Executive Summary

This deliverable presents the final version of the MODACloudML languages and models. This version of the language includes support for both the IaaS and PaaS levels and relies on the MODAClouds model-driven approach which defines three layers of abstraction: the Cloud-enabled computational Independent Model (CCIM), the Cloud Provider-Independent Model (CPIM), and the Cloud Provider-Specific Model (CPSM). In addition, this deliverable introduces design heuristics or success factors, hereafter called guidance, as well as cloud patterns which help mitigate various pitfalls when designing multi-cloud applications.
Members of the MODAClouds consortium:

- Politecnico di Milano
- Stiftelsen Sintef
- Institutul E-Austria Timisoara
- Imperial College of Science, Technology and Medicine
- SOFTEAM
- Siemens Program and System Engineering
- BOC Information Systems GMBH
- Flexiant Limited
- ATOS Spain S.A.
- CA Technologies Development Spain S.A.

Published MODAClouds documents

These documents are all available from the project website located at http://www.modaclouds.eu/
Contents

INTRODUCTION ........................................................................................................................................... 4
  1.1 CONTEXT AND OBJECTIVES .............................................................................................................. 4
  1.2 STRUCTURE OF THE DOCUMENT ..................................................................................................... 5

ACHIEVEMENTS ........................................................................................................................................... 6
  2.1 MODACLOUDML MODELS ............................................................................................................... 7
    2.1.1 The Cloud-enabled Computation Independent Models ................................................................. 7
    2.1.2 The Cloud Provider Independent Models ..................................................................................... 7
    2.1.3 The Cloud Provider Specific Models ............................................................................................ 18
  2.2 CLOUD PATTERNS ............................................................................................................................. 24
    2.2.1 Guidance and success factors ....................................................................................................... 24
    2.2.2 Selected Patterns ......................................................................................................................... 28

CONCLUSION ............................................................................................................................................... 29

BIBLIOGRAPHY .......................................................................................................................................... 30

APPENDICES ............................................................................................................................................. 31

A APPLYING MODACLOUDML TO THE CONSTELLATION SERVER CASE ........................................... 32
  2.3 USE CASE: CONSTELLATION SERVER ................................................................................................. 32
  2.4 DESIGN ALTERNATIVE AND DEPLOYMENT MODELS ................................................................. 32

B EXTENDED CLOUD PATTERNS ........................................................................................................... 37
  THE ‘EXTERNAL CONFIGURATION STORE’ PATTERN .............................................................................. 37
  THE ‘LEADER-FOLLOWERS’ PATTERN ....................................................................................................... 38
  THE ‘RUNTIME RECONFIGURATION’ PATTERN ......................................................................................... 38
  THE ‘PROVIDER ADAPTER’ PATTERN ........................................................................................................ 40
Introduction

1.1 Context and objectives

The landscape of cloud computing encompasses an ever-growing number of providers offering a multitude of infrastructure-as-a-service (IaaS) and platform-as-a-service (PaaS) solutions. In this landscape, application providers are facing the emergent need to execute and manage multi-cloud applications [1] (i.e., applications that can be deployed across multiple cloud infrastructures and platforms). In particular, they seek to exploit the peculiarities of each cloud solution as well as to combine the delivery models of IaaS and PaaS to enable optimisation of performance, availability, and cost. However, this is a complex task as stated in the CORDIS report on cloud computing [2], "whilst a distributed data environment (IaaS) cannot be easily moved to any platform provider (PaaS) […] it is also almost impossible to move a service/image/environment between providers on the same level."

Several projects aim at promoting interoperability and preventing vendor lock-in, but they are not sufficient to properly manage the complexity of development and administration of multi-cloud applications [3] at both design- and run-time. In particular, existing cloud solutions typically focus on supporting either IaaS or PaaS, but not both.

MODACloudML enables managing multi-cloud applications in a cloud provider-independent way while still exploiting the peculiarities of each IaaS and PaaS solution. By supporting both IaaS and PaaS, MODACloudML enables several levels of control of multi-cloud applications: (i) in case of executing on IaaS or white box PaaS solutions; full control with automatic provisioning and deployment of the entire cloud stack from the infrastructure to the application, or (ii) in case of executing on black box PaaS solutions; partial control of the application (note that if parts of the multi-cloud application executes on IaaS or white box PaaS, MODACloudML provides full control of those parts). This way, MODACloudML promotes the DevOps method [4] that aims to achieve better delivery life-cycle by integrating development and operation activities.

The model-driven engineering approach adopted by the MODACloudML platform allows the developers to build the system at various levels of abstraction. The three levels are: (i) the Cloud-enabled Computation Independent Model (CCIM) to describe an application and its data, (ii) the Cloud-Provider Independent Model (CPIM) to describe cloud concerns related to the application in a cloud-agnostic way, and (iii) the Cloud-Provider Specific Model (CPSM) to describe the cloud concerns needed to deploy and provision the application on a specific cloud.

Figure 1 depicts the distribution of WP4 models across these three layers of abstraction. The models manipulated by the MODACloudML IDE are grouped within the dashed box. The models out of the box can be used to support their specifications. The models at the CCIM level are used to semi-automatically generate part of the CPIM models. In particular, the Service Definition Models and the Service Orchestrations Model, which can partially be generated through reverse engineering techniques, are used to initiate the Design Alternatives and deployment models whilst the CCIM data models are used to initiate the CPIM data models. Finally, the CPIM models are refined into CPSM models. A description of the purpose of each of these models is available in D4.2.1, whilst a description of the whole set of models manipulated within the MODAClouds platform is available in D3.2.2.
This document presents the MODACloudML language and the models used at these various abstraction layers. The modelling concepts and the metamodels of each of them are detailed and illustrated in appendix using the Project management case study. In addition, this document presents a set of cloud patterns and best practices that can be used by application developers and providers to ease the design and management of multi-cloud applications.

### 1.2 Structure of the document

The remainder of the document is organized as follows. Section 2 briefly summarizes the main achievements and evolutions of MODACloudML during the second year of the project. Section 3 presents the metamodels and highlight their evolutions compared to D4.2.1 including the CIM, CPIM, and CPSM levels. These models are detailed in appendices and applied to the Project Management Server case study. Section 4 introduces cloud patterns and guidances that have been used or extended so far in the context of the MODAClouds project.
Achievements

<table>
<thead>
<tr>
<th>Objectives</th>
<th>Achievements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Modelling concepts for the models at the CCIM level.</td>
<td>At this CCIM level the models and metamodels remain similar to the one presented in D4.2.1 and the work has focused on the improvement of their support by the MODACloudML IDE.</td>
</tr>
<tr>
<td>Refine the existing modelling concepts of the Design alternative and deployment models at both the CPIM and CPSM levels.</td>
<td>The existing modelling concepts in the design alternatives and deployment models have been refined in particular to better reflect the dependencies between the various software components to be deployed (cf. Sections 2.1.2.1).</td>
</tr>
<tr>
<td>Extend the Design alternatives and deployment models with concepts to support PaaS solutions.</td>
<td>Design alternatives and deployment models have been extended with new component and port types in order to support the specification of deployments that exploit at the same time or separately IaaS and PaaS solutions (cf. Sections 2.1.2.1).</td>
</tr>
<tr>
<td>Extend the Design alternatives and deployment models with concepts to support a configuration management tool.</td>
<td>Design alternatives and deployment models have been extended with the necessary concepts to support the integration of the Puppet configuration management tool with CloudML (cf. Section 2.1.2.1).</td>
</tr>
<tr>
<td>Refine the existing modelling concepts for the data models at both the CPIM and CPSM levels.</td>
<td>See D4.4.2</td>
</tr>
<tr>
<td>Refine the existing modelling concepts for the Resource models at both the CPIM and CPSM levels.</td>
<td>Resource models at both the CPIM and the CPSM have been refined to better reflect the current cloud offerings (cf. Sections 2.1.2.3 and 2.1.3.3).</td>
</tr>
<tr>
<td>Extend the Resource models to the PaaS level.</td>
<td>Resource models at both the CPIM and the CPSM have been extended with PaaS concepts (cf. Sections 2.1.2.3 and 2.1.3.3)</td>
</tr>
<tr>
<td>Identify a set of cloud patterns and guidance based on the project experiences.</td>
<td>In Section 2.2, we review state of the art cloud patterns relevant in a multi-cloud context. Some of the selected patterns are extended based on the MODAClouds experiences and key design heuristics and guidance relevant when designing multi-cloud applications are proposed.</td>
</tr>
</tbody>
</table>
2.1 MODACloudML models

In this section we detail the various MODACloudML models and present the evolution of their metamodels compared to D4.2.1.

2.1.1 The Cloud-enabled Computation Independent Models

At the CCIM level, an application is described as a set of high level services following a Service Oriented Architecture (SOA) [5]. The application is presented as a set of business-aligned reusable services that can be combined into high-level business processes and solutions within the context of an enterprise. These models at this level of abstraction involve three main concepts: a set of services, an orchestration, and a set of usage models.

At CCIM level models and metamodels remain similar to the one presented in D4.2.1 and work has focused on the improvement of their support by the MODACloudML IDE (i.e., development of features such as model validation or reverse engineering of application’s code to generate part of the CCIM models as well as derivation of part of the application code from CCIM models). For a detailed presentation of the models and metamodels please refer to D4.2.1. For a detailed presentation of the MODACloudML IDE please refer to D4.3.2.

2.1.2 The Cloud Provider Independent Models

At the CPIM level MODACloudML proposes a new approach to describe the deployment, provisioning and data models of multi-cloud systems in a provider-agnostic way. Compared to D4.2.1, the main evolution of these models has been to offer support for PaaS solutions.

2.1.2.1 Design Alternative and deployment model

CloudML is a Domain-Specific Modelling Language (DSML) to model the provisioning and deployment of multi-cloud applications (i.e., to specify design alternatives and deployment model), which is developed in collaboration between the MODAClouds and PaaSage\(^1\) projects and is a sub-part of MODACloudML. CloudML has been designed based on the following requirements:

- **Cloud provider-independence** (R1): CloudML shall support a cloud provider-agnostic specification of the provisioning and deployment of applications in the cloud. This will simplify the design of multi-cloud applications and prevent vendor lock-in.
- **Separation of concerns** (R2): CloudML shall support a modular, loosely-coupled specification of the provisioning and deployment so that the modules can be seamlessly substituted. This will facilitate the maintenance as well as the adaptation of the deployment topology.
- **Reusability** (R3): CloudML shall support the specification of types or patterns that can be seamlessly reused to design a deployment. This will ease the evolution as well as the rapid development of different variants of a system.
- **Abstraction** (R4): CloudML shall provide a single domain-specific language and abstraction for specifying deployment on both IaaS and PaaS in a provider-independent or -specific way. Moreover, the MODAClouds architecture involves models@runtime (see D3.2.2) to enable the continuous evolution of the system with no strict boundaries between design-time and runtime activities. CloudML shall then be reused by the models@runtime and provide the necessary concepts so that it can continuously provide an up-to-date representation of the deployment of the running system.

\(^1\) [http://www.paasage.eu/](http://www.paasage.eu/)
• **White- and black-box infrastructure** (R5): CloudML shall support IaaS and PaaS solutions. This will enable various degrees of control over underlying infrastructures and platforms of multi-cloud applications.

In particular, CloudML allows developers to model the provisioning and deployment of a multi-cloud application at two levels of abstraction: (i) the Cloud Provider-Independent Model (CPIM), which specifies the provisioning and deployment of a multi-cloud application in a cloud provider-agnostic way (addressing the requirement R1); (ii) the Cloud Provider-Specific Model (CPSM), which refines the CPIM and specifies the provisioning and deployment of a multi-cloud application in a cloud provider-specific way. This two-level approach is agnostic to any development paradigm and technology, meaning that application developers can design and implement their applications based on their preferred paradigms and technologies.

CloudML is also inspired by component-based approaches [6], which facilitates separation of concerns (R2) and reusability (R3). In this respect, deployment models can be regarded as assemblies of components exposing ports, and bindings between these ports.

In addition, CloudML implements the *type-instance* pattern [7], which also facilitates reusability (R3) and abstraction (R4). This pattern exploits two flavours of typing, namely *ontological* and *linguistic*, respectively [8]. Figure 2 illustrates these two flavours of typing. SL (for Small Linux) represents a reusable type of virtual machine. It is linguistically typed by the class VM (for Virtual Machine). SL1 represents an instance of the virtual machine SL. It is ontologically typed by SL and linguistically typed by VMInstance.

CloudML models can be edited using the MODACloudML IDE. In addition, an eclipse-based editor that supports a new textual syntax and offers features such as syntax highlights, auto-completion, and on-site validation has been developed. Through the usage of these two tools deployment models can be serialized in two formats, namely the JavaScript Object Notation (JSON) and the XML Metadata Interchange (XMI).

The CloudML models and metamodels are represented as plain Java objects and ecore and are available at https://github.com/SINTEF-9012/cloudml. The JSON and XMI serialization codecs are based on Kotlin (http://kotlin.jetbrains.org) and the Kevoree Modeling Framework (KMF) [9]. The textual syntax editor (available at https://github.com/SINTEF-9012/cloudml-dsl) exploits the Xtext framework [10].

Figure 3 depicts the inheritance relationship between all the concepts (i.e., type and instance level elements) of the CloudML metamodel. **Appendix A.2 provides a detailed description of all the concepts as well as sample models conforming to this metamodel applied to the Project Management Server case study.**

---

2 http://www.eclipse.org/modeling/emf/
In the following, we provide a description of the new concepts introduced within the CloudML metamodel since the release of D4.2.1.

Figure 4 presents the type part of the CloudML metamodel. As in the former metamodel, all the new concepts in this metamodel can be associated with Resources and Properties (cf. D4.2.1).

A Component represents a reusable type of component of a cloud-based application, this concept has been refined with subtypes. A Component can be an ExternalComponent, meaning that it is managed by an external Provider (e.g., an Amazon RDS Database container), or an InternalComponent, meaning that it is managed by CloudML (e.g., a JEE container). This mechanism enables supporting both IaaS and PaaS solutions through the single abstract concept of component. The property location of ExternalComponent represents the geographical location of the data centre hosting it (e.g., location = "eu-west-1", short for West Europe). The properties credentials and serviceType represent the authentication information needed to access to this specific service and its type (e.g., database, application container), respectively.

An ExternalComponent can be a VM (e.g., a virtual machine running GNU-Linux). The properties minCores, maxCores, minRam, maxRam, minStorage, and maxStorage depict the lower and upper bounds of virtual compute cores, RAM, and storage, respectively, of the required virtual machine (e.g., minCores = 1, minRam = 1024). The property OS represents the operating system to be run by the virtual machine (e.g., OS = "ubuntu"). All these constraints are optional and do not have to be defined in the CPIM.

A HostingPort represents a hosting interface of a component. A HostingPort can be a ProvidedHost, meaning that it provides hosting facilities (i.e., it provides an execution environment) to another component (e.g., a virtual machine running GNU/Linux provides hosting to a JEE container), or a RequiredHost, meaning that an internal component requires hosting from another component (e.g., Constellation requires hosting from a JEE container).
A *Hosting* represents a reusable type of hosting binding between a *Required* and a *ProvidedHost* (e.g., a Servlet container is contained by a virtual machine running GNU/Linux).

A *Cloud* represents a collection of virtual machines on a particular cloud provider. One cloud provider can host several *Clouds*.

In order to support Puppet as a configuration management engine that could be used by the CloudML deployment engine to install and configure part of an application on a specific VM, we have defined the concept of *PuppetResource* which extends *Resource*. The properties *masterEndpoint* and *repositoryEndpoint* depict the endpoint of the Puppet master node responsible for configuring a Puppet client and of the repository containing the Puppet manifests (describing the configuration), respectively. The properties *repositoryKey* and *username* depict the credentials to be used to access the manifest repository. The property *configurationFile* depicts the name of the manifest to be used by the Puppet master in order to perform the configuration. Finally, the *configure hostname* command describes how to modify the hostname of a VM and enables its identification by the Puppet master. The integration of Puppet with CloudML is an ongoing work and further tests and development will be performed during the third year of the project.

These types can be instantiated in order to form an assembly of components that specifies a deployment model. Each instance is identified by a unique identifier and refers to a type. Figure 5 presents the instance part of the CloudML metamodel.
2.1.2.1.1 Constraints on the CPIM metamodel

Using OCL constraints, we formalized the dependencies between types and instances as well as constraints to ensure that a deployment model makes sense. The set of constraints defined in D4.2.1 has been extended and updated to the new metamodel. In the following we present the added constraints.

A first set of constraints (see Listing 1) has been specified to ensure that a deployment model follows the naming policy adopted within CloudML. Each component, communication, and hosting instance and type created has a unique name within the type scope. Within the scope of a component, each port has a unique name. A port can be identified uniquely within a deployment model by combining its name with the name of the component it belongs as follows: component-name.port-name. Similarly, within the scope of a CloudMLElement, each property and resource has a unique name.

---

Instances of each type have unique names

context CloudMLModel:
inv types_with_unique_name:
    self.components->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name)
    and self.communications->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name)
    and self.hostings->forAll(h1, h2 | h1 <> h2 implies h1.name <> h2.name);

---

Instances of each instance have unique names

inv instance_with_unique_names:
    self.componentInstances->forAll(ci1, ci2 | ci1 <> ci2 implies ci1.name <> ci2.name)
    and self.communicationInstances->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name)
    and self.hostingInstances->forAll(h1, h2 | h1 <> h2 implies h1.name <> h2.name);

---

All ports of a component type have unique names

context Component:
inv ports_with_unique_names:
    self.providedHosts->forAll(h1, h2 | h1 <> h2 implies h1.name <> h2.name);
    and self.providedCommunications->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name);

---

All ports of a component type have unique names

context InternalComponent:
inv ports_with_unique_names:
    self.requiredCommunications->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name);

---

All ports of a component instance have unique names

context CloudMLElement:

context ComponentInstance:
inv ports_with_unique_names:
   self.requiredCommunicationInstances->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name);

-- All ports of a component instance have unique names

class InternalComponentInstance:

context InternalComponentInstance:
inv ports_with_unique_names:
   self.providedHostInstances->forAll(h1, h2 | h1 <> h2 implies h1.name <> h2.name);
   and self.providedCommunications->forAll(c1, c2 | c1 <> c2 implies c1.name <> c2.name);

-- All properties and resources of a CloudMLElement have unique names

class CloudMLElement:
invariant different_properties_in_CloudMLElement:
   properties->forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
   and resources->forAll(r1, r2 | r1 <> r2 implies r1.name <> r2.name)

Listing 1 OCL constraint to ensure a proper naming of a deployment model

A second set of constraints (see Listing 2) ensures that each communication or hosting port is contained by a component. These constraints are particularly important to ensure that all the ports used into a Communication or a Hosting are associated to a component and that the deployment model consist in a well-formed connected graph without open edges. This property has to be valid at both the type and instance levels.

class Component:
inv provided_component_ports_should_point_to_component:
   self.providedCommunications->forAll(p | p.owner = self)
   and self.providedHosts->forAll(p | p.owner = self);

class InternalComponent:
inv requiredHost_owner_is_self:
   self.requiredHost.owner = self;
inv requiredCommunications_owner_is_self:
   self.requiredCommunications->forAll(p | p.owner = self);

class ComponentInstance:
inv component_instance_ports_belong_to_instance:
   self.providedCommunicationInstances->forAll(p | p.componentInstance = self)
   and self.providedHostInstances->forAll(p | p.componentInstance = self);

class InternalComponentInstance:
inv requiredHost_owner_is_self:
   self.requiredHost.owner = self;
inv requiredCommunications_owner_is_self:
   self.requiredCommunications->forAll(p | p.owner = self);

Listing 2 OCL constraints to ensure that the deployment model is a connected graph

In CloudML, internal components can be composed of other internal components in order to ease the reusability of sub-parts of a deployment model. In order to prevent recursive containment (i.e., a composite component containing itself), which might lead to a deployment that never ends, a composite component is not allowed to contain itself (see Listing 3).

-- A composite component is not allowed to contain itself

class Component:
invariant recursion_in_parts_of_component:
   not(self.contains(self, self));

Listing 3 OCL constraint on composite components
Communication and Hosting types bind together ports of specific types. Accordingly, the constraint specified in Listing 4 ensures that Communication and Hosting instances must bind together instances of these port types.

-- A communication instance should always bind ports instances with the same type as the one defined in its type
context CommunicationInstance:
inv communication_instance_correct_port_instantces:
    self.requiredCommunicationInstance.type = self.type.requiredCommunication
    and self.providedCommunicationInstance.type = self.type.providedCommunication;

-- a Hosting instance should always bind ports instances with the same type as the one defined in its type
context HostingInstance:
inv containment_instance_correct_port_instance:
    self.providedHostInstance.type = self.type.providedHost
    and self.requiredHostInstance.type = self.type.requiredHost;

Listing 4 OCL constraints to ensure that bindings involve port of the proper type

Similarly, a component type contains a set of port types (i.e., ProvidedCommunication, ProvidedHost, and RequiredCommunication and RequiredHost in case it is an internal component). The constraint defined in Listing 5 ensures that an instance of a component type can only contains port instances of one of these port types.

context ComponentInstance:
inv component_port_instances_of_correct_type:
    self.providedCommunicationInstances->forAll(p | self.type.providedCommunications->includes(p.type))
    and self.providedHostInstances->forAll(p | self.type.providedHosts->includes(p.type));

context InternalComponentInstance:
inv internal_component_port_instances_of_correct_type:
    type.oclIsKindOf(InternalComponent)
    and requiredCommunicationInstances->forAll(p | typeoclAsType(InternalComponent).requiredCommunications->includes(p.type))
    and requiredHostInstance.type = type.oclAsType(InternalComponent).requiredHost;

Listing 5 OCL constraints to ensure that component instances contain port instances of the proper type

2.1.2.2 Data model
The Data model span over all three abstraction levels (CCIM, CPIM and CPSM) as indicated in Figure 1. The description of the data model can be found in D4.4.2 and are not repeated here. D4.4.2 provides a full overview of the data models and the data modelling activities including the data models at CCIM, CPIM and CPSM levels and the mapping between these models.

2.1.2.3 Resource models
As part of MODACloudML, resource models represent typical cloud environment and can be used as a catalogue of available resources and classical cloud patterns. These resource models are particularly relevant for the analysis of non-functional characteristics as well as to help the cloud application developer and the cloud application provider in designing, optimizing and deploying a multi-cloud system. Indeed, they provide a description of the resources available and the relationship between them. In this section we present the evolution of the initial models presented in D4.2.1 and their extension to PaaS solutions.
At the CPIM level the goal of the resource model presented in Error! Reference source not found. is to represent a pattern of the general structure of a cloud environment, without expressing in detail the features of cloud services offered by the available cloud providers.

A cloud system is always owned by a Cloud Provider, which is a specialization of the general abstract concept of Provider. Typically, a Cloud Provider offers to the end users several Cloud Services. A Cloud Service can be classified into three main classes: IaaS-Service, PaaS-Service, and SaaS-Service. An IaaS-Service is composed of one or more Cloud Resources and ResourcePools, while a PaaS-Service is composed of one or more CloudPlatformServices and, similarly, a SaaS-Service is composed of one or more CloudSoftwares. Cloud Softwares can either be deployed on CloudPlatformServices or run directly on ResourcePools (i.e., as it will be discussed later, a pool of computing resources running at an infrastructure layer). Vice versa, an ApplicationServiceProvider offers SaaS-Services.

A CloudElement represents either a CloudIaaSResource, a CloudPlatformService or a CloudSoftware, and can be characterized by a Cost Profile (see D5.2.2 for further details).

A ResourcePool aggregate one or more homogeneous Compute resources, which are CloudIaaSResources (see Error! Reference source not found.), and might contain AdaptivePolicies.

Error! Reference source not found. represents other features of CloudIaaSResources, CloudPlatformServices, and CloudSoftwares, which are also specializations of CloudElements.
Some CloudElements can be characterized by a Location (expressed in terms of Region, Sub Region and Virtual Area), which specifies where the (virtual) hardware infrastructure providing the cloud resources is located.

A CloudIaaSResource represents the minimal resource unit of a given IaaS-Service and can be classified into a Compute unit or a Cloud Storage unit. A Compute unit represents a general computational resource, like a Virtual Machine (VM).

A PaaS-Service is a software framework exposing a defined Application Programming Interface (API) that can be used to develop custom applications and services. The platform also provides an execution environment for such custom applications and services. A CloudPlatformService always runs on at least one Cloud Compute resource. Frontend, Backend, MiddlewareService, and Database are the four possible specializations of a CloudPlatformService. Frontend platforms can host frontend services, which are directly exposed to end users and are supposed to interact with them and thus providing data to backend services. Backend services are hidden to end users and are supposed to process data coming from frontend services eventually providing them with some intermediate results. MiddlewareServices can host services like message queues or task queues, which are used to decouple Frontend instances from Backend instances. A databaseService can store structured or semi-structured data over cloudStorage. We details these CloudPlatformServices in the following.
A CloudSoftware is an application or a service that can be deployed on CloudPlatformServices or can run directly on Compute Resources. A CloudSoftware can be provided by a Cloud Provider within its offered SaaS or can be provided by an Application Service Provider who uses the Cloud Platforms or directly the Cloud Resources offered by a Cloud Provider. Finally, a CloudSoftware can be possibly classified into REST Software or SOAP Software, like stateless or stateful services, depending on whether it is a REST- or a SOAP-based software.

Error! Reference source not found. shows detailed features of a CloudIaaSResource, which is a particular Cloud Element. A Cloud Element can be tied to one or more Cloud Elements through point-to-point Links in order to create a virtual network of Cloud Elements.

Figure 8 CPIM-IaaS (Purple: resource allocation profiles, Green: Compute and storage resources, Yellow: IaaS resource).

A Resource Pool is a set of Compute resources associated to an Allocation Profile, which contains a set of Allocations specifying how the number of allocated instances within the Resource Pool changes in a certain reference period. A Cloud Storage unit is a resource able to store any kind of unstructured or structured data and it can be classified into FilesystemStorage or BlobStorage. A FilesystemStorage is a storage unit based on a file system, so it contains files and folders organized into a hierarchical structure. Virtual Hard Disks (VHDs) are concrete examples of FilesystemStorage units. A BlobStorage is a storage unit based on a flat structure composed by data containers.

Considering more in details PaaS-Service models Error! Reference source not found. describes MiddlewareServices. Within MODAClouds we consider the Queue, Memcache, and SSLEncryption services.
**Queue**s are classified in **MessageQueue** and **TaskQueue** and are associated also to a **CloudStorage** resource that provides persistent messages. A **SSLEncryption** service encrypts a specific **Link** resource.

![Diagram of CPIM-PaaS Middleware Services](image)

Figure 9. CPIM-PaaS Middleware Services.

Figure 10 shows **Database (DB)** services. A DB service is able to store (exploiting a **CloudStorage** resource) structured or semi-structured data and can be classified into **Relational DB** or **NoSQL DB**. A **Relational DB** is a database platform based on the relational model, so database entries are organized into tables and rows (homogeneous within a table) and can be accessed and manipulated through relational query languages. A **NoSQL DB** is a database platform based on a distributed architecture, into which data is not required to have a relational structure. Furthermore, these databases use query languages different from SQL and they cannot guarantee all the ACID properties (Atomicity, Consistency, Isolation, Durability). Databases are considered as cloud platform services because they provide interfaces to access structured or semi-structured data and can be configured by the user, but it is their underlying infrastructure (IaaS level) backup and recovery are usually managed by the Cloud provider. **Relational DBs** can include also **ReadReplica** services to improve application performance.

![Diagram of CPIM-Database Services](image)

Figure 10. CPIM-Database Services.
Among NoSQL databases, *KeyValueStores* allow to retrieve information (values) given their key, *ColumnStores* organize data are inside structures named Columns, which in turn are contained inside Column Families, and are indexed by key. Typically, data inside the same Column (for every key) are persisted together. *DocumentStores* manage semi-structured data, called documents. Each document may contain several unique identifier fields, used to index other fields that can be of any type, simple or nested, including other documents. Finally, *MultiModelDBs* provide characteristics of at least two of the above mentioned data models.

### 2.1.3 The Cloud Provider Specific Models

At the CPSM level, the design alternatives and deployment models as well as the data models are refined to include provider-specific concerns. Compared to D4.2.1, the main evolution of these models has been to offer support for PaaS solutions.

#### 2.1.3.1 Design Alternative and deployment model

In the following, we show how a deployment model at the CPIM level can be refined into a CPSM and detail the list of provider specific attributes that refine a CPIM into a CPSM. Appendix A provides a detailed description of all the concepts as well as samples of CPSM applied to the Project Management Server case study.

A deployment model at the CPSM level consists of an enrichment of the instances of the corresponding CPIM with provider-specific information. This enrichment mainly affects external components.

The transformation from CPIM to CPSM consists in: (i) adding the actual data resulting from the resolution of the constraints defined in the external component types (e.g., actual number of cores, memory size), and (ii) adding data required for the deployment and management of the application that are provider-specific. Thanks to this enrichment, one may retrieve data about the actual resources provisioned including how they can be accessed and how they can be configured. Such data is particularly useful during the process of configuration of the components and their relationships as well as for adaptation by models@runtime. In order to obtain the provider-specific information, this transformation can be either parameterized with data from the resource models at the CPSM level or it can retrieve these information directly from the providers.

Figure 11 presents an example of this process. The first step consists of the specification of the provider on which the CPIM instances will be deployed (e.g., the virtual machine running GNU/Linux called SL1 will be provisioned in the private cloud). A request is then sent to the models@runtime for details on virtual machines available from this provider according to the constraints defined in its type. As a consequence, the engine will interact with the provider in order to retrieve this information before updating the metadata associated to the instance (e.g., small instance in UK location). Similar information can be obtained for PaaS solutions.
The following table summarizes the attributes used to enrich the CPIM with provider-specific information.

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExternalComponentInstance</td>
<td>Public address</td>
<td>The IP public address to access the external component</td>
</tr>
<tr>
<td>ExternalComponentInstance</td>
<td>Status</td>
<td>The status of the component (i.e., running, stopped, error)</td>
</tr>
<tr>
<td>ExternalComponentInstance</td>
<td>ProviderSpecificType</td>
<td>The type of resource allocated by the provider (e.g., m1.micro)</td>
</tr>
<tr>
<td>ExternalComponentInstance</td>
<td>ID</td>
<td>The unique identifier of the service defined by the provider (e.g., identifier of an instance of Amazon RDS database)</td>
</tr>
<tr>
<td>VMInstance</td>
<td>Image ID</td>
<td>The unique identifier of the image run by the VM</td>
</tr>
<tr>
<td>VMInstance</td>
<td>Network ID</td>
<td>The unique identifier of the network the VM belong to</td>
</tr>
<tr>
<td>VMInstance</td>
<td>CPU</td>
<td>The actual CPU power allocated by the provider (between the minCPU and maxCPU constraints defined at the CPIM level)</td>
</tr>
<tr>
<td>VMInstance</td>
<td>RAM</td>
<td>The actual RAM allocated by the provider (between the minRAM and maxRAM constraints defined at the CPIM level)</td>
</tr>
<tr>
<td>VMInstance</td>
<td>Storage</td>
<td>The actual Storage size allocated by the provider (between the minStorage and maxStorage constraints defined at the CPIM level)</td>
</tr>
</tbody>
</table>
2.1.3.2 Data models

As mentioned, the description of the data model can be found in D4.4.2 and are not repeated here. D4.4.2 provides a full overview of the data models and the data modelling activities including the data models at the CCIM, CPIM and CPSM levels and the mapping between these models.

2.1.3.3 Resource models

A resource model at the CPSM level can be thought as a particular instance of the resource model at the CPIM level representing the features of the cloud services offered by a particular cloud provider. In this section we will present one example of Cloud Provider-Specific Model, referred to Amazon Web Services (AWS) which provides services both at IaaS and PaaS level.

Considering Amazon cloud, we can distinguish some relevant IaaS services like: Elastic Compute Cloud (EC2), Simple Storage Service (S3), Elastic Block Store (EBS), Relational Database Service (RDS), the NoSQL database DynamoDB, Simple Queue Service (SQS), and ElastiCache. Amazon itself has to be considered as a realisation of the Cloud Provider concept expressed within the CPIM. Since a Cloud Provider offers one or more Cloud Services, we can extend the relation provides to all the aforementioned cloud services. Some of them can be classified as CloudIaaSResources, like EC2, S3 and EBS; the others (RDS, DynamoDB, SQS and ElastiCache) can be classified as CloudPlatformServices.

Considering that S3 provides storage with a flat file system, an S3 Instance is a realization of the Blob Storage concept at the CPIM level. We can define a Cost Profile for this type of instances that represents the cost variability. Such profile is necessary due to the fact that the S3 service is charged based on different price ranges, depending on the allocated capacity. On the other hand, EBS provides volumes with standard file systems. An EBS Instance is thus a realisation of the File system Storage.

Finally, Amazon EC2 provides a set of EC2 Instances, which are realisations of a Compute Resource. EC2 Instances are characterized by different EC2 Prices, which are essentially the Costs from the CPIM. Moreover, it is possible to specify EC2 Auto Scaling Groups, which are sets of homogeneous EC2 Instances and can be referred to as a realisation of the Resource Pools concept at the CPIM level. EC2 Scaling Policies can be defined on these EC2 Auto Scaling Groups as the rules appointed to control the scaling activities. An EC2 Scaling Policy derives from a generic Scaling Policy of the CPIM. For an EC2 instance it is possible to specify the Location in order to control reliability and delays.

Figure 12 exemplifies the modelling of an EC2 Micro Instance, showing also the relation with the SubRegion and Region of the CPIM. Amazon instances do not provide any permanent local storage, so they usually use an external EBS Volume.

Within EC2, it is possible to define Virtual Areas (EC2 Micro Virtual Private Cloud) within sub-regions (in this case eu-west-1b that, in turn, belongs to the region eu-west-1) and to associate EC2 instances to them. The Figure shows also other features of EC2, such as the possibility to define Resource Pools (EC2 Micro Auto Scaling Group) composed by EC2 instances with the same configurations. It is also possible to associate to these pools Allocation Profiles (EC2 Micro Allocation Profile) that specify the number of instances of a given type are allocated in a given time period.

---

3 aws.amazon.com
At the time of writing, Amazon provides 23 different instance types. Figure 13 shows the partial CPSM associated to EC2 (only a few EC2 instance types are represented and details about CPU and memory are omitted for simplicity).

Figure 14 and Figure 15 show an overview of the CPSM representations of DynamoDB and RDS (only a few RDS instance types are represented). RDS is a PaaS Cloud Service composed of one or more RDS...
Instances, each of them represents a Relational DB that, in turn, is a Database Cloud Resource. The diagram in Figure 14 shows also some realizations of an RDS Instance, such as the RDS Micro DB Instance, the RDS Small DB Instance, the RDS Medium DB Instance and the RDS Large DB Instance. An RDS Instance is composed by a Compute instance executing the RDBMS (MicroDBInstance, SmallDBInstance, LargeDBInstance) and a Cloud Storage providing the needed storage capacity (RDS_Storage).

DynamoDB, on the other hand, is a NoSQL Database and in particular it belongs to the KeyValueStore category. As for RDS, DynamoDB is a Cloud Platform Service that realizes the Database concept.
For what concerns storage services, S3 and EBS are composed of S3 Instances and EBS Volumes respectively (see Figure 16). Considering that S3 provides storage with a flat file system, an S3 Instance is a realization of the Blob Storage that, in turn, is a Cloud Storage and then a Cloud IaaS Resource.

![Diagram of AWS CPSM - EBS and S3 Overview](image)

Figure 16. AWS CPSM - EBS and S3 Overview (yellow: CPIM classes, cyan: CPSM classes)

Figure 17 describes the case of the Simple Queue Service (SQS), a Cloud Platform Service that implements a Message Queue Service and hence it is a realisation of the Middleware class.

![Diagram of Amazon Simple Queue Model](image)

Figure 17 Amazon Simple Queue Model (yellow: CPIM classes, cyan: CPSM classes)

Finally, in Figure 18 the ElastiCache service is reported. It is a special Middleware Service that allows the cloud end user to easily install in-memory cache services (in particular Memcached and Redis are currently supported) to improve the performance of the overall application. ElastiCache can run on
different types of EC2 instances (in the figure only a subset of the actual types is reported) from *Micro* to *xLarge*.

Figure 18 Amazon ElastiCache (yellow: CPIM classes, cyan: CPSM classes, green: external softwares)

### 2.2 Cloud patterns

While working on multi-cloud applications, we identified a set of design heuristics or success factors, hereafter called guidance, which help mitigate various pitfalls when designing multi-cloud applications. These guidance complement the numerous design and architecture patterns that have already been identified over the past decade [11], [12], [13] as the cloud technologies matured. Along with the above referred patterns, we review below a selection of the patterns that were relevant in the MODAClouds context. Please note that we do not detail cloud offering patterns since an important set of these patterns is already provided by the resource models (D4.2.1 and Sections 2.1.2.3) especially at the CPIM level.

#### 2.2.1 Guidance and success factors

We review below the key design heuristics and guidance that were relevant in the MODAClouds context and consequently particularly relevant when designing multi-cloud applications, namely:

- **Autoscaling**: Key success factor for building robust scalable applications.
- **Compute partitioning**: Key design heuristics that help building system that can easily be maintained and deployed on cloud platform and infrastructure.
- **Multiple datacentre deployment**: Key success factors that ensure successful deployments across several cloud providers.
- **Instrumentation and telemetry**: Key success factors in building feedback about the runtime performance of the system and its underlying platform and infrastructure.

We present these patterns below, furthermore, based on our experiences and findings developing the MODAClouds approach we have made some extensions to these guidance and patterns.
2.2.1.1 Compute partitioning guidance

Why compute partitioning is important? Cloud computing provides application developers with the ability to exploit IT resources that can be provisioned with minimal management effort. Cloud applications typically exploit this feature by relying on multiple distributed IT resources, which might evolve for instance by scaling in or out. In such context, modularity and loose coupling between the blocks composing an application are key factors to exploit cloud properties. This is even more important in the context of multi-cloud applications which relies on resources possibly offered by multiple providers with their own specificities.

However, they require applications designed to support such distribution by applying the separation of concerns principle. This principle advocates decomposing and encapsulating the features of an application into modular and reusable blocks. Indeed, as expressed in [12], monolithic applications based on a single component are often less suitable for a cloud environment, in particular due to their inability to be scaled out efficiently. In addition, minimizing the coupling between application components improves the ability to manage and adapt the application or a subpart of it.

In order to achieve this, application developers can follow the Computing partitioning guidance from [11] and apply the loose coupling and distributed application patterns [12].

MODACloudML leverages this guidance and propose to decompose applications into logical components and help the user in allocating these components on cloud resources. At the CCIM level application are represented as a set of reusable services following a Service Oriented Architecture (SOA). These services are allocated to cloud resources in the deployment models which are component-based. The decision of allocating the various application services on the cloud resources is made through a Quality of Service analysis which relies on QoS models also exploiting a component-based approach. This way, all models involved during the design of cloud-based application apply the separation of concern principle.

In addition, MODAClouds advocates as a best practice to design cloud application and allocate application components to cloud resources through the workflow illustrated by Figure 19(detailed explanation of this figure are available in D3.2.2). This workflow involves four actors: the QoS engineer, the Feasibility Study engineer, the Application provider, and the Application developer.

The process starts at the CCIM level with the Application developer specifying the functional model of the application as a set of services together with requirements and a model of the data to be used by the application. Then, a decision process is started to enrich the previously produced models with insight on candidate services that can be exploited to deploy the application on the cloud. The QoS engineer then analyse the various design alternatives in order to identify the most suitable. The resulting CPIM models are then refined by adding cloud-dependent information in order to obtain a model that can be used to effectively deploy the cloud-application.

Relevant patterns involved: Loose coupling, compute partitioning, distributed applications, integration provider
2.2.1.2 Provider-independent design and deployment of cloud applications

Why provider-independent design is important? Different cloud providers support varying sets of services that may be reused by applications. Providers of multi-cloud applications seek to exploit the peculiarities of each cloud solution and to combine the delivery models of IaaS and PaaS in order to optimise performance, availability, and cost. Developers therefore need a way to design multi-cloud applications so that they can take profit of this variability instead of being hindered by it.

A solution to achieve this consists in separating the design of the application from the technical specification of the underlying infrastructure as for instance suggested by the Model-Driven Architecture (cf. MDA) [14] through the usage of platform-independent (PIM) and platform-specific models (PSM). Platform-independent models should provide the right level of abstraction so that they can be used to generate PSMs targeting different platforms whilst still exploiting their peculiarities. The challenge is then to identify the right level of abstraction as well as the concepts relevant at both the PIM and PSM levels.

In accordance with this approach, the MODAClouds [3] as well as the PaaSage [15], REMICS [16], and ARTIST [17] projects, advocate the usage of cloud provider-independent and cloud provider-specific models. In particular, MODACloudML drives cloud-application providers and developers through the following design activities.

Activities at CCIM level:
- Breakdown the cloud-application into independent services.
- Definition of the interfaces and data types for each service.
- High level service orchestration scenarios.
Activities at CPIM level:
- Definition of cloud provider-independent deployment scenarios. Each scenario defines:
  - which services are implemented by the application and which services are reused from providers.
  - the profile of the required IaaS or PaaS resources and the allocation of services to resources.
- Definition of cloud provider-independent deployment scripts.

Activities at CPSM level:
- Selection of cloud provider-specific resources.
- Configuration of specific cloud provider account.

Relevant patterns involved: Loose coupling, multiple datacentre deployment

2.2.1.3 Instrumentation and telemetry

Why instrumentation and telemetry is important? As for legacy application, the management of cloud-based application involves monitoring activities to diagnose the health of the overall system. Intuitively, a typical monitoring activity consist in observing the cloud infrastructure of an application by collecting metrics related to CPU, memory, disk and network either directly from the VMs or using the provider’s platform APIs. However, especially in a multi-cloud context where monitoring interfaces are likely to be incompatible and provider-specific, the monitoring activity is highly subject to vendors lock-in. In addition, it is not enough to only monitor a physical machine or a VM to measure the application resource consumption, detecting SLA violations and managing resources efficiently.

Application-level monitoring is of significant importance in the cloud context, and especially for SLA management. Indeed, due to virtualization as the basis for resource sharing, multiple VMs can be run on a single physical machine or even multiple applications can be run on a single VM. As a result, per-application monitoring in such a shared environment is essential to keep applications healthy and guarantee QoS. Therefore, it is not enough just to monitor a physical machine or a VM to measure the application resource consumption, detecting SLA violations and managing resources efficiently. In addition, complex applications that must scale including by bursting across clouds can generate huge volume of data to be collected by the monitoring mechanisms and can overwhelm simple monitoring techniques.

As recommended in the instrumentation and telemetry guidance from [11], the monitoring solution can collect and highlight high level events and alarms that give more efficiently insights about the health of the application. MODAClouds design-time platform as well as the MODAClouds monitoring platform pushes users to follow these guidance by: (i) allowing the definition of monitoring rules at both the infrastructure and application in a provider-independent way, and (ii) enabling the design of monitoring rules describing how incoming stream of data have to be processed, and what output should be produced when certain conditions have been verified.

Relevant patterns involved: instrumentation and telemetry, loose coupling, multiple-datacentre deployment

In order to support designers in following these guidance and in designing multi-cloud applications they can exploit a set of predefined cloud patterns. Some of these patterns are presented in the following section.
2.2.2 Selected Patterns

We survey below the existing design and architecture patterns that were used in the MODAClouds, namely:

- **Broker [18]:** Ensure that the components remaining independent one another regarding execution platform, communication protocol, reference, and data representation. The resulting flexibility eases the maintenance and later evolution of the cloud applications.

- **Integration provider/Application proxy [12]:** Encapsulate the technicalities resulting from the integration of two or more third party services into its own component, ensuring loose coupling between them. Cloud applications aggregates third party services, which may be discontinued over time. The integration provider pattern eases the replacement of services through the life-time of cloud applications.

- **Elasticity manager [12]:** Get feedback about the resources usage of component application in order to automatically adjust the underlying platform and infrastructure. The pay-as-you-go idea inherent to Cloud Computing requires getting an accurate picture of the resources one consumes. The elasticity manager helps automatically maintain a consumption that fits the needs.

- **Elastic Load Balancer [12]:** Scrutinize the incoming load of application components, and adjust accordingly the number of active replicas serving the load balancer. As for the elasticity manager, the elastic load-balancer pattern helps tailoring load balancing to the real needs of the system.

In addition, we present below the six patterns that were specifically extended in the MODAClouds projects, namely:

- **External configuration store:** Outsourcing the management of configuration to a separate service in the architecture. Leveraging cloud technologies requires flexibility in deployment and configuration of application components. Setting up an external configuration store brings such flexibility, which in turn, ease the transition to cloud technologies.

- **Leader-Followers:** Dynamically delegate the management of a sub part of the architecture to a separate component. Cloud applications often aggregate several sub system, whose total complexity exceeds the capacity of a single maintenance team. The leader election pattern help reduce this complexity by dividing maintenance responsibility.

- **Runtime reconfiguration:** Built-in the ability to dynamically take into account external changes made to configuration. Leveraging Cloud dynamicity requires to quickly account for changes (more customer, new services, etc.). Runtime reconfiguration is a viable solution for reducing the response time in a cloud setting.

- **Provider adapter:** Ensure that the implementation of a component remains agnostic to any specific cloud provider. As cloud technologies mature, application may need to transition from one provider to another. The provider adapter pattern ensures that such transitions will not trigger any unexpected yet significant reworking of the application components.

- **CPIM IaaS cloud offering pattern (cf. D4.2.1):** At the CPIM level, the goal of the IaaS cloud offering pattern is to represent the general structure of an Infrastructure-as-a-Service cloud environment, without expressing in detail the features of cloud services offered by the available cloud providers. This catalogue of resources expressed in D4.2.1 as resource models is particularly useful as a basis for the specification of cloud-related models.

- **CPIM PaaS cloud offering pattern (cf. Section 2.1.2.3):** At the CPIM level, the goal of the IaaS cloud offering pattern is to represent the general structure of a Platform-as-a-Service cloud environment, without expressing in detail the features of cloud services offered by the available cloud providers. This catalogue of resources expressed in D4.2.1 as resource models is particularly useful as a basis for the specification of cloud-related models.

Appendix B provides a detailed view of each of the four firsts extended patterns (the two other are presented in D4.2.1 and in Section 2.1.2.3, respectively).
Conclusion

In this deliverable we presented how the models initially defined in D4.2.1 have evolved especially in order to support the modelling of application exploiting PaaS solutions. In addition, this deliverable presents a set of cloud patterns and guidance that have been used or extended in the context of MODAClouds and help mitigate various pitfalls when designing multi-cloud applications. Finally, appendices provide detailed descriptions of the defined models and metamodels applied to the Constellation Server case study.
Bibliography

Appendices
Appendix A

A  Applying MODACloudML to the Constellation Server case

2.3 Use case: Constellation Server

The Constellation Server prototype (Project Management Server case study) is an advanced repository to store models defined using Modelio, a modelling tool. The server provides advanced functionalities such as models storage in remote repositories, models versioning management, organization of collaborative work around these models, and supports the execution of high resource consuming processing operations on these models.

As depicted in Figure 20, this prototype is organized around two main types of components: the Administration Services and the Agents Services. The Administration Services are web-based applications which combine the administration and configuration features for all the services provided by the Constellation platform. Based on a JEE infrastructure, this service is one of the most important interfaces between the clients and the application. On their side, the External Agents are independent applications that provide specific high resource consuming services to the Project Management Server prototype (e.g. Remote Model Fragment Agent, Modelling Conference Agents). These agents can be deployed on demand on specific cloud instances (IaaS or PaaS depending on their implementation).

![Figure 20 Overall architecture of the Constellation server.](image)

2.4 Design Alternative and deployment models

A CloudMLModel consists of CloudMLElements, which can be associated with Property and Resources. A Resource represents an artefact (e.g., scripts, binaries, configuration files, etc.) adopted to manage the deployment life-cycle (e.g., download, configure, install, start, and stop). The three
The main types of CloudMLElements are Component, Communication, and Hosting. Figure 21 presents the types that have been defined using the MODACloudML IDE in order to deploy the Constellation Server.

A Component represents a reusable type of component of a cloud-based application. A Component can be an ExternalComponent, meaning that it is managed by an external Provider (e.g., the Constellation Data, see Listing 6), or an InternalComponent, meaning that it is managed by CloudML (e.g., the JEE container or the administration service). This mechanism enables supporting both IaaS and PaaS solutions through the single abstract concept of component. The property location of ExternalComponent represents the geographical location of the data centre hosting it (e.g., location="eu-west-1", short for West Europe). The properties credentials and serviceType represent the authentication information needed to access this specific service and its type (e.g., database, application container), respectively.

An ExternalComponent can be a VM (e.g., a virtual machine running GNU-Linux, see Listing 7). The properties minCores, maxCores, minRam, maxRam, minStorage, and maxStorage depict the lower and upper bounds of virtual compute cores, RAM, and storage, respectively, of the required virtual machine (e.g., minCores=1, minRam=1024). The property OS represents the operating system to be
run by the virtual machine (e.g., OS="ubuntu"). All these constraints are optional and do not have to be
defined in the CPIM.

```java
vm AdministrationNode{
    provider: Flexiant, os: "ubuntu", is64
    ram: 1024.., cores: 1.., storage: 50..
    securityGroup: "Constellation", sshkey: "cloudml", groupName: "Constellation"
    provided host AD
}
```

Listing 7 An example of VM type

**Components** are connected through two kinds of ports. A **CommunicationPort** represents a
communication interface of a component. A **CommunicationPort** can be a **ProvidedCommunication**,
meaning that it provides a feature to another component (e.g., the Administration service provides a
JMS interface, see Listing 8), or a **RequiredCommunication**, meaning that it consumes a feature from
another component (e.g., the Administration server requires a JDBC interface, see Listing 8). Only
internal components can have a **RequiredCommunication** since they are managed by CloudML. The
property **isLocal** represents that the component requesting the feature and the component providing
the feature have to be deployed on the same external component. The property **isMandatory** of
**RequiredPort** represents that the **InternalComponent** depends on this feature (e.g., the Administration
server depends on Constellation data and hence Constellation data has to be deployed before the
Administration server, see Listing 8).

A **HostingPort** represents a hosting interface of a component. A **HostingPort** can be a **ProvidedHost**,
meaning that it provides hosting facilities (i.e., it provides an execution environment) to another
component (e.g., a virtual machine running GNU/Linux provides hosting to a JEE container, see
Listing X), or a **RequiredHost**, meaning that an internal component requires hosting from another
component (e.g., the Administration server requires hosting from a JEE container, see Listing 8).

```java
internal component AdministrationService {
    resource adServiceResource{
        download: "wget -P ~ script.sh",
        install: "sudo bash script.sh"
    }
    provided communication jmsProvided {port: 0}
    required communication jdbcRequired {port: 0, mandatory
    required host jeeRequired { ("language" : "Java")
}
```

Listing 8 An example of internal component

A **Communication** represents a reusable type of communication binding between a **Required**- and a
**ProvidedCommunication** (e.g., the Administration server communicates with the Agent manager using
JMS on the default port since the property port is set to 0, see Listing 9). A **Communication** can be
associated with **Resources** specifying how to configure the components so that they can communicate
with each other.

```java
communication adService2agent{
    from AgentManager.jmsRequired
    to AdministrationService.jmsProvided,
    resource adService2agentResource: {
        download: "wget -P ~ script.sh",
        configure: "sudo bash script.sh"
}
```

---

Public Final version 1.0, Dated 30/09/2014 34
A Hosting represents a reusable type of hosting binding between Required- and a ProvidedHost (e.g., a JEE container is contained by a virtual machine running GNU/Linux, see Listing 10). A Hosting can be associated with Resources specifying how to configure the components so that the contained component can be deployed on the container component.

`execution JEE2VM{`  
`from JEEContainer.adRequired to AgentNode.AD`  
`}`

These types can be instantiated in order to form an assembly of components that specifies a deployment model. Each instance is identified by a unique identifier and refers to a type (see Listing 11). The deployment model of the Constellation server example can then be specified as depicted in Figure 22.

Listing 9 An example of communication type

Listing 10 An example of execution type.

Finally, at the CPSM level the deployment model is refined with provider specific information as depicted in Figure 23.
Figure 23 A CPSM of the project management case study
B Extended Cloud Patterns

The ‘External Configuration Store’ Pattern

Within MODAClouds the configuration of an application not only includes the properties associated to the functional behavior of an application but also the configuration (including deployment) of the underlying infrastructure. Accordingly, we extended the configuration store pattern to encompass the overall information needed for the deployment and configuration of a multi-cloud application.

- **Intent:** Outsourcing configuration and deployment information for any particular components or services into a separate service.
- **Known as:** Managed configuration [12]
- **Motivating Example:** In the Project Management Server case study, one might need to replicate an agent in order to improve performances. Instantiating a new constellation agent without modifying the implementation implies that all the configuration and deployment information have to be externalized.
- **Applicability:** This pattern is especially relevant when:
  - the application contains several instances of the same component (or group of components), whose configuration must be synchronized.
  - the configuration of the various components will have to be dynamically adjusted to accommodate various load and/or usage patterns;
  - similar reconfigurations need to be triggered on several parts of the application.
- **Structure:**

![Configuration Store Diagram]

- **Consequences:** To exploit a configuration store implies the following:
  - applicative component must have been designed to interact with the configuration store. This may impact the implementation of each components;
  - introduction of a single point of failure.
- **Sample implementation:** Various technical solutions are available to build a configuration store:
  - Ad-hoc database/service, can provide applicative components with proper configuration information. For instance, in the MODAClouds context, all deployment and configuration resources are managed in an external file server. CloudML extensively relies on this server to orchestrate the deployment and configuration.
  - General configuration management tools such as ESCAPE for instance.
- **Known uses:** CloudML, Puppet, Chef.
The ‘Leader-Followers’ Pattern

Within MODAClouds the leader-followers pattern can be used to elect a component responsible for configuring and managing sub part of the execution environment of an application. A master node has the knowledge of the other peers (sometimes including their status) and generates or tune their configuration or deployment accordingly.

• **Intent**: Delegating the management (including configuration of the execution environment) of a component or a group of components to a specific one to reduce the complexity of the overall management.

• **Known as**: Leader election pattern \[11\]

• **Motivating Example**: The storm application used in the Smart City case study enable the delegation of processing activities to a collaborative set of “worker” components. Increasing the number of worker permit to distribute and absorb workload peaks. However, deploying and managing a large number of workers is a complex task.

• **Applicability**: This pattern is especially relevant when:
  o the application contains numerous instances of the same component (or group of components) whose configuration and deployment must be synchronized or simultaneously updated.

• **Structure**:
  o One component takes the responsibility to coordinate, manage, or configure other components

• **Consequences**
  o The leader election pattern reduces the complexity of managing the overall application, by delegating to specific component the management of the related subsystems.
  o The system should provide mechanisms to dynamically select the leader, or a new leader in case of outage or failure otherwise it introduces a single point of failure.
  o The loss of control to local components may hinder to capacity to finely orchestrate complex management activities at the system scale. Synchronizing local and global management activities may also be more difficult.

• **Sample implementation**: In the case of deploying a storm cluster as done in the Smart City Urban Safety Planner use case, CloudML registers each “supervisor” node (followers) into the “nimbus” (leader), and then generates the relevant configuration file used to configure each supervisor.

• **Known uses**: Storm, CloudML

The ‘Runtime Reconfiguration’ Pattern

Within MODAClouds the runtime reconfiguration pattern is extended to the dynamic adaptation of the application deployment using the models@runtime architecture. This architecture, by synchronizing a
representation of the deployment of an application with the actual application enables third-parties to adapt only selected parts of the deployment.

- **Intent**: Dynamically reconfigure applicative components, application frameworks, and execution environments to minimize the downtime in a production setting.
- **Motivating Example**: In the Project Management Server case study, one might need to replicate an agent in order to improve performances without impacting the running application.
- **Applicability**: This pattern is especially relevant when an application or the deployment of an application needs to be reconfigured dynamically at runtime, such as adapting logging policies, updating database connections, or deploying new services.
- **Structure**
  - A key point in the runtime reconfiguration pattern is to give applicative components the ability to detect and integrate external changes in their configuration, without having to remotely restart them.
  - Another key point consists in maintaining a representation of the configuration and deployment causally connected to the running system.

- **Consequences**
  - The mechanisms needed to detect and apply external changes in configuration may have a significant impact on the design of applicative components.
  - Monitoring the status of the running system might impact its performances.
- **Sample implementation**
  - CloudML maintains a MODACloudML deployment model causally connected to the running system. On the one hand, any modification to the CPIM will be reflected in the CPSM and, in turn, propagated on-demand onto the running system. On the other hand, any change in the running system will be reflected in the CPSM, which, in turn, can be assessed with respect to the CPIM. This way, by exploiting the MODACloudML deployment model, the models@runtime environment seamlessly bridges the gap between the runtime and design-time activities.
- **Known uses**: CloudML
The ‘Provider adapter’ pattern

The provider adapter pattern is particularly relevant in the context of managing multi-cloud application. Within MODAClouds, this pattern has been applied to the MODACloudML supporting tools and extended to the language itself through the concept of cloud provider-independent models that can be automatically or semi-automatically refined into cloud provider-specific models.

- **Intent**: Ensure that application components do not directly rely on the services offered by cloud providers, but rather leverage upon abstract APIs to reduce the coupling between application and cloud providers.

- **Motivating Example**:
  o Automating the deployment of Project Management case study requires provisioning several VMs and configuring the associated database. This database is not meant to be coupled to a specific cloud provider and one may wish to be able to reuse deployment script regardless of the targeted cloud provider. The Provider adapter pattern addresses such coupling between cloud application and cloud provider.

- **Applicability**: This pattern is especially relevant when:
  o application components are not written for a specific single cloud provider, and may move to or across other providers for maintenance reasons for instance.

- **Structure**:
  o Application components use an abstraction of the services offered by several cloud providers: the provider adapter. It is thus possible to move components from one cloud to another by simply updating the service end-points between the adapter.

- **Consequences**
  o Postpones the selection of a cloud provider later in the development cycle.
  o Forces the identification of non-standard services offered by only a few providers, and whose usage may tightly couple the user to the provider.

- **Sample implementation**:
  o Design high level interfaces abstracting the provider-specific services to deploy the database solution on various clouds.

- **Known uses**:
  o The jclouds library provides a provider adapter a Java library so that application can use cloud service without relying on provider specific services. Jclouds supports more than 20 clouds providers.
  o The MODAClouds CPIM library.
  o CloudML.