SeaClouds Project
D3.2 Discovery, design and orchestration functionalities

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, WP SeaClouds Design-Time modelling and orchestration
Deliverable code	D3.2
Deliverable Title	Discovery, design and orchestration functionalities
Nature	Report
Dissemination Level	Public
Due Date:	M18
Submission Date:	Date of actual submission – 02nd April 2015
Version:	1.0
Status	Draft
Author(s):	Marc Oriol (UPI), Simone Zenzaro (UPI), Leonardo Bartoloni (UPI), Mattia Buccarella (UPI), Antonio Brogi (UPI), Diego Pérez (POLIMI), Miguel Barriuso (NURO), Roi Sucasas (ATOS), José Carrasco (UMA), Adrián Nieto (UMA), Miguel Barrientos (UMA), Javi 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)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	09/03/15	First ToC	Marc Oriol (UPI), Antonio Brogi (UPI)
0.2	18/03/15	First contributions	Leonardo Bartoloni (UPI) Simone Zenzaro (UPI), Mattia Buccarella (UPI), Marc Oriol (UPI), Miguel Barriuso(NURO), Roi Sucasas (ATOS), José Carrasco (UMA), Adrián Nieto (UMA), Miguel Barrientos (UMA), Diego Pérez (POLIMI)
0.3	24/03/15	Second contributions	Leonardo Bartoloni (UPI), Simone Zenzaro (UPI), Mattia Buccarella (UPI), Marc Oriol (UPI), José Carrasco (UMA), Javi Cubo (UMA), Adrián Nieto (UMA), Miguel Barrientos (UMA), Diego Pérez (POLIMI)
0.4	26/03/15	Version to be revised	Marc Oriol (UPI), Simone Zenzaro (UPI), Leonardo Bartoloni (UPI), Mattia Buccarella (UPI), Javier Cubo (UMA)
0.5	01/04/15	Revision	Francesco d'Andria (ATOS), Andrea Turli (Cloudsoft)
1.0	02/04/15	Stable version after reviews	Marc Oriol (UPI)

Table of Contents

List of figures

List of tables

Executive Summary

This document describes the specification of the application to be deployed by SeaClouds, and how it evolves through different models, until reaching a multi-cloud solution that orchestrates the different modules of the application into concrete PaaS and IaaS services.

This deliverable is structured in two parts. In the first part, we address the specification of the different models used along the process. In the second part, we describe the architecture and design of the different components of SeaClouds that process the above models.

In the first part, we first refine and consolidate the Application Model Lifecycle presented in the previous deliverable D3.1 [1], defining the different stages of the Application Models generated and consumed in the SeaClouds lifecycle. Secondly, we elicit and consolidate the required pieces of information that should be present in these models, identifying the characteristics and dependencies of the topology, as well as the properties and QoS to be hold on the application and its components. Then, we describe how these elements are represented on the different models of the lifecycle: first by means of a graphical human-readable model, and ultimately by a formal machine-readable TOSCA YAML specification. Finally, we validate this TOSCA YAML specifications by instantiating the case studies defined in SeaClouds.

In the second part, we describe the architecture and design of the different components of SeaClouds that generate and/or use the aforementioned models. Particularly, we specify the Parser of TOSCA YAML, and the architecture and design of the Discoverer and the Planner.

1. Introduction

1.1. Glossary of Acronyms

Acronym	Definition
AAM	Abstract Application Model
ADP	Abstract Deployment Plan
API	Application Programming Interface
APP	Application
DAM	Deployable Application Model
DB	Database
DBMS	Database Management System
FAT	File Allocation Table
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure-as-a-Service
NTFS	New Technology File System
OS	Operating System
PaaS	Platform-as-a-Service
PAYG	Pay-As-You-Go
PHP	Hypertext Preprocessor
QoB	Quality of Business
QoS	Quality of Service
SLA	Service Level Agreement
SLO	Service Level Objective
SSL	Secure Sockets Layer
TOSCA	Topology and Orchestration Specification for Cloud Applications
UML	Unified Modelling Language
URL	Uniform Resource Locator
VM	Virtual Machine
WP	Work Package
XML	Extensible Markup Language
YAML	YAML Ain't a Markup Language

Table 1. 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 model is proposed to represent all the related information.

2.1. Application Model Lifecycle

The main purpose of the application model is to keep a track of the constituents of a multi-cloud application during its lifecycle through the SeaClouds platform. Figure 1 shows the application model lifecycle of the platform.

The initial input for SeaClouds is, on the one hand, an abstract application, which is instantiated by the user and described through an Abstract Application Model (AAM). This model contains the definition of all the modules of the application, their relationships, and the user's requirements. These requirements are both technical and QoS requirements that may apply to the whole application and/or to the constituent modules. On the other hand, the Discoverer provides the Clouds Offerings Model. This model includes the list of available cloud offerings (for both PaaS and IaaS) from service providers, with information regarding their technical characteristics and QoS information.

These two models are processed by the Matchmaker and the Optimizer, which generate as output an Abstract Deployment Plan (ADP). The ADP is an intermediate result where all the modules of the application are instantiated by concrete services that provide the functionality required, meeting the technical and QoS requirements.

Then, the DAM Generator augments the information specified in the ADP and generates a Deployable Application Model (DAM). The DAM contains the information needed by the SeaClouds Deployer to deploy, configure and execute the application (e.g. with all required information about credentials).

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. If there is a violation on the QoS and replanning is required, the Planner is triggered to generate a new Concrete Application Model, which will follow the same Application Model lifecycle.

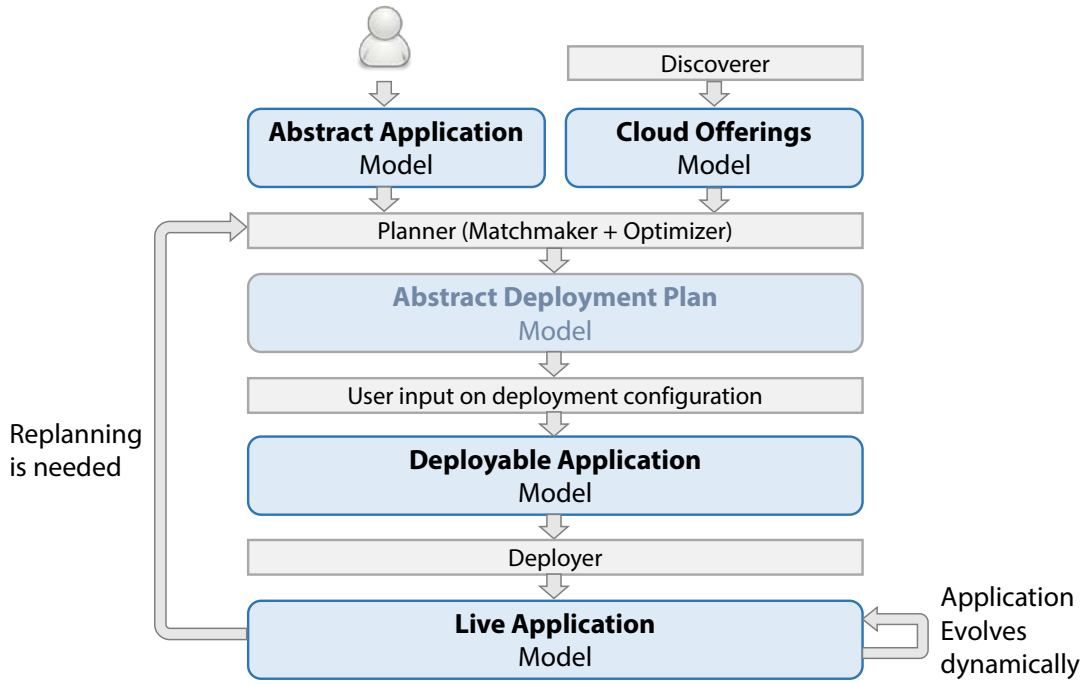

Figure 1. Application Model lifecycle

2.2. Topology and properties required

2.2.1. Topology

The user is required to specify the topology information about the application to be deployed. Topology information can be categorized in two layers: The deployment layer and the logic layer.

Deployment layer:

This layer specifies the different components and artefacts of the application to be deployed in a multi-cloud system. It describes the different software components of the application, and for each component, the properties of IaaS or PaaS that are required. These properties include both technical and QoS requirements (see Section 2.2.2 for details). This is the minimal required information that the Planner needs (particularly, the matchmaker and optimizer components) in order to select a suitable set of cloud offerings and provide a set of solutions where to deploy each of the application's components in order to meet the specified requirements.

Logic layer:

The logic layer allows specifying a more detailed structure of the functionality and defines the relationships and interaction between the different components of the application. This information allows the SeaClouds platform to evaluate global properties, and decide the deployment strategy keeping those global properties into account (e.g. deploying two components that interact heavily in the same cloud infrastructure to reduce network latencies).

The interaction between components are instantiated by defining the available operations of the components and its relationships. In its simplest form, each component has only one "use" operation, and the relationship between these "use" operations reflect functional dependencies between components. In its more complete form, it is possible that one component exposes operations with different performance or QoS requirements, for which more than one operation should be defined.

The logic layer may include also some benchmarking information to predict the behaviour of the component for optimization purposes.

As an example, we consider a simple application composed by a web interface and a database (see Figure 2). The application has two different usage patterns, one being normal user operation and the other being administrative usage. Normal user operation usually involves two or three queries to the database, while administrative usage may trigger a database backup, which is a slow operation. The QoS requirement for normal usage would thus be different from the administrative usage, for which it may not be possible to guarantee the same timings. In this case it would be desirable to define two operations "NormalUsage" and "AdministrativeUsage" on the web interface with different QoS requirements, and similarly two operations "Query" and "Dump" on the

database with different benchmark values, and define the two relationship between these operations: the first would be stating that the "NormalUsage" operation will invoke an average of 2.5 "Query" operations, while "AdministrativeUsage" operation will invoke the "Dump" operation, finally the 4 operations would be annotated with benchmark information and when needed with QoS requirements.

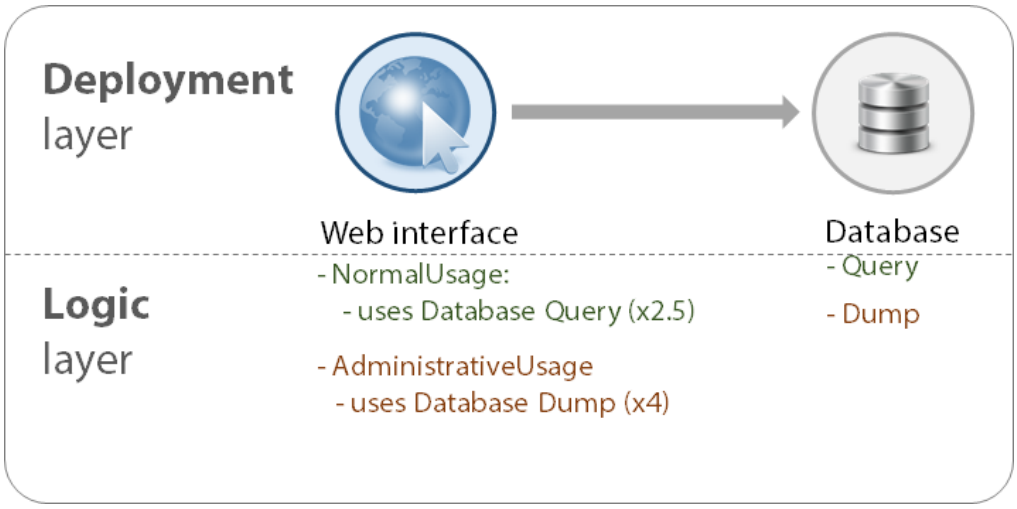

Figure 2. Example of Topology Model

2.2.2. IaaS & PaaS properties

In this section we describe the important properties for both IaaS and PaaS that the user may require for each of the software components of the application to be deployed.

The main idea of IaaS is to provide a virtualized computing resource. IaaS providers enable the user to deploy virtual machines into the service provider's premises, so the user can benefit of the advantages of the service providers' capabilities (e.g. scaling operations in order to adjust resources on demand). In the context of SeaClouds, the interest is to identify the capabilities and properties that define these virtual resources.

Unlike the IaaS, the PaaS allows a cloud user to deploy specific resources or applications that run into the provider's platforms, supporting also all the basic advantages of the cloud computing field (scalability, elasticity, etc…). PaaS providers offer support to a specific number of services and systems, depending on the capabilities of the offered platform itself. As an example, without including any additional software within the application being deployed, the application relies on pre-installed resources such as databases, web servers, etc., with everything running on a specific operating system. This pre-installed software stack is what makes the platform. In the context of SeaClouds, the interest is to identify the capabilities and properties of these platforms.

The list of elicited properties comply with the data requirements as defined in D2.1 [2], and include both technical properties and QoS.

Technical properties:

Basic properties (IaaS and PaaS)
Each IaaS and PaaS is typically recognized by its service name. This property is simply a string that often identifies the provider as well as the cloud (e.g., “Google App Engine”, “Microsoft Windows Azure”, “Red Hat OpenShift”, etc.).
Even if the provider’s name is often included within the service name string, another string, dedicated to the provider’s name only is potentially useful (e.g., some user could lead the matchmaking process by favourite providers).

Computational Features (IaaS)
The two most interesting features that describe a virtual resource from the point of view of its computational power are the *number of cpu*s and the *memory size*.
The number of cpus represents the cardinality of computational units. This number is related to the computational speed. When an application needs to perform cpu intensive tasks this is one of the important parameters to tune.
The memory size represents the memory available for the applications running on the virtual resource.

Networking (IaaS)
Networking properties define features about bandwidth costs, load-balancing attributes, and the number of available IPs

Storage (IaaS)
The storage properties define *disk capacity* and the type of *file system* needed by the cloud application.

Location (IaaS and PaaS)
Location property represents the location of the offering. It can be defined with different levels of detail. It is possible to specify a wide area like “eu_west” or more precise location “ie_Dublin”.

Scaling capabilities (IaaS and PaaS)
Deploying an application to the cloud brings many advantages, one of them is self-adapting the platform to the real needs of the cloud application.

If the application needs more resources, cloud platform with scalable features can adjust the amount of such resources also automatically. There are two kinds of scalability operations: vertical and horizontal scaling.
Vertical scaling is useful when the application needs to increase the amount of resources in order to fulfil its performance standard. It is also possible to reduce the amount of resources when not needed with a decreased cost in the cloud for such application.
Horizontal scaling allows to instantiate the same machine multiple times.

IaaS-like properties within PaaS

It is likely that, in spite of the structural differences between a PaaS and a IaaS, some properties are common to both types of clouds. In particular, it is possible to find specifications of the underlying architecture even relating to a PaaS cloud (e.g., core frequency, memory and storage). For this reason, the same set of properties that is used to characterize a IaaS is also included in the generic description of a PaaS.

Groups of properties

There are certain properties that can, in principle, vary in number. When relating to such properties, one shall think of them as grouped in their suitable group, rather than reason about each independently of the others. A couple of examples can be mentioned: the database support and the programming language support. These two supports are groups of properties, whose size depend on the number of databases and programming languages respectively supported. In the follow-up of the document, a more detailed table, that describes the properties more in detail, will be organized grouping the properties, if needed.

Database Support (PaaS)

The set of supported DBMS (e.g., mysql, postgresql, oracle, and other common DBMS) is hardly missing in the offerings of all the PaaSes. Each supported database is represented by a boolean attribute whose name is the database name itself, followed by "_support".

Programming languages support (PaaS)

The properties about the supported programming languages follow the same logic adopted by the database support. For each supported programming language, one corresponding boolean value is set equal to true. Even in the programming language case, properties shall be considered as a group of properties.

Other properties (PaaS)

One cloud could excel in other less common aspects and specific services with respect to other clouds. This could make such clouds the preferred choice for the matchmaker during the allocation of the applications onto the available resources. These resources involve SSL features, auto-scaling, automatic recovery from failures, information about used virtual machines, domains, IP, etc. Each of this property is represented by one corresponding boolean attribute. The full list can be found in the follow-up of the document.

QoS properties:

Availability (IaaS and PaaS)
The service availability is one of the most important characteristics that contribute to the evaluation of the quality of service. It is, in general, a percentage value that shows for how much time the service could be successfully reached and utilized. It may happen that such information is provided as a list of percentages, each described by a string containing notes about the value. As an example, a broker may provide availabilities for instants of one hour each or for wider periods like past weeks, past months, etc. Since we are viewing through the perspective of properties, the *availability* is the least value the contract of a certain provider guarantees to the cloud users.

Bandwidth pricing (IaaS and PaaS)
Another important resource, provided by the clouds, is the connection bandwidth. Consequently, a pricing plan for the provider becomes an essential property for the quality of service of the cloud itself and for its performance over networking applications.
The *bandwidth* is typically assigned a cost per some quantity of bytes exchanged (e.g., dollar/KB, dollar/MB, etc...) and it is often charged independently of the direction of the data (i.e., incoming data and outgoing data).

Service pricing (IaaS and PaaS)
The pricing of the service depends on the business model the cloud itself decided to apply to enter the market. In general, one cloud could provide, at the disposal of the cloud consumer, different pricing plans, each based on the PAYG paradigm. In order for attributes to suitably model the service pricing, a complex type is defined.
The *platform_pricing* node is a list of complex types defined by means of the following properties:

- *name*, a string identifying the name of the pricing under consideration
- *price_per_hour*, floating-point value, expressed in dollars
- *cpu*, a scalar value expressing the clock frequency of the processor of the cloud in MHz
- *memory*, a scalar value expressing the amount of available RAM memory in MB
- *local_storage*, a scalar value expressing the amount of available disk quota in GB

IaaS and PaaS concepts overview

In the following tables, we summarize the list of properties described.

Table 2. Technical properties

Property name	Description	Applies to
num_cpus	Number of cpus	IaaS
mem_size	Memory size	IaaS
disk_size	Disk Size	IaaS
load_balancing	Load balancing support	IaaS
scaling_horizontal	Horizontal scaling support	IaaS
scaling_vertical	Vertical scaling support	IaaS
storage_type	The type of filesystem for the storage (e.g. SATA)	IaaS
storage_file_system	The file system type for the storage. (e.g. NTFS)	IaaS
service_name	Name of the offering	PaaS, IaaS
provider_name	Name of the provider	PaaS, IaaS
location	Geographic information about the location of the servers	PaaS, IaaS
api_restricted	Boolean, telling whether the API at the developer's disposal are limited to or customized for some specific feature (e.g., networking)	PaaS
auto_failover	Boolean, true when the cloud has the capability of automatically recover after a failure	PaaS
auto_scaling	Boolean, true when the cloud has the capability of automatically scale, depending on the outside demand	PaaS
process_based	Boolean, true if the functioning of the cloud can be thought as a pool of workers doing the assigned tasks.	PaaS

self_hostable	Boolean, true when the provider offers a self hostable option that simulates/duplicates the platform features and functionality	PaaS
vm_based	Boolean, true when the deploying application is going to be hosted and executed upon a virtual machine.	PaaS

Table 3. Database support

Database support		
MySQL_support	Boolean, true when it is possible to attach to a MySQL database.	PaaS
PostgreSQL_support	Boolean, true when it is possible to attach to a MySQL database.	PaaS
Oracle_support	Boolean, true when it is possible to attach to an ORACLE database.	PaaS
SQLite_support	Boolean, true when it is possible to attach to an SQLite database.	PaaS
MongoDB_support	Boolean, true when it is possible to attach to a Mongo database.	PaaS

Table 4. Programming languages support

Programming languages support		
DotNet_support	Boolean, true when it is possible to develop the deploying application in any of the .Net languages (C#, J#, VB.Net, F#, etc…)	PaaS
Python_support	Boolean, true when it is possible to develop the deploying application in Python.	PaaS
Java_support	Boolean, true when it is possible to develop the deploying application in Java.	PaaS
Go_support	Boolean, true when it is possible to develop the deploying application in Go.	PaaS

Table 5. Quality of Service

Quality of Service		
availability	Percentage; how much time one cloud has been reachable for	PaaS, IaaS
bandwidth_pricing	Cost for exchanged data over Internet Connection	PaaS, IaaS
service_pricing	Cost of the offering	PaaS, IaaS

2.3. Graphical TOSCA Model

This section provides the technical information for the following models: Abstract Application Model, Cloud Offerings, Abstract Deployment Plan, Deployable Application Model and Live Model. All these Models are implemented using the TOSCA YAML syntax [6], which provides a more compact and maintainable syntax than his TOSCA XML ancestor, used in D3.1[1].

2.3.1. AAM TOSCA Model

The AAM TOSCA Model (see Figure 3) is the model instantiated by the user that describes the topology of the application and its requirements (for both the application and each of its constituent modules). The structure of the AAM TOSCA Model is composed by the following elements: Modules, Relationships and Application operation.

- Modules are the basic building blocks of the application. A module can be either a IaaS (of type `seaclouds.Types.Compute`) or a PaaS (of type `seaclouds.Types.Platform`). Each module defines its functional requirements as `Properties` (e.g. num_cpus).

- Relationships are connections between modules. Each relationship has a name and a set of properties that provides useful information for the optimization process (e.g. the average usage count). Relationships are identified by `seaclouds.Types.Relationship` types (e.g. `Uses`).

- Application operation is a description of a functionality of a module and is identified by the TOSCA type `seaclouds.Types.Logic`. In the AAM, an operation is enriched with QoS information. There are two categories of QoS information: QoS_info and QoS_requirements.

`QoS_info` defines information about benchmarked properties that are useful for optimization purposes.

`QoS_requirements` defines the required QoS for that operation (e.g. the execution time must be below 3 seconds).

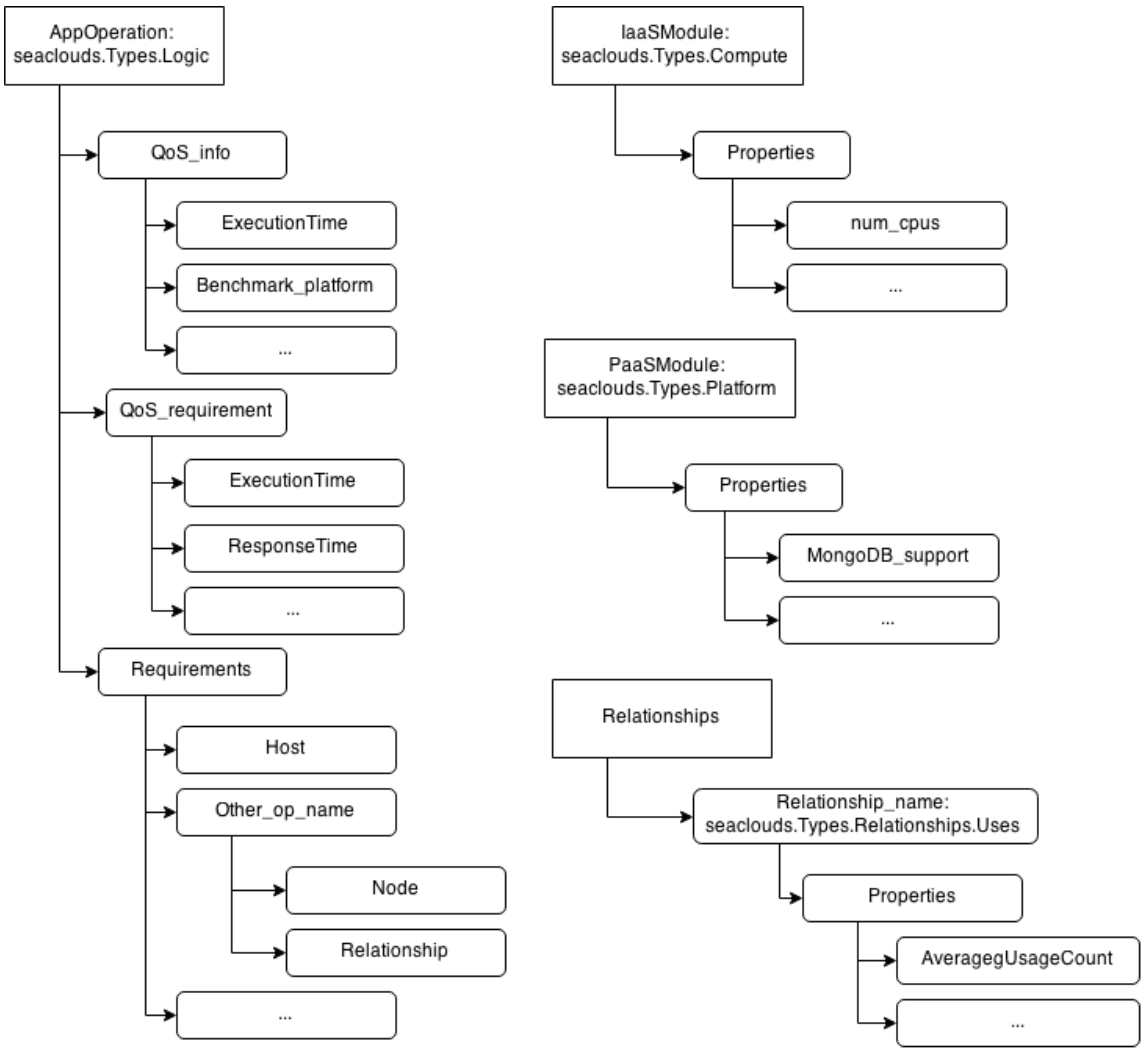

Figure 3. Graphical TOSCA Model for AAM

2.3.2. Cloud Offerings TOSCA Model

Cloud offerings identifies two main cloud offering category types: Compute and Platform (see Figure 4).

The TOSCA type `seaclouds.nodes.Compute` extends the TOSCA type `tosca.nodes.Compute` and is used to describe IaaS offerings while the TOSCA type `seaclouds.nodes.Platform` is used to describe PaaS offerings.

Exploiting the TOSCA type system, each cloud offering is considered an extension of these two basic types. These node types already describe part of the interesting information of the cloud offerings, for example the IaaS offerings already have defined properties about virtual machines like the number of cpu cores or the memory and disk size but also information about scaling support and location. Similarly happens for the PaaS offering model.

TOSCA type system exploitation allows to have extensible property set when a new offering is added to the repository. This approach helps to address the various features of PaaS offering.

For example if the support for a particular language is not present in the `seaclouds.nodes.Platform` definition, extending this type into a PaaS offering allows to add such language support to the set of features naturally.

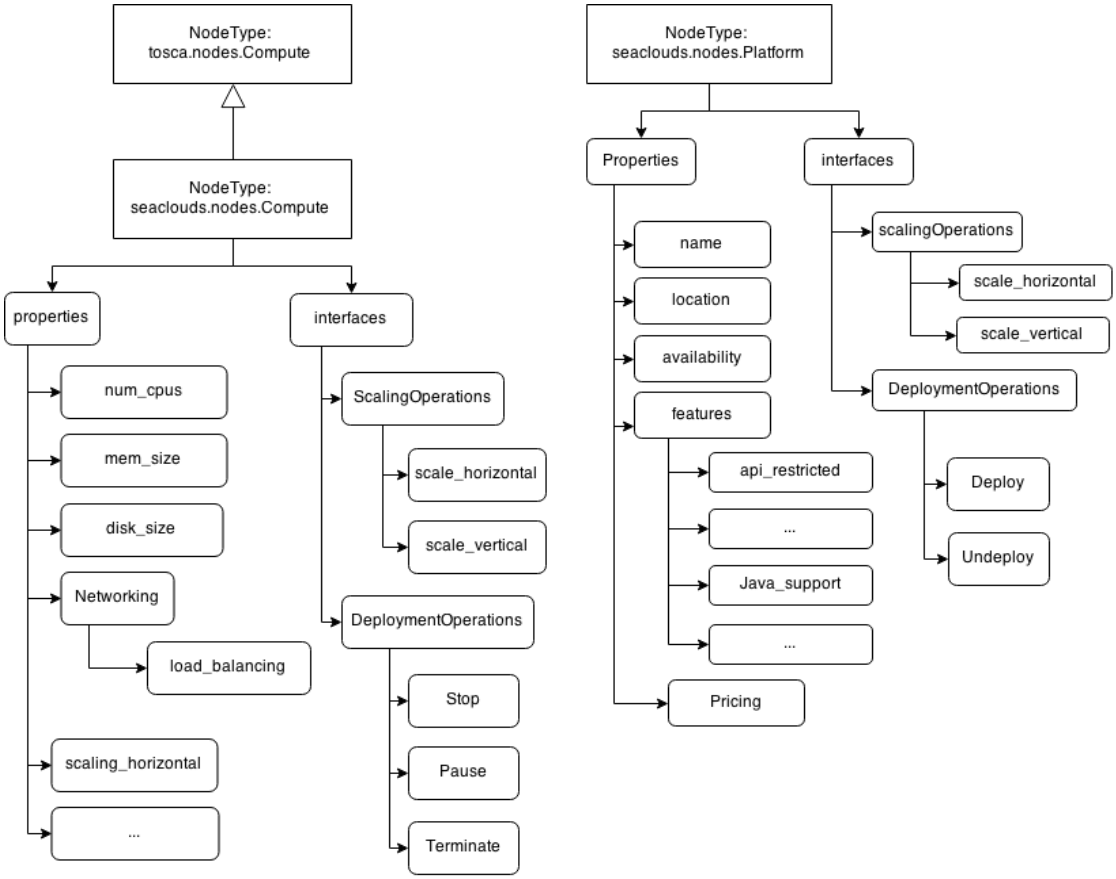

Figure 4. Graphical TOSCA Model for the Cloud Offerings

2.3.3. ADP Model

The ADP TOSCA Model is the internal model, previous to the DAM, that describes the deployment plan. The ADP TOSCA Model describes the topology of the application with concrete services on each module from IaaS and PaaS providers. That is, each of the modules of the application in the AAM are mapped to a concrete service that fulfill the required technical properties and QoS defined for that module. Furthermore, the application (i.e. composition of modules) satisfies also the requirements defined at the application level.

As shown in Figure 5, the structure of the ADP TOSCA Model is similar to the elements presented in the AAM, with the particularity that the modules in AAM are replaced by concrete Cloud Offerings as defined in the Cloud Offerings Model (i.e. `seaclouds.nodes.Platform` for PaaS services and `seaclouds.nodes.Compute` for IaaS).

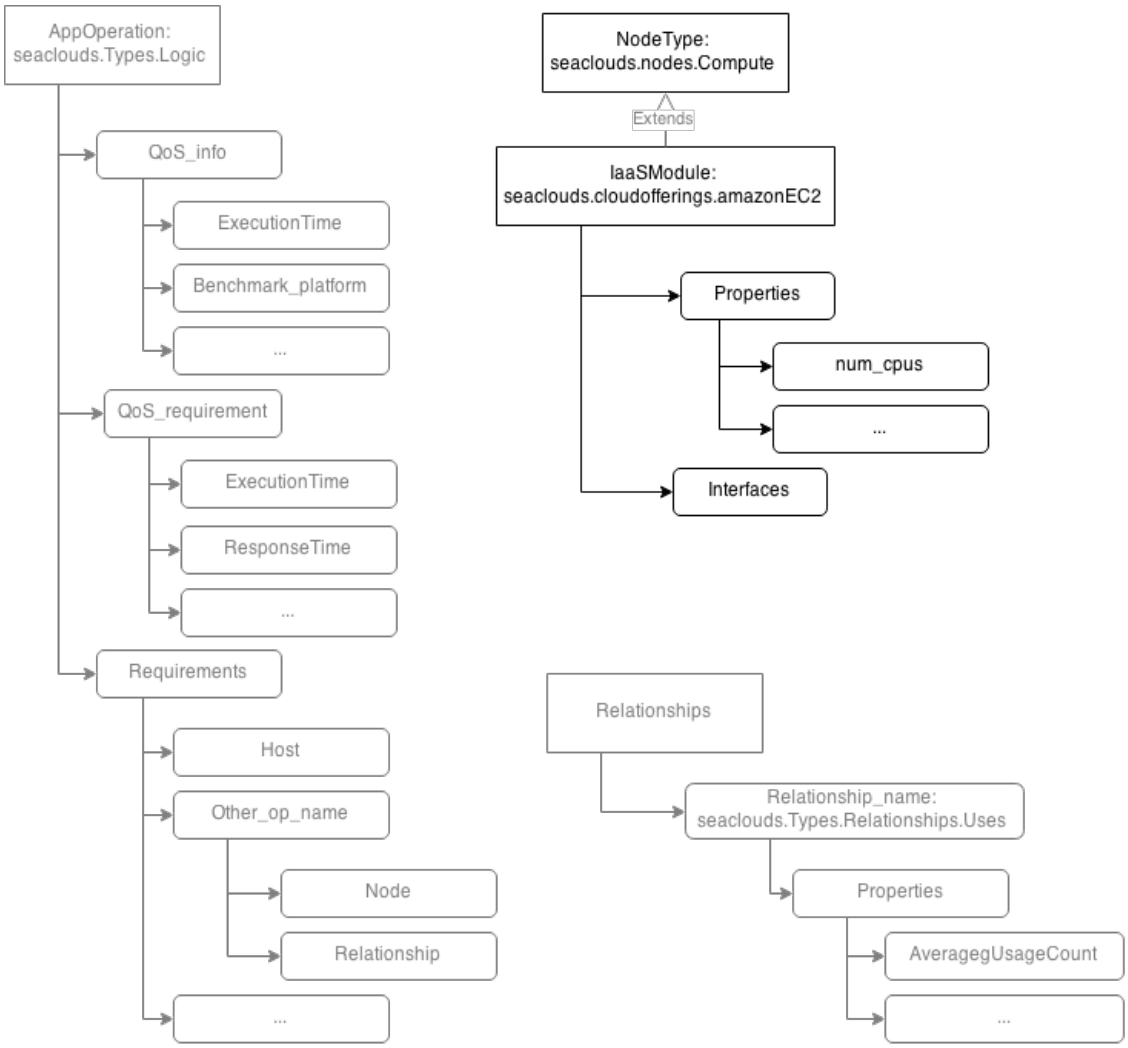

Figure 5. Graphical TOSCA Model for the ADP

2.3.4. DAM and Live Model

The DAM Model augments the information stored in the ADP with the data required to perform the deployment of the different modules of the application on the selected clouds (e.g. credential, reconfiguration policies, etc.). Specifically, the DAM is generated by the DAM generator by using the ADP internal model and interacting with the user to retrieve the required information. Overall, DAM describes the complete application structure and the deployment plan to distribute the different elements over the target locations. The structure of the DAM Model includes the information of the ADP and the following elements:

- the configuration parameters of the application modules
- the credentials required by the selected clouds for hosting application modules
- the reconfiguration policies.

The live Model contains the current status of the application, as specified in DAM, including the following additional runtime information:

- structure of application modules (including additional modules that could be created at runtime, due to resizing dynamic entities, such as clusters)
- current policy status of each application module
- runtime information regarding QoS
- available effectors (start, restart, stop, etc.), executed when the policy thresholds/target are violated, for each application module.

The DAM and Live Models are not used to generate a deploymen plan, but to execute a deployment and maintain runtime information of the application respectively. For such a reason, their formal definition is out of scope of this deliverable, and they will be developed and presented in their respective deliverables taking also into account the requirements and considerations as defined in D2.1 [2].

2.4. TOSCA YAML schema

In this section we describe the formalization of the TOSCA YAML schemas [6] for the models that have been defined in section 2.3, which include the AAM, Cloud Offerings and ADP. As described previously, the DAM and Live Models are out of the scope of this deliverable.

2.4.1. AAM

Here we describe how to formalize the AAM in SeaClouds TOSCA YAML. As described previously, the AAM is structured on two layers: The Deployment layer and the Logic Layer.

Deployment layer

The Deployment layer is composed of several modules. A module is a minimal deployable entity, and it is defined by a `node_template` in the AAM. The type of the node template specifying a module is usually `seaclouds.nodes.Compute` or `seaclouds.nodes.Platform`, but can also be a more specific type if the user wants to specify a particular instance (for example `seaclouds.cloudofferings.Amazon.EC2`). Figure 6 depicts the TOSCA YAML Schema for this layer.

The properties of these modules define technical requirements on the cloud offering which should be selected to deploy the module, such as available memory for IaaS or the services provided for PaaS. These properties are further explained and their scheme is defined in the next section.

Modules (Deployment Layer)

node_templates:
  <module_name>:
  type: <module_type>
  properties:
  <property_name> : <property_value>
  …

Figure 6. TOSCA YAML Schema of the Modules (Deployment Layer)

Logic layer

The logic layer is composed of the functionalities of the modules and the dependencies between them. The functionalities are expressed as node templates with the `seaclouds.nodes.Logic` type. The `Logic` nodes must have a `host` named requirement which refers to the module implementing the functionality being described. The dependency from functionalities provided by other modules in the application can be expressed using requirement linking to other logic nodes with a relationship of the type `seaclouds.relationships.Uses`. The `Uses` relationship has a property `average_usage_count` which defines how many times the target functionality need to be used in average to provide the functionality being described.

In `Logic` nodes the user can also define two properties: `qos_requirements` and `qos_info`, that respectively specify the QoS requirements for the given functionality and the benchmark information needed to compute them (e.g. required cpu_time and benchmark of the used platform). Figure 7 depicts the TOSCA YAML Schema for this layer.

Functionalities and Dependencies (Logic Layer)

```
relationship_types:
  seaclouds.relationships.Uses:
    valid_targets: [seaclouds.nodes.Logic]
    properties:
      average_usage_count:
        type: float
        default_value: 1.0

node_types:
  seaclouds.nodes.Logic:
    derived_from: tosca.nodes.Root
    properties:
      qos_info:
        type: seaclouds.types.QosInfo
      qos_requirements:
        type: seaclouds.types.QosRequirement
    requirements:
      - host:
          node: tosca.nodes.Root
          relationship: tosca.relationships.HostedOn

relationship_templates:
  <relationship_name>:
    type: seaclouds.relationships.Uses
    properties:
      average_usage_count: 2

node_templates:
  <functionality_name>:
    type: seaclouds.nodes.Logic
    properties:
      qos_requirements:
        <property_name>: <property_value>
        …
      qos_info:
        <benchmark_platform> : <property_value>
        <property_name>: <property_value>
        …
    requirements:
      - host: <host_module>
      - <dependency_name>:
          node: <other_functionality_name>
          relationship: <relationship_name>
      - …
```

Figure 7. TOSCA YAML Schema of Functionalities and Dependencies (Logic Layer)

2.4.2. Cloud Offerings

Here we describe how to define the Cloud Offerings in SeaClouds TOSCA YAML. The different cloud offerings are defined as `node_types` and are structured in the form of `seaclouds.nodes.Compute` for IaaS and `seaclouds.nodes.Platform` for PaaS (see Figure 8). Each of the cloud offerings contains the technical and QoS properties, which can be simple elements, or complex structs defined in the property types (see Figure 9). We also provide an example of cloud offering for PaaS (see Figure 10) and IaaS (see Figure 11).

Node Types

```
tosca_definitions_version: tosca_simple_yaml_1_0

node_types:
  seaclouds.nodes.Compute:
    derived_from: tosca.nodes.Compute
    properties:
      scaling_horizontal:
        type: string
        constraints:
          valid_values: [no,manual,auto]
      scaling_vertical:
        type: string
        constraints:
          valid_values: [no,manual,auto]
      storage_type:
        type: string
        valid_values: [sata, sas, ssd, scsi, …]
      storage_file_system:
        type:string
        valid_values: [ntfs, fat32, …]
      storage_size:
        type: scalar-unit
      location:
        type: seaclouds.types.Location
      networking:
        type: seaclouds.types.NodeNetworkInfo

  seaclouds.nodes.Platform:
    properties:
      service_name:
        type: string
      location:
        type: seaclouds.types.Location
      service_availability:
        type: list
          entry_schema:
```

```
            properties:
              name:
                type: string
              percentage:
                type: float
                constraints:
                  - in_range: [0, 100]
             notes:
                type: string
              bandwidth_pricing:
               type: list
               entry_schema:
                 properties:
                    outbound_pricing:
                      type: string
                    description:
                      type: string
                    inbound_pricing:
                      type: string
service_features:
  properties:
    api_restricted:
      type: boolean
    auto_failover:
      type: boolean
    auto_scaling:
      type: boolean
    process_based:
      type: boolean
    self_hostable:
      type: boolean
    vm_based:
      type: boolean
    MySQL_support:
      type: boolean
    Go_support:
      type: boolean
    Java_support:
      type: boolean
    Python_support:
      type: boolean
    MongoDB_support:
      type: boolean
    platform_pricing:
      type: list
      entry_schema:
        description: TOSCA YAML for payg paas model
          properties:
            name:
              type: string
```

```
                price_per_hour:
                  type: float
                  constraints:
                    - greater_or_equal: 0
                cpu:
                  type: string
                memory:
                  type: integer
                  constraints:
                    - greater_or_equal: 0
                local_storage:
                  type: integer
                  constraints:
                    - greater_or_equal: 0
```

Figure 8. TOSCA YAML Schema for IaaS and Paas Offerings

Property types

```
tosca_definitions_version: tosca_simple_yaml_1_0

property_types:
  seaclouds.types.NodeNetworkInfo:
    outbound_bandwidth:
      type: scalar-unit
    inbound_bandwidth:
      type: scalar-unit
    load_balancing:
      type: bool
    number_of_ipv4:
      type: integer

  seaclouds.types.Location:
  seaclouds.types.Location.Europe:
    derived_from: seaclouds.types.Locations
  seaclouds.types.Location.Europe.Germany:
    derived_from: seaclouds.type.Locations.Europe
  ...
```

Figure 9. TOSCA YAML Schema for property types.

IaaS Offering Example

```
tosca_definitions_version: tosca_simple_yaml_1_0

property_types:
  seaclouds.types.os
  seaclouds.types.os.linux:
    derived_from: seaclouds.types.os
  seaclouds.types.os.linux.centos:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.linux.debian:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.linux.fedora:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.linux.gentoo:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.linux.rhel:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.linux.suse:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.linux.ubuntu:
    derived_from: seaclouds.types.os.linux
  seaclouds.types.os.windows:
    derived_from: seaclouds.types.os
  seaclouds.types.os.windows.2008:
    derived_from: seaclouds.types.os.windows
  seaclouds.types.os.windows.2012
    derived_from: seaclouds.types.os.windows
  seaclouds.types.os.freeBSD:
    derived_from: seaclouds.types.os

node_types:
  seaclouds.nodes.Compute.Amazon:
    derived_from: seaclouds.nodes.Compute
    properties:
      operating_system:
        type: seaclouds.types.os
    attributes:
      load_balancing: true

  seaclouds.nodes.Compute.Amazon.c1.xlarge:
    derived_from: seaclouds.nodes.Compute.Amazon
    attributes:
      location:
seaclouds.types.Locations.AM.US.OR.Portland
      operating_system: seaclouds.types.os.linux.ubuntu
      num_cpus: 8
      mem_size: 7 GB
```

```
      disk_type: sata
      local_storage: 1.6 TB
```

Figure 10. IaaS offering example

PaaS Offering Example

```
tosca_definitions_version: tosca_simple_yaml_1_0

nodetypes:
  seaclouds.nodes.Platform.GoogleAppEngine:
    derived_from: seaclouds.node.Platform
    attributes:
      service_name: 'Google AppEngine'
      service_availability:
        - name: 'Service Level Agreement (SLA)'
          percentage: 99.95
        - name: '30 Days'
          percentage: 99.9037

      bandwidth_pricing:
        outbound_pricing: 'First 1GB = Free, Over 1GB = $0.120/GB'
        inbound_pricing: 'No Cap = Free'

      service_features:
        api_restricted: true
        auto_failover: true
        auto_scaling: true
        process_based: true
        MySQL_support: true
        Go_support: true
        Java_support: true
        Python_support: true

       platform_pricing:
         - name: 'B1'
           price_per_hour: 0.080
           cpu: '600MHz'
           memory: 128
           local_storage: 0
         - name: 'F1'
           price_per_hour: 0.050
           cpu: '600MHz'
           memory: 128
           local_storage: 0

       service_location:
seaclouds.types.Locations.EU.IE.Dublin
```

Figure 11. PaaS Offering example

2.4.3. ADP Model

An ADP model is constructed by replacing the module_types of the modules of the AAM with concrete types corresponding to the specific cloud offerings chosen for deployment. Hence, the ADP Model does not require the definition of new elements, but reuses the same elements as defined previously in the AAM. The only requirement, is that an ADP Model enforces that a type defined in a module of the topology should be of a specific cloud Provider (see Figure 12).

Modules (Deployment Layer)

node_templates:
  <module_name>:
  type: <module_type> //shall be of concrete cloud provider
  properties:
  <property_name> : <property_value>
  …

Figure 12. TOSCA YAML Schema for the Modules in the ADP

2.5. Case Studies specification in TOSCA YAML

In this section we validate the defined TOSCAL YAML specification by instantiating them into the use cases of the SeaClouds Project, namely, Cloud Gaming and Softcare [8].

2.5.1. Cloud Gaming specification

Use case description

The Cloud Gaming use case consists on a platform that provisions, manages and runs online games. Figure 13 depicts the different modules of the platform, and are detailed below:

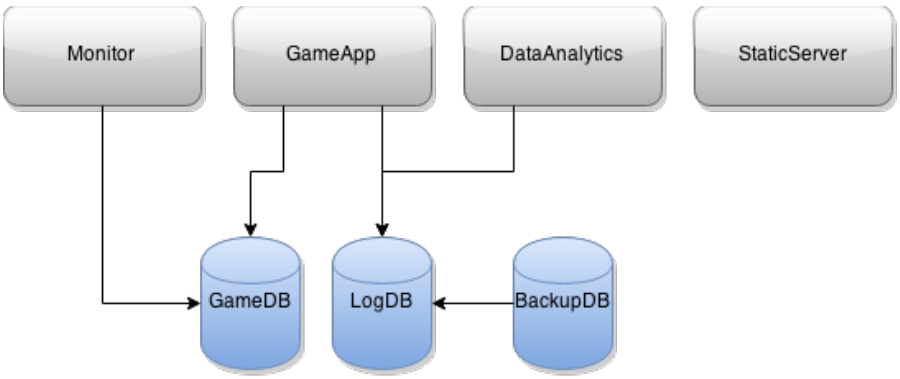

Figure 13. Cloud Gaming topology

Webserver(s) are needed in order to make available the game to the players (nuro's clients) anything capable of processing http and https requests (i.e: apache2, nginx, node.js, etc….)

- StaticServer (web space for the static data): some little static data like graphics and a welcome index page for the audience of the game. from here the game player is also able to register an account to start playing.
- GameApp: the brain of the game, the logic and the rules of how the game works are written here in the PHP programming language.
- DataAnalytics: developed for evaluating the game internally (nuro's privileged information), to evaluate the current number of players, the weekly average number of players, to find cheaters, to evaluate the progress and acceptation/penetration of the game in the market, etc…
- Monitor: this module is responsible of accounting the amount of players as well as the tendency (growing, decreasing) of each value monitored, the average in several time periods (10 seconds, 1 minute, 1 hour, 1 day, 1 week, 1 month)

Object-relational database(s) are needed for storing and retrieving the data generated by the game users.

- GameDB: A database with highly volatile data related to each player situation and progress (playerID, player nick, player level, player score, player resources, etc...)
- LogDB: A database with a log of all the actions and messages received and sent by the player
- BackupDB: A database with backup information.

AAM specification

In this section we show how the AAM for the Cloud Gaming platform can be instantiated following the TOSCA YAML specification. For readability reasons, here we present an extract of the document, For the complete TOSCA YAML AAM file, the user may download it from [3].

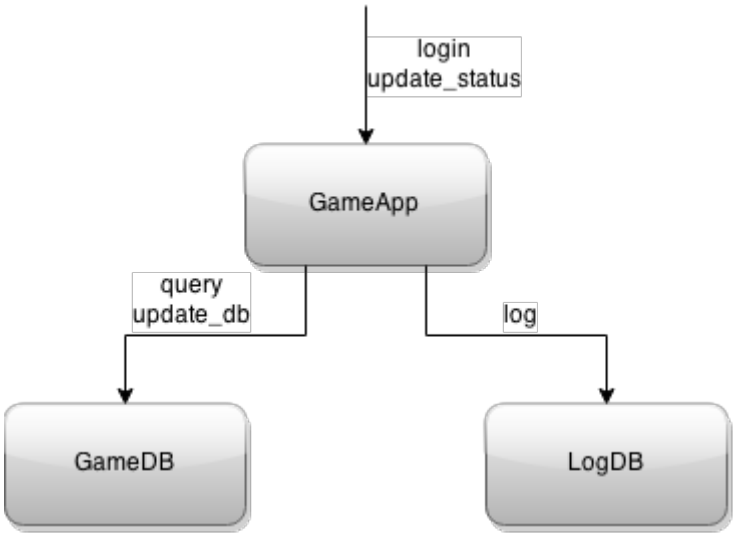

Figure 14 Excerpt of the Cloud Gaming topology

We consider the topology subset shown in Figure 14, which is composed of the GameApp, Game DB, LogDB and their relationships. GameApp connects to GameDB to query or update the database, for instance, when users log-in to validate the password or to perform an update of the status of the game. Similarly it updates the LogDB every time an action must be logged.

We describe below some technical characteristics and QoS for that topology:

The AAM defines that the module `game_app` must be a IaaS with at least 4 cpus, vertical scaling support and to be located in Germany. The `game_db` module must be a PaaS that should be deployed in Germany. The `log_db` is a IaaS with at least 100 GB of disk capacity.

These represents technical requirements and are defined in SeaClouds TOSCA YAML as follows:

```
node_templates:
  game_app:
    type: seaclouds.nodes.Compute
    properties:
      location: seaclouds.types.Locations.Europe.Germany
      num_cpus: 4
      scaling_vertical: auto

  game_db:
    type: seaclouds.nodes.Platform
    properties:
      location: seaclouds.types.Locations.Europe.Germany
      mongoDB_support: true
```

```
  log_db:
    type: seaclouds.nodes.Compute
    properties:
      disk_size: 100 GB
```

The relationships between these modules are described as follows:

```
 relationship_templates:
    game_app.update_status.to.game_db.update:
      type: seaclouds.relationships.Uses

    game_app.update_status.to.game_db.query:
      type: seaclouds.relationships.Uses
      properties:
        average_usage_count: 2

    game_app.update_status.to.log_db.log:
      type: seaclouds.relationships.Uses

    game_app.login.to.game_db.update:
      type: seaclouds.relationships.Uses

    game_app.login.to.game_db.query:
      type: seaclouds.relationships.Uses

    game_app.login.to.log_db.log:
      type: seaclouds.relationships.Uses
```

Finally, we define the operations. In particular we show `game_app.login` operation assuming that it is hosted by game_app and requires to perform three operation (`query_db`, `update_db`, `update_log`) each with their qos requirements. Similarly we define the `update_status` operation.

```
node_templates:
  game_app.login:
      type: seaclouds.nodes.Logic
      requirements:
        - host: game_app
        - query_db:
            node: game_db.query
            relationship:
game_app.login.to.game_db.query
        - update_db:
            node: game_db.update
            relationship:
game_app.login.to.game_db.update
        - update_log:
```

```
            node: log_db.log
            relationship: game_app.login.to.log_db.log

  game_app.update_status:
      type: seaclouds.nodes.Logic
      properties:
        qos_requirements:
          execution_time: 2.0
        qos_info:
          execution_time: 1.02
          benchmark_platform: seaclouds.nodes.amazonEC2
      requirements:
        - host: game_app
        - query_db:
            node: game_db.query
            relationship:
game_app.update_status.to.game_db.query
        - update_db:
            node: game_db.update
            relationship:
game_app.update_status.to.game_db.update
        - update_log:
            node: log_db.log
            relationship:
game_app.update_status.to.log_db.log
```

2.5.2. SoftCare specification

Use case description

The SoftCare solution is a software application that aims at developing an innovative and integrated solution for the use of social inclusion tools by elderly people and for the general management (self-management included) of their medical problems.

Figure 15 depicts the different modules of the SoftCare solution, which are described below:

- Softcare Core app: A web service based application that has the main logic and also acts as the interface between all the SoftCare solution main components. It's a SOAP based web services application developed in Java 1.7, running under Tomcat, and using different frameworks (Apache CXF, Spring and Hibernate). It requires a low response time and a very high availability. It also requires to be deployed in a private PaaS.

- Softcare DB: The main database used by the Sofcare Core app. Requires MySQL 5.5 or higher.

- Softcare Desktop client application: (This application won't be deployed in any cloud provider). It's a desktop application used by the elderly people that will connect with the Softcare Core app

- SoftCare Web applications: It consists on the Softcare Web client app and the Softcare Web admin app. These modules are required to be deployed in a PaaS with support for Java 1.7 and Tomcat. The Softcare Web client app requires to be deployed in a public PaaS, whereas the Softcare Web admin app requires to be deployed in a private PaaS.

- Forum Web application: a forum application that uses the Forum DB, and the Multimeda Respository Application. It is required to be deployed in a PaaS with support for Java 1.7 and Tomcat.

- Forum DB: database for the Forum Web application. It requires MySQL 5.6.

- Multimedia Repository Application: multimedia repository based on Lily Data Repository. It is required to be deployed in a private IaaS, with Linux, Java 1.6, Hadoop, HBase, ZooKeeper and SOLR.

- Services app: the Softcare solution also includes a Services application that will connect to third party services offered by cloud providers (like an email delivery service) in order to use them. It is required to be deployed in a public PaaS with Java 1.7 and Tomcat support.

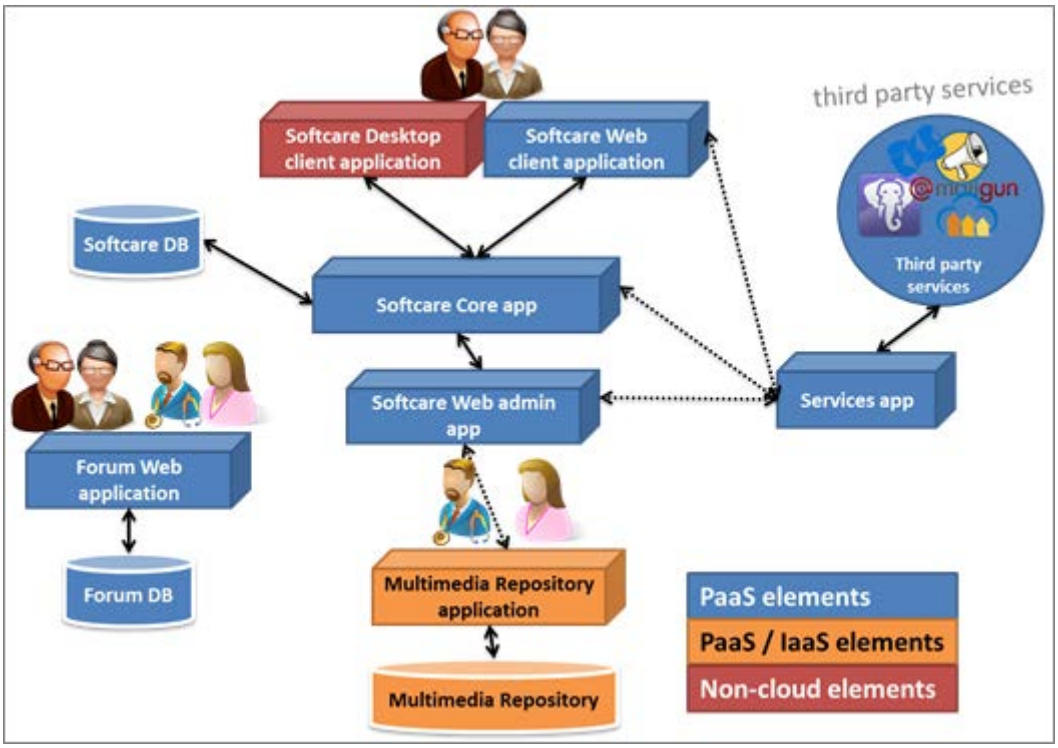

Figure 15. Softcare topology

AAM specification

In this section we show how the AAM for the Cloud Gaming platform can be instantiated following the TOSCA YAML specification. For readability reasons, here we present an extract of the document. For the complete TOSCA YAML AAM file, the user may download it from [3].

We will consider the topology subset shown in Figure 16, composed by Softcare Core App and Sofcare DB also considering the relationship between them.

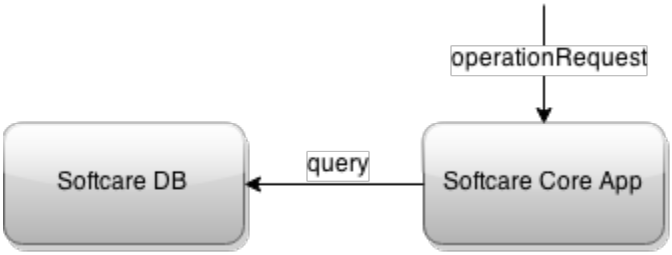

Figure 16. Excerpt of the SoftCare topology

First we define the two modules `SoftcareCoreApp` and `SoftCareDB` with their requirements.

```
node_templates:
  SoftcareCoreApp:
    type: seaclouds.nodes.Platform
    properties:
      Java_support: true
      Java_version: [1.7,1.7]
      MySQL_support: true
      private_paas: true

  SoftcareDB:
    type: seaclouds.nodes.Platform
    properties:
      location: seaclouds.types.Locations.Europe
      MySQL_support: true
      MySQL_version: [5.5,5.7.5]
```

We establish the relationship between the core application and the DB and the one describing the request of operation to the core application. Below we exploit the TOSCA relationships.

```
relationship_templates:
  operationRequestTo.SoftcareApp:
```

```
    type: seaclouds.realtionships.Uses
  SoftcareCoreApp.query.SoftcareDB:
    type: seaclouds.relationships.Uses
```

Then we define the quality of service requirements as `Logic` node templates. We consider 99.9% as a measure of high availability and at most one second as low response time.
In order to exemplify better how to define AAM we assume that the every operation request to the core application requires querying the DB.

```
node_templates:
  operationRequest:
    type: seaclouds.nodes.Logic
    properties:
      qos_requirements:
        response_time: 1 s
        availability: 99.9
    requirements:
      - host: SoftcareCoreApp
      - query_db:
          node: SoftcareDB.query
          relationship: SoftcareCoreApp.query.SoftcareDB
```

3. Parser of TOSCA

As it was mentioned above, SeaClouds uses the TOSCA syntax to describe and encapsulate in an agnostic manner the information provided by the different components. For example, as we can see in Section 2.3.2, the features, capabilities, requirements and other aspect of the cloud providers will be modelled using a subset of the TOSCA specification which allows to take advantages of the standard.
In this section, we present the TOSCA parser which will be used by the different SeaClouds components to process the information during the life-cycle of any application management. Then, in order to facilitate the aforementioned task, the goals of the TOSCA parser is provided a useful tool that allows the necessary SeaClouds components to manage the TOSCA specification in a portable and reusable way.

TOSCA propose two representations to describe its syntax, XML and YAML. Although, TOSCA Simple Profile in YAML was aimed to provide a more accessible syntax as well as a more concise and incremental expressiveness of the standard, an application description could become a very extensive YAML file that should be processed to use the contained information. The main goals of the parser is to provide a unified and useful tool that allows processing a TOSCA description and generating and object representation. Thus, each SeaClouds component that needs to manage a TOSCA

profile, both input and outputs, could use the parser to accomplish its work. Following, we list the components which interacts potentially with the TOSCA parser:

- Dashboard: Saves the User Input in TOSCA.
- Discoverer: Saves the Cloud offers information in TOSCA.
- Planner (Matchmaker & Optimizer): Read the user input, read the discoverer information, saves DAM in TOSCA
- Deployer: Reads the DAM in order to deploy and manage the application
- SLA: Reads the QoB from the DAM.

3.1. Using a TOSCA subset

TOSCA is a full-expressive standard which specify a particular methodology to describe and wrap the cloud application structure (components and relationships), and how they must be orchestrated (by means of a plan) in a portable way to increase a vendor-neutral ecosystem. Moreover, they describe the mechanisms which must be implemented by the clouds to support standard-based application deployment and management. However, as we have mentioned previously, we adapt the standard specification to our requirements so we will not use all features provided by TOSCA.

As we presented above, SeaClouds focuses on Nodes Types and Templates, Properties, Capabilities for describing the clouds offering, and applications description such as AAM and DAM. However, the implementation and management of the applications is not addressed using the TOSCA mechanisms, we use more flexible technologies to this goals, which will be implemented in the deployer. The definition and maintenance of an orchestration plan is a complex and error-prone task. The plans have to define each necessary step to deploy and configure the application taking into account all the properties and requirements of the providers. Also, according to the current mechanism proposed, the plan's operations should be modified in case of changing the cloud providers in which modules are deployed. This is because the modification of the providers is performed by substituting the artefacts that implement the deployment operations. In Figure 17, we can see a diagram that show the used and dropped TOSCA elements in SeaClouds. The used elements are highlighted by a blue background, in the same way the not used elements are indicated by a red background.
Taking in account these restrictions, the first draft of the proposed parser does not address the processing of the orchestration plan and the implementations artefacts management (such as the implementations of node types operations).

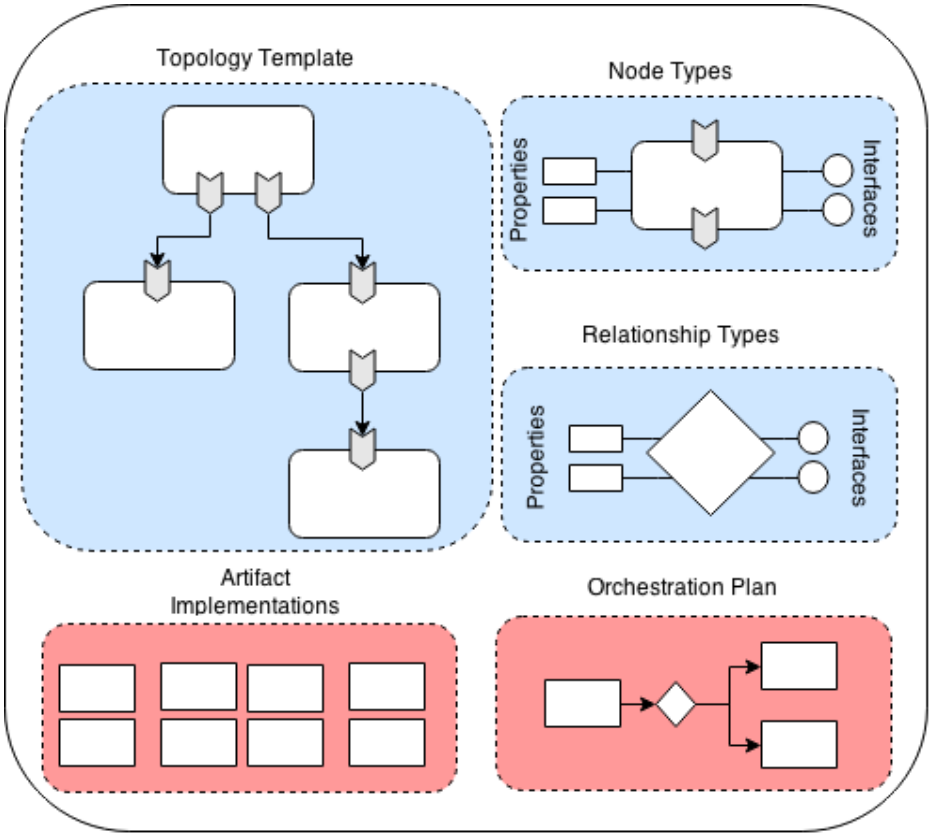

Figure 17. The TOSCA elements which are used in SeaClouds.

Following these usage restrictions, several applications and cloud offering descriptions examples are shown in previous sections.

3.2. Design decisions

In this sections, we describe the TOSCA parser that is designed according to the usage restrictions of the elements.

As we have mentioned previously, the main goal of the parser is to generate an object representation from a TOSCA description. The parser defines and implements a mapping between TOSCA files and a Java object model representing them. In Figure 18, an UML class diagram shows the Java representation of TOSCA concepts such as **NodeTypes**, **Properties**, **Capabilities**, **values**, etc.

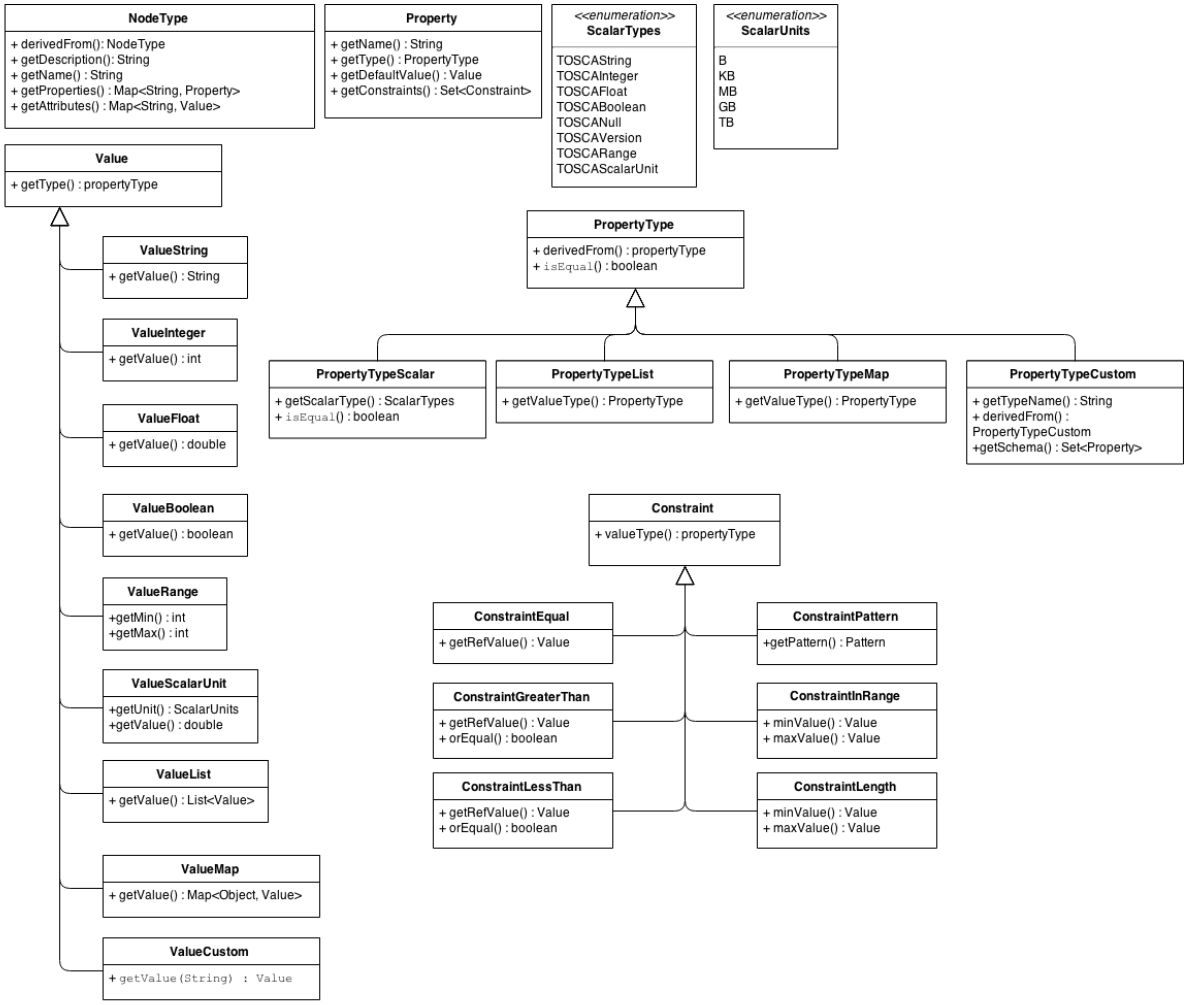

Figure 18. TOSCA elements model.

The parser provides to any SeaClouds component an object representation of the basic TOSCA elements, according to the schema specification. However, in many contexts in SeaClouds components it is necessary to have a more specific representation of some tosca types, for example for the matchmaking of cloud offerings description it may be useful to have a java interface corresponding to NodeTypes `seaclouds.node.Compute` or `seaclouds.node.Platform`. In this case the specific interface can be bound to the generic NodeType or NodeTemplate object using Java `Proxy` class [4]. The specific interfaces of the parser for the aforementioned SeaClouds TOSCA documents such as the Cloud Offering, AAM, ADP and DAM are yet to be defined because they are tightly bound to the actual implementation of the modules using them.

4. Discoverer

4.1. Discoverer Architecture & Design

The discoverer component of SeaClouds is responsible of providing the planner with information about the available cloud offerings. In order not to rely on a single source of information the architecture of the discoverer is modular. The main module aggregates the information from the adapter modules and exposes it to the rest of the SeaClouds platform. The adapter modules are responsible of converting the information from heterogeneous sources into TOSCA YAML format for cloud offerings, which will then be consumed by the main module. The modular approach allows also for easy extension of the discoverer if some of the used sources of information becomes unavailable and must be replaced. Figure 19 depicts the architecture of the Discoverer.

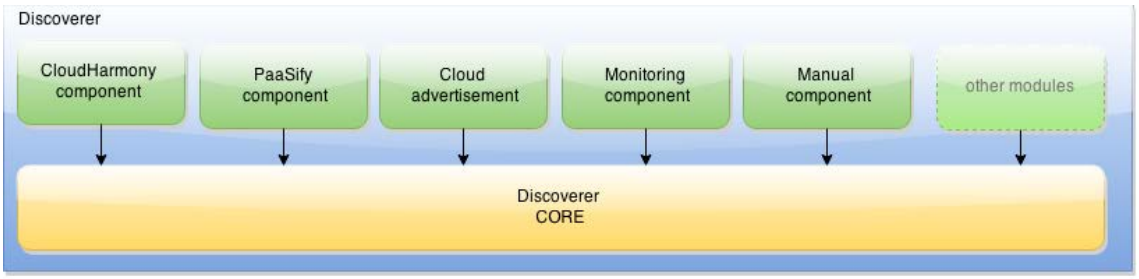

Figure 19. Discoverer architecture

4.2. Discoverer Modules

The adapter modules retrieve information from various sources. For example, different cloud comparing services, an API which allows cloud provider themselves to announce their own offerings, a module which allows for manual input of data, etc. The initial modules that we define are the following:

4.2.1. CloudHarmony component

CloudHarmony [9] is a repository of cloud offerings of different nature. Most of the information stored in CloudHarmony can be mapped to the required properties that are defined in the cloud offering TOSCA YAML model used by SeaClouds. CloudHarmony provides a query API which allows the information to be fetched in JSON format. The discoverer CloudHarmony component fetches this data, converts it into TOSCA YAML format and stores it in the Discoverer CORE.

4.2.2. PaaSify component

PaaSify [10] is the second directory that SeaClouds will rely on for populating the local repository of cloud offerings. The PaaSify web site focuses mostly on two main strength points: i. searching the repository can be performed by adding filters that shrink the size to the suitable offerings according to the specified characteristics; ii. the accuracy of the fetched data is hardly wrong because the repository is filled manually by users within the community or by the administrator of the offered clouds himself.

From the SeaClouds perspective, PaaSify will be used to populate the local repository of cloud offerings, basically by getting the JSON files from to the PaaSify profile directory, then convert them in TOSCA YAML, in such a way that they can be used locally by SeaClouds. The PaaSify JSON files can be retrieved through Github.

4.2.3. Cloud advertisement

After launching the SeaCloud platform, we expect the cloud providers to be interested in having their own offers listed in the SeaCloud discoverer directory, in order to increase their visibility. We propose a model for which cloud providers can notify the SeaCloud platform of updates to their offering, and provide the information in the TOSCA YAML format for the Cloud Advertisement component. We see two possible strategies for this interaction: the first one is the PUSH approach, where the cloud provider submits a list of their updated services whenever changes occur, the second is the PULL approach, where the cloud providers register a URL for their offers, with the updated list of cloud offering that can be fetched when needed.

The PUSH approach has the advantage that the cloud providers do not have to setup a URL with the purpose of advertising to SeaClouds, whereas the PULL approach allows the cloud providers to advertise automatically to other frameworks beyond the scope of SeaClouds.

4.2.4. Monitoring component

SeaClouds monitoring components will be deployed on various cloud services. The monitoring results of these components can be used to as an alternate source of data for some of the properties of the cloud offerings stored in the repository, in order to complete or validate the information already present. The monitoring component of the discoverer can get this information from the live model and update the data of the repository accordingly.

4.2.5. Manual component

Even if automated systems are available, manually introducing cloud offerings information should be kept as an option. Manually maintaining a list of cloud offerings may in some cases have lower cost than maintaining the software for automatic detection. On the other hand, some technical information may not be available in a machine readable format for automatic detectors to read. Under these circumstances, the manual intervention of an administrator could be preferred.

5. Planner

The Planner component is in charge of providing a set of deployment plans that define where each application module will be deployed. Given an AAM the Planner will generate a set of ADPs that meet the requirements specified by the user.

The generation of a deployment plan consists of two steps:

1. matchmaking the suitable offerings for each module;

2. optimizing the set of suitable offerings reducing the number of possible configurations.

5.1. Planner Architecture & Design

The planner architecture is composed, as shown in Figure 20, by three modules: Matchmaker, Optimizer and DAMGenerator.

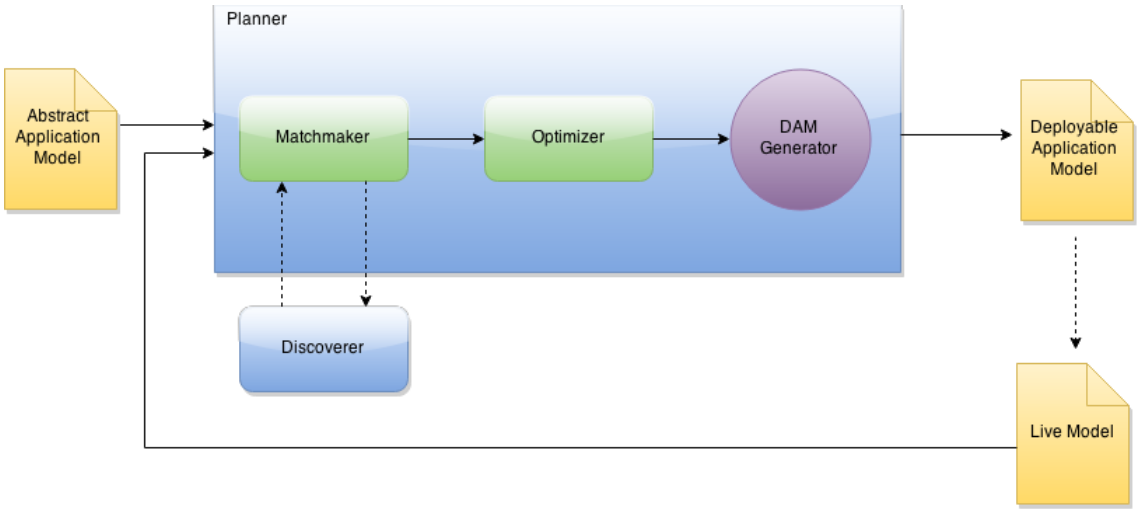

Figure 20. Planner Architecture.

At design time, in order to plan the deployment of an application, the Planner component requires as input an AAM. At run time, for replanning, the Planner requires also the information about the Live Model. The planner expose *plan* and *replan* as the two main functionalities that, respectively address the first planning of an application and the replanning process.

Figure 21 shows the sequence diagram for the planning process. The dashboard will trigger the planner requiring its planning functionality and giving the AAM for the application to be planned. The planner calls the Matchmaker that will get (a stream of) cloud offerings from the discoverer and will look for the suitable offerings for each AAM module. When the matchmaking process ends, the Planner requires the optimization step from the Optimizer that will select a set of optimal deployment proposal. Finally, for this set of deployment proposals, the DAM generator generates the DAM.

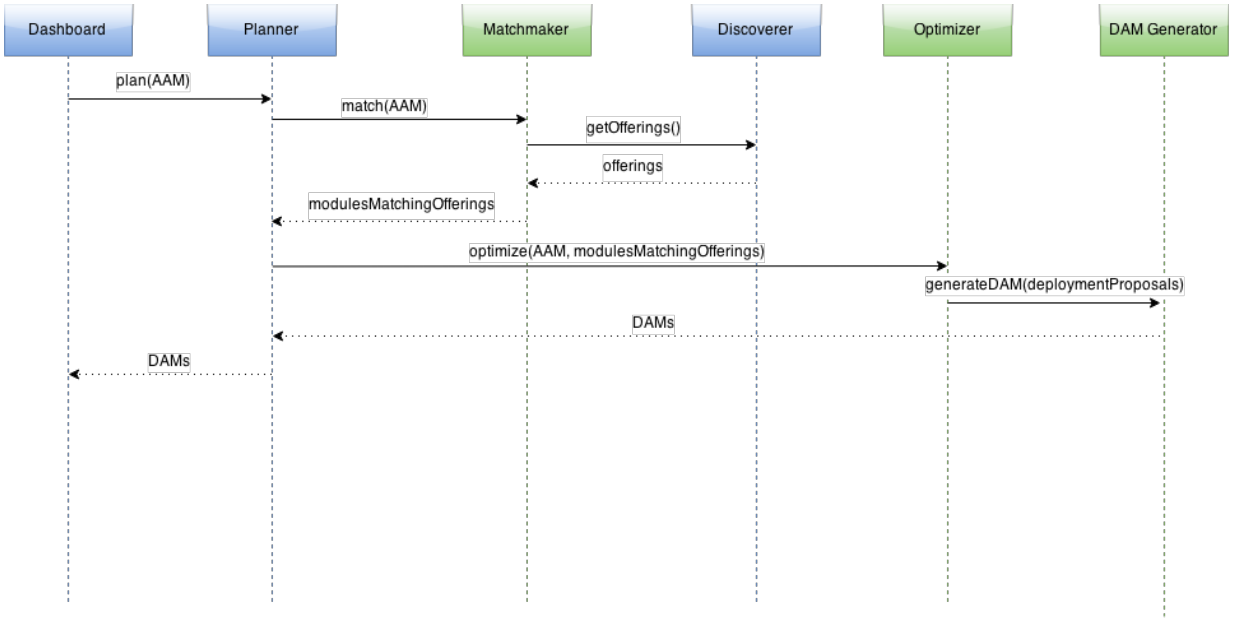

Figure 21.Sequence diagram of planning

After the first successful deployment of an application with SeaClouds, it can happen that for different reasons (e.g. monitoring or SLA violations) a replanning is required. The process of replanning an application is similar to planning but, having already deployed the application, the Planner can leverage the Live Model information to better optimize the deployment proposals. In particular the Live model contains runtime information about the actual deployed application and the causes that triggered the replanning process.
Figure 22 shows the sequence diagram for replanning.

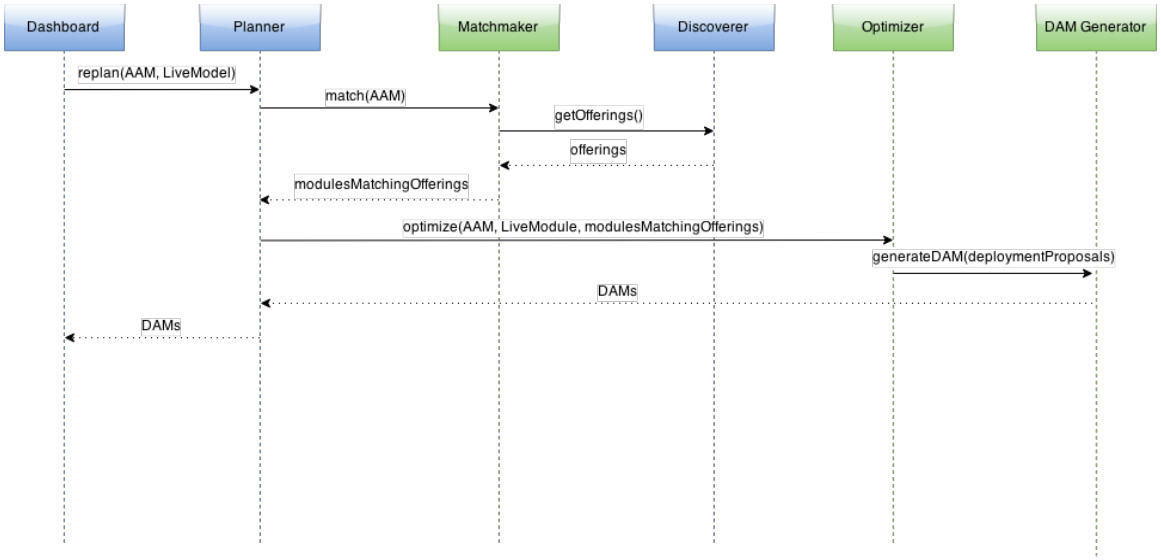

Figure 22. Sequence diagram of replanning

5.2. Planner Modules

5.2.1. Matchmaking

The Matchmaker is in charge of matching the user requirements for the different modules as described in the AAM with the actual offerings from the Discoverer of SeaClouds (see Figure 23).

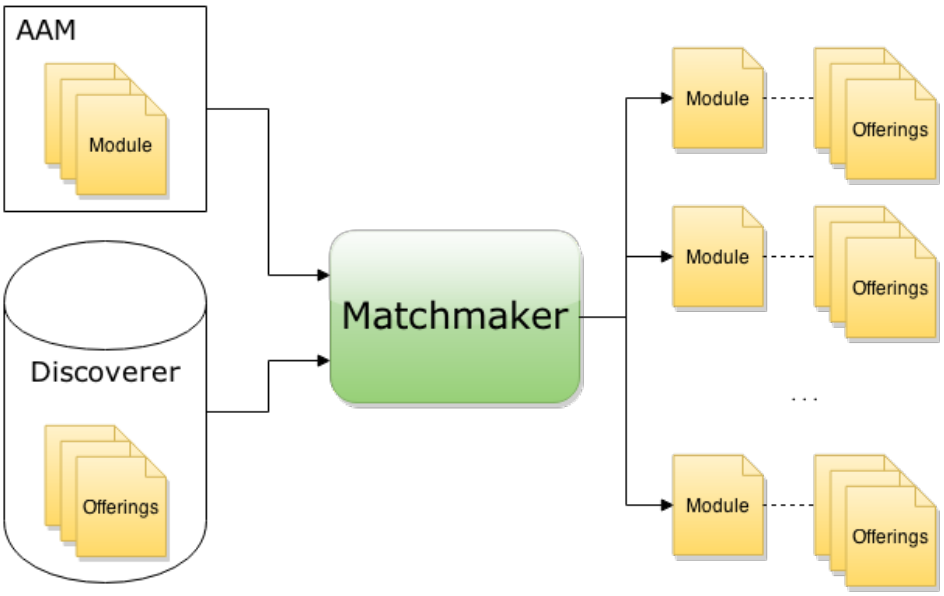

Figure 23. Design view of the Matchmaker

Input

The input for the matchmaker is a set of application modules and a set of cloud offerings.

Output

The output of the matchmaker is the collection of matching offerings for each module.

Assumptions

The matchmaker assumes that, for each functional property, the value domain of such property has a partial order (e.g. more cpu cores are better than less).

Matchmaking Process

The matchmaking process looks for every cloud offering that fulfil the technical requirements defined into the modules of the AAM.

The matchmaker assumes that each module contains a set of technical requirements and that a (partial) order is defined for each technical requirement.

A suitable offering for a module is a cloud offering that has equal or better (according to the ordering for each property) properties for all the defined technical requirements in the module.

5.2.2. Optimization

This section describes the interfaces offered by the optimization module and the design of its internal behaviour. The goal of the optimization module is to provide the planner with an optimization problem solving method, which can find an appropriate orchestration of the cloud services. That is, whereas the Matchmaker identifies the cloud offerings that meets the requirements at the module level, the Optimizer identifies the compositions that satisfies the global application requirements (e.g., application's performance and availability) while minimizing both the cost incurred on the usage cloud computing resources and the effort on the migration of modules (the latter is only considered if the optimization module is invoked during a reconfiguration process of replanning).

Offered interfaces

Optimization module offers the following methods:

- Optimize(String AAM, String suitableCloudOffers)

This method produces a set of candidate partial plans where, in each plan, each module is associated to one and only one cloud service. In case of being possible to define the number of replicas of a service, the information for the deployment of the application module over such service is enhanced with the initial number of replicas to use that is suitable to deal with the application expected utilization and satisfy its performance and availability requirements. More information about the interface and invocation of Optimize method is provided in [5].

- Reoptimize(String AAM, String cloudOffers, String currentDAM, String reasonOfReplanning)

This method produces a set of candidate partial reconfiguration plans. In each of these candidates it is specified the information for changing from the current DAM that is no longer valid to a computed alternative deployment that overcomes the current DAM problems. In case of being possible to define the number of replicas of a service; then the information for the deployment of the application module over such service is enhanced with the initial number of replicas to use that is suitable to deal with the application expected utilization and satisfy its performance and availability requirements.

Internal behaviour

The design of Optimize method regarding its required and provided information is illustrated in Figure 24, while the sequence diagram is illustrated in Figure 25 and described below.

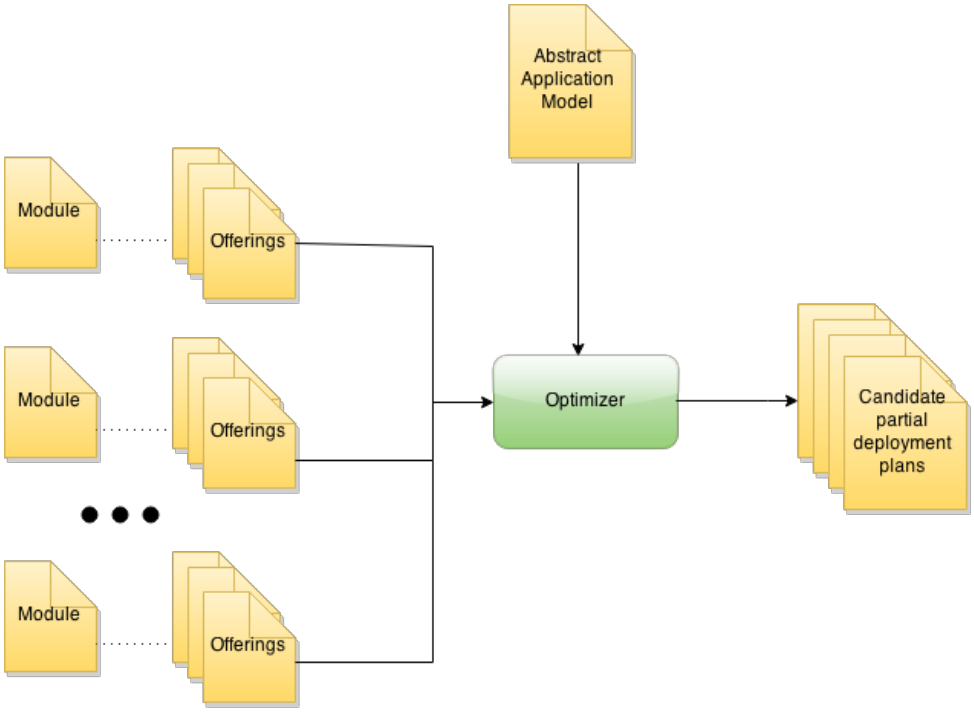

Figure 24. Design view of Optimize method interfaces

As presented in the interfaces description, optimizer receives an invocation of its method "optimize" with parameters AAM, and suitableCloudOffers. First, it uses operations getTopology, getQoSinformation and getCloudOffers of TOSCA parser to obtain its required information in a structured manner, which includes:

- Information of the application topology, dependencies between modules and the set of suitable cloud offers for each module, which are stored in the AAM.

- The QoS information of the application, which comprises the QoS requirements and QoS properties that are necessary for calculating its quality (e.g., for computing the expected performance of the application, it is necessary to know the expected number of user requests per minute that it will have to serve), which are stored in the AAM.

- Information regarding the properties of each of the suitable cloud offers (e.g., its capability to scale horizontally, availability, etc.), which are stored in suitableCloudOffers.

After this information has been collected, it is created the object that implements the optimization problem solving. The optimizer implements more than one optimization problem solving methods.
The decision of which one should be used is a decision of the optimizer component itself since but the SeaClouds user is not required to be an expert in this domain, but it is helpful for further research goals in order to compare the behaviour of different methods.

After this, it is invoked the method solve of the instantiated optimization problem solving method having as parameters the application topology, information of cloud offers, application QoS requirements and application QoS properties. This output of this method consists of a list with the best solutions it has been able identify.

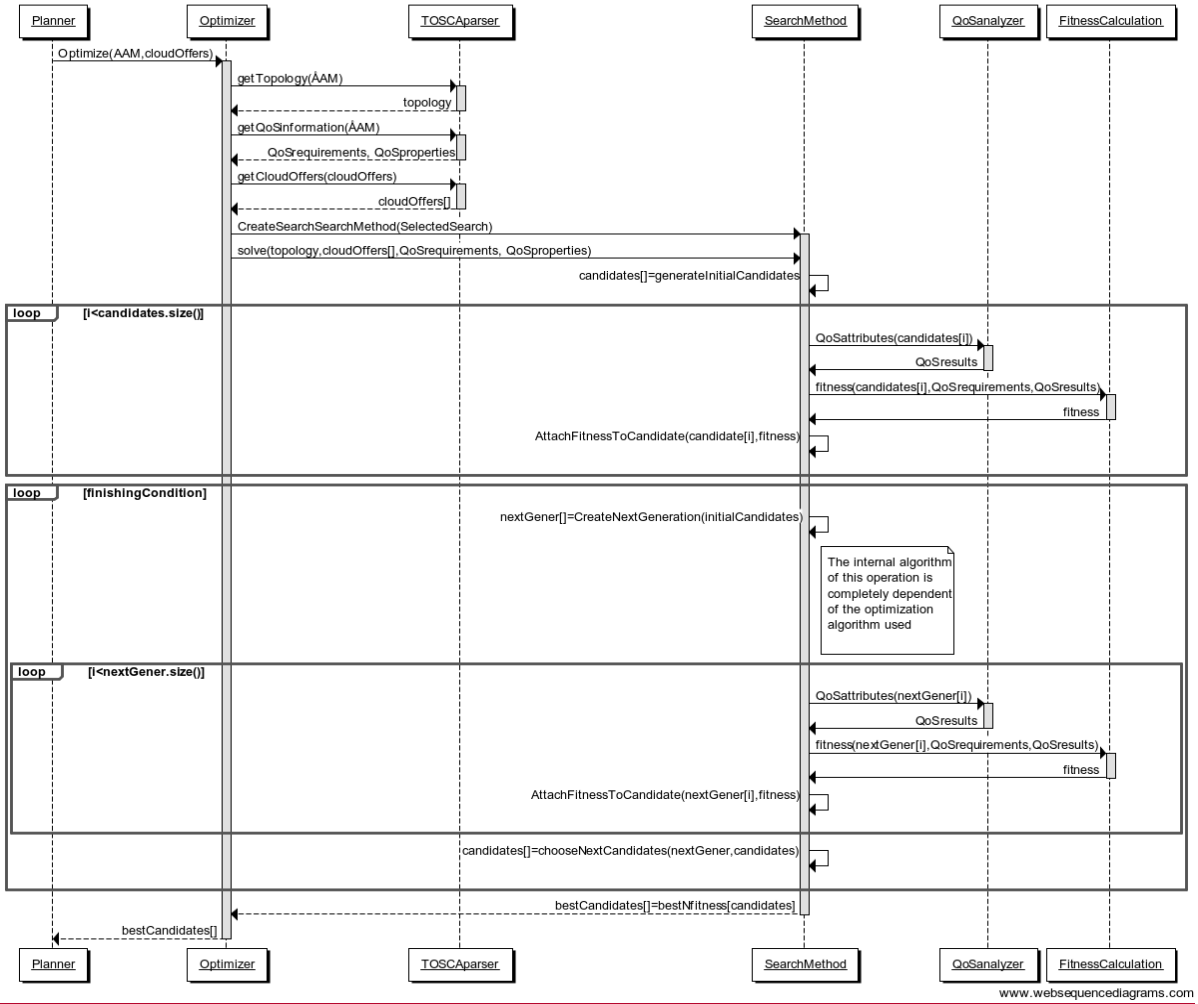

Figure 25. Sequence diagram of Optimize

Regarding the internals of solve method, the rest of the Sequence Diagram provides a high level view of the their behaviour. Slight variations in this behaviour are possible, according to the particularities of the actual optimization method instantiated in each case. The design view provided in the rest of the sequence diagram extends the view of

the optimizer behavior provided in an algorithmic form in D3.1 [1] since it concretizes the classes and objects that implement the optimization operations such algorithm. There is a large matching between the operations in the first specification algorithm in D3.1 [1] and the operations in the sequence diagram in Figure 25, notwithstanding the matching is not complete. The reason is that the requirements for the optimizer have changed during this time interval. Some examples of differences are:

- In D3.1 [1] it was assumed that the optimizer provided only one candidate solution -the one that the fitness function considered the best- while the current optimizer requirements are that a list of candidates is provided to the user -the list of the N candidates that the fitness function considered the best ones- and s/he can choose one among them.

- In the first design provided in D3.1 [1] the *fitness* function was responsible for both compute the quality attributes of a candidate with respect to different properties (e.g., performance and availability) and to calculate the fitness of a candidate from these quality attributes. In the current design the calculation of the *fitness* of a candidate still uses its quality attributes but the computation of such quality attributes is decoupled from the *fitness* function itself. Moreover, the fitness calculation of a candidate is also decoupled from the concrete search method used in each moment. These decisions were taken to improve the maintainability and extensibility of the optimizer, allowing easier additions of *SearchMethod* and *FitnessCalculation*. In this way, the set of implemented search methods is more easily extendable since they do not need to implement a fitness function but reuse one of the already existent ones. Accordingly, since the calculation of QoS attributes is not dependent of the optimization method or fitness function used but they are related to other research areas (e.g., to the Software Performance Evaluation [Smith] in the case of computing the application expected response time), we have decide to include these capabilities in a package different from the optimization methods and fitness functions. Following the current design, the set of *SearhMethod* and *FitnessCalculation* is more easily extendable. As a corollary, this separation increases also the trustability of the comparison between optimization methods due to the execution of the same code for the QoS analysis and evaluation solution fitness.

In this context, the *solve* method first generates a random set of initial solution candidates. For each candidate it calculates its quality attributes and, using such attributes and the quality requirements, its fitness value.
At this point, an iterative process of optimization begins its execution using the initial random set of candidates. It will finish when a “finishingCondition” is satisfied. The firsts operations inside the iterative process create the next generation of solutions using the

set of current candidate solutions. The algorithm to produce the next possible solutions follows the two step optimization process given in D3.1 [1] (i.e., a first step for deciding the cloud provider and a second step for the choice of a concrete offer among the suitable set of the provider), while its particularities are completely dependent on the optimization problem solving method implemented. Next, it is calculated the fitness of each solution in the new generation. The last operation of the iterative process (called *chooseNextCandidates* in Figure 25) consists in choosing as candidate solutions the "appropriate" ones from the set of current candidate solutions and next generation according to the optimization method instantiated. This operation is also completely dependent of the optimization method instantiated, hence the choice of next candidates may not correspond to the selection of the possible solutions with the highest fitness values. When *"finishingCondition"* of the iterative process is satisfied, *solve* method returns the N alternatives with best fitness among the current list of candidate solutions.

5.2.3. DAM generation

The DAM generation process starts after the Optimizer has obtained the ADPs and the user has selected the most suitable one. This process consists on transforming the selected ADP into a fully deployable model in the form of DAM. To this end, more information has to be introduced in the model (e.g. credentials, reconfiguration policies,…). Such information is retrieved by asking the user the missing information required by the Deployer, and including this information using the TOSCA specification.

The DAM generation process is heavily dependant on the Deployer component. The integration with the Deployer will be consolidated in subsequent deliverables and will drive the technical details and design decisions for the transformation of the orchestrated solution in ADP into an automatically deployable model in DAM.

6. Conclusions

In this deliverable we have described the process and artefacts for the generation of a deployment plan that orchestrates the different modules of the application.

In the first part of the deliverable, we have consolidated the Application Model Lifecycle, which includes the AAM, Cloud Offerings, ADP, DAM, and Live Model. Secondly, we have elicited the required pieces of information that should be present in the models used for the generation of a deployment plan. Then we have described how these elements are represented: first by means of a graphical human-readable model, and then by a formal machine-readable TOSCA YAML specification. Finally, we have validated the specification on the use cases of Cloud Gaming and Softcare.

In the second part, we have described the architecture and design of the different components of SeaClouds that are involved in the generation of a deployment plan. We have defined the structure and design of the Parser of TOSCA YAML, which builds an object-oriented representation of the TOSCA YAML model; and then we have described the design and architecture of the components of SeaClouds involved in the generation of the deployment plan. Namely, the Discoverer and the Planner.

7. References

[1] SeaClouds Project Team, Public Project Deliverable. “D3.1. Discovery, Design and Orchestration Functionalities: First Specification”, October 2014.

[2] SeaCloud Project Team, Public Project Deliverable. “D2.1 Requirements for the SeaClouds Platform. V1.5”, March 2015.

[3] Repository of TOSCA YAML files https://github.com/SeaCloudsEU/SeaCloudsPlatform/tree/master/planner/core/src/main/resources

[4] Java Dynamic Proxy Specification https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.htm

[5] SeaCloud Project Team, Public Project Deliverable. “D4.5 Unified dashboard and revision of Cloud API“, March 2015.

[6] OASIS, “TOSCA Simple Profile in YAML Version 1.0”, Committee Specification Draft 02, December 2014.

[7] OASIS, “Topology and Orchestration Specification for Cloud Applications, v1.0”, 2013. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

[8] SeaCloud Project Team, “D6.1 Case study extended description”, 2014

[9] CloudHarmony web page https://cloudharmony.com/ (last retrieved March 2015)

[10] Paasify Comparative and Supported Providers. http://www.paasify.it/vendors (last retrieved March 2015)