SeaClouds Project

D4.1 Definition of the multi-deployment and monitoring strategies

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	WP4. SeaClouds run-time environment
Deliverable code	D4.1
Deliverable Title	Definition of the multi-deployment and monitoring strategies
Nature	Report
Dissemination Level	Public
Due Date:	M12
Submission Date:	10th October 2014
Version:	1.0
Status	Draft
Author(s):	Miguel Barrientos (UMA), Jose Carrasco (UMA), Javier Cubo (UMA), Francesco D'Andria (ATOS), Francisco Durán (UMA), Alex Heneveld (CloudSoft), Adrián Nieto (UMA), Andrea Turli (CloudSoft), Ernesto Pimentel (UMA), José María Álvarez (UMA)
Reviewer(s)	Ana Juan Ferrer (ATOS), Michela Fazzolari (UPI)

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 definition of the strategies to deploy on specific cloud providers, and to extract the information necessary for performing monitoring of deployed applications from the description of the SeaClouds orchestration defined in Deliverable D3.1 [1]. Specifically, the document describes how the modules of a cloud application are deployed on heterogeneous clouds, and how the monitoring and SLA services are initialized and executed. A real use case, namely the Nuro (early) case, is used to illustrate the application of the strategies defined in this document.

1. Introduction

The main objective of the WP4 (SeaClouds run-time environment) is the design of a unified management and monitoring infrastructure to help on the distribution of cloud-based applications over multiple and heterogeneous clouds, and to re-distribute its modules in these clouds during execution, considering a universal set of metrics to guarantee the desired properties and requirements, including Quality of Services (QoS) and Quality of Business (QoB).

Following the Figure 1, corresponding to the Initial Architecture of the SeaClouds Platform defined in Deliverable D2.2 [2], we overview how the different components interact. Specifically, in this document we present the SeaClouds run-time environment composed by the Deployer, Monitor and SLA components, after present the Discoverer and Planner in Deliverable D3.1, corresponding to the SeaClouds design-time. A more detailed flow representing all the interactions among the different components, and representing functionalities, is explained (apart from in D2.2) more in detail in Deliverable D5.4.1 related to the Initial version of sw platform [3] (we refer the reader there to not repeat here the sequence already described in that document).

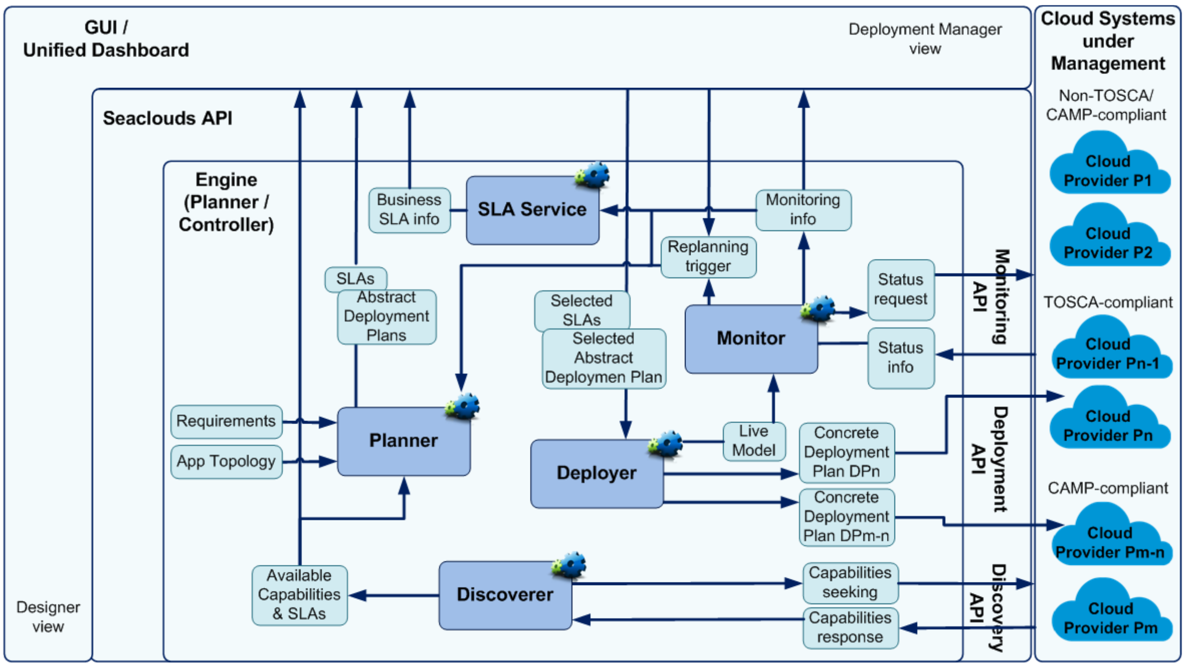

Figure 1. Initial Architecture of the SeaClouds Platform (see deliverable D2.2 [2])

The multi-cloud deployment component, presented in Section 2, will be in charge of generating a concrete deployment plan for each target cloud platform. Concrete deployment plans include all the needed steps to be performed to actually deploy a (set of) application module(s) on each specific cloud platform. In order to perform this, the Deployer component gets an abstract deployment plan, generated by the Planner component [1], together with the SLAs of the selected services.

The monitoring component is described in Section 3. Its purpose is to monitor the deployed application modules, and to verify the QoS properties for specific modules and the SLA of the whole application, specified at design time. The Monitor component will be connected to a reconfiguration component (which will be completed in the next phase of the project, in Deliverable D4.3 [4]) capable of performing an evolution or migration of modules composing the application when it is required. Thus, SeaClouds will preserve the soundness of the SeaClouds orchestration specification.

The SLA service is introduced in Section 4. It enables the SLA management of business-oriented policies. Given a component to generate the formal documentation describing agreements between the parties involved (customers, application providers and cloud providers), the SLA service will guarantee all the agreements associated with the fulfillment of the non-functional properties.

To facilitate reuse and modularity, the operations corresponding to the deployment, monitoring and SLA service (in addition to the planner) will be provided as a unified API, and a dashboard will be provided (deliverable D4.2 describe a first design of the SeaClouds API [5]).

A distinguishing aspect of the SeaClouds platform is that it intends to build on top of two OASIS standards initiatives: TOSCA (to represent application topologies) and CAMP (to use standardized artifacts and APIs that need to be offered by PaaS clouds to manage the building, running, administration, monitoring and patching of applications in the cloud). It is, however, worth noting that the Deployer and the Monitor does not require cloud providers to be TOSCA or CAMP compliant. The Deployer can generate concrete deployment plans for non-TOSCA/CAMP compliant providers as needed [2], and the Monitor can take non-TOSCA/CAMP compliant monitoring agents (as will be explained in subsections 3.3 and 3.4).

To illustrate the strategies defined, we report in Section 5 how the deployment and monitoring components, and the SLA service, have been used in a real case study, namely the early version of the Nuro case study, which is detailed in Deliverable D6.3.1 [6] (to be delivered together with the current deliverable at month M12).

Section 6 presents our conclusions of the deliverable. Two annexes provide further details on different aspects of the deliverable, namely on default metrics and user-defined properties (Annex A), and on the SLA service used (Annex B).

1.1 Glossary of Acronyms

Acronym	Definition
PaaS	Platform-as-a-Service
QoS	Quality of Service
QoB	Quality of Business
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
DAM	Deployable Application Model
YAML	YAML Ain't Markup Language
AWS	Amazon Web Services
VM	Virtual Machine
IP	Internet Protocol
REST	Representational State Transfer
UID	Unique Identifier
JSON	Java Script Object Notation
JVM	Java Virtual Machine
XML	Extensible Markup Language
SLO	Service Level Objective

Table 1: Acronyms

2. Multi-cloud deployment process

In this section, we present our approach to the multi-deployment process and the mechanisms to manage, orchestrate and maintain the application modules distributed over the target cloud providers. These are provided according to the specification of the initial architecture described in Deliverable D2.2 (Initial Architecture). To understand the architecture of the deployer and its deployment strategy (defined later), we first introduce some of the main concepts used in the Deployer component, as well as the plan specified to deploy the cloud application.

2.1 Deployer concepts

The SeaClouds Deployer module contains the mechanisms in charge of deploying and managing the cloud applications. Here, we introduce the concepts required to manage an application and its components, which include the entities hierarchy and the sensors, effectors and policies.

2.1.1 Mapping the application modules: Entities

Entity is considered the central concept in a deployment. An entity represents a resource under management, which can be either a *base* entity (an individual machine or a software process) or a logical collection of entities.

Fundamental to the processing model is the capability of entities to be *parents* of other entities (the mechanism by which collections are formed), with each entity having a single parent entity, up to the privileged top-level ***application*** entity, which has no parent.

As entities are software constructions, they can be extended, overridden, and modified. Entities can have events, operations, and processing logic associated with them, and it is through this mechanisms that management is carried out.

For instance, the module of a web application which contains the business logic will be represented using an entity. This describes the application's nature, the technological requirements, and a set of operations (such as deploying the application on an application server). Also, this entity could define dependencies among modules. Thus, the application's components, dependencies and other elements are **normalized** (or **semi-standardized**) which is very useful to deploy and manage the deployed applications in a homogenized way.

This approach has some additional advantages, since we can consider the entities like agnostic elements; they are not dependent of any particular technology and define a flexible methodology to perform heterogeneous deployments of multi-cloud applications. This eases the description of the application's structure (topology specification) and the adaptation of the application's components to the constraints and properties of the cloud providers used, addressing vendor lock-in and portability issues.

The main functionalities of an entity are:

- Provisioning the entity in the given location or locations;
- Holding configuration and state (attributes) for the entity;
- Reporting monitoring data (sensors) about the status of the entity;
- Exposing operations (effectors) that can be performed on the entity; and
- Hosting management policies and tasks related to the entity.

2.1.2 Application and Parent

Every entity has a (unique) ***parent*** entity, which creates and manages it. There is only one exception: *application entities* do not have parents, they are the top-level entities created and managed externally, manually or programmatically.

All entities, including applications, can be the parent of other entities. A "**child**" entity is typically started, configured, and managed by its parent. For example, an application may be the parent of a web cluster, and the cluster can, in turn, be the parent of its web server processes. In the management console, this is represented hierarchically in a tree view (see Figure 2). This hierarchy allows the application's components to be managed according to their deployment dependencies, making easier their description and organization. For example, a JBossServer parent entity could contain several child entities such as deployed applications, runtime-modules.

Please note that this hierarchy contains only the deploying dependencies, and does not contain functional relationships between components. E.g., the functional dependency between a database that is used by a web application will most probably not appear in the deployment hierarchy of entities. Functional relations will be defined through the topology description (Deployable Application Model, DAM).

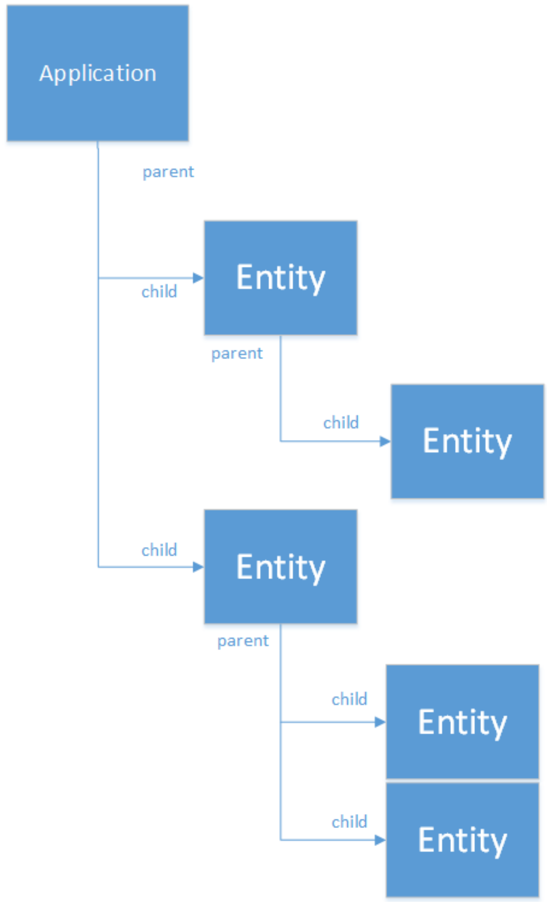

Figure 2. Entity application hierarchy

2.1.3 Sensors, effectors and policies

Sensors (activity information and notifications) and **effectors** (operations that can be invoked on an entity) are defined by entities as static fields on the entity.[1] **Policies** perform the active management, by tracking changes on monitored entities and performing changes on them if necessary. Entities can have zero or more Policy instances attached to them.

- **Sensors** can be updated by their entities or associated tasks, and sensors from an entity can be subscribed to by its parent or other entities to track changes in an entity's activity.

- **Effectors** of an entity can be invoked by an entity's parent remotely (its manager), and the invoker is able to track the execution of that effector. Effectors can be invoked by other entities.

- **Policies** can subscribe to sensors from entities or run periodically. When they execute, they can perform calculations, look up other values, and, if deemed

[1] In future releases, the communication between sensors, policies and effectors will happen through the SeaClouds API.

necessary, invoke effectors or emit sensor values from the entity they are associated with.

In Figure 3, we use a simple example to illustrate the aforementioned concepts. We show a server cluster, a topology component which is represented by an entity. We describe the policy that manages the system's behavior.

This policy is in charge of maintaining the scalability of the system and ensuring the server response time. In this case, the cluster contains several server instances to support a web application.

The entity that represents the cluster has a sensor to indicate the overhead and the response time of the system.

Also, in order to allow the management of the cluster, the entity has an effector to add or remove an instance to its server set.

Thus, the first image shows how the requestTime notification is received by the policy (1), which checks the cluster status, and decides to add a new server to the cluster (2). The third image shows the final cluster (3).

These concepts are used in the Deployable Application Model to describe the application topology and its management and orchestration in a flexible way.

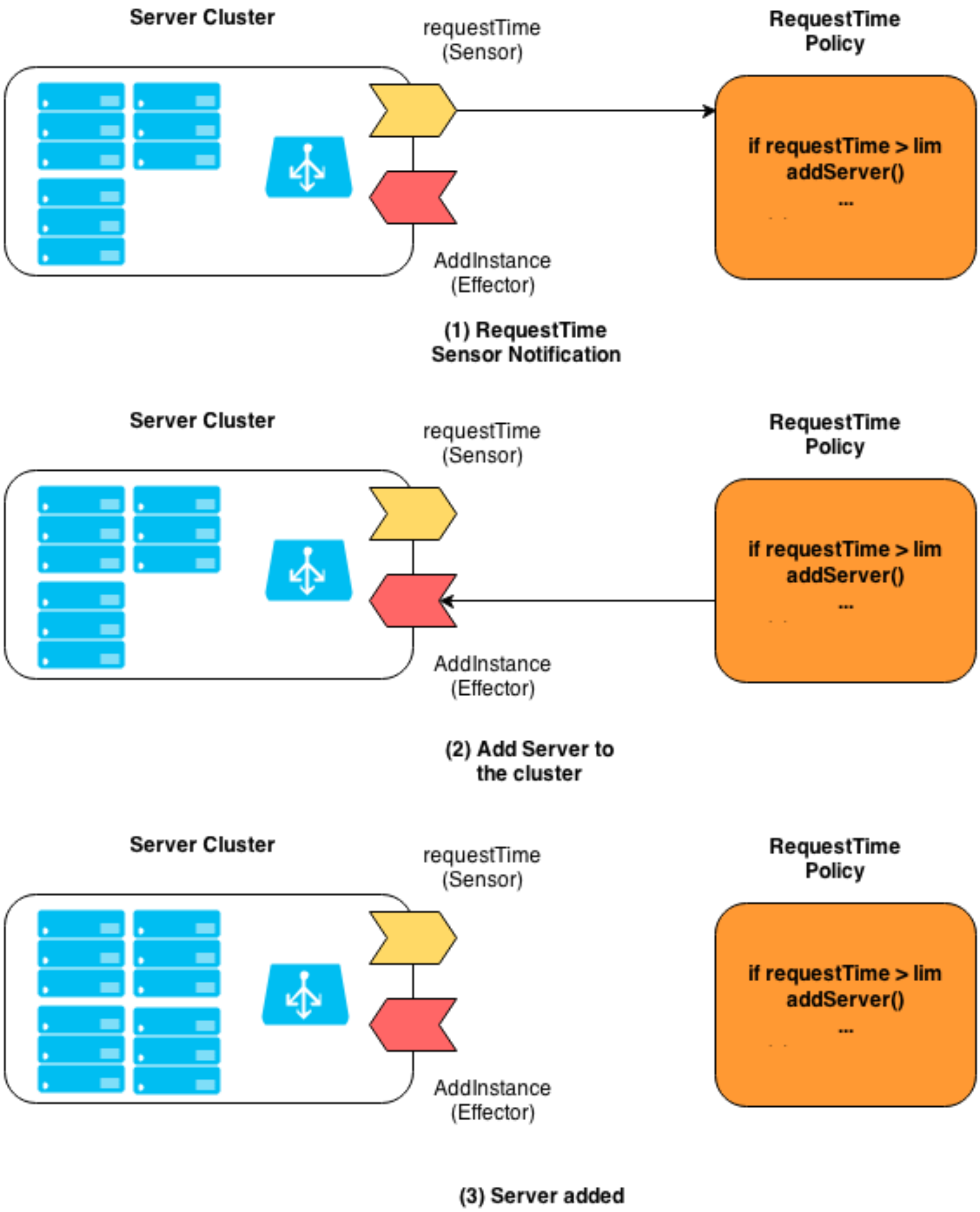

Figure 3. Sensor, Effector and Policy example

2.2 Deployable Application Model analysis (DAM)

In this section, we present the Deployable Application Model (DAM) to specify the structure of an application, the abstract deployment plan. Following the CAMP specification, DAM uses a YAML syntax allowing the application topology to be described. An application is composed of several modules and relationships, which are essential to maintain the application structure, dependencies among modules, and information on how they are related.

Thus, the Deployer receives a DAM, which specifies all the required information on the application to deploy, its distribution and orchestration, and then it follows the instructions to deploy the application using the indicated services. After deployment, the Deployer maintains the management of the application, and monitors its status.

The SeaClouds Initial Architecture [2] describes the Deployer component. Different engines could be used to deploy applications. In our current solution, Brooklyn is being used as Deployer engine, to accomplish the heterogeneous management of the cloud providers. Thu, we define the DAM based on the YAML Blueprint specification of Brooklyn [7].

2.2.1 Outline of the Deployable Application Model

Below, we outline the DAM schema. Within this outline we also provide an overview of the main components needed to describe the application and its management.

name: name of application
location: a location specification element as a string or map.

services: this block contains the entities which compose the application.
- serviceType/service: service reference
name: human readable name of entity.
id: id of entity.
location: location (provider service) where the entity will be deployed (**target cloud provider**).
config: (features and requirements of the entity).
children: a list of child entities specifications which will be configured as children of this entity.
policies: list of policy specifications which add behavior to the entity.
enrichers: enrichers of this entity.
initializers: values needed to configure this entity (key values).

The schema shows the elements needed by the Deployer in order to deploy, monitor and manage the application components over the target providers. We describe each of these elements in more detail in the following sections.

2.2.2 Root Elements

In the first abstraction level, the DAM is composed by **Root Elements**.

Name	Specifies the name of the application in a human readable way. This name is specified by the SeaClouds user.
Location	This key references the **global deployment location** by default. If an

	application entity does not describe its own location key, it will be deployed on the provider described by the global location (see location section for more detailes).
Services	**ServiceSpecification.** This element contains all the application's components description (as explained in Section 2.2.1 entities is the general concept for services, agents, artifacts, etc.).

Table 2. DAM. Root elements

Here we can see an example of the **RootElements**.

```
name: java-cluster-db-example
location: localhost
services:
- serviceType:
           ....
           ServiceSpecification
            ....
```

The following section describes **ServiceSpecification**.

2.2.3 ServiceSpecification

Within the services block, a list of values maps should be supplied, with each map defining a ServiceSpecification. The following map describes the application's components, their features and requirements.

service/serviceType/service_type	Each **ServiceSpecification** should declare the service type of the application components, indicating what type of service is being specified there. For example, a JBossServer. It contains the Entity Class that describes the service. The following formats are supported for defining types: ● io.package.JavaEntityClass ● java:io.package.JavaEntityClass E.g.: entity.webapp.jboss.JBoss7Server describes a JBossServer7 entity.
id	This field defines the identifier

	component. It is the key used by the Deployer to find and manage the application components.
name	This field describes the application name in a human readable way. The Deployer does not use this field to identify the application elements; it uses the id field (see above).
location	This field denotes the provider to be used to deploy the application element. If this field is not included, the application will use the global deployment location described in the **RootElements** (see the Location section below for more details).
config	This element describes the configuration of the entity. It can contain arbitrary values, which depend on the entity specification. E.g, the port of a server entity (see the Config section).
children	This field is a list of ServiceSpecification descriptions that will be configured as children of this entity. An entity could have several entity children, which have to be specified through this field (see the Children section).
policies	With this field we can specify the policies to be used by an entity. E.g., Resizable policy. They are references to policies of entities in the YAML plan, and are are referenced using the Entity Class. E.g., test.policy.TestPolicy. Each policy could be a map that references and initializes policy parameters.
enrichers	Enrichers follow the policy methodology. In this case, an enricher wraps a policy to process the data and generate a processed result. Like policies, enrichers are specified through entity classes, configured through the key value map. E.g., the Delta Enricher converts absolute sensor values into a delta.

initializers	A list of EntityInitializer instances to be constructed and run against the entity. Each initializer is defined as a map, by providing type and config as keys. An EntityInitializer instance allows arbitrary customization of an entity whilst it is being constructed, such as adding dynamic sensors and effectors.

Table 3. DAM. Service specification

2.2.4 Location: The key of the multi-deployment

Location specifications are supplied either for the entire application (at the root element level) or for a specific ServiceSpecification. Locations define the cloud providers used to deploy the application components. Moreover, the providers could have several data centers distributed around of the world. In this case, the location clause allows us to point to a concrete region. The next example shows how AWS is selected to distribute a module over its data center in US East, Northern Virginia (http://docs.aws.amazon.com/general/latest/gr/rande.html) using alternative supported formats for specifying locations.

```
location:
  jclouds:aws-ec2:
    region: us-east-1
    identity: AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
```

or

```
location:
  jclouds:aws-ec2:us-east-1
    identity:AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
```

or

```
location:aws-ec2:us-east-1
    identity:AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
```

As the provider's credentials are necessary to use the cloud services, they will be specified in the Deployer's properties file. If specified in the properties file, the deployer will manage the credentials for the deployment mechanisms. Then, the location only has to describe the service needed for the deployment:

```
location:
  jclouds:aws-ec2:
```

```
    region: us-east-1
```

or, with a more compact notation,

```
location:
  jclouds:aws-ec2:us-east-1
```

or

```
location:aws-ec2:us-east-1
```

DAM allows the specification of multiple locations for an entity:

```
location:localhost
location:named:my_openstack
location:aws-ec2:us-west-1
```

The Deployer engine usually has some predefined providers (by default) to specify the situation when several pre-existing nodes are using. Thus, for example, the called "byon" provider, predefined in the Brooklyn engine could be used in this format:

```
location:
  byon:
    user:root
    privateKeyFile:~/.ssh/key.pem
    hosts:
    -81.95.144.58
    -81.95.144.59
    -seaclouds@159.253.144.139
    -seaclouds@159.253.144.140
```

The following example shows an application that specifies the location in which to deploy.

```
name: sample-single-jboss
description: Single JBoss example
services:
- serviceType: entity.webapp.jboss.JBoss7Server
 name: jboss1
 id: jboss1Server
 location: aws-ec2:us-west-1
```

This location methodology clearly describes the final cloud provider specification and has several advantages for multi-deployment. Each entity describes the target cloud over which it will be deployed through a location clause. Thus, the Deployer hides the final constraints and features of the cloud providers used to distribute the application component. In the next example, we show an application topology that uses several services from different platforms, in this example, HP Cloud (http://www.hpcloud.com/) and AWS (http://aws.amazon.com/).

```
name: multi-deployment-example
services:
- serviceType:entity.webapp.ControlledDynamicWebAppCluster
  name: My Web
  location: aws-ec2:us-west-2
  ...

- serviceType:  entity.database.mysql.MySqlNode
  id: db
  name: My DB
  location: hpcloud-compute:az-1.region-a.geo-1
  ...
```

Please note that the Deployer requires some knowledge to manage and use the services of the target cloud provider in order to adapt the distribution of the application component to the selected providers. In fact, the homogenization mechanisms use the provider interfaces to accomplish this task.

2.2.5 Configuration

All entities have a map that contains configuration information. This can contain arbitrary values, typically keyed under staticConfigKey fields on the Entity subclass. These values are inherited, so setting a configuration value at the application level will make it available in all its sub-entities unless it is overridden.

Configuration is propagated when an application "goes live", so configuration values must be set before this occurs. The configuration values define several entity features, as the implementation artifact path, the http port, the proxy http port, and so on.

```
config:
      key1: value1
      key2: value2
      key3: value3
```

The values are specified like a key map, such as:

```
name: java-cluster-db-example
services:
```

```
- serviceType: entity.webapp.ControlledDynamicWebAppCluster
  name: My Web
  location: aws-ec2:us-west-1
  config:
    wars.root:  example.war
    http.port: 9280+
    proxy.http.port: 9210+
```

Thus, the entity's configuration allows specifying all the features necessary for its configuration. However, we have to keep in mind that the mentioned configure fields contain the values used to configure the real application component that is represented by the entity.

2.2.6 Children

An entity could have some child entities; the parent entity manages their lifecycles. The children of an entity are defined like a ServiceSpecification again, a list of services which will be managed by the parent (see 1.2.3 Service Specification). It is a smart way to specify the children entities, reusing the entity specification list recurrently.

```
-service Type: typeOfService
   name: ...
   location: ...
  children: ServiceSpecification
```

The following example illustrate the children specification:

```
name: java-cluster-db-example
description: Simple Web App
location: aws-ec2:us-west-2
services:
- serviceType: entity.basic.SameServerEntity
  name: parent
 children:
  - serviceType: entity.webapp.ControlledDynamicWebAppCluster
    name: My Web
    ...
  - serviceType: entity.database.mysql.MySqlNode
    id: db
    name: My DB
    ...
```

2.2.7 Services constraint

A DAM defines the features and requirements of the provider over which the application components are to be deployed. These requirements specify the physical properties of the deployment services, such as, memory, storage or operating system, as the following example illustrates:

```
provider: aws
id: m3.medium
memory: 3.75                    #GB
storage: 4                      #GB
os: linux
```

The user specifies these constraints to optimize or limit the execution of the applications, e.g. consume controls and so on. The Deployer allows these constraints to be specified by using the following elements:

```
name: name of application
services:
- serviceType: typeOfService
  location:...
  name: name of component
  provisioning.properties:
    constraint1: value1
    constraint2: value2
    constraint3: value3
    ...
```

provisioning.properties	This field defines, using a key-value map, the features to compose the elements that specify the constraints of the deployment environment. **Note** that the *provisioning.properties* will be ignored if deploying to *localhost* or *byon fixed-IP* machines.
constraintN:valueN	This field is a pair (constraint-value) that specifies a particular constraint or feature and its expected value in the deployment environment E.g.: minRam: 8192 minCores: 4

Table 4. DAM - Service constraints

Table 3

The following example illustrates the use of the fields described above:

```
name:simple-vm
services:
-type:.entity.basic.EmptySoftwareProcess
  name:VM
  provisioning.properties:
    minRam:8192
    minCores:4
    minDisk: 100
```

This will create a VM with the specified parameters in the chosen location. The entity is called "VM", and the hostname and IP address(es) are reported as sensors. Moreover, there are many more provisioning.properties supported here, including:

- A user to be created (if not specified it creates a new user with the same username that SeaClouds is running under).
- A password for the user or a publicKeyFile and privateKeyFile (defaulting to keys in ~/.ssh/id_rsa{.pub} and no password).
- MachineCreateAttempts may be used for unreliable clouds.
- And other options, like imageId, userMetadata and disk and networking options (e.g. autoAssignFloating for private clouds).

2.3 Deployer Architecture

As described in the Initial Architecture of the SeaClouds Platform, Deliverable D2.2 the Deployer is in charge of deploying the application on the cloud providers infrastructure, by executing the instructions contained in the Abstract Deployment Plan (ADP), specified as a concrete plan, the DAM, that comes into the Deployer as an output from the Planner. As a result of the execution of the DAM plan, the Deployer will deploy, manage and monitor the applications, and thus it will generate a live model of the managed applications (the consortium is working currently on this, and the results more stable will be delivered in Deliverable D4.3, Design of the run-time reconfiguration process). The Deployer will have some healing capabilities to repair a managed application that is violating one or more constraints. Therefore, it has to monitor not only the deployment activities but also the running applications and report the failures.

Thus, the Deployer receives a DAM, and then it performs the application components and resources distribution using the target cloud services. Following the architecture described in D2.2, the DAM is taken from the Dashboard, coming as a Planner output.

This interaction between the aforementioned SeaClouds components will be orchestrated using the SeaClouds API (a first specification will be delivered in D4.2). Figure 4 shows an initial approach for the Deployer architecture, describing its modules and the responsibilities. In this figure the high-level layout description of the components that constitute the Deployer is shown. Please note that the API layout specifies the operations that define the set of Deployer functionalities [8].

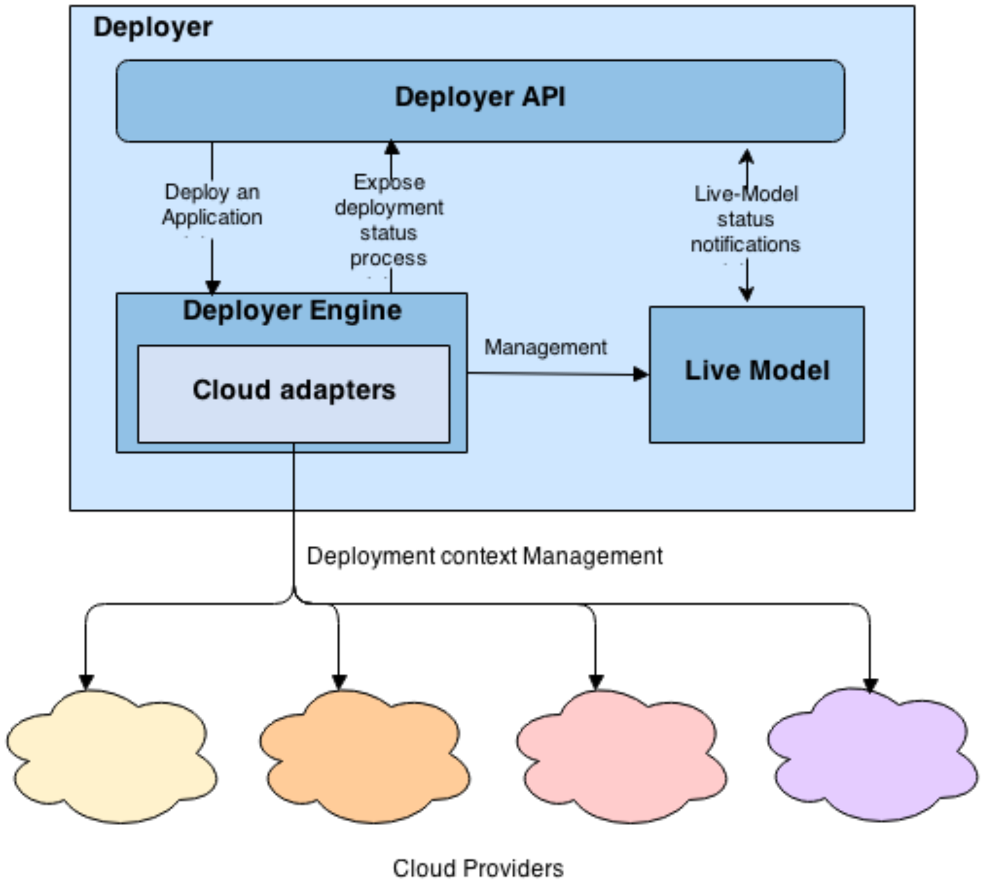

Figure 4. Initial Architecture of the Deployer module.

The main element of our SeaClouds Deployer is the Deployer Engine. The Deployer Engine receives a DAM through its Deployer API and executes the DAM (Deploy an application action in Figure 4). As the Deployer Engine is cloud-agnostic, it is able to deploy applications to the different cloud provider services using multiple Cloud Adapters. During the deployment process, the Deployer Engine monitors the status of the task in order to maintain the knowledge about the distribution. In this case, as aforementioned, we use Brooklyn as the Deployer Engine.

Once the application has been deployed, the Deployer Engine uploads the live model. This model contains the data structure that, in turn, contains a profile about the topology of the real application distribution over the target providers (using the cloud adaptors). The profile includes information about *(i)* relationships and dependencies, and used services, and *(ii)* about the deployment context as IP machines, O.S used, real listener port, metadata of the applications components, etc.

The Monitor and the Dashboard use the live model (see Figure 1) to maintain the status of the application according to the constraints and features that have been described by the user. In addition, the Dashboard checks the live model in order to maintain the graphic representation of the application distribution.

In Figure 5, the early run-time prototype of the Deployer component is depicted (some minimum parts, so far, have been already implemented, and the result of the early prototype will be showed in the first Demo Review). It contains the different elements that will compose the aforementioned component. In addition, several elements used and produced by the Deployer are included, such as the DAM and the *DeployerNotificationEvent*.

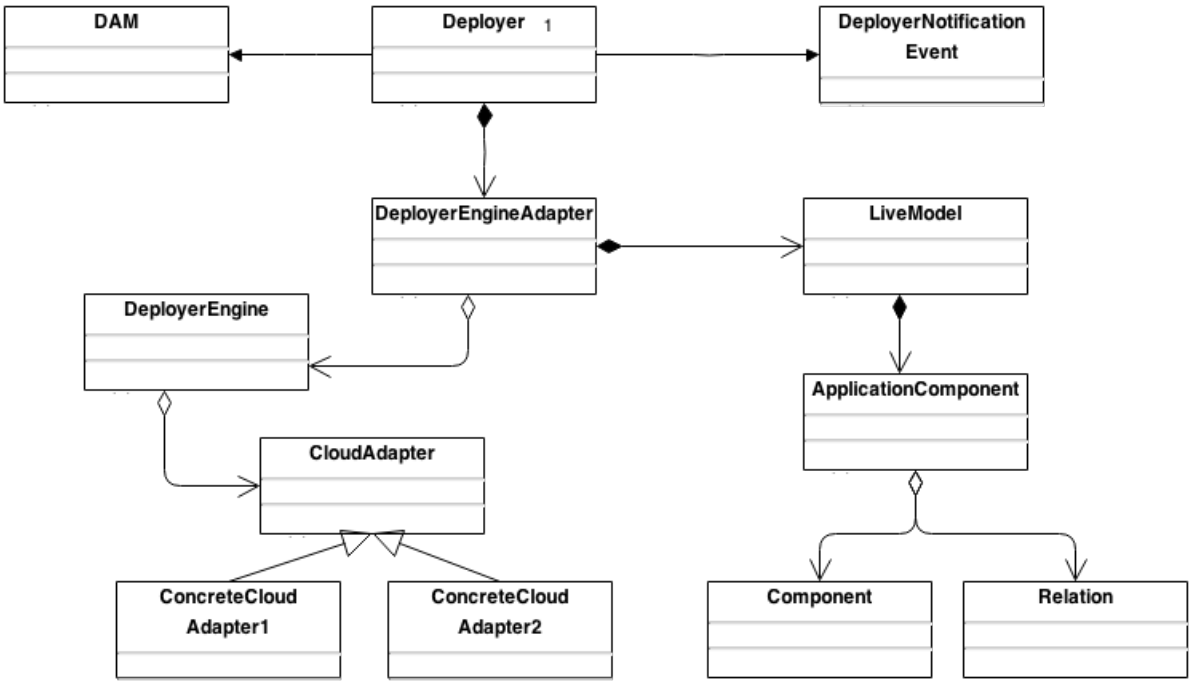

Figure 5. Run-time prototype. Deployer module

It is worth mentioning the current version of SeaClouds relays on Brooklyn as deployer engine, that is in charge of distributing the applications' modules over different locations (clouds, local or remote VM, etc). In this regard, our intention is to add several features to Brooklyn in order to widen its capabilities to fulfill the goals of SeaClouds. So far, Brooklyn only supports Java applications but, although Java is a language widely used in cloud applications, SeaClouds includes as one of its goals support for other languages, for example PHP. In Deliverable D5.4.1, we describe the architecture to add PHP support to Brooklyn. We have based our development on the Brooklyn structure, as we have mentioned above, to take of advantage of current Brooklyn management mechanisms. Therefore, please, refer to that document for the implementation details of the SeaClouds early prototype.

2.3.1 Live Application Model

The live model includes the information present in the Deployable Application Model, plus some important extensions:

- Sensor values
- Policies
- Additional entities (if creation of some entities results in children)
- What effectors (operations) are available, e.g. is migrate supported?

All aspects of the live model are exposed through a REST API where entities can be navigated, and all current information about children, sensor values, and policies can be accessed. By assigning UID's as part of the deployable model, these ID's can be tracked in the live model and live information about the application modules can be accessed, as needed, by the planner or by an operator.

2.3.2 Post-deployment strategy

The Deployer manages the application once it has been deployed (see Figure 6). It should be noted that the migration task is one the most complex features of the project life-cycle. Although it is briefly presented here, it is presently an ongoing work, and it will be presented in more detail in a next deliverable, D4.3, on month M16.

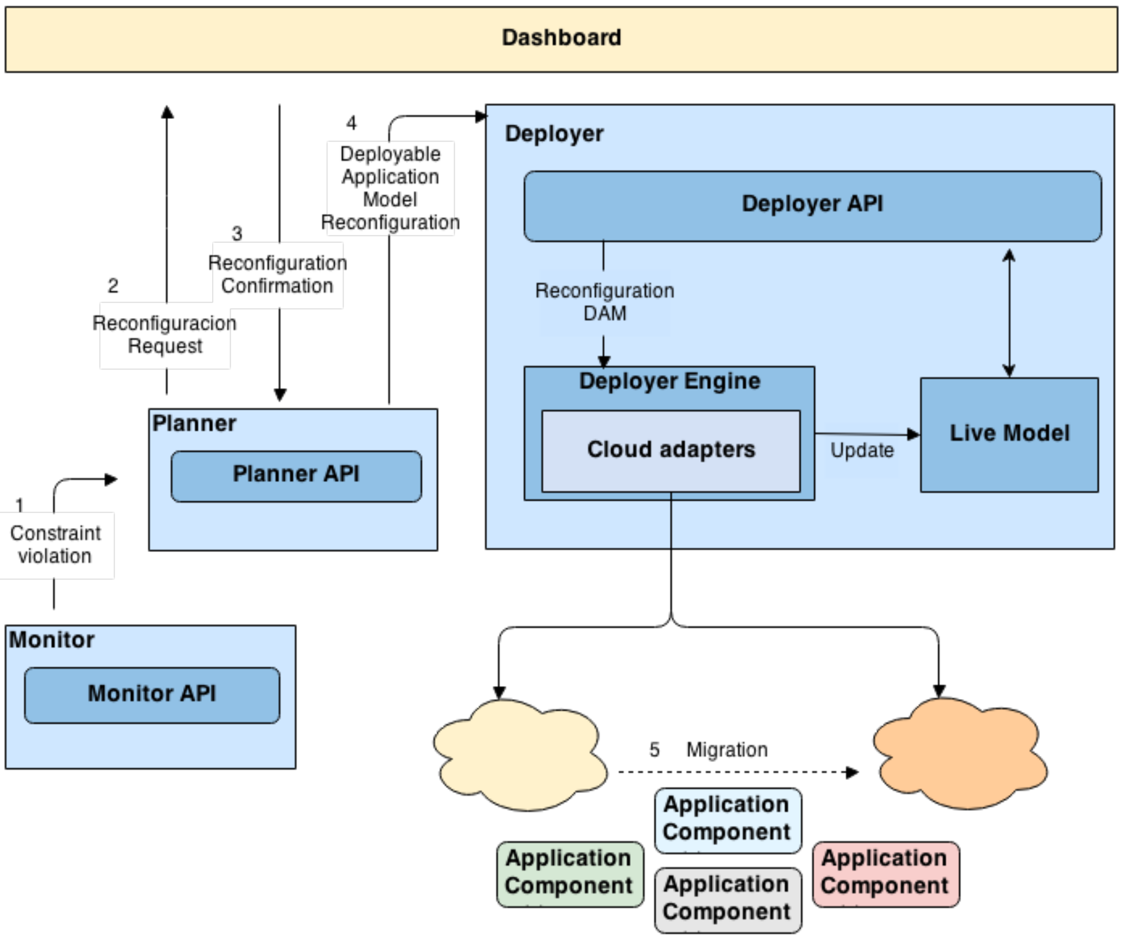

Figure 6. Deployment strategy - Interaction between modules

As Figure 6 shows, the Deployer is in charge of migrating the modules over the different cloud providers and it contains the necessary mechanisms to manage the target cloud services for distributing the application component. When an application is deployed, the live model is built and the Monitor module begins to check its status using the API (as it is explained in the Monitor Section, Section 3).

When a constraint is violated, the Monitor triggers an event to manage this issue; it could produce (among others) a policy reconfiguration (e.g. add a server to minimize a response time) or a migration of a component. In the second case, the Monitor sends a notification to the Planner to replan (Deployable Application Model Configuration) the application distribution. The Planner gets the current application distribution status from the Live Model using the Deployer API. The Planner uses that information to generate a replan that contains the necessary information to migrate the application components over different cloud providers. Please note that the aforementioned components redistribution does not modify the application components dependencies, so it does not affect to the application topology.

Then, the Planner sends the replan to the Dashboard, for the user to validate it. Afterwards, Deployer Engine receives the replan, again through the Deployer API. Following, the replanning process migrates (relocates) the necessary application components to mitigate the constraint violation that triggered this chain of events.

3. Monitoring process

In this section, we propose an initial approach for the Monitoring architecture, describing its elements and the responsibilities assigned to the Monitor component.

3.1 Monitoring Architecture

Following the SeaClouds Initial Architecture presented in Figure 1, the Monitor component is in charge of retrieving the monitoring data from deployed applications, making it available to the rest of the components in SeaClouds. Also, it is responsible for the control and enforcement of QoS properties and SLA, as well as forwarding violations of these properties to the interested subscribed modules. The retrieved data and related information must be completely available through the Monitor API. In Figure 7 we can see the main layout for the components that constitute the Monitor.

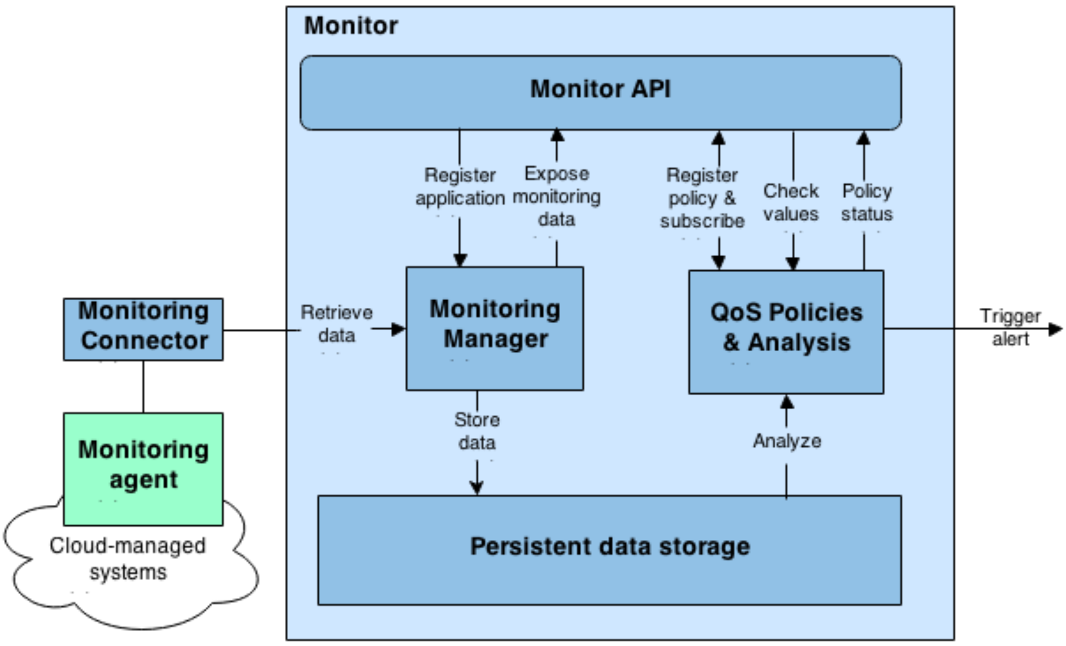

Figure 7. Monitoring architecture diagram

As shown in Figure 7, the Monitor functionality is focused on the Monitoring Manager, which acts as a registry for new applications deployed and it also has the necessary mechanisms to retrieve the data generated from the Monitoring agents, to make it publicly available through the Monitor API and to store it for later usage and analysis.

Monitoring data and QoS properties can be generated from the different layers that compose the deployed applications, composing a wide family of metrics that shapes its current state (in Annex A, a list of default metrics and user-defined properties is given). Moreover, due to the different nature of the software layers found at applications, many monitoring tools and infrastructure can be found. This architecture presents an abstraction layer over those tools, using several Monitoring Connectors as translation and routing mechanisms, in order to adapt metrics and properties from the different sources (we denote them as Monitoring Agents) into the internal Monitoring model.

The Monitor is also responsible for the creation and management of analysis mechanisms, such as rules or policies, that operate over the data that are being retrieved. It will allow external agents to subscribe to notifications on rules violations.

3.2 Monitoring Manager

The Monitoring Manager holds the necessary logic to coordinate the elements inside the Monitor component. The main task that the Monitoring Manager needs to perform is to route the monitoring data, such as sensor values, from the deployed applications in SeaClouds to the Monitor API.

Also, in order to provide records of previous information, the Monitoring Manager is connected to a Persistent Data Storage that allows it to track the evolution of the sensors' values.

3.3 Monitoring Connector

The Monitoring Connector translates the proprietary application monitoring service into SeaClouds standard metrics. This allows us to use different ways to connect to a data source. By using different connectors we can monitor from large application servers to a single Apache instance (see Figure 8).

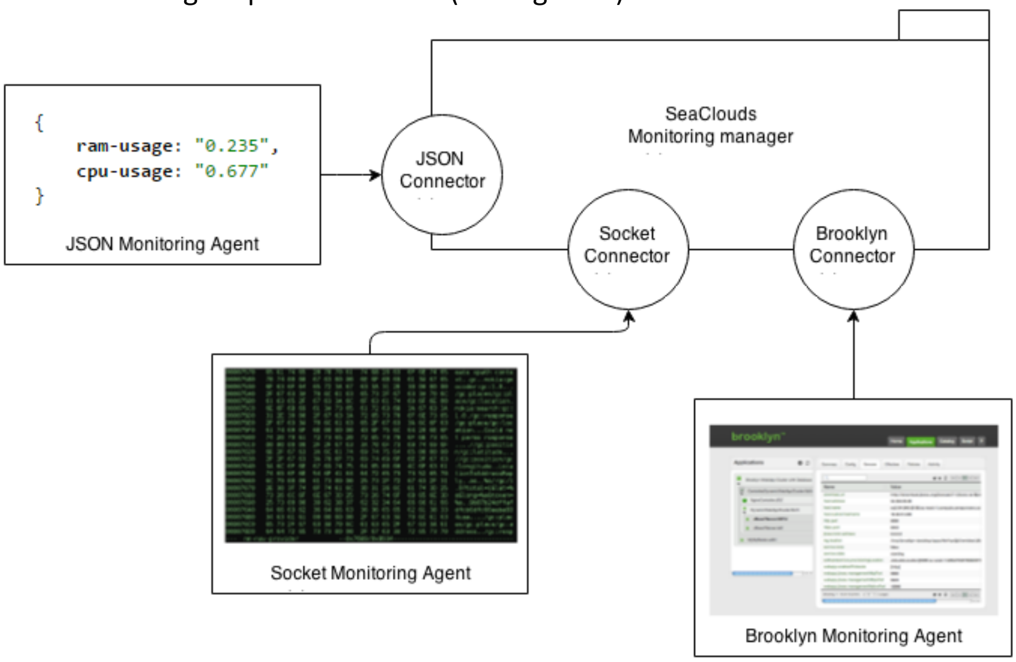

Figure 8. Monitoring Connector Example

3.4 Monitoring Agents

The Monitoring Agents are the main providers of raw sensor values, like the average CPU usage, or the average response time (Figure 9). The data parsing into SeaClouds objects will be done by a specific Monitoring Connector that will serve as a routing point that allows including this information as available data in the Monitor module.


```
{
  - database1: {
        status: "Connected",
        connection_time: "0.00018191337585449218750",
        inser_log_time: "0.01504206657409667968750",
        analytics_time: "0.00101685523986816406250",
        update_log_time: "0.00021004676818847656250"
    },
  - request_analytics: {
      - minute: {
            seconds: "60",
            avg_response_time: "1.376749992",
            max_response_time: "2.75350",
            request_count: "2",
            player_count: "1"
        },
      - hour: {
            seconds: "3600",
            avg_response_time: "1.376749992",
            max_response_time: "2.75350",
            request_count: "2",
            player_count: "1"
        },
```


Figure 9. Example of monitoring data provided by a Monitoring Agent

Then, the concept of agents is an abstraction layer for all the providers of any kind of monitoring data. Brooklyn will be one of them, having a Brooklyn-connector to route the Brooklyn sensors into SeaClouds. Another example of agent could be also a single web service which returns data directly into SeaClouds.

In practice, the Monitoring Agents will be composed of different types of service endpoints that supply any kind of information coming from every level of the application deployed, from low level QoS properties, such as "average disk usage" to higher level performance metrics or even specific properties like database-related "insert query response time". This approach creates homogeneity over the diverse nature of properties, simplifying any analysis task related to them.

It is worth mentioning, no more explanations are provided in this document about the Monitoring mechanism implemented, since they are detailed in Deliverable D5.4.1. That document describes the work done for the main Monitoring Agent that is considered at the point of the first prototype, which is mainly implemented by Brooklyn, acting as a management entity over the deployed applications. The objective is to provide Brooklyn with the ability to create user-defined sensors that will make HTTP requests to a JSON-based service that will provide monitoring information. In order to implement this functionality, some Brooklyn mechanisms have been extended and others have been generated. The contents of this contribution and the different mechanisms in which it relies on are explained in D5.4.1. There, are described the mechanisms used to retrieve the information generated in different kind of services that generate data, usually related to application metrics monitoring. We could distinguish two big approaches on maintaining and updating this information: notifications and feed mechanisms. While the first one relies on a subscribe and notify mechanism working on events changes, the second one requires a periodic request or polling over some service. According to the polling approach that Brooklyn, as a Monitoring Agent, the different existing mechanisms, also are called Data feeds.

3.5 QoS Policies & Analysis

This component will be in charge of managing the monitored data by performing some minor analysis and creating mechanisms (such as policies or rules) that will perform QoS-level evaluations. External components (e.g. Planner module or SLA service) will be able to create policies and to subscribe to events generated by them.

Policies will perform a simple and direct analysis on the actual state of the system, by constantly checking the value of necessary metrics. This kind of analysis will consist in the evaluation of a simple property (e.g., metric_value {operator} threshold) that is built over the monitored metrics. More advanced or complex analysis can be performed by using the events generated by policies or by querying the Persistent Data Storage system.

3.6 Persistent Data Storage

The Persistent Data Storate module defines the Persistence Layer used by the Monitor module to provide the retrieved information within certain periods of time. Stored data can be used by the QoS Analysis element to perform advanced analysis over the data stored on it and also by external components, such as SLA Service, through the Monitor API.

While this layer will keep track of the information retrieved by the Monitoring Agents, the need of storing a complete evolution of all the metrics can cause several problems related to excessive information storage. In any case, the Persistent Data Storage will maintain a record of the events generated by the QoS Policies and general information about the Monitor module as well, since this information could be used by the SLA Service to perform necessary QoB properties enforcement.

3.7 Monitoring Strategy

Figure 10 shows an overall diagram of the interaction between the Monitor and the rest of the modules in SeaClouds.

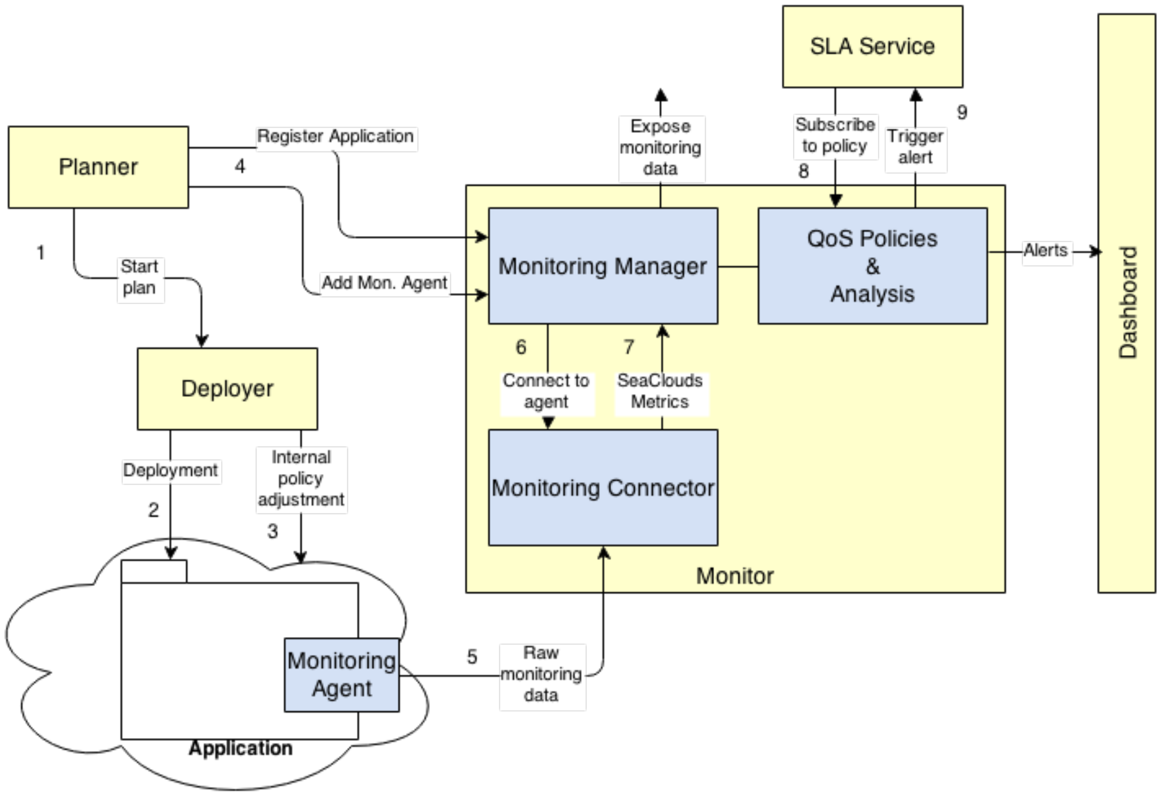

Figure 10. Monitoring strategy - Interaction between modules

The registration of a new application into the SeaClouds context is one of the most useful processes to exemplify how monitoring data are managed among components.

The monitoring process starts once the target application enters the deploying phase. At this point, the Deployer component will notify the Monitoring Manager, through the Monitor API, registering the new application in the Monitoring Manager. The registration process will require information about the Monitoring Agent in use, so that metrics can be collected and exposed to the rest of the SeaClouds modules by using Monitor API.

In Figure 11, an initial class diagram for a run-time prototype is described. It shows how the relations among components are established, according to the monitoring strategies explained in the previous section, including the element introduced at the initial architecture, such as *MonitoringAgents*, *MonitoringConnectors* and *MonitoringManager*. In the next deliverables this information will be completed with a more detailed description about the working implementation. Note that, as aforementioned, in Deliverable D5.4.1, we present the HttpSensor support for the Monitoring Agent used in our first prototype, where we based our Monitoring extending the mechanisms used in Brooklyn.

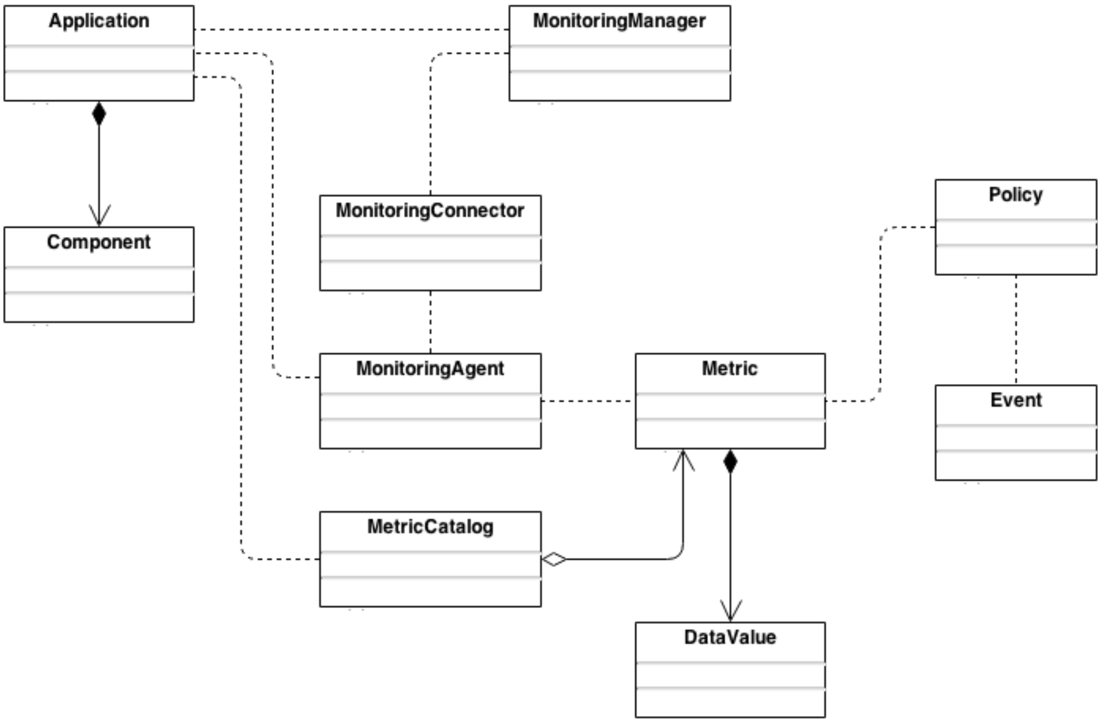

Figure 11. Run-time prototype. Monitor module

4. SLA Service

In the context of the SeaClouds project, the SLA Service enables the Service Level Agreements (SLA) management of business-oriented policies. Firstly, it represents the component responsible for generating the formal documentation describing electronic agreements between the parties involved in the process: customers, application providers and cloud providers.

Moreover, at runtime, the SLA Service is in charge of supervising that all the agreements (SLA guarantees) are respected by assessing the business penalties (QoB: Quality of Business) associated with the fulfillment of these non-functional properties (some examples are provided below).

The SLA policies are based on WS-Agreement specification, which defines schemas for SLA Templates and SLA Agreements (look at Annex B).

However, in the scope of this project, SLAs do not aim at representing a contractual relationship between the customers consuming virtualized cloud resources and the vendors that provide them. SLAs describe the service that is delivered, the functional and non-functional properties of the resource, and the duties of each party involved.

Figure 12 gives a detailed overview of the software components belong to the SLA Service and how they are related to other SeaClouds services.

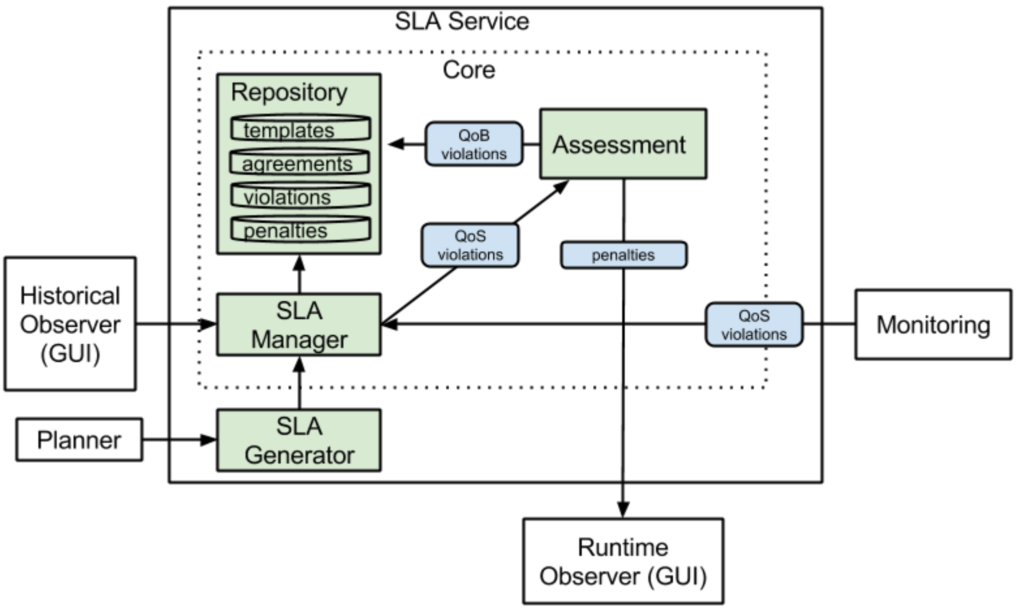

Figure 12. SLA Service architecture diagram

SLA Generator is a RESTful WebService that performs the automatic generation of SLA agreements (compliant with the WS-Agreement specification) on behalf of Cloud providers, interacting with the SeaClouds planner component. The agreement will rely on the description of offerings and the QoB policies specified by the application provider during the cloud resources matchmaking phase as well as the resource capabilities advertised by the cloud provider.

The ***SLA Manager*** is a RESTful WebService as well. This component is mainly in charge of virtualizing the persistent layer as a WebService, allowing it to manage (remotely) SLA Templates and SLA agreements, storing them in a repository, retrieving and updating them. It also receives the QoS violations from the monitoring service and forwards them to the Assessment component.

The implementation of the interfaces is suggested by WS-Agreement and will be detailed in the Deliverable D4.2.

The SLA Manager (see Figure 13) supports the following features:

- To create, update, delete and obtain Agreements.
- To create, update, delete and obtain Templates.
- To obtain, start and stop Enforcements.
- To create, update, delete and obtain Providers.
- To obtain Violations.

On the other hand, it provides the logic to filter information according to some parameters, such as:

- Agreements that can be filtered by the specific provider.
- Templates that can be filtered by the specific provider or resource.
- Violations that can be filtered by agreement-id, service name, range of dates (from, until) or a provider.

SLA Repository represents the SLA persistent layer. It provides useful views to manage the persistence of SLA Templates, agreements and their relationship. Moreover, it provides a place where QoB Violations are stored and retrieved together with the relative business penalties. The following picture depicts the first early version of the SLA Repository data model, also more explained in detail in Deliverable D5.4.1.

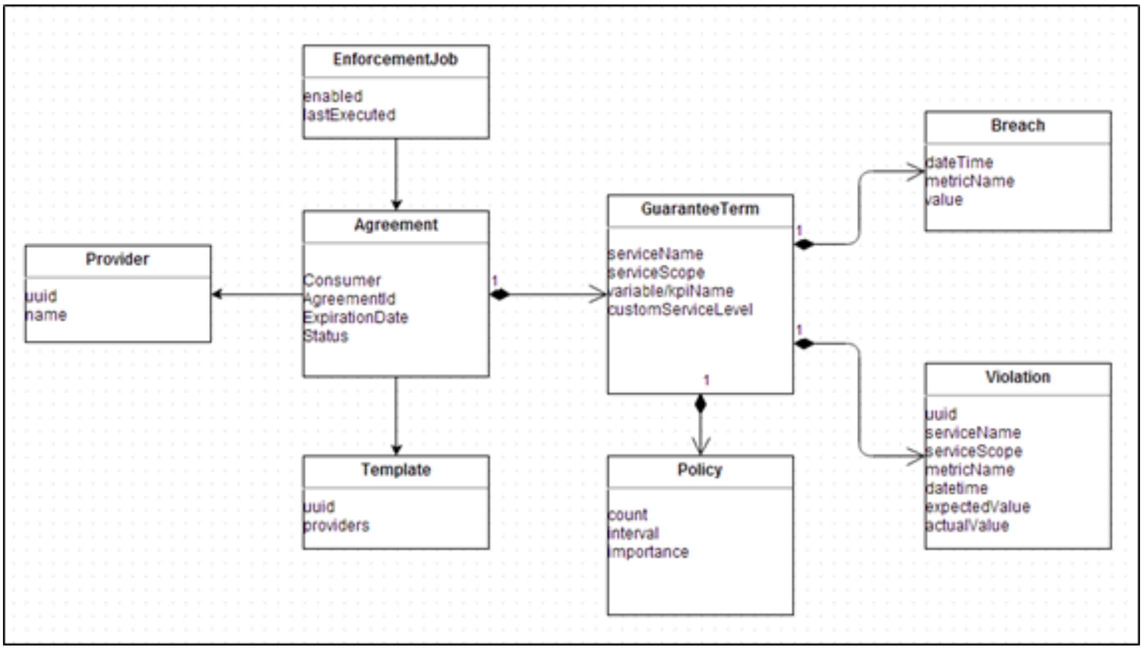

Figure 13. Support of the SLA Manager

Finally, the ***Assessment*** component is a library component in charge of supervising that the QoB policies specified in the agreement are respected. It processes all measurements related to the execution of a business application sent by the monitoring service. It checks whether the measurements are within the thresholds established in agreement for QoB metrics. Whenever the execution of the business application does not satisfy these conditions, the component makes use of policies to take appropriate recovery actions. Moreover it may notify any observer (like an accounting component) of raised penalties.

5. Nuro (early) Case: SeaClouds in action at Run-time

In order to illustrate SeaClouds at run-time (deployer, monitoring, and SLA), we use an early version of the Nuro Case study, more detailed in Deliverable 6.3.1. The Nuro application is structured into a PHP module for the business logic, and a MySQL module as database (i.e., they are entities), as shown in Figure 14.

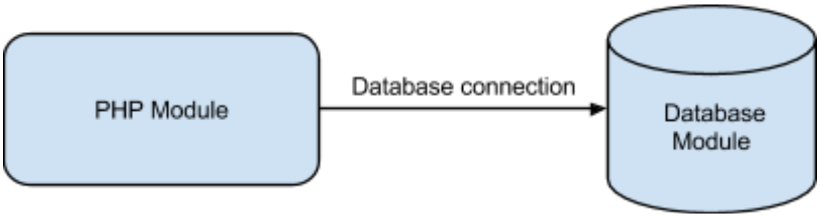

Figure 14. Nuro case application modules

A possible deployment of the modules of the Nuro case application is presented in Figure 15.

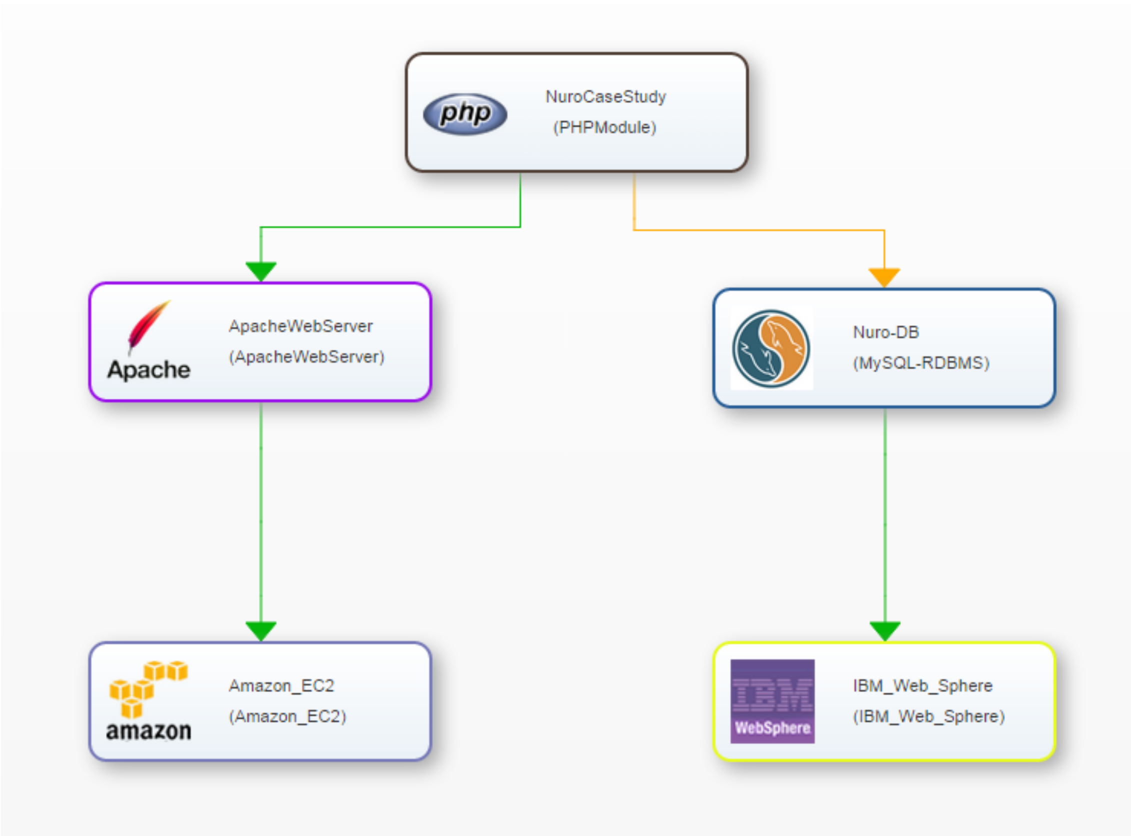

Figure 15. Representation of the Concrete Deployment of the modules of the Nuro (early) Case Study

Once the application is described through the Application Model, it is sent to the Planner. The Planner uses the data obtained by the Discoverer component to select the target cloud services according to the defined metrics (for the Nuro Case, see Tables 4 and 5 below) and other application requirements. Then, after generates the Abstract Deployment Plan (ADP), the Deployable Application Model (DAM) is composed, which contains the deployment instructions (see Deliverable D3.1, where a more specific explanation about the generation of the ADP is given). Below, a possible DAM example is shown:

```
name:PHPHelloWorld
services:
-serviceType:entity.webapp.apache.ApacheServer
  name:ApacheServer
  location:aws-ec2:us-west-2
  brooklyn.config:
    http_port:80
    app_git_repo_url:
        https://seaclDem:seaclouds@bitbucket.org/seaclDem/nurocasestudyphp5-5.git
    db_connection_file_config: config/config.php
    db_connection_config_params:
  g_DatabaseHost:formatString("%s",component("db").attributeWhenReady("host.address")
)
     g_DatabasePort:component("db").attributeWhenReady("mysql.port")
     g_DatabasePassword:component("db").attributeWhenReady("mysql.password ")
     g_DatabaseUser: root
  brooklyn.initializers:
  - type: brooklyn.entity.software.http.HttpRequestSensor
    brooklyn.config:
      name: nuro.analytics_time
      uri:formatString("%s/sensor.php",component("apache").attributeWhenReady("
 host.ad dress"))
      jsonPath: $.database1.analytics_time
      targetType: double
-serviceType:entity.database.mysql.MySqlNode
  id:db
  name:MyDB
  location: hpcloud-compute:az-1.region-a.geo-1
```

This DAM describes a PHP entity and a MySQL database, both deployed and configured by the Deployer Engine, which distributes the applications modules over the cloud services which are pointed in the DAM. In this case, the web application is deployed using AWS service (aws-ec2:us-west-2) and the database is built over HP service (hpcloud-compute:az-1.region-a.geo-1).

Once the application is distributed, the Deployer sends an event to the Monitor. This event triggers a synchronization process between the Deployer and the Monitor, to carry out the application monitorization. Figure 16 shows a brief example of the process described so far.

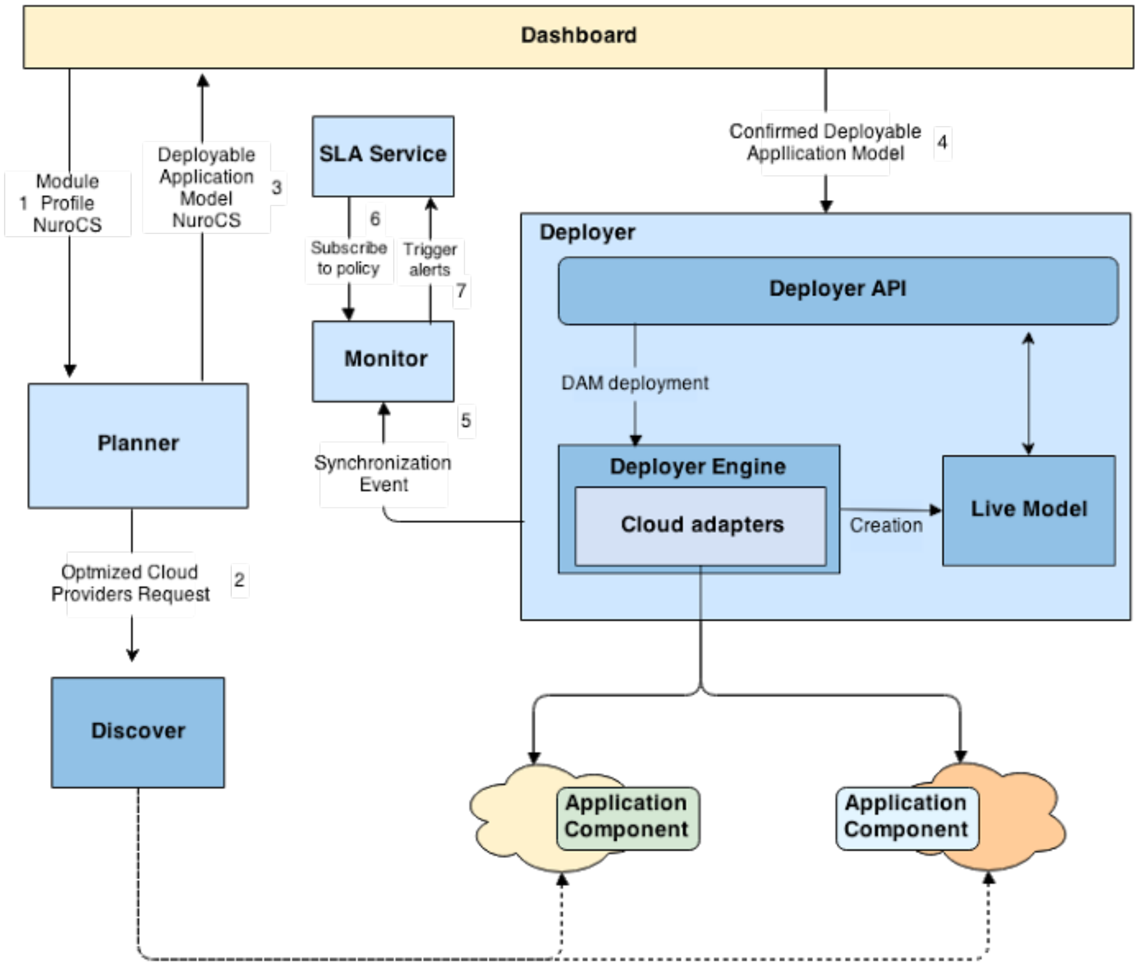

Figure 16. Nuro (early) Case Study deployment diagram

Nuro's monitoring requirements include several application-level and database metrics, such as execution times for different kinds of simple and complex queries (see Figure 17). In this flow, the SLAs will be connected to the Monitor (as depicted in Figure 10), checking the QoS violations, and describing the service that is delivered, the functional and non-functional properties of the resource, and the duties of each party involved.

The main source of monitoring data is implemented by a PHP service that exposes this information as a JSON document. The mechanism used to retrieve this information and to make it available to the rest of SeaClouds components is an element of Brooklyn (Brooklyn Monitoring Agent, as was already mentioned in the Monitoring Agents section), by making use of *HttpFeeds* and the necessary sensors that will constantly update the value of monitoring data generated by the *Nuro Monitoring service.*

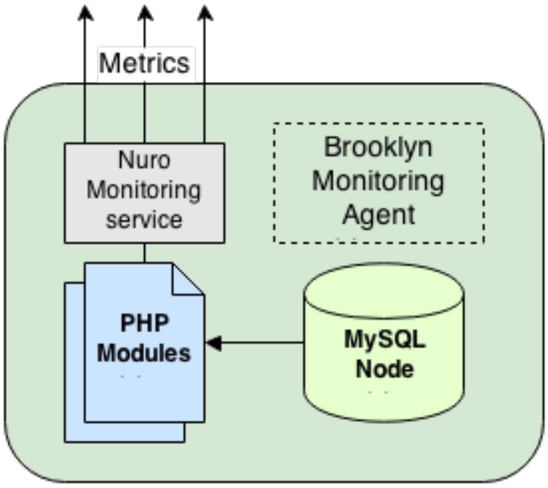

Figure 17. Nuro case-study monitoring strategy

The metrics generated by *Nuro Monitoring service* are the following:

Database		
Name	**Type**	**Description**
status	String	Database connection status.
connection_time	Double	Elapsed connection time to database.
insert_log_time	Double	Elapsed time inserting log to database.
analytics_time	Double	Elapsed time on generating the analytics queries.
update_log_time	Double	Elapsed time updating log to database.

Table 5. Nuro metrics. Database

Request analytics		
Name	**Type**	**Description**
avg_run_time	Double	Average PHP run time.
max_run_time	Double	Maximum PHP run time.
avg_request_time	Double	Average request duration time.
max_request_time	Double	Maximum request duration time.
request_count	Integer	Total number of requests.
player_count	Integer	Total number of players.

Table 6. Nuro metrics. Request analytics

("Request analytics" metrics are generated for different periods of time: every minute, hour and day.)

At the SLA level, we propose two levels (or layers) of agreements:

1. Application Provider - Cloud Provider layer
2. User - Application Provider layer

Application Provider - CloudProvider layer

In this level, the provider is the cloud provider, and the consumer is Nuro. This means that Nuro expects the cloud providers to offer some guaranteed service.

There will be one agreement per module. In Nuro's case study, two agreements will be considered (one for the PHP module, and another one for the MySQL module).

The QoS in this kind of agreements should only refer to metrics that are independent from the other modules. E.g. runtime(php) = runtime(total) - runtime(mysql)

Example of agreement, with the following desired QoS constraints (constraints to be fulfilled by the CloudProvider):

- runtime(php) < 1000ms
- availability(php, day) > 0.99
- requests_minute_worker < 400

The Monitoring Manager evaluates all these constraints.

A proposed agreement based on WS-Agreement specification is shown below:


```
Agreement:
  AgreementId: nuro-php-0001
  Name: Agreement for Nuro Case Study
  Context:
    ServiceProvider: AmazonAWS
    Customer: NURO
  Terms:
    - ServiceReference: { Name: nuro, ServiceName: default, Value: "http://nuro.seaclouds.eu" }
    - ServiceProperties:
        Name: qos_properties
        ServiceName: default
        Variables:
          - { Name: AVG_RUN_TIME, Metric: "xs:double", Location: /sensor/avg_run_time }
          - { Name: DAY_AVAIL, Metric: "xs:double", Location: /sensor/day_avail }
          - { Name: REQUESTS_MINUTE_WORKER, Metric: "xs:double", Location:
/sensor/requests_minute_worker }

    - GuaranteeTerm:
        Name: "AvgRunTime"
        ServiceScope: { ServiceName: default, Value: /game }
        ServiceLevelObjective:
          Target:
            Constraint: AVG_RUN_TIME < 1000

  - GuaranteeTerm:
        Name: "Availability"
        ServiceScope: { ServiceName: default, Value: / }
        ServiceLevelObjective:
          Target:
            Constraint: DAY_AVAIL < 1000

  - GuaranteeTerm:
        Name: "Requests"
        ServiceScope: { ServiceName: default, Value: / }
        ServiceLevelObjective:
          Target:
            Constraint: REQUESTS_MINUTE_WORKER < 400
```

The main purpose of the SLA Service is to handle QoB. Since it is not possible to impose penalties to cloud providers, currenlty this layer only serves as a store of QoS violations.

The violation of QoS constraints should result in recovery actions by the Deployer to correct the situation.

User - Application Provider layer
In this level, the provider is Nuro, and the consumer is any Nuro's customer. This means that Nuro offers the SLA agreement to their customers.

There will be one agreement per full application. The constraints in this kind of agreement should only refer to observable metrics by the end user. For example, in Nuro's case study, request times or availabilities.

The violations of these high-level constraints can be used in QoB rules.

The application provider can use the agreement of the frontend component in Application Provider - Cloud Provider level as a template for generating an agreement at this level, and modify the constraints if desired.

A QoB rule is defined like this:

- A constraint over a metric provided by sensors (e.g. runtime < 2000ms). A non-fulfilled constraint is considered a breach.
- A time window where the number of breaches must be below a threshold (e.g. 5 breaches/day). If this time window is violated, a QoB violation is raised, and the business values take place (discount, migrations, downrating, other recovery actions…)

For example, considering the previous QoS rules whose metrics were suitable for this level (constraints 1 and 2), the following QoB rules can be defined:

- QoB1
  - runtime(request) < 2000ms
  - 5 violations/day
  - 10% discount
- QoB2
  - availability(day) > 0.99
  - 10 violations/month
  - migration

Using the previously defined QoB constraints, a proposed agreement is:


```
Agreement:
  AgreementId: nurocasestudy0001
  Name: Agreement for Nuro Case Study
  Context:
    ServiceProvider: NURO
    Customer: any
  Terms:
    - ServiceReference: { Name: nuro, ServiceName: default, Value: "http://nuro.seaclouds.eu" }
    - ServiceProperties:
```

```
    Name: qos_properties
    ServiceName: default
    Variables:
      - { Name: AVG_REQUEST_TIME, Metric: "xs:double", Location: /sensor/avg_request_time }
      - { Name: DAY_AVAIL, Metric: "xs:double", Location: /sensor/day_avail}

- GuaranteeTerm:
    Name: "AvgRequestTime"
    ServiceScope: { ServiceName: default, Value: /game }
    ServiceLevelObjective:
      Target:
        Constraint: AVG_REQUEST_TIME < 2000
    BusinessValueList:
      CustomBusinessValue:
        Policy: { Count: 5, Duration: P1D }
        Action: 10% DISCOUNT

- GuaranteeTerm:
    Name: "Availability"
    ServiceScope:
      ServiceName: default
      Value: /
    ServiceLevelObjective:
      Target:
        Constraint: DAY_AVAIL > 0.99
    BusinessValueList:
      CustomBusinessValue:
        Policy: { Count: 10, Duration: P1M }
        Action: MIGRATION
```

6. Conclusions

In this deliverable we have presented the multi-deployment and monitoring strategies, as well as the SLA service, as three of the main components of the SeaClouds architecture.

Along the document we have described how the modules of the cloud application are deployed in heterogeneous clouds, and how the monitoring service and the SLA service are initialized and executed. A real use case, the Nuro (early) case is used to illustrate the application of the strategies defined in this document. Also some Annexes with some details on metrics and properties, as well related to the SLA agreement have been presented at the end of the document. About the early prototype, along the document, apart from present some design details, we have also referred to Deliverable D5.4.1 where the implementations details are provided.

Annexes

A. Default metrics and user-defined properties

The definition of properties and metrics gathered by monitoring agents may differ depending on the tools and methods used for the generation of the properties and also on the different nature of the agents. This problem can cause duplication and inconsistency of rules and properties built over monitoring data.

In this section we define a set of metrics, based on general usage of different platforms and applications, covering a wide range of commonly used properties. Monitoring connectors will be in charge of translating and adapting the metrics provided by different agents into SeaClouds' own set of metrics when possible, and creating new ones when these are not possible to adapt or the metrics are defined by the user itself at application level.

We will classify these metrics according to the nature of systems being monitored, going from low level layers to concrete application-based metrics.

Basic platform metrics

Java-based platforms (VM metrics)		
Name	**Type**	**Description**
Used heap memory	Long	Current heap size (bytes)
Init heap memory	Long	Initial heap size (bytes)
Committed heap memory	Long	Committed heap size (bytes)
Max heap memory	Long	Max heap size (bytes)
Non heap memory usage	Long	Current non-heap size (bytes)
Current thread count	Integer	Current number of threads
Peak thread count	Integer	Peak number of threads
Start time	Long	Start time of Java process
Up time	Long	Uptime of Java process (millis, elapsed since start)
Process CPU Time	Double	Process CPU time (total millis since start)
Process CPU Time Fraction last	Double	Fraction of CPU time used, reported by JVM (percentage, last datapoint)
Process CPU Time Fraction window	Double	Fraction of CPU time used, reported by JVM (percentage, over time window)
Available processors	Integer	Number of available processors

System load average	Double	System load average for the last minute[2].
Total physical memory size	Long	Total physical memory size
Free physical memory size	Long	Free physical memory size

Table 7. Java-based platforms metrics.

Machine level		
Name	**Type**	**Description**
Uptime	Duration	Elapsed time since machine started.
Load average	Double	System load average for the last minute.
CPU usage	Double	Current CPU load.
Free memory	Long	Current free memory.
Used memory	Long	Current used memory.
Total memory	Long	Current total memory.

Table 8. Machine level metrics

Server pool		
Name	**Type**	**Description**
Available count	Integer	The number of locations in the pool that are unused
Claimed count	Integer	The number of locations in the pool that are in use

Table 9. Server pool metrics.

Database metrics

Database node		
Name	**Type**	**Description**
Read pending	Long	Current pending ReadStage tasks
Read active	Integer	Current active ReadStage tasks
Read completed	Long	Total completed ReadStage tasks
Write pending	Long	Current pending MutationStage tasks
Write active	Integer	Current active MutationStage tasks
Write completed	Long	Total completed MutationStage tasks
Thrift port latency	Long	Latency for thrift port connection (ms) or null if down
Thrift port latency in window	Double	Latency for thrift port (ms, averaged over time

[2] The system load average is the sum of the number of runnable entities queued to the available processors and the number of runnable entities running on the available processors averaged over a period of time.

		window)
Reads per second last	Double	Reads/sec (last datapoint)
Writes per second last	Double	Writes/sec (over time window)
Reads per second in window	Double	Reads/sec (over time window)
Writes per second in window	Double	Writes/sec (over time window)

Table 10. Database metrics.

Web application servers metrics

Web Server		
Name	**Type**	**Description**
Total accesses	Long	Number of accesses
Total KBytes	Long	Total traffic in KBytes
CPU Load	Double	Percentage CPU load
Uptime	Long	Total up time (in seconds)
Request per second	Double	Requests per second
Bytes per secoond	Double	Bytes processed per second
Bytes per request	Double	Bytes per request
Busy workers	Integer	Number of busy workers
Idle workers	Integer	Number of idle workers

Table 11. Web server metrics.

Web Application Server		
Name	**Type**	**Description**
Request count	Integer	Request count
Error count	Integer	Request errors
Total processing time	Integer	Total processing time, reported by webserver (millis)
Max processing time	Integer	Max processing time for any single request, reported by webserver (millis)
Processing time fraction last	Double	Fraction of time spent processing, reported by webserver (percentage, last datapoint)
Processing time fraction in window	Double	Fraction of time spent processing, reported by webserver (percentage, over time window)
Bytes received	Long	Total bytes received by the webserver
Bytes sent	Long	Total bytes sent by the webserver
Request per second last	Double	Reqs/sec (last datapoint)
Request per second in window	Double	Reqs/sec (over time window)

Table 12. Web application server metrics.

Web Application Server Cluster		
Name	**Type**	**Description**
Request count per node	Double	Cluster entity request average
Error count per node	Integer	Cluster entity request error average
Requests per second last per node	Double	Reqs/sec (last datapoint) averaged over all nodes
Requests per second in window per node	Double	Reqs/sec (over time window) averaged over all nodes
Processing time fraction in window per node	Double	Fraction of time spent processing reported by webserver (percentage, over time window) averaged over all nodes")
Total processing time per node	Integer	Total processing time per node (millis)

Table 13. Web application server cluster metrics.

B. WS-Agreement

The SLA Service follows the concepts and XML structures for agreements and templates that are defined in the WS-Agreement specification. There the agreement template describes the service offer, and an agreement describes an instantiation of a template associated to the customer. WS-Agreement (see Figure 18) specifies the XML structure to define agreements and templates, and a two layers interface of web services for operation. This implementation is focused on the XML structure, and defines REST interfaces to operate.

The following text is a summary of the specification, obtained from [9], and slightly modified to include QoB handling.

The ***WS-Agreement language*** defines the data types for expressing the content of an agreement. This language is defined independently from the WS-Agreement protocol and can therefore be used in a wide set of scenarios, for example with other protocol bindings. It is defined in the form of XML schema and describes the data types and the structure of the Agreement document, the Agreement Template document, and the Agreement Offer document.

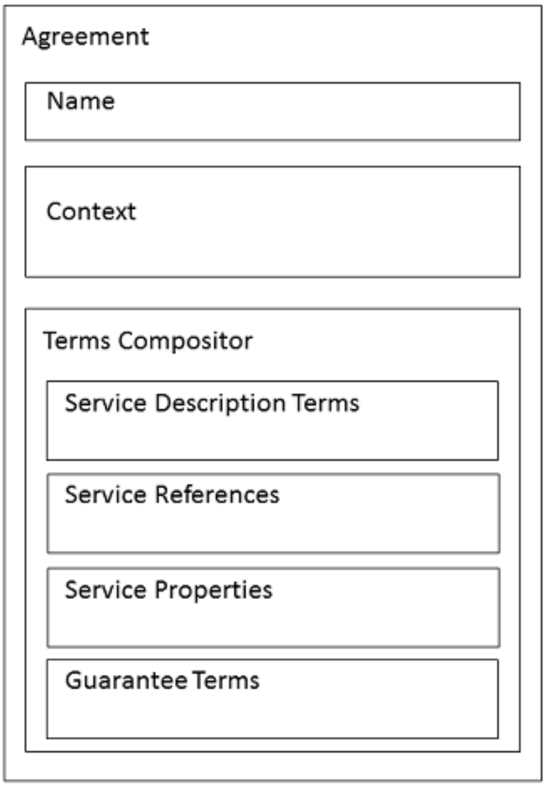

Figure 18. WS-Agreement

The above picture depicts the basic structure of an agreement. An agreement contains an agreement identifier, its name, an agreement context and a term compositor with a detailed description of the service to provide.

The ***agreement context*** contains a set of meta data that is associated with an agreement. First of all, the context provides information on the parties that created the agreement. The context therefore contains two elements: Agreement Initiator and Agreement Responder. The content of these two elements is not further specified. They can contain an arbitrary, domain specific description of each party in order to resolve them to real world entities. Such a description can for example be an endpoint reference or a distinguished name that identifies the party in a security context. Additionally, the agreement context associates each party with its respective role in the agreement by specifying which party is the service provider and which is the service consumer. By that, the involved parties in the agreement are identified and bound to their roles. The agreement context may also contain an expiration time that defines how long an agreement (and an associated service) is valid. The context should also reference the template that was used to create the agreement. This is of particular importance for the agreement offer validation process. Besides that, the agreement context may contain any domain specific data.

In general, each term in WS-Agreement is identified by a term name. Service terms additionally contain a service name. The service name allows the semantically grouping of multiple service terms in an agreement. A single service in an agreement can therefore be described by multiple service terms. Each service term describes a different aspect of the service. Service terms can be distinguished into Service Description Terms, Service References, and Service Properties.

Service Description Terms give a functional description of the service to provide. Therefore, the service description terms contain a domain specific description of the service. This can comprise a complete or a partial description. Since the WS-Agreement is designed to be domain independent, the content of a service description term can be any valid XML document. Both parties involved in the agreement (agreement initiator and agreement responder) must understand the domain specific service description, e.g. the XML schema for the domain specific language must be known to both parties.

Service References provide a way to refer to existing services within an agreement. This can be useful when a certain service quality should be provided for an existing service rather than for a new one. Similar as in service description terms, service references can contain an arbitrary XML document that describes the service reference. Here the same rules and restrictions apply as for service description terms.

Service Properties are the last type of service terms. From an abstract point of view, they provide a way to define variables in the context of an agreement. A variable definition comprises a name, a metric, and a location. The location refers to a distinct element in the agreement, e.g. by using an XML query language such as XPath. Service Properties are used to define and evaluate guarantees in WS-Agreement.

Guarantee Terms are the second group of agreement terms. They provide the required capabilities to express service guarantees in agreements, define how guarantees are assessed and which compensation methods apply in case of meeting or violating the service guarantees. Guarantee terms consist of four elements: a Service Scope, a Qualifying Condition, a Service Level Objective and a Business Value List.

Service Scope specifies which services in the agreement are covered by the guarantee. Since a single agreement can comprise a set of different services, one guarantee may apply to one or more services.

Qualifying Condition specifies preconditions that must be fulfilled before a guarantee applies. Not all guarantees apply during the whole lifetime of an agreement. The qualifying condition specifies the preconditions that must be met before a guarantee is evaluated.

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 business (QoB). The QoB properties of a service are defined in the service description terms of an agreement. These are the agreed service properties. At the service provisioning time the actual QoB properties are derived from the service monitoring system. This service level objective essentially defines how the agreed QoB properties are related to the actual QoB properties. It defines a logical expression that can be assessed in order to determine the fulfillment of a guarantee.

In the context of SeaClouds, the QoS properties are assessed as policies of the Monitoring Service, which communicates the QoS violations to the SLA Service.

Business Value List defines the penalties and rewards that are associated with a guarantee. WS-Agreement already defines a model to express business values for guarantees. These predefined business values range an abstract importance of a guarantee to arbitrary value expressions, such as Euro or US-Dollar.

References

[1] Deliverable D3.1 "Discovery design and orchestration functionalities: first specification"

[2] Deliverable D2.2 "Initial architecture and design of the SeaClouds platform"

[3] Deliverable D5.4.1 "Definition of the multi-deployment and monitoring strategies".

[4] Deliverable D4.3 "Design of the Run-time Reconfiguration process" (to be delivered on Mont M16).

[5] Deliverable D4.2 "Cloud Application Programming Interface".

[6] Deliverable D6.3.1 "Case Studies Preliminary implementation".

[7] BluePrint Description: http://brooklyncentral.github.io/v/0.7.0-SNAPSHOT/use/guide/defining-applications/yaml-reference.html, CloudSoft, 2014.

[8] Brooklyn API: https://brooklyn.incubator.apache.org/v/0.7.0-M1/use/api/index.html

[9] Guide to WS-Agreement Language: https://packcs-e0.scai.fraunhofer.de/wsag4j/wsag/wsag-language.html