped by an Abstract Service. Thus, for each Module, the user is required to provide a description of the environment needed by the module itself to run. The module requirements will be mapped into this environment. In the following we give an example of this process.

Let consider a web application composed of three modules that needs three services to run: a WebServer, a Database and a Scheduler. The following requirements have been specified for the WebServer: it should support PHP 5.2 or 5.5 and HTTP over SSL connections. Thus, for the PHP module, the user will provide the description of the WebServer service and its characteristics. In this case the user is specifying an Abstract Service, since she has no preference about the kind of WebServer to be used for her application. The matchmaker component will be in charge of finding all the possible Concrete Services offered by cloud providers that realize a certain Abstract Service. On the other hand, the user can decide to directly specify a Concrete Service, for example an ApacheWebServer. The process to be followed is exactly the same and in this case the work for the matchmaker component is easier since it will look for exact matching between Concrete Services.

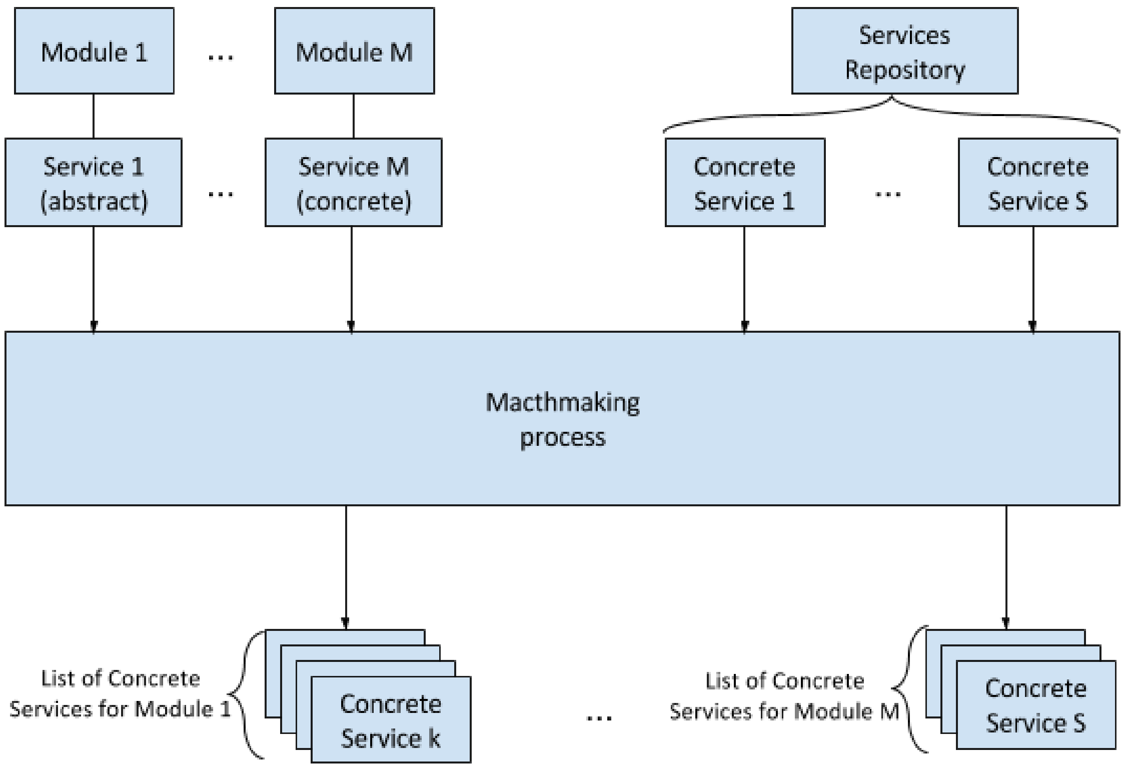

Figure 11. Initial approach for the matchmaking process

A high-level algorithm for the optimization problem is shown in the following:

```
while (notEmpty(listOfModules)) loop
    foreach (module in listOfModules) loop
        desiredService = getService (module);
        foreach (service in offeredServices) loop
            if (match (service, offeredService) then
                add(service, candidateService(module));
            endif;
        endfor;
    endloop;
endloop;
```

4.2 Optimization

The optimization activity of the Planner component decides, among all the options that satisfy the technology requirements provided by the matchmaking activity, a suitable combination of cloud resources that allows the application to satisfy its quality requirements, at the same time that considers the utilization of cloud elasticity.

This a challenging task from the research point of view since approaches in the state of the art on this topic do not completely offer all the functionalities that the Planning module of SeaClouds would provide. In Annex C, we have summarized approaches appeared in the scientific literature that deal with resource allocation challenges, which are strongly related to the ones faced by the SeaClouds optimization activity.

The rest of this section explains in detail the outputs provided by the optimization activity, the input information that is required in order to create the outputs, and the process to create the outputs from the input information.

4.2.1 Required and Provided Information

Once the optimizer activity has executed, the Planner will be able to provide information regarding:

- Allocation of each base module onto one or more locations of independent availability.
- Allocation of modules into concrete resource types; i.e., decide for the execution of each module using PaaS or IaaS, among the feasible alternatives provided by the matchmaking activity.
- For each module that has been decided to make use of PaaS, the characteristics of the platform are provided.
- For modules that make use of IaaS, the type of virtual machines in which they will execute and number of them are provided.
- Scaling mechanisms for each module that executes on IaaS. This output contains information for the application to take advantage of the elasticity offered by cloud providers.

For creating this information, principles of the fields of performance and availability evaluation are exploited. According to the information usually required by research works in these areas, the optimizer will require the following information as input:

- Application topology. This information is described in object Topology in the application model. It is useful for performance and availability evaluation.
- Candidate services for each base module, provided by the previous matchmaking. This is used for availability, performance and cost evaluation.
- Quality Requirements. This information is described in object Quality of Service in the application model.
- Expected application workload. This information is used for performance evaluation.
- Operational profile of the application, including:
    - Routing between modules
    - Service demand of each module
    - Amount of data transfer between modules in terms of size of messages

  This information is used for performance and evaluation.

- Maximum parallelism level of modules. This information is used for performance evaluation. It is an important information for modules that execute in IaaS, in order to make an appropriate decision regarding the power and number of cores of the VMs in which they will execute.

Network connection speed between different locations and inside each location.

4.2.2 Optimization Process

The process is based on heuristic search algorithms to obtain a solution that meets requirements. The rationale under the decision of using this kind of algorithms is that they are well suited for finding a solution starting from a set of candidate ones. This is especially useful in the SeaClouds domain problem, where replanning calls (i.e., the Planner executions done at the application's runtime due to some problem in its current deployment) ask for a new deployment solution. In order to avoid reconfiguring the whole application after a problem in some module is found, it is advisable that the planner does not propose a solution that is completely different from the currently deployed configuration.

It is worth noting that this is not the only strategy that could be adopted. The optimization could be also based on other types of approaches in which the maximum "distance" between the current deployment that is no longer suitable and the deployment proposed as a new solution is included as an additional constraint. However, heuristic searches already adopt this concept in their basic definition.

Based on the quality requirements of SeaClouds case studies, the metrics that the optimization activity considers are availability, performance and cost.

Figure 12 shows the decisions that are made by the optimizer, depicting with blue dots the variation points in the problem it faces. Each Base Module in the application (being x_0 of them) can be executed in one or more cloud provider zones. Therefore, if there is x_1 locations, a module can execute in 2^{x1} possibilities. In each location, it may require either IaaS or PaaS resources offered by the cloud. This adds another variation point where two candidate alternatives exist.

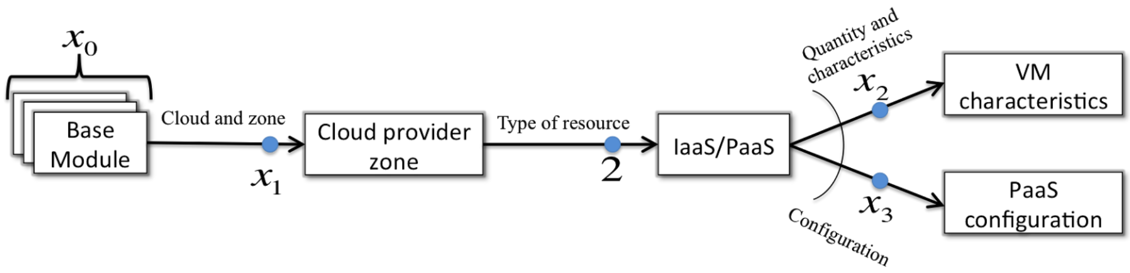

Figure 12. Initial approach for the optimization process

For the alternative of IaaS utilization, the optimizer has to decide the type of virtual machines to use in the selected zone, and the quantity of them. This variation point is denoted with x_2. Being x_2 type of machines, and the budget allowing the use of up to k_i VMs of type *i* (being *i* in the interval [1, ...,x_2]), the number of alternatives in this variation point are $\sum_i k_i$. For the alternative of PaaS utilization, the optimizer has to decide the values of the parameters that the platform allows to configure.

Due to the amount of variation points and the quantity of alternatives of each point, a single heuristic algorithm may not be able to manage the execution in order to find a suitable solution. To solve this challenge, the optimization algorithm is split up into two steps that are executed iteratively. These steps are depicted in Figure 13 and explained in the following.

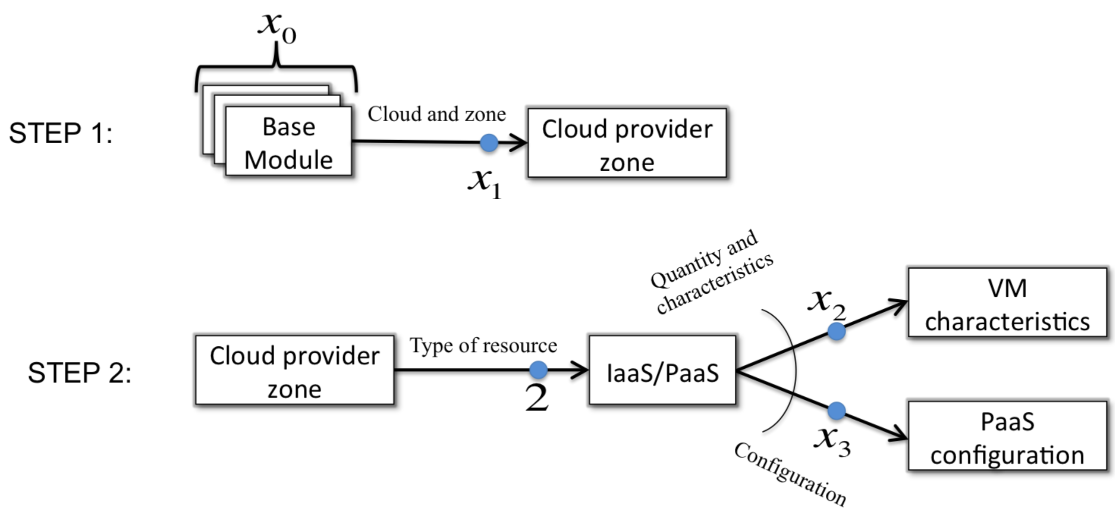

Figure 13. Two-steps optimizer

The first step is in charge of proposing a distribution of the Base modules into cloud provider's locations. This step uses a heuristic search among the candidate distributions. The objectives of this heuristic search is to find a module distribution that satisfies the availability requirement at a bounded cost.

The second step uses a heuristic search to decide the type of resource to use in each of the decided locations. In case of deciding for IaaS, it also searches for the suitable set of virtual machines to use. In case of deciding for PaaS, it searches for a suitable platform configuration. The objective of this heuristic searches is to find the set of resources for each location that allow the application to satisfy its performance requirements at a bounded cost.

If the second step cannot find a suitable solution the process iterates from the first step. The first step will receive information from the second step regarding the locations that caused the most important troubles in terms of a mixture of performance and cost. In this way, the first step will give less weight to such association of modules with locations, then reducing the likelihood that these locations are proposed again when generating the next distribution of modules over locations.

If the second step succeeds, the iterative heuristic process finishes and all the outputs of the optimization process have been generated, except for the scaling mechanisms.

A high-level algorithm for the optimization problem is shown in the following, where searchNeighborBetterThanCandidatePlan method implements the two step heuristic search:

```
candidatePlan=generateInitialRandomCandidate();
bestPlan=candidatePlan;

while (not stopCondition) loop
        newPlan=searchNeighborBetterThanCandidatePlan();
        if(fitness(newPlan)>fitness(candidatePlan)) then
          candidatePlan=newPlan;
        else
          if(fitness(candidatePlan)>fitness(bestPlan)) then
               bestPlan=candidatePlan
               end if;
               candidatePlan=generateInitialRandomCandidate();
        end if;
end loop;
return bestPlan;
```

Note: an example of stopCondition is "Certain number of cycles elapsed without improving bestPlan"

Finally, if the algorithm succeeds searching for a solution that satisfies the quality requirements, the scaling mechanisms of each module that executes in IaaS are created, taking into account the cost of running the chosen solution and the maximum budget for the application.

Example:
For the sake of understandability, next paragraphs describe a simple example of the optimizer execution.

Given an application that consists of two modules M1 and M2 where each request to M1 also requires an execution of M2, as shown in Figure 14. The demand of a request in isolation requires 0.25 seconds in M1 and 0.2 seconds in M2.

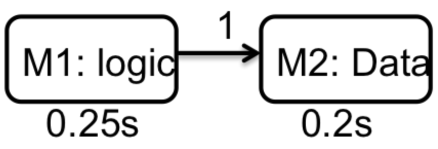

Figure 14. Example of system properties provided in the abstract application model and operational profile.

The usual workload of the application consists of 200 requests per minute. The quality requirements state that:

- Requests have to be answered in less than 1 second in average.
- The system availability should be higher than 99.9%.

- The system costs should not exceed 200€ per month.

The input provided by the matchmaking activity for M1 states that there are two cloud providers for allocating it, offering 5 and 4 different options as IaaS respectively. In turn, for M2 there are also two cloud providers, the first provider offering the first 2 PaaS options and 2 IaaS options, while the second provider offers 1 PaaS option and 4 IaaS options. Each of the options are associated with the information regarding their quality in terms of the performance, availability and cost they provide. The scheme in Figure 15 summarizes this information.

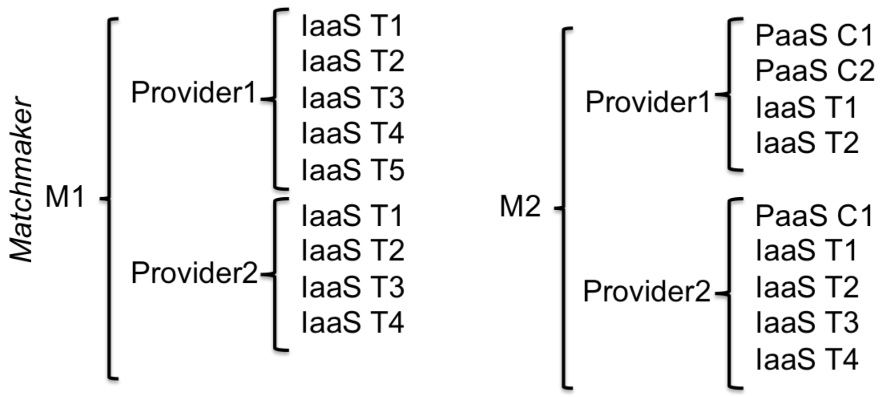

Figure 15. Example of options for each module

Regarding the rest of inputs, the maximum parallelization level of modules states that both M1 and M2 are fully parallelizable, and the network speed between providers has a default value (e.g., 20 MB/s).

An example trace of the behavior of the heuristic algorithm is the following:

- It finds a local optimum by proposing the utilization of Provider1 for M1 and Provider2 for M2 in its first step, and in the utilization of three instances of the option "IaaS T4" for M1 and "PaaS C1" for M2. This local optimum solution is represented in Figure 16.

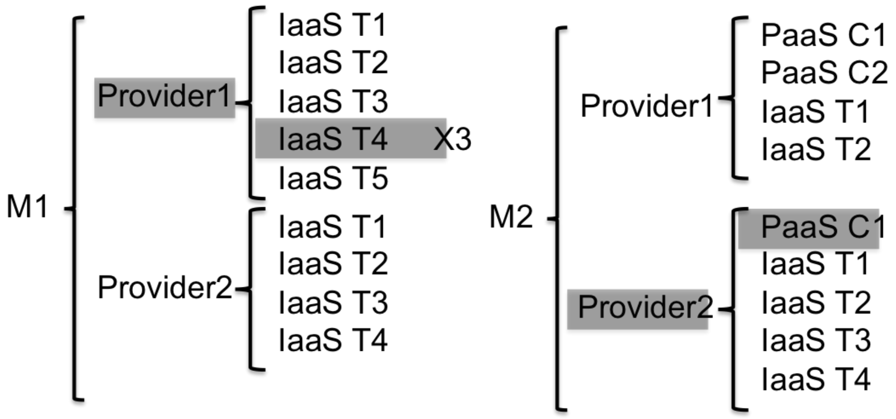

Figure 16. Local optimum found by the heuristic search

- The fitness function states that the solution is not suitable because it satisfies the performance and cost requirements but not the availability one.
- The algorithm continues the iteration to find another local optimum. In this case it still proposes to execute M1 in Provider1 using three instances of "IaaS T4", but now M2 is executed in "Provider 1" using one instance of "IaaS T1". This local optimum is represented in Figure 17.

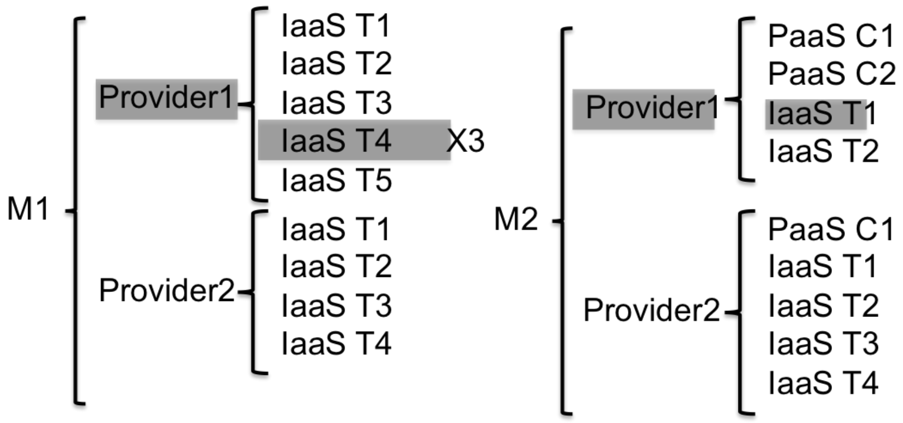

Figure 17. Alternative local optimum found by the heuristic search

- The fitness function in this case states that the local optimum is suitable because it satisfies all the quality requirements and gives it a score based on the values of each of its quality attributes.
- The algorithm continues iterating to find other local optimum satisfying all the quality requirements and having a higher score than the one already obtained. The algorithm will stop when it iterates a certain number times without having found another optimum that improves the score of the current best solution. In that case, it will return the best solution found during all the process.

4.2.3 Abstract Deployment Plan

The ADP generated as output by the planner describes the distribution of the application modules into multiple cloud provider services, so that user requirements are satisfied. Specifically, it includes the concrete services associated with each Base module and the policies to manage the scaling mechanism of each module, to guarantee both performance and cost requirements. The plan is a part of the Deployable Application Model introduced in section 2.1.

We have already mentioned in this document that TOSCA defines a well-built specification and best practices for describing a full-detailed application topology. The standard provides flexibility to describe any application module and its properties, requirements and capabilities. Thanks to these two last features, TOSCA defines a methodology to describe the relationships between modules in a generic way, but maintains the configurability and the requirements to establish and manage the relationships among modules. To this end, we propose to exploit the TOSCA language to define the ADP. Nevertheless, we consider the XML specification could be hard to understand by a regular end-user. Therefore, we propose to base our ADP specification on the TOSCA simple profile specification (the first TOSCA YAML) [22], which is more human-readable.

Then, an example of ADP (considering aspects of the Nuro early case) is reported in the following by exploiting the TOSCA simple profile specification.

```
tosca_definitions_version: tosca_simple_1.0
description: TOSCA simple profile for nuro web application:a php
custom
> application, a web server, and mysql database on two  different
services.
inputs:
 db_name:
      type: string
      description: The name of the database.
 db_user:
      type: string
      description: The username of the DB user.
 db_pwd:
      type: string
      description: The database admin account password.
 db_port:
      type:integer
      description: Port for the MySQL database
 mysql_version:
      type: integer
      description: Version of mysql DB
 php_version:
      type: integer
      description: Version of php
node_templates:
 nuroCaseStudy:
      type: seaClouds.nodes.WebApplication.PHP
      properties:
            version: { get_input: php_version}
      requirements:
               -  host: webServer
               -  database_endpoint: nuroDatabase
      interfaces:
            create: php_install.sh
            configure: php_configure.sh
            start: php_start.sh

 nuroDatabase:
      type: seaClouds.nodes.Database.MySQL
      properties:
      port: { get_input: db_port }
```

```
            db_name: { get_input: db_name }
            db_user: { get_input: db_user }
            db_password: { get_input: db_pwd }
            version: { get_input: mysql_version}
      capabilities:
            database_endpoint:
      requirements:
         -  host: amazon.i2.xlarge
      interfaces:
            create: mysql_db_install.sh
            start: mysql_db_start.sh
            configure: mysql_db_configure.sh
 webServer:
      type: seaClouds.nodes.WebServer.Apache
      requirements:
         -  host: hp-standard.small
      interfaces:
            create: webserver_install.sh
            start: webserver_start.sh

 amazon.i2.xlarge:
      type: seaClouds.nodes.Compute
      properties:
            # compute properties (flavor)
            disk_size: 800
            num_cpus: 4
            mem_size: 30.5
            # host image properties
            os_type: Linux
            region: USeast
            cost: 0.853

 hp.standard.small:
      type: seaClouds.nodes.Compute
      properties:
            # compute properties (flavor)
            disk_size: 10
            num_cpus: 2
            mem_size: 2
            # host image properties
            os_type: windows
            region: USeast
            cost: 0.09

group:
  webserver_group:
      #the nuroCaseStudy and the webServer nodes will be scaled
together
      members: [nuroCaseStudy, webServer]
      policies:
      # Specific policy definitions are considered domain specific in
TOSCA
      - my_scaling_policy:
        #example of Brooklyn policy embedded in TOSCA simple profile
        brooklyn.policies:
         - policyType: brooklyn.policy.autoscaling.AutoScalerPolicy
           brooklyn.config:
            metric: $brooklyn:sensor
("brooklyn.entity.webapp.DynamicWebApp
            Cluster", "webapp.reqs.perSec.windowed.perNode")
            metricLowerBound: 10
            metricUpperBound: 100
            minPoolSize: 1
            maxPoolSize: 5

node_types:
 seaClouds.nodes.WebApplication.PHP
      derived_from: tosca.nodes.WebApplication
      properties:
            version:
             type: string
      requirements:
            - host: seaClouds.nodes.WebServer.Apache
            - database_endpoint: seaClouds.nodes.Database.MySQL

 seaClouds.nodes.Database.MySQL:
      derived_from: tosca.nodes.Database
      properties:
            version:
             type: string
      requirements:
            - host: seaClouds.nodes.Compute
      capabilities:
            database_endpoint: seaClouds.capabilities.DBendpoint.MySQL

 seaClouds.nodes.WebServer.Apache
      derived_from: tosca.nodes.WebServer
      requirements:
            - host: seaClouds.nodes.Compute

 seaClouds.nodes.Compute
      derived_from: tosca.nodes.Root
      properties:
            # compute properties
            num_cpus:
             type: integer
             constraints:
                  - greater_or_equal: 1
            disk_size:
             type: integer
             constraints:
                  - greater_or_equal: 0
            mem_size:
             type: integer
             constraints:
                  - greater_or_equal: 0
            # host image properties
            os_type:
             type: string

            # Compute node’s primary IP address
            ip_address:
             type: string
      capabilities:
            host:
             type: Container
             containee_types: [tosca.nodes.SoftwareComponent]
outputs:
 website_url:
      description: URL for PHP application.
      value: { get_property: [hp.standard.small, ip_address] }
```

5. Preliminary Prototype Description

Based on the specification of the Planner in the previous section, which includes a matchmaking process and then an optimization process, in this section, the preliminary design and descriptions of the Planner component are provided, which include the first design of UML class diagram and the related APIs.

5.1 UML Class Diagram

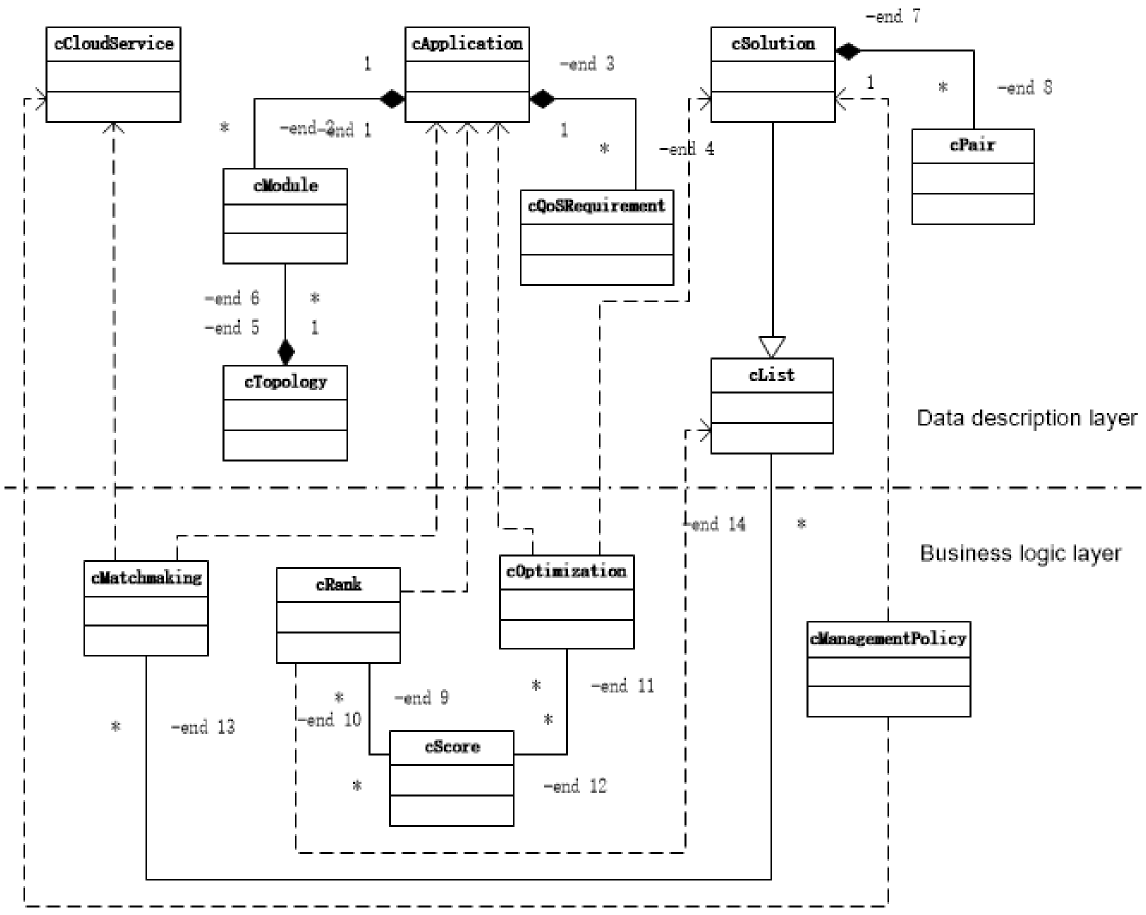

Figure 18. UML class diagram of the Planner

The classes in this diagram are divided into two layers, namely, data description layer and business logic layer, which will be described in detail below.

Data description layer:

- **cApplication:** describe the cloud application provided by the user, which consists of classes cModule, cQoSRequirement, and cTopology.
- **cModule:** describe the modules that compose the cloud application.
- **cQoSRequirement:** describe the QoS requirements of modules and/or the whole application.

- **cTopology:** describe the topology of cloud application and related operations, i.e., the relationships among the modules that compose the application.
- **cCloudService:** describe the cloud offerings advertised by cloud providers (obtained from the Discoverer component).
- **cList:** class of (linked) list, which is used to store the candidate cloud offerings for each module.
- **cPair:** store the pair of module and its corresponding cloud offering, and also other related information. This can be a class or a structure, depending on the convenience.
- **cSolution:** can be inherited from the cList class, and represents the resulted (optimal) solution(s) that is composed of cPair objects. In addition, it also stores the scores of the resulted solution(s).

Business logic layer:

- **cMatchmaking:** the class that implements the matchmaking service. Given a set of cCloudService and a cApplication as inputs, the matchmaking process will generate a cList for each cModule in the application.
- **cScore:** the class for calculating scores for the candidate cloud offerings (considering together the price, availability, performance and so on).
- **cRank:** for a module (cModule), rank the candidate cloud offerings list (cList) in accordance with the score, or select *n* cloud offerings with the highest scores and rank them.
- **cOptimization:** the class that implements the optimization process. Given a cApplication and the cList for each cModule in the application, the optimization process will generate the optimal solution(s).
- **cManagementPolicy:** set the management policies (e.g., scaling policies) for each resource (or cloud zone).

In this design, the class cApplication consists of multiple cModule(s) and cQoSRequirement(s), and the relationships among these cModule(s) are described by cTopology, and it represents the application to be deployed on (multiple) clouds.

cApplication, and cList are the inputs of cRank, which calls cScore to calculate the related “score” for all the candidate cloud offerings of each module, considering the factors as availability, performance and cost, and then output an ordered list of these candidates.

cOptimization takes the ordered list of cloud offerings for every module as input, and implements a combined optimization process to get the optimal or suboptimal solutions, which is stored in cSolution.

Class cSolution is inherited from cList, and it contains multiple pairs of modules and their corresponding cloud offerings (cPair), which represents a distribution of application modules onto (multiple) available clouds.

At last, cManagementPolicy is used to set the scale policies for each cloud zone in cSolution.

In addition to these classes described above, there is also another wrapper class cOptimizer in this design, whose inputs are cApplication (including cModule, cQoSRequirements, and cTopology) and cList, while the cSolution is its output.

5.2 API Design

The initial API design of the planner component is as follows, and more detailed information can be found from deliverable D4.2.

Methods	**Parameters**	**Output**	**Description**
GET CandidateList	Application, CloudService	CandidateList	Gets the list of candidate clouds offerings for each module
GET Solution	Application, CloudService, CandidateList	ResultedSolution	Gets the solutions and returns
GET SimilarSolution	CurrentSolution	NewSolution	Gets a solution that is not very different from the current used, and this will be useful for replanning
POST Application	Application		The application will be deployed
POST Reconfiguration Confirmation	Application		The reconfiguration plan is approved.

Table 2. The initial API design of the planner

6. Concluding Remarks

In this deliverable, we have presented the first specification of the SeaClouds discovery functionality, of the SeaClouds formalism to specify properties and requirements, of the SeaClouds application topology model and of the SeaClouds planning policies. The inputs needed by the platform are analysed and a SeaClouds application meta-model is proposed to represent all the related information. The OASIS standard TOSCA is employed to represent the application topology, and a possible mapping from the proposed application meta-model into TOSCA representations is also presented. To facilitate the discovery of capabilities and services offered by cloud providers, a cloud meta-model is proposed and also mapped into TOSCA representations. The first specification of the planner including a matchmaking process and an optimization process is presented, and a very early prototype design of the planner component is also provided.

In the next deliverable, the second and final documentation of the SeaClouds discovery functionality, of the SeaClouds formalism to specify properties and requirements, of the SeaClouds application topology model and of the SeaClouds planning policies will be delivered.

Annex:

A. Background on TOSCA

In this section, we would like to provide a simple and compact introduction to TOSCA (OASIS Topology and Orchestration Specification for Cloud Applications) [19], which aims at solving the problem of deploying and flexibly managing complex multi-service applications in the cloud, by providing a language to describe and manage complex cloud applications in a portable, vendor-agnostic way. The following descriptions are mainly taken from [19, 20].

TOSCA provides an XML-based modeling language, whose purpose is to allow formalizing the structure of each cloud application as a typed topology graph, and the management tasks as plans. In TOSCA, an application is represented as a ServiceTemplate, which is shown in the following figure. The ServiceTemplate is in turn composed by a TopologyTemplate and (optionally) by some management Plans.

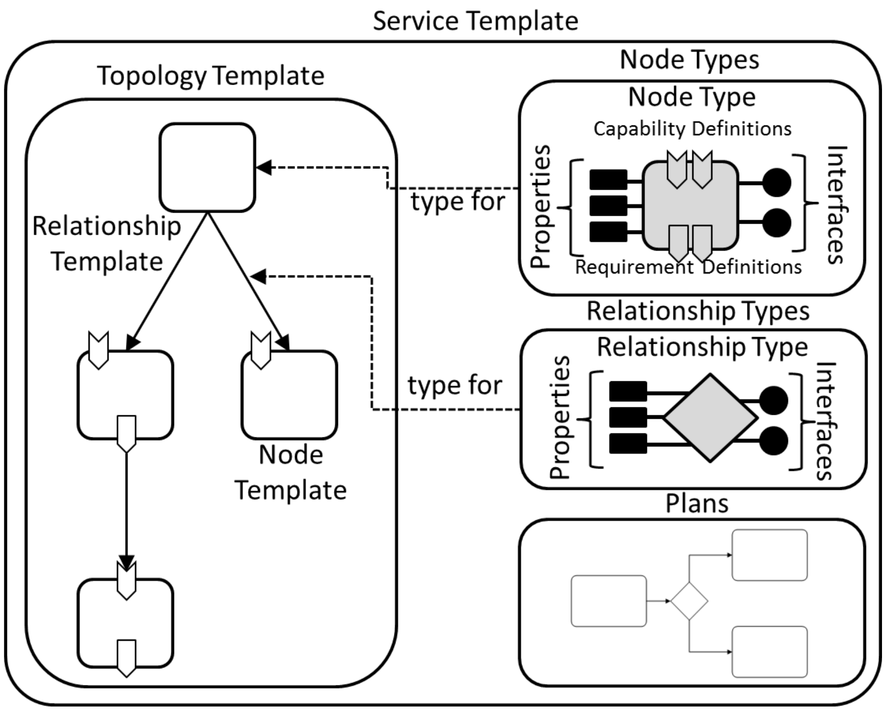

Figure 19. TOSCA Service Template

In this representation, the topology of a multi-component application is represented by means of TopologyTemplates. A TopologyTemplate is essentially a typed graph whose nodes are the application components, and whose edges are the relations between these application components. Syntactically speaking, the application components and their relations are represented by means of typed NodeTemplates and RelationshipTemplates, respectively.

Each application component appears in the topology as a NodeTemplate, and each NodeTemplate is in turn typed. This is because the purpose of NodeTemplates is to define the application-specific features of components (e.g., actual property values, QoS, etc.), while the purpose of the corresponding types is to describe the structure of the features to be specified.

The structure of the features exposed by an application component is defined by means of NodeTypes. More precisely, a NodeType specifies the structure of the observable properties of an application component, the management operations it offers, the possible states of its instances, the requirements needed to properly operate it, and the capabilities it offers to satisfy other components requirements. Syntactically speaking, properties are described with PropertiesDefinitions, operations with Interface and Operation elements, requirements with RequirementDefinitions (of certain RequirementTypes), and capabilities with CapabilityDefinitions (of certain CapabilityTypes).

Moreover, NodeTypes do not specify which are the artifacts required to instantiate and operate application components, since that is the purpose of NodeTypeImplementations. Each NodeTypeImplementation refers to the NodeType whose implementation is under definition and specifies its DeploymentArtifacts and ImplementationArtifacts. The former are the contents (viz., ArtifactTypes and ArtifactTemplates) needed to materialize instances of application components, while the latter are those which implement management operations offered by application components.

With respect to the relations between application components, they can be modeled by means of RelationshipTypes, RelationshipTypeImplementations, and RelationshipTemplates. A RelationshipType defines the structure of a generic relationship between a ValidSource (i.e., a NodeType or a node’s RequirementType) and a ValidTarget (i.e., a NodeType or a node’s CapabilityType). It also allows to describe the operations which can be performed on the source and on the target of the relationship (via SourceInterfaces and TargetInterfaces, respectively), its observable properties, and the possible states of its instances. Each RelationshipType requires to be connected with the artifacts implementing the operations it offers. This is the purpose of RelationshipTypeImplementations, each of which refers to a RelationshipType and specifies its ImplementationArtifacts. More precisely, a RelationshipTypeImplementation links each operation offered by a NodeType with the ArtifactTypes and ArtifactTemplates implementing it. As for nodes, types and type implementations only describe relations in a generic way. Once placed in the topological description of a certain application, they become application-specific and thus require to be described by means of RelationshipTemplates (to describe application-specific features).

TOSCA allows artifacts to represent contents of any type (e.g., script, executable program, installable image, configuration file, library, etc.). This requires to describe artifacts along with the metadata needed to properly access them. The structure of such metadata is described by means of ArtifactTypes, while links to concrete artifacts (and values of invariant metadata) that can be specified by employing ArtifactTemplates.

With respect to management plans, TOSCA prescribes to use workflows to describe Plans (so as to leverage of their suitability to handle errors, exceptions and human interactions), but it does not mandate the use of specific workflow language. Furthermore, Plans are distinguished on the basis of their planType. There are only two predefined types of plans: the BuildPlan type models plans which initially create a new instance of a service template, while the TerminationPlan type is for plans used to terminate the existence of a service instance.
For non-functional features, TOSCA employs policies to describe non-functional behavior and/or quality-of-service (QoS) that an application and its components can declare to expose. Similar to the other entities in the TOSCA standard, a policy has an abstract PolicyType definition and is instantiated by defining a PolicyTemplate. While the PolicyType describes the structure and required parameters of a policy, the PolicyTemplate is used to define a specific policy instance. ServiceTemplates, NodeTemplates, and RelationshipTemplates can then declare their non-functional features by referring the PolicyType and/or PolicyTemplate describing them.

In addition, a ServiceTemplate can also describe the functional and non-functional features it exposes externally. More precisely, the (optional) BoundaryDefinitions element allows to specify the properties, capabilities, requirements and operations of internal components which are externally visible. It also allows to expose management plans as operations and to describe the non-functional properties of the complex application.

TOSCA also prescribes the format to archive application specifications along with the installable and executable files needed to properly instantiate the specified applications. This is because the modeling language described above only allows developers to specify the application topology and its management and to give it in a Definition.tosca document. Such document must be packaged together with the artifacts implementing its components so as to make all such artifacts available to the execution environment.

To this end, the TOSCA specification defines an archive format called CSAR (Cloud Service ARchive) to package application specification together with concrete implementation and deployment artifacts. A CSAR is a (compressed) zip file containing at least the Definitions and TOSCA-Metadata directories. The Definitions directory contains one or more Definitions.tosca documents. These documents contain the TOSCA definitions describing the cloud application. More precisely, exactly one of them must contain the ServiceTemplate defining the structure and behavior of the whole cloud application, while the others can be devoted to supporting definitions (so as to modularize the application specification). Additionally, CSARs can also be devoted to contain TOSCA definitions to be reused in other contexts. For instance, a CSAR might be used to provide a set of NodeTypes (with their corresponding implementations) to be employed as building blocks while specifying new cloud applications. A TOSCA-Metadata directory contains the TOSCA.meta file. Its purpose is to describe metadata about the other files in the CSAR by means of blocks, which in turn consist of a set of name-value pairs. More precisely, the first block of the

TOSCA.meta file provides metadata about the CSAR itself (e.g., version, creator, etc.), while each other block points to a file in the CSAR and describes its metadata.

An application specification is packaged in a CSAR archive with the purpose of deploying it on TOSCA-compliant cloud platforms, which can offer TOSCA containers to process CSAR archives, and thus to deploy and operate the application. TOSCA containers can deploy applications by processing the CSAR archives in two different ways. On one hand, Imperative Processing takes the CSAR and deploys the application according to the workflow defined as a BuildPlan in the corresponding ServiceTemplate. On the other hand, Declarative Processing deploys the application by trying to automatically excerpt a deployment plan from the application's TopologyTemplate.

B. Analysis of Common Services Offered by Cloud Providers

Cloud solutions allow customers to develop cost-effective applications by exploiting the self-adaptation capabilities of cloud services. In order to exploit these properties, the identification of common offered cloud services is essential. Moreover, the scaling policies and the pricing schemes adopted by cloud providers need to be characterized. To this end, in this Annex we offer an analysis of these capabilities (services, prices, policies), with the aim of identifying common characteristics in order to describe them with a suitable representation of a cloud meta-model.

We analyze separately PaaS and IaaS providers, since they offer services at different levels.

Analysis of PaaS capabilities
Among the existing PaaS provider, we have selected four providers (already supported by the Cloud4SOA project), namely RedHat Openshift, CloudBees, Heroku and Cloud Foundry. The last one is an open source PaaS platform that can used to build customized PaaS.

Usually, PaaS providers offer environments for application development and hosting, which automates the provisioning, management and scaling of applications. A PaaS provider generally manages a set of servers, in which end-user applications run. Multiple servers are managed by a broker, which coordinates orchestration and automation. Each server provides a multi-tenant environment, which is shared by end-user application. The isolation of each application is guaranteed since they run on a secure container inside the server. This container is the basic unit of compute resources that a PaaS provider offers.

RedHat Openshift
General capabilities: RedHat OpenShift Origin supports several languages (java, node, perl, php, python, ruby, etc.) and frameworks (rack, wsgi, psgi, node.js, rails, django, jboss, tomcat, etc.). In addition, the most common databases can be used, both relational and non-relational (mysql, postgresql, mongodb, amazon rds).

The secure container inside an OpenShift instance is called Gear. An end-user application normally needs an environment, which provides the languages and services the application needs. These modular entities are called Cartridges in OpenShift. Cartridges can be web frameworks, databases, monitoring services, or connectors to external backends and they are deployed to one or more gears. Web and database cartridges get their own gears, while cartridges for logging and monitoring will have access to all gears.

Scaling policies: OpenShift supports the manual scaling as well as auto-scaling applications. The OpenShift infrastructure monitors incoming web traffic and can automatically add or remove application gears to handle changes in request volume. When a scalable application is created, this is set to scale automatically depending on the load it receives. In particular, the following thresholds can be established: 1) Scale up if the number of concurrent requests exceed 90% of max concurrent requests over one period. 2) Scale down if the number of concurrent requests fall below 49.9% of max concurrent requests over three consecutive periods. These thresholds cannot be currently configured: max concurrent requests is fixed to 10 requests, and a period is 20s.

Pricing schemes: OpenShift provides three monthly plans (namely free, bronze, silver) with different characteristics. The free plan is limited to three small gears and 1GB storage per gear, but it is totally free and it is available worldwide. Silver and bronze plans give the user more flexibility (small, medium and large gears, more storage allowed, etc). The silver plan provides all the features of a bronze plan plus the Red Had premium support. Both silver and bronze plans are available in North America (the U.S. and Canada) and Europe (EU member states, Iceland, Israel, Norway, Switzerland, and Russia).

In Table 3 the main characteristic of each gear together with its price are reported. The additional storage cost is set to $1.00/GB/month.

Gear size*	RAM (MB)	Base Storage (MB)	cost ($ per hour)
small	512	1000	0.02
medium	1000	1000	0.05
large	2000	6000	0.10

Table 3. OpenShift pricing scheme for compute resources

Heroku

General capabilities*:* Heroku offers a PaaS provider to build maintainable and scalable applications without worrying about the infrastructure. The supported languages are Ruby, Node.js, Python, Java, PHP, Clojure and Scala. Moreover, Heroku supports PostgreSQL as database-as-a-service.

The secure container inside a Heroku instance is called Dyno. A Dyno is like a virtualized UNIX container to run user application components. Three different Dyno sizes are available, each size having different memory and CPU characteristics. Languages or frameworks support is offered by means of Buildpacks: Ruby, Python, Java, Clojure, Node.js and Scala are all implemented as buildpacks.

Scaling policies: Heroku does not support automatic scaling. Applications on Heroku can be scaled instantly from the command line or Dashboard. Each application has a set of running dynos that can be scaled up and down. Dynos can also be scaled vertically, providing them with more memory and CPU share.

Pricing scheme: Heroku's pricing is based on resources actually used. Dynos can be scaled and databases can be added. In Table 4 the characteristics and pricing for dynos are reported.

Dyno*	RAM (MB)	Base Storage (MB)	cost ($ per hour)
1x	512	0	0.05
2x	1024	0	0.10
Px	6000	0	0.80

Table 4. Heroku pricing scheme for compute resources

Heroku's PostgreSQL database is available in multiple tiers (Hobby, Standard, Premium and Enterprise) and each tier allow the user to choose among a variety of plans. The different plans with their characteristics and pricing are summarized in Table 5.

Tier/plan	DB size (GB)	Row limit (K)	Max queries /hours	Max connecti ons	Expected Availability (%)	Cache (GB)	cost ($ per month)	SLA
hobby/free	-	10	-	20	99.5	0	0	no
hobby/basic	-	10000	-	20	99.5	0	9	no
standard/0	64	no	-	60	99.9	400	50	no
standard/2	256	no	-	200	99.9	1700	200	no
standard/4	512	no	-	400	99.9	7500	750	no
standard/6	1024	no	-	500	99.9	34000	2000	no
standard/7	1024	no	-	500	99.9	64000	3500	no
premium/0	64	no	-	60	99.95	400	200	no
premium/2	256	no	-	200	99.95	1700	350	no
premium/4	512	no	-	400	99.95	7500	1200	no
premium/6	1024	no	-	500	99.95	34000	3500	no
premium/7	1024	no	-	500	99.95	64000	6000	no
enterprise	*	*	*	*	*	*	*	yes

*Enterprise plans, in addition to features seen in premium databases, come with a Service Level Agreement: if availability is longer than expected that month is free.

Table 5. Heroku pricing scheme for storage resources

Cloud Foundry (Open Source)

General capabilities: Cloud Foundry is an open PaaS and provides a choice of clouds, frameworks and application services. As an open source project, there is a broad community both contributing and supporting Cloud Foundry. The languages supported are java, ruby, node, scala, go, groovy and several frameworks can be chosen: spring, rails, sinatra, play, tomcat.

The secure container is called Warden in CloudFoundry and manages isolated, ephemeral, and resource-controlled environments. These isolated environments can be limited in terms of CPU usage, memory usage, disk usage, and network access. The only currently supported OS is Linux.

Scaling policies: there are two levels at which Cloud Foundry scales, whether automatically or not. The first is at the Cloud Foundry infrastructure level and it is the responsibility of the PaaS operator that implements Cloud Foundry. The operator needs to monitor the load on the various servers and launch additional or terminate idle ones as appropriate.

The second level is at the individual application level and is primarily expressed in how many app execution engines are "running" the application and it is the responsibility of each application's owner.

Analysis of IaaS capabilities

The following IaaS will be considered: Amazon Web Services, Google Cloud Platform, IBM SoftLayer, HP Cloud, Rackspace Cloud, Apache CloudStack, RedHat OpenStack, and Microsoft Azure (optional).

Compute resources

IaaS capabilities are usually delivered in form of Virtual Machines that includes different types of resources. The price of an offering is affected by the type and the amount of a resource. Common characteristics for compute resources are reported in the Table 6.

	AWS	Google	SoftLayer	HP Cloud	RackSpace	CloudStack	OpenStack	Azure
vCPU	yes	yes	yes	yes	no	no	yes	yes
memory	yes	yes	yes	yes	yes	yes	yes	yes
disk	yes	-	yes	yes	yes	yes	yes	yes
image (OS)	yes	yes	yes	yes	yes	yes	yes	yes
id	yes	yes	yes	yes	no	yes	yes	yes
location	yes	yes	yes	yes	yes	yes	yes	yes
price	yes	yes	yes*	yes	yes*	-	-	yes*

*prices are provided through a calculator

Table 6. Common characteristics for compute resources

Storage resources

There are three types of storage: block, file, and object. Each type offers their own advantages and has their own use cases.

Block Storage: is persistent storage organized into unstructured "blocks", each the same length. There is no concept of “file” at this level. Generally, using block storage offers the best performance, but it is low-level. Block storage can be either "locally attached", or it can be "network" attached, in a SAN. An ordinary disk drive, RAID array, or USB storage key are examples of locally attached "block storage".

Block storage devices typically are formatted with a filesystem, such as Linux's ext3 or btrfs, or Microsoft's FAT32 or NTFS.

File Storage: provides access to a file system. This is the most familiar kind of storage. Users of file storage have access to files and can read and write to either the whole file or a part of it. File systems are what operating systems provide on all of our personal computers. In a shared environment, file storage is often seen as a network drive.

Object Storage: does not provide access to raw blocks of data, nor offer file-based access. Object storage provides access to whole objects, or blobs of data and generally does so with an API specific to that system. Unlike file storage, object storage generally does not allow the ability to write to one part of a file. Objects must be updated as a whole unit (Amazon S3, Rackspace Cloud Files). Object storage excels at storing content that can grow without bound. Perfect use cases include backups, archiving, and static web content like images and scripts. One of the main advantages of object storage systems is their ability to reliably store a large amount of data at relatively low cost.

Common characteristics that can be specified for storage resources are reported in table 7 and Table 8.

Block Storage	AWS	Google	SoftLayer	HP Cloud	RackSpace	CloudStack	OpenStack	Azure
location	yes	-	-	no	no	-	-	yes
size	yes	-	-	yes	yes	yes	yes	yes
operations	yes	-	-	yes	no	-	-	yes
snapshots	yes	-	-	yes	no	-	-	no
volume backups	yes	-	-	yes	no	-	-	no

Table 7. Common characteristics for block storage resources

Object Storage	AWS	Google	SoftLayer	HP Cloud	RackSpace	CloudStack	OpenStack	Azure
location	yes	yes	no	no	no	-	-	yes
size	yes	yes	yes	yes	yes	yes	yes	yes

operations	yes	yes	yes	yes	no	-	-	yes
monthly data transfer*	yes	yes	yes	yes	yes	-	-	yes

*some providers refer to monthly data transfer as bandwidth

Table 8. Common characteristics for object storage resources

C. Resource Allocation Strategies in Cloud Computing

In this Annex, we have summarized approaches appeared in the scientific literature that deal with resource allocation challenges, which are strongly related to the ones faced by the SeaClouds optimization activity.

Specifically, the table below (Table 8)[1], summarizes the approaches according to:
(a) The application **domain**, i.e., if a single cloud or a multicloud environment is considered;
(b) The cloud **level** interested by the resource allocation, i.e., if the resource is at IaaS level, or PaaS level or SaaS level;
(c) The considered QoS Requirements in the selection and allocation of resources, e.g., cost, response time, utilization, availability, and so on;
(d) **QoS evaluation** methods;
(e) The adopted **allocation algorithm**;
(f) The **validation** of the proposed approaches.

Paper	**Domain**	**Level**	**QoS Req**	**QoS Eval**	**Allocation algorithm**	**Validation**
[1] 2011	Multicloud	IaaS	max resource usage, max availability, min cost	multiobjective scheduling algorithm (including migration aspect)	genetic approach	-web-based application -Simulation (no real data)
[2]2009	Multicloud	IaaS	min cost (both on-demand and reserved)	Stochastic Optimization model (no migration)	stochastic integer programming 2-stages recourse	academic example, extensive simulation
[3] 2013	singlecloud	IaaS	min energy, min network traffic, max revenue	multiobjective optimization (no migration)	memetic algorithm	real data (private, amazon, rackspace) no real application, several scenarios generated randomly
[4] 2012	Multicloud	IaaS	performance, cost, load balancing,	optimization max capacity with constraints	Integer programming (Ample/Cplex)	experimentation with data from

[1] Goal of the Table 9 is to show the main and recent trends in the field rather than present an exhaustive analysis of the literature, which is out of the scope of this deliverable.

				on hw configuration, number of VM, load balancing (no migration)		amazon EC2-US, EU, ElasticHost
[5] 2011	Multicloud	IaaS	cost (different prices mechanisms)	optimization problem (no migration)	integer programming	experiments with data from amazon
[6] 2011	Multicloud	IaaS	min cost or max capacity	optimization problem extend [4] with VM migration	integer programming	experimentation with data from amazon EC2-US, EU, ElasticHost
[7] 2013	Multicloud	IaaS	resource utilization, max revenue	Markov Decision process	simple-value iteration method	academic example
[8] 2014	Multicloud	IaaS	performance, cost	all possible resource allocation, ordering solutions by cost, exhaustive search	ad hoc algorithm	realistic data from benchmark [11], comparison single/multiple clouds
[9] 2014	single cloud	IaaS	min cost, response time,	optimization problem	queueing+ integer programming+ local search	experimentation with data
[10] 2014	single cloud	IaaS	cost (different prices mechanisms), revenue	adaptation plan algorithm based on heuristic	queueing theory	experimentation with real workload data
[12] 2014	single cloud	IaaS	autoscaling performance	heuristic	rule-based heuristic-based, QN	simulation
[13]2012	single cloud		autoscaling, performance	Rule based algorithm	QN+greedy algorithms	simulation
[14] 2014	single cloud	IaaS	utility functions	topology-based optimization	partial ordering of the viable topologies (theoretical work)	simulation

Table 9. Overview of related work

In the following we shortly describe the main characteristics of these approaches. Concerning the domain, we have mainly reported the approaches related to multi-cloud environments, which are the SeaClouds target [1,2,4,5,6,7,8]. For the sake of completeness, also single cloud solutions [2,9,10,12,13,14] have been studied, but only few of them have been included in the Table to show the main trends in the area. One of the main weakness point of the existing approaches, concerns the level dimension: to the best of our knowledge, only IaaS solutions have been proposed so far for the management of resource allocation in the cloud.

As concerns the quality requirements, most of the approaches focus on the minimization of the overall cost [1,2,4,5,6,7,8,9,10], some of them include availability [1], performance [4,8,9,12,13], resource utilization [1,6,7], load balancing [4] or network traffic [3] and their trade-off. To be effective, QoS evaluation approaches should rely on models representing the systems in an accurate/realistic way. Several approaches reported in Table 2 rely on the definition of simple aggregate QoS functions (like sum, product, max, and average) that can be easily defined and managed. However, due to dependencies between different applications or between applications and resources these aggregation functions could lead to quality estimation that represent optimistic (or pessimistic) bounds rather than a realistic estimation.

Most of the adopted solutions rely on the definition of a (multi-objective) optimization problem [1,2,3,4,5,6,9,14], other approaches adopt techniques based on markov Decision processes [7], heuristic definition [10,12], or ad-hoc solution [8,13]. Accordingly, the allocation algorithms use integer programming [2,4,5,6,9], genetic algorithms [1,3], ad-hoc algorithms [8,12,13] and queueing theory [9,10,12,13].

An investigation of the validation strategies, shows that several approaches perform experiments based on generated examples [1,2,7] or apply a case study based validation [3,4,5,6,8,9,10,12,13,14]. Some of them use real data [3,4,5,6,8,9,10] while others simply rely on simulation [12,13,14].

References

[1] Marc E. Frincu and Ciprian Craciun. 2011. Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing* (UCC '11). IEEE Computer Society, Washington, DC, USA, pp. 267-274. DOI=10.1109/UCC.2011.43 http://dx.doi.org/10.1109/UCC.2011.43

[2] Sivadon Chaisiri, Bu-Sung Lee, Dusit Niyato: Optimal virtual machine placement across multiple cloud providers. APSCC 2009: 103-110.

[3] Fabio López Pires and Benjamín Barán. 2013. Multi-objective Virtual Machine Placement with Service Level Agreement: A Memetic Algorithm Approach. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing* (UCC '13). IEEE Computer Society, Washington, DC, USA, pp. 203-210. DOI=10.1109/UCC.2013.44

[4] Johan Tordsson, Rubén S. Montero, Rafael Moreno-Vozmediano, and Ignacio M. Llorente. 2012. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Gener. Comput. Syst.* 28, 2 (February 2012), pp. 358-367.

[5] J.L. Lucas-Simarro, R. Moreno-Vozmediano, R.S. Montero, I.M. Llorente. Dynamic Placement of Virtual Machines for Cost Optimization in Multi-Cloud Environments. In *Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011)*, pp. 1-7, July 2011.

[6] Wubin Li; Tordsson, J.; Elmroth, E., "Modeling for Dynamic Cloud Scheduling Via Migration of Virtual Machines," In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pp.163-171, Nov. 29-Dec. 1 2011.

[7] Oddi, G.; Panfili, M.; Pietrabissa, A; Zuccaro, L.; Suraci, V., "A Resource Allocation Algorithm of Multi-cloud Resources Based on Markov Decision Process," *2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom),* pp. 130-135, 2-5 Dec. 2013.

[8] Simon S. Woo, Jelena Mirkovic, Optimal application allocation on multiple public clouds, Computer Networks, Volume 68, 5 August 2014, pp. 138-148, ISSN 1389-1286, http://dx.doi.org/10.1016/j.comnet.2013.12.001.

[9] D. Ardagna, G. Gibilisco,M. Ciavotta, A.Lavrentev “A Multi-Model optimization Framework for the Model Driven Design of Cloud Applications”, To appear, SSBSE 2014.

[10] D.Perez-Palacin, R. Mirandola, R. Calinescu “Synthesis of Adaptation Plans for Cloud Infrastructure with Hybrid Cost Models” to appear Euromicro SEAA 2014.

[11] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (IMC '10). ACM, New York, NY, USA, 1-14. DOI=10.1145/1879141.1879143

[12] Tighe, Michael; Bauer, Michael, "Integrating cloud application autoscaling with dynamic VM allocation," *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1-9, 5-9 May 2014, doi: 10.1109/NOMS.2014.6838239

[13] Moreno Marzolla, Raffaela Mirandola: A Framework for QoS-aware Execution of Workflows over the Cloud. CLOSER 2012: 216-221

[14] Vasilios Andrikopoulos, Santiago Gomez Saez, Frank Leymann, Johannes Wettinger: Optimal Distribution of Applications in the Cloud. CAiSE 2014: 75-90

[15] https://brooklyn.incubator.apache.org/

[16] The ModaCloud Project. Public deliverable D4.2.1
http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D4.2.1_MODACloudMLDevelopmentInitialVersion.pdf

[17] http://aws.amazon.com/cloudformation/aws-cloudformation-templates/

[18] https://wiki.openstack.org/wiki/Heat

[19] OASIS: TOSCA 1.0 (Topology and Orchestration Specification for Cloud Applications),
Version 1.0 (2013), http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf

[20] Antonio Brogi, Jacopo Soldani, PengWei Wang: TOSCA in a Nutshell: Promises and Perspectives. In *Proceedings of the 3nd European Conference on Service-Oriented and Cloud Computing (ESOCC 2014)*, pp. 171-186, Manchester, UK, September 2-4, 2014.

[21] Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery - a modeling tool for tosca-based cloud applications. In: Service-Oriented Computing. Springer (2013), pp. 700-704.

[22] OASIS: TOSCA Simple Profile 1.0
http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csd01/TOSCA-Simple-Profile-YAML-v1.0-csd01.html#_Toc385247586

[23] https://www.ogf.org/documents/GFD.107.pdf

[24] http://en.wikipedia.org/wiki/Service_level_objective