Melodic
Big data cloud

Title:

Provider agnostic interface definition & mapping cycle

Multi-cloud Execution-ware for Large-scale Optimised Data-Intensive Computing

H2020-ICT-2016-2017
Leadership in Enabling and Industrial Technologies; Information and Communication Technologies

Grant Agreement No.:
731664

Duration:
1 December 2016 -
30 November 2019

www.melodic.cloud

Deliverable reference:
D4.1

Date:
09 April 2019

Responsible partner:
UULM

Editor(s):
Daniel Baur

Author(s):
Daniel Baur, Daniel Seybold

Approved by:
Ernst Gunnar Gran

ISBN number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D4.1 Provider agnostic interface definition & mapping cycle

Executive summary:

This deliverable presents the Executionware component of the Melodic project. The tasks of the Executionware are: (a) the allocation of resources from a heterogeneous multi-cloud environment, (b) the usage of those resources to deploy and run (data processing) tasks and (c) monitoring the runtime context of the running tasks.

This document focuses on the provider agnostic interface used to abstract syntactic and semantic differences in the cloud providers' APIs and the required mapping to translate the agnostic interface to concrete implementations on the providers' side. In addition, it presents a first draft of the resource management layer, focusing on resource advertisement to Melodic's Upperware. Finally, the deliverable gives an outlook for a refined resource management layer and the data processing layer that will span on top of it.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Document	
Period Covered	M1-16
Deliverable No.	D4.1
Deliverable Title	Provider agnostic interface definition & mapping cycle
Editor(s)	Daniel Baur
Author(s)	Daniel Baur, Daniel Seybold
Reviewer(s)	Gregoris Mentzas, Marcin Prusiński
Work Package No.	4
Work Package Title	Executionware
Lead Beneficiary	Ulm University
Distribution	PU
Version	1.0
Draft/Final	Final
Total No. of Pages	36

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table of Contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

List of Figures

List of Tables

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

1 Introduction

The purpose of this deliverable is to depict the design and the implementation of Melodic's Executionware. As seen in the overview of Melodic's architecture in Figure 1, the Executionware fulfills four main tasks: (a) it provides a cloud-agnostic interface to access features of multiple cloud providers in a harmonized way, (b) it delivers resource management capable of allocating and managing resources from those providers, (c) it supplies a data processing layer on top of resource management able to execute the user's defined processing tasks and (c) it, finally, provides monitoring services gathering runtime information of the managed resources and deployed tasks.

Within the context of the Melodic project, the Executionware has two major points of interaction: (a) the Adapter component of the Upperware and (b) the API offered by the cloud providers. For the interaction with the Upperware, the Executionware provides a RESTful API giving the Upperware access to its resource management capabilities and monitoring services. On cloud provider side, the Executionware implements a provider agnostic interface that is then mapped to the data format of the respective cloud provider allowing the Executionware to allocate and manage resources across multiple providers.

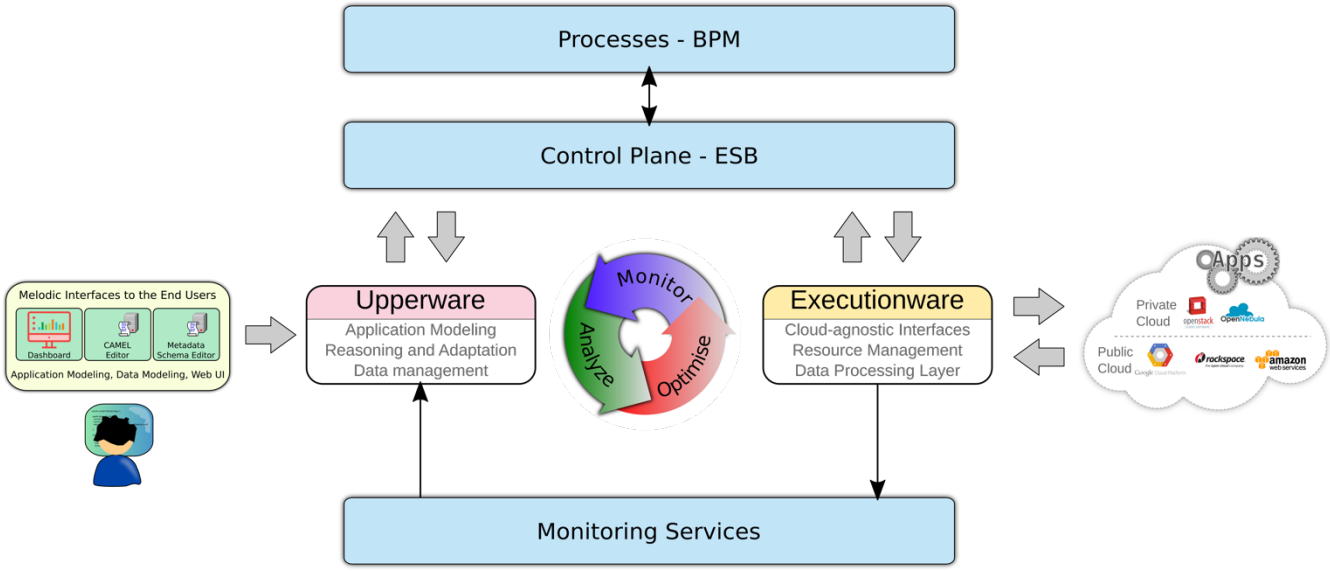

Figure 1: Melodic Architecture [1]

1.1 Scope of the document

This document is intended for the general audience that is interested in how the Executionware achieves cloud agnostic resource management across multiple cloud providers. The work of this deliverable depends on the System Specification D2.1 [2] and the Architecture and Feature Definitions D.2.2 [1] deliverables.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

1.2 Structure of the document

The rest of this document is structured as follows: Section 2 discusses related work in the domain of the Executionware. Afterwards, Section 3 derives the features needed to enable the Executionware's task of managing resources across multiple cloud providers. Section 4 shortly presents the architecture of the Executionware. Next, Section 5 discusses the implementation of the Executionware while Section 6 depicts its integration and documentation. Finally, Section 7 gives an outlook for future work before Section 8 will conclude the deliverable.

2 Related Work

Due to the continuous evolvement of Cloud computing the heterogeneity of offered Cloud service models as well as the actual Cloud provider APIs complicate the orchestration of distributed applications in a multi-cloud environment [3][4]. Yet, the results of the Cloud research community address this heterogeneity and offer unifications for different Cloud service models [5]. In the following, we revisit existing solutions and outline their adoption and required extensions in Melodic in order to enable the orchestration of data-intensive applications in a multi-cloud environment. First, we introduce approaches for the mapping of Infrastructure as a Service (IaaS) and second for Platform as a Service (PaaS) Clouds. Afterwards, we depict concepts that work across those Cloud levels. Finally, we introduce concepts for the unified management of heterogeneous resources, such as physical machines, virtual machines or containers.

2.1 IaaS Mapping

With the adoption of Cloud computing and the growing amount of Cloud providers, the need for a unified representation of IaaS resources is realised by the Cloud research community and standardization bodies in order to prevent users from vendor lock-in. Further, a unified resource mapping eases the deployment of multi-cloud application scenarios and provides even more flexibility to the application owners in terms of application adaptations like scaling or migration.

In this respect, different standards and model specifications try to tackle the Cloud abstraction on the level of resource definition. The Open Cloud Computing Interface (OCCI)[1] targets the definition of an API for Cloud resources with the focus on IaaS. There have been some attempts to implement OCCI on private Clouds (e.g., OpenStack[2] or rOCCI [6]), but wide adoption and commercial usage is still missing.

[1] http://occi-wg.org/
[2] http://occi-wg.org/2012/07/18/occi-in-openstack/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

The Cloud Infrastructure Management Interface (CIMI)[3] provides a model for the management of interactions between an IaaS provider and the service consumer. Attempts for CIMI to be implemented in OpenStack and the retired Apache Deltaclouds[4] have been performed but are already inactive and therefore wide adoption is missing as well.

Yet, concepts from these approaches have been adopted by the Cloud community, resulting in a set of abstraction layer libraries for different programming languages. These libraries provide the mapping of the providers resource offerings to generic resource templates. In addition, they support a subset of the providers' storage APIs. Well adopted libraries are Apache jclouds[5] for Java, Apache Libcloud[6] for Python and Fog[7] for Ruby. All of these libraries provide a single interface to users abstracting all the IaaS provider-specific characteristics. Through such an abstraction layer, multi-cloud application deployment is enabled through facilitating the provision and deployment of IaaS resources.

In order to enable adaptive multi-cloud deployment for data-intensive applications, the usage of such abstraction libraries is not sufficient as the orchestration of the applications across multiple Cloud providers [7] [8] is not in their scope. Yet, the libraries provide the tool to build orchestration tools on top of them to enable multi-cloud orchestration.

Cloud orchestration tools typically rely on abstraction libraries but are able to manage the deployment of the whole application as well as the complete lifecycle of the involved resources [7]. The latter two capabilities are usually covered by a dedicated Domain specific language (DSL) to express the required information. Besides application deployment, orchestration tools may also exhibit application monitoring and adaptation features.

Apache Brooklyn[8] is an orchestration tool for modelling, monitoring, and managing applications through autonomic blueprints that define an application using a declarative YAML syntax, which complies with the CAMP [9] standard and exposes many of the CAMP REST API endpoints.

Cloudify[9] is an orchestration tool that builds upon a TOSCA-aligned modelling language for describing the topology of the application which is then deployed to allocated Cloud resources. As in TOSCA [10], Cloudify splits the blueprint in a type and a template definition. Types define abstract reusable entities that are to be referenced by templates. The types therefore define the structure of the template, by, e.g., defining the properties that a template can have/must provide. The template then provides the concrete values for these types. This mechanism is used for

[3] https://www.dmtf.org/standards/cmwg
[4] https://deltacloud.apache.org/
[5] https://jclouds.apache.org/
[6] https://libcloud.apache.org/
[7] http://fog.io/
[8] https://brooklyn.apache.org/
[9] http://cloudify.co/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

specifying/annotating the nodes as well as the relationships between them within an application topology.

Fully commercial tools such as Scalr[10] build as well upon the aforementioned abstraction libraries but focus only on the usage of public Cloud providers. Further, their closed core impedes accessing and extending their core functionalities.

As existing Cloud Orchestration Tools (COTs) only target a subset of the functionalities of a holistic COT [7] and do not focus explicitly on data-intensive applications, Melodic pursues the established Cloudiator COT [11] [12], which provides multi- and cross-cloud support, monitoring and runtime adaptation mechanisms and native container support. Cloudiator's IaaS abstraction layer builds upon the jclouds library by adding several extensions and enhancements. Further, Cloudiator enables the multi- and cross-cloud deployment of applications by providing a powerful interface for a unified resource mapping, which is introduced in Section 3.

2.2 PaaS Mapping

While the IaaS level focuses on the provisioning of resources to run Cloud applications, e.g., virtual machines, the PaaS level covers the provisioning of resources in conjunction with application-centric run-time environments or application servers. Besides the runtime environment for the actual application, PaaS offers additional services, such as database management systems or load balancers, which can be added to the environments.

Yet, the APIs of existing PaaS offerings tend to be even more heterogeneous [13] than those of the IaaS offerings, which makes the orchestration of services across multiple PaaS providers a very challenging task[11]. A first approach to standardize the deployment of Cloud applications with respect to PaaS is provided by OASIS CAMP [9]. CAMP specifies an interoperable protocol to package and deploy applications and interfaces for self-service provisioning, monitoring, and control. However, the CAMP standard is not yet adopted by popular PaaS providers, such as Heroku[12], OpenShift[13] or CloudFoundry[14].

The heterogeneity of existing PaaS providers is analyzed in [13] and a profile for common capabilities for PaaS offerings is presented. In this scope, a model is derived, which represents the three main aspects of PaaS offerings: infrastructure, platform and management. Moreover, a fine-grained classification of PaaS into IaaS-centric, generic and SaaS-centric PaaS is derived, based on the level of provided control mechanisms.

[10] https://www.scalr.com/
[11] https://paasfinder.org/vendors
[12] https://www.heroku.com/
[13] https://www.openshift.com/
[14] https://www.cloudfoundry.org/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Similar to the IaaS mapping approaches, there are approaches for PaaS that provide a uniform PaaS API for multiple PaaS providers. The COAPS API [14] defines a unified description allowing a PaaS provider independent representation of applications based on a model for the application and its environment. Furthermore, the COAPS API provides a REST API for the life-cycle management of the application (e.g., *createApplication*, *destroyApplication*, etc.) that internally maps the calls to the APIs of the actual chosen PaaS providers.

A similar approach is followed by Nucleus [15], providing also a PaaS provider agnostic API supporting four PaaS providers. Nucleus focuses on the separation between platform- and application-centric API interactions and offers, compared to the COAPS API, additional management features, such as scaling, monitoring of applications or the management of services and deployment region.

The PaaS Unified Library (PUL) [16] adopts the concept of providing a unified PaaS API via a REST service or as a library. PUL supports a comprehensive set of resource and application management operations, such as *deploy, undeploy, start, stop, bind service* or *scale application*.

A first approach towards a PaaS orchestration tool is presented by PaaSHopper, a middleware for orchestrating application across multiple PaaS providers. Similar to the previously introduced approaches [13]–[16] PaaSHopper middleware enables the composition of multiple application components running at different PaaS providers into one application. Yet, PaaSHopper only focuses on the PaaS service model.

Within Melodic, Cloudiator's revised modular architecture enables the integration of different PaaS mapping approaches. A first implementation is building upon the PUL library, which provides the required feature set and allows an easy integration. Further details are described in Section 3.1.2.

2.3 Cross-Level Mapping

While the previously introduced mapping concepts of mapping only focus on a dedicated Cloud service level, recent advances in the Cloud research also target the mapping across the IaaS and PaaS service level. A first approach for cross-level mapping is introduced by [17], identifying the challenges in cross-level API mapping and the orchestration of applications. Further, a preliminary COT is presented, which exploits CloudML [18] models to unify different Cloud service levels and enable cross-level orchestration.

Another cross-level mapping and orchestration approach is introduced in [19] which extends the IaaS orchestration tool Apache Brooklyn[15] with the capabilities to additionally orchestrate applications over PaaS services. Applications are described in provider-agnostic TOSCA models [10]. While [19] only focuses on the deployment aspect of cross-level orchestration, an extended

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

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

version of the respective tool focuses as well on the adaptation aspect of cross-level orchestration [20]. In this context, [20] introduces an algorithm for migrating applications between IaaS and PaaS providers. Yet, cross-level monitoring and additional level-specific adaptation actions in different levels of abstraction are not supported. For instance, even at the IaaS level, support for component scaling is missing. Based on the above analysis, the available cross-level COTs only support a subset of the desired COTs features [7] for cross-level mapping and orchestration. Hence, Cloudiator will enable the execution of complex adaptation actions for multi- and cross-cloud scenarios.

2.4 Resource Management

Resource management has been thoroughly studied in different contexts like Cluster, Grid or Cloud Computing. Apache Mesos [21] uses a two level scheduling approach, splitting the resource management to a central unit and multiple frameworks. The central scheduler offers each framework a set of resources and leaves it to the framework to select the best-fitting one. It uses the Dominant Resource Fairness (DRF) algorithm [22] to achieve fairness in a multi-tenancy environment, where multiple tenants compete for the resources.

Apache Aurora[16] and Marathon[17] build upon Mesos, providing support for long-running services and containers.

Google Borg [23] uses a central resource manager especially designed for handling large clusters sizes. In contrast to Mesos' DRF algorithm, Borg uses quotas and priorities for the scheduling decisions.

High Performance Computing (HPC) resource managers like Moab[18], TORQUE[19] or SLURM[20] in general use large backlogs (scheduling queues) to achieve high utilization.

In contrast to existing solutions that typically target the scheduling on a static, prior known set of resources with objectives like fairness across multiple users and high utilization, Cloudiator aims at scheduling on a dynamic set of resources that are acquired on demand.

3 Features

To be able to deploy distributed applications in a heterogeneous multi-cloud environment, multiple features are required that are derived from the Melodic's System Specification D2.1 [2] and Initial Feature Definition D2.2 [1]. These features are presented in this section.

[16] http://aurora.apache.org/
[17] https://mesosphere.github.io/marathon/
[18] http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-suite-enterprise-edition/
[19] http://www.adaptivecomputing.com/products/open-source/torque/
[20] https://www.schedmd.com/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

First, a provider agnostic interface is required to hide the syntactic and semantic differences between the different cloud providers. Additionally, mapping logic needs to exist, capable of mapping the agnostic interface to the different API implementations of the providers. Cloudiator features this layer for IaaS as well as PaaS clouds.

Second, one needs to be able to supply a provider agnostic description of the application that needs to be deployed. This is the task of our job description framework.

Additionally, a resource management layer is required, capable to use the provider agnostic interface and mapping layer to allocate matching resources. We rely on an offer-based approach were a matchmaking entity, e.g., the Upperware of Melodic, can retrieve a set of node advertisements and select the best-matching node candidate.

Having allocated the resource using the resource management layer, the Executionware's deployment logic is responsible for deploying the described application on the given resources.

Finally, a monitoring layer measures the current deployment context, reporting runtime deviations to interested entities.

3.1 Provider agnostic interface & mapping

As different cloud providers and especially different cloud models still differentiate their service based on their API, supporting multiple cloud providers requires a common mapping of the different APIs. For this purpose, the Executionware features a provider agnostic interface mapping. Its task is to harmonise the APIs of multiple providers to a common interface, thus abstracting those differences from other components. This abstraction is split into two different layers, based on the supported cloud levels: IaaS and PaaS.

3.1.1 IaaS

The IaaS layer is responsible for providing a compute service abstraction for IaaS clouds. It is split into two main services: (i) the *Compute Service* providing actions to manipulate compute resources, e.g., virtual machines and (ii) the *Discovery Service* allowing the discovery of compute entities required to create virtual machines. Both interfaces are depicted in Figure 2 on page 13. The implementations of those interfaces are done in so called drivers, each providing the unique mapping required for the API of a provider. A list of supported cloud providers can be found in Table 2.

The compute service interface offers two main operations: (a) the *createVirtualMachine()* operation used for both creating and starting virtual machines by referencing the image, the hardware (flavor) and the location used for creating the virtual machine and (b) the *deleteVirtualMachine()* operation used for deleting prior created virtual machines. As the virtual machine needs to be accessed using a remote connection (SSH or WinRM), it is the contract of the *createVirtualMachine()* to also assign a public IP Address if necessary and attach credentials

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

required for login. Additionally, the *Compute Service* offers a link to retrieve the corresponding *Discovery Service*.

The *Discovery Service* interface provides methods for: *i)* retrieving a single entity of the discovery model and methods for *ii)* retrieving all available entities of one type. The entities that are discoverable and their relationships are shown in the discovery model depicted in Figure 3 while the single entities are explained in detail in Table 1.

The relationships between Location, Image and Hardware depict an "is-available-in" relationship, as Locations are typically independent installations of the same cloud management software meaning that not all images resp. hardware may be available in all locations of a cloud provider. To ensure uniqueness of IDs in a multi-cloud environment, the original identifier issued by the cloud provider is replaced with a globally unique but stable ID while the original ID is retained in the providerId attribute.

To provide pricing information and increase the availability and quality of the meta-data attached to the discovered offerings, we connect to meta-data services like CloudHarmony[21] providing this information for multiple cloud providers.

[21] https://cloudharmony.com/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

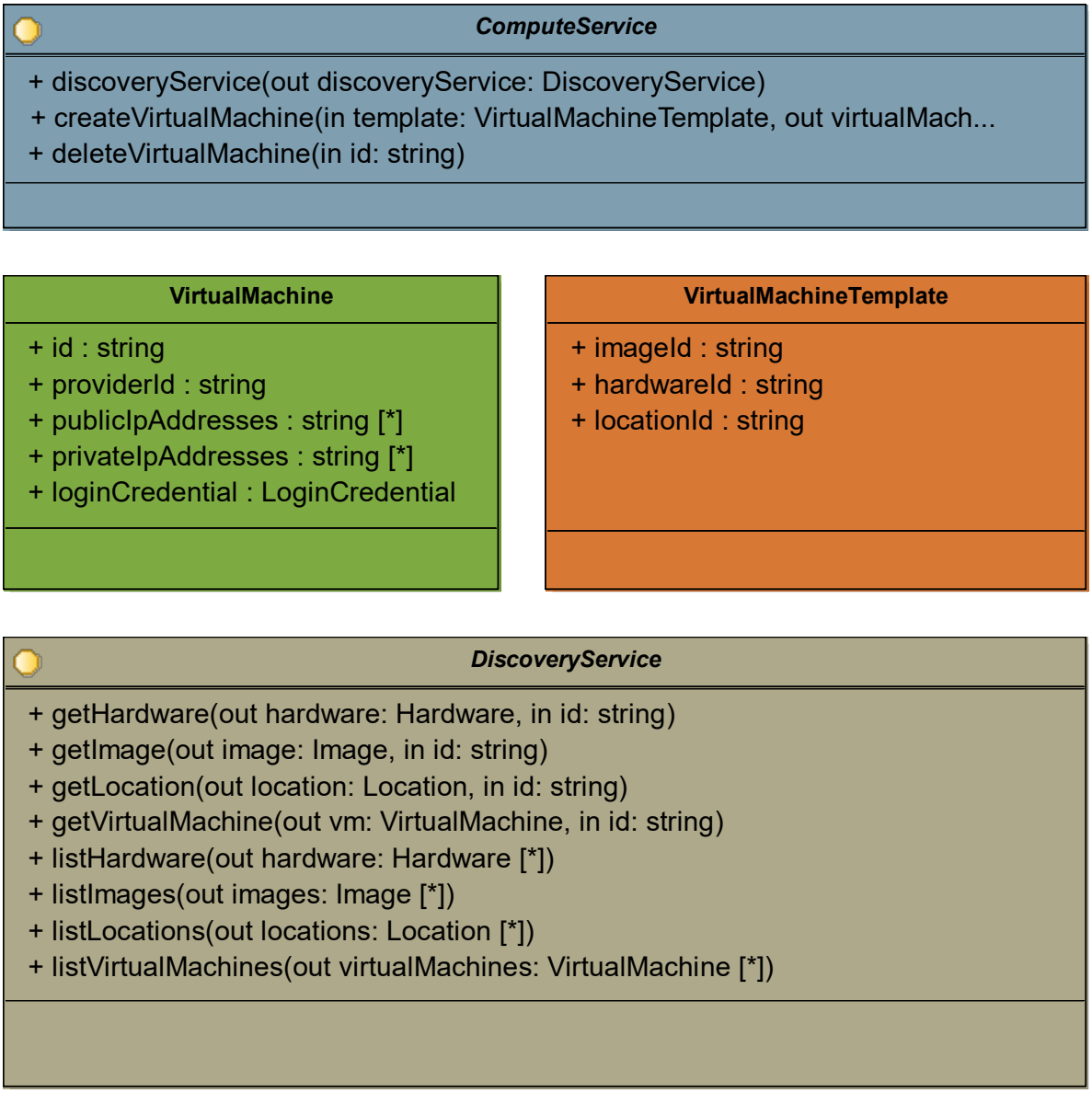

Figure 2: ComputeService and DiscoveryService Interface

Table 1: IaaS Compute Entities

Entity	Description
Hardware	Defines the computational resources of a node by number of cores, disk space and RAM.
Image	Represents the basic setup of a node, i.e. the operating system. The operating system is defined by its version, its architecture (32- or 64-bit), its family (e.g., Ubuntu) and its type (e.g., Unix).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Location	Represents (virtual) locations offered by a provider. They are stored in a hierarchical relationship with the scopes host, (availability) zone and region. It also contains geographical information like country or latitude/longitude of the (physical) datacenter.
Price	Defines the price as a function of Location, Image and Hardware as prices typically differ based on Location, Image (license fees) and Hardware. We normalise the price for the duration of one hour and USD.
VirtualMachine	The created virtual machine. Provides access information by exposing its (public/private) IP Addresses and access credentials.

Table 2: Supported Cloud Providers

Provider	**URL**	**Status**
Openstack	https://www.openstack.org/	Stable
Amazon Elastic Compute Cloud	https://aws.amazon.com/ec2/	Stable
Google Compute Engine	https://cloud.google.com/compute	Beta
Microsoft Azure	https://azure.microsoft.com	Alpha
Profitbricks	https://www.profitbricks.de	Alpha

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

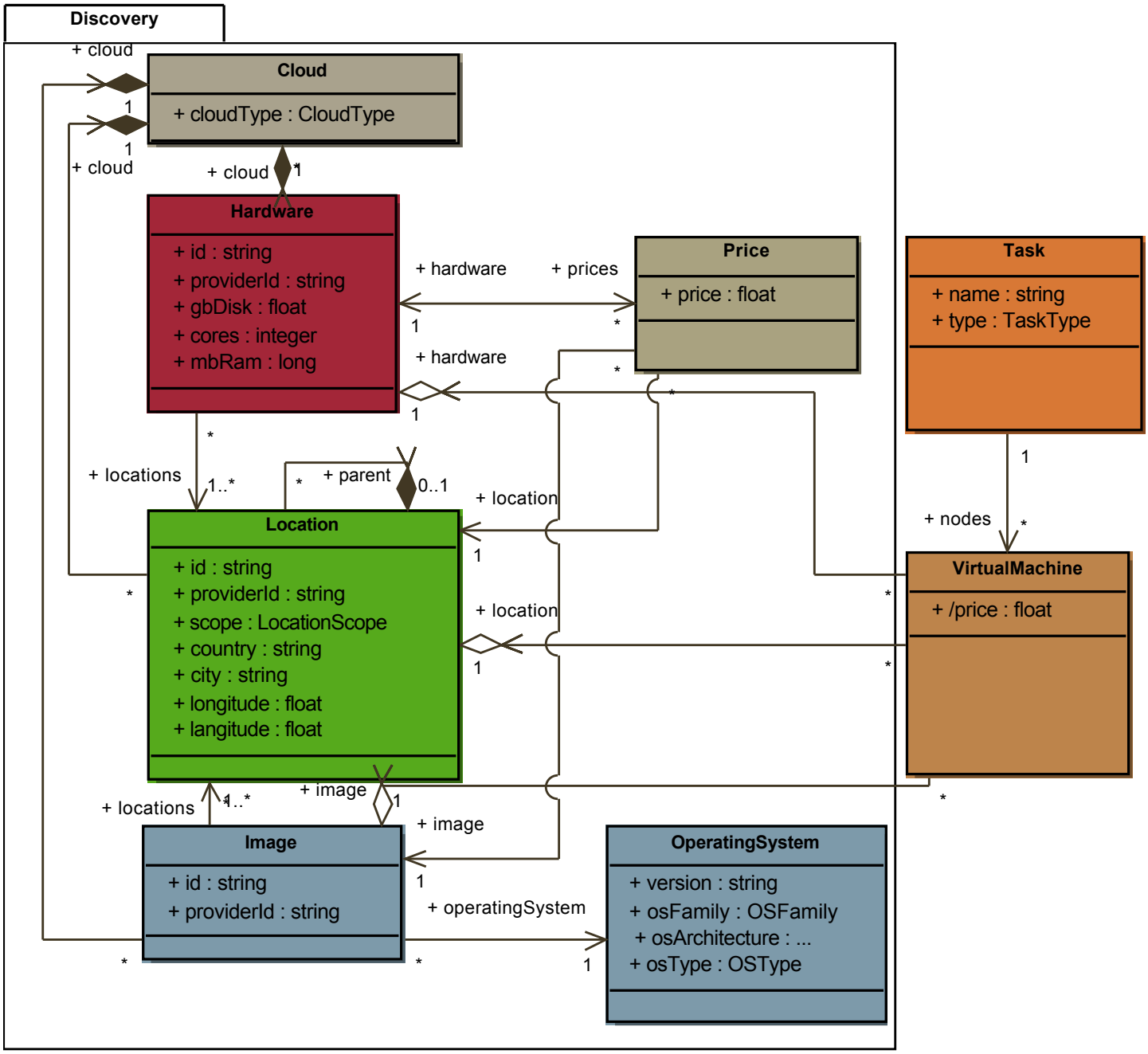

Figure 3: Discovery Class Model

3.1.2 PaaS

The PaaS layer is responsible for providing a platform service abstraction for PaaS Clouds. The *PlatformService* provides the necessary operations to manage *PlatformEnvironments* at the PaaS Clouds via the *createPlatformEnvironment()*, *updatePlatformEnvironment()* and

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

deletePlatformEnvironment() operations as depicted in Figure 4. The interface to interact with a specific platform is enabled via platform drivers. A platform driver can be a platform specific driver, such as OpenShift REST Client[22] for the OpenShift platform[23], or abstraction layers, such as the PaaS Unified Library (PUL) [16]. A list of supported PaaS providers can be found in Table 4. A *PlatformEnvironment* comprises all resource entities that are required to deploy an application at a specific platform, namely the *Platform*, *PlatformHardware*, *PlatformRuntime* and *PlatformService*. A high-level description of these entities is provided in Table 3 and the technical component description is depicted in Figure 4.

While the PaaS landscape is even more heterogeneous than the IaaS landscape [24] [13], there is currently no support for the automatic discovery of *PlatformEnvironments* in Melodic. Hence, each *PlatformEnvironment* and its respective entities need to be created based on a predefined model or configuration file. Yet, services such as PaaSfinder[24] ease the collection of platform specific details to define the *PlatformEnvironments*. For the long term, the usage of PaaSFinder or similar services could be used to enable the automatic discovery of *PlatformEnvironments*.

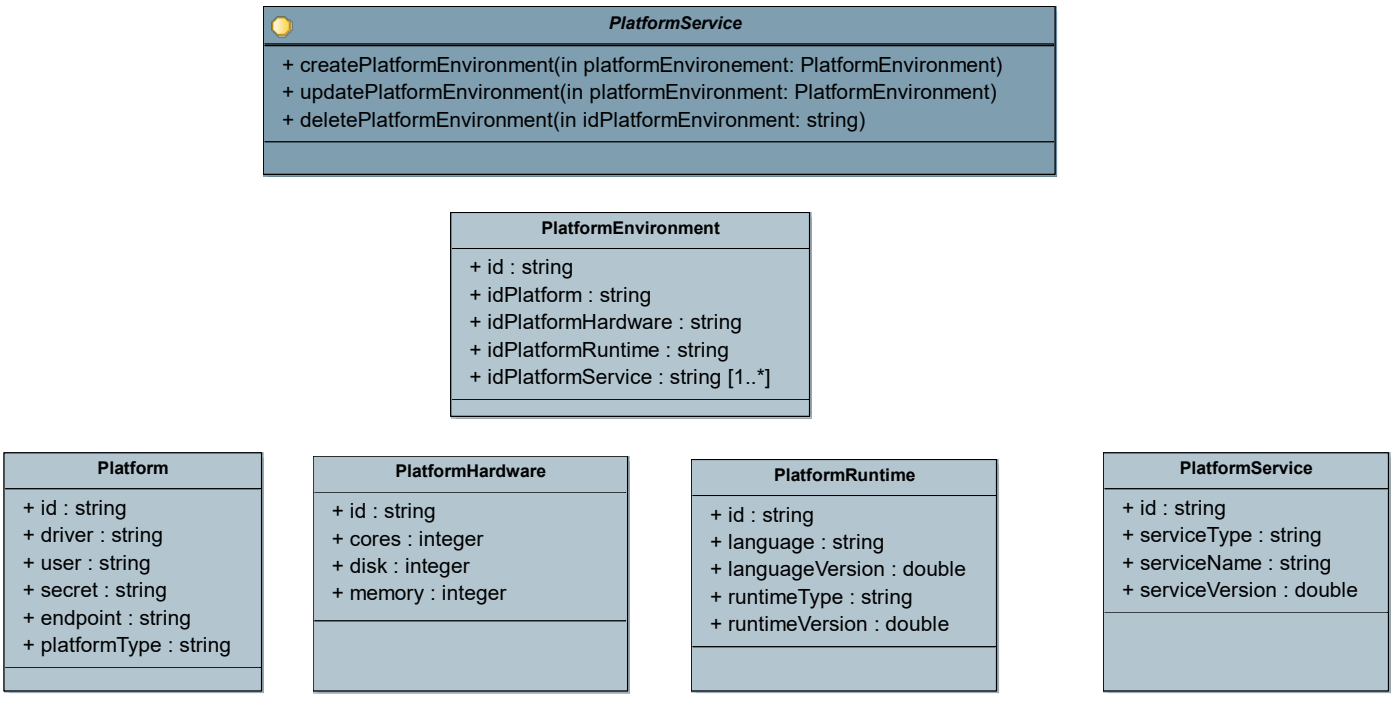

Figure 4: PlatformService Interface and Plaform Entities

[22] https://github.com/openshift/openshift-restclient-java
[23] https://www.redhat.com/en/technologies/cloud-computing/openshift
[24] https://paasfinder.org/vendors

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table 3: PaaS Entities

Entity	Description
Platform	Defines the basic attributes of the PaaS provider, i.e., the PaaS type, API endpoint and version
PlatformHardware	Represent the basic resources of a PlatformEnvironment, i.e. number of cores, disk space and memory. PlatformHardware attributes might be omitted as not all PaaS providers support the full set of PlatformHardware attributes
PlatformRuntime	Represents the supported runtimes of the Platform to deploy applications. Runtimes might be plain runtime environments, such as the JVM, or application containers, such as Tomcat or JBoss.
PlatformService	Represents additional services of the PaaS provider which can be linked to the actual applications. Typical services are databases, load balancers or message queues.

Table 4: Supported PaaS Providers

Provider	URL	Runtimes	Services	Status
Heroku	https://www.heroku.com/	Java, PHP	ClearDB	Stable
OpenShift Online	https://www.openshift.com/	Java	MySQL	Stable
OpenShift Origin	https://www.openshift.org/	Java	MySQL	Beta
CloudFoundry	https://www.cloudfoundry.org/	Java, PHP	-	Beta
Pivotal	https://pivotal.io/	Java, PHP	-	Alpha
IBM Bluemix	https://www.ibm.com/cloud-computing/bluemix	Java	-	Alpha

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.2 Job Description

To be able to deploy applications and data processing jobs in a multi-cloud environment, the Executionware requires a description of the entities it should deploy. This description includes the artifact (e.g., an executable) to be deployed, (communication) dependencies between entities in case of a distributed application, and requirements depicting the resource demands for selecting the resource on which the entity will be deployed.

We, therefore, use a three-layered approach for representing the user's application: (i) *Jobs* which represent a logical group of several (ii) *Tasks* which describe the executable artifact and its properties. Finally, (iii) a *Process* depicts an instantiation of a task, representing the process running on the resource. The entities of the job description framework are depicted in Figure 5.

A task can define multiple interfaces that depict the deployment process required for it. We currently offer (i) a *LifecycleInterface* where the user describes actions to be executed during specific steps of the Task's lifecycle, e.g., while installing or starting it, (ii) a *DockerInterface* where the user can refer to a Docker image, (iii) a *SparkInterface* describing a Spark data processing job and (iv) a *PlatformInterface* deploying the task on a PaaS-platform.

A task also defines communication dependencies to other Tasks of the same Job, by expressing if it either provides a port to another Task or requires a port of another Task.

We also differentiate between two TaskTypes: batch and service. While batch represents a task that runs once and then exits, a service represents a task that is running infinitely.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

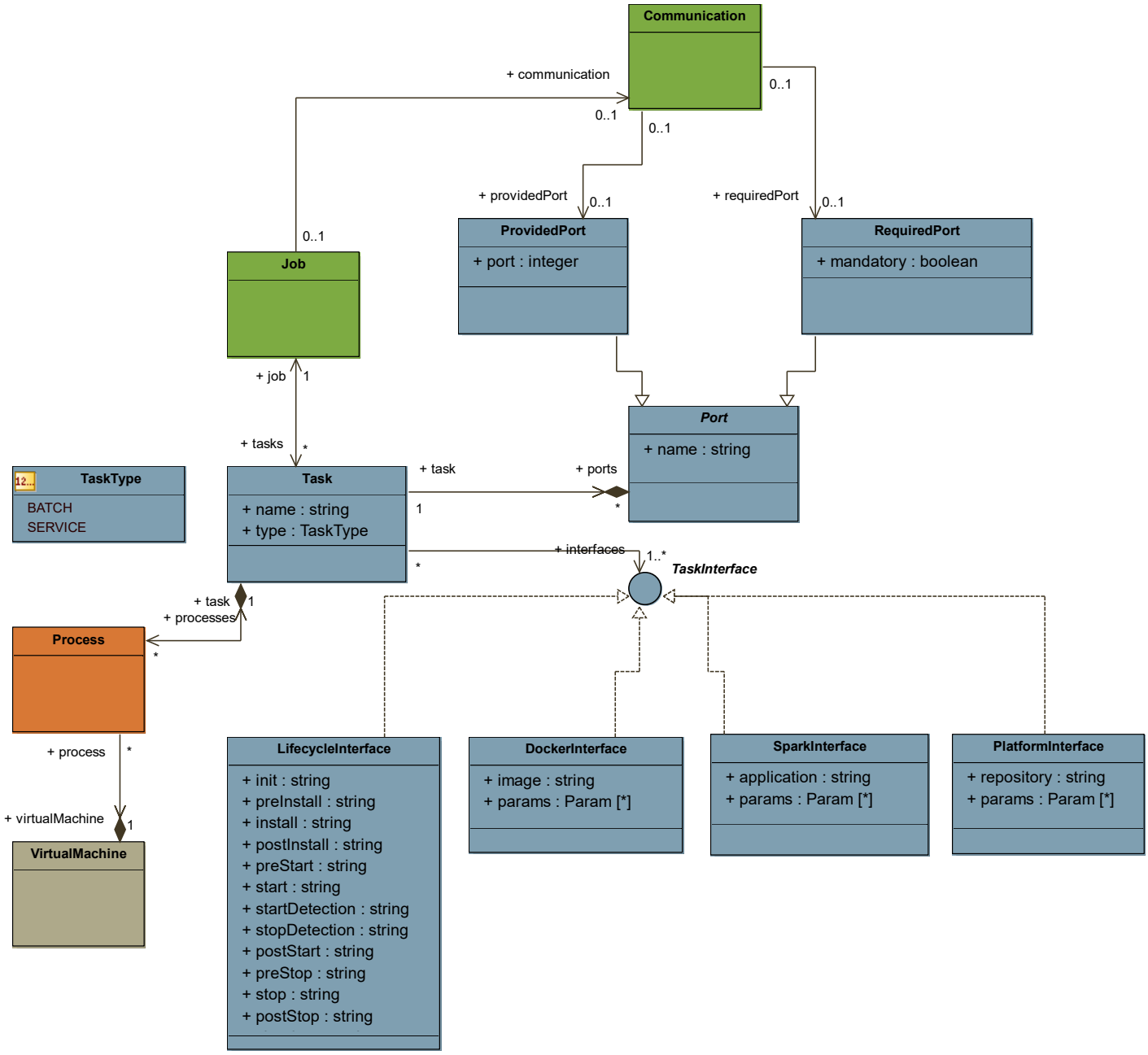

Figure 5: Job Description Framework

3.3 Resource Management

The tasks of the resource management layer of the Executionware are to (i) advertise all possible resources it is capable to create or has already created so that a matchmaking service can select fitting resources for the demand defined by each Task (see Section 3.2 on page 18) and (ii) allocate the selected resource from a multi-cloud environment. The allocated resource will then be passed to the deployment layer responsible for the Task deployment on this resource.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.3.1 Resource Advertisement

To be able to select a matching offer, the possible solution space needs to be known. This is the task of the resource advertisement feature of the Executionware. For this purpose, it relies on the resource discovery mechanism of the provider agnostic interface (see Section 3.1 on page 11), providing information about all possible offer combinations of cloud providers. This information is then used to generate all valid combinations while considering the constraints expressed by the relations, thus, e.g., respecting that a specific Hardware offer is only available in a single Location.

As this solution space is, depending on the number of considered cloud providers, possibly very large, we limit the amount of resources considered, by giving the user the possibility to express requirement constraints targeting each Task he desires to have executed. For example, there may be Tasks that require a specific amount of RAM as they otherwise are not able to start.

The Executionware currently offers three ways to express such requirements depicted in Figure 6: (i) using attribute requirements, (ii) using the object constraint language (OCL[25]) and (iii) identifier requirements where the user can directly select the resource he desires to use.

By using attribute requirements, the user expresses requirements by directly referencing attributes of the discovery model depicted earlier in Figure 3 on page 15. The requirements are expressed in the form

<AttributeRequirement> ::= <Class> <Attribute> <Operator> <Value>.

An example for an attribute requirement that restricts resources to at least four cores would be expressed by *Hardware.Cores >= 4*.

Identifier requirements allow the user to select the concrete offers he wants to use, by expressing the identifiers for Hardware, Image and Location. As a result, exactly one node candidate will be returned.

OCLRequirements use the object constraint language (OCL) for expressing requirements on the object model depicted in Figure 3 on page 15. In contrast to the attribute requirements, this allows specifying more complex expressions. Currently not all possible expressions in OCL are supported. Table 5 gives an overview of supported expressions, their descriptions and an example.

To generate the node candidates the Executionware executes two steps: (i) filtering the base entities (Hardware, Image and Location) by applying the constraints expressed on them and (ii) generating all eligible combinations, considering the constraints expressed by the relationships

[25] http://www.omg.org/spec/OCL/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

of the object model (see Figure 3 on page 15). The generated node candidates are returned to the caller.

Table 5: OCL Requirements

OCL Expression	Description	Example	Explanation
forAll()	Every node needs to fulfill the expressed constraint	nodes->forAll(n | n.hardware.cores >= 4 implies n.hardware.ram >= 4096	Every node that has at least 4 cores needs to also have at least 4096 MB of RAM
exists()	At least one node needs to fulfill the expressed constraint	nodes->exists(location.geoLocation.country = 'DE')	At least one node needs to be located in Germany
select()	Selects the nodes fulfilling the expressed constraint and returns a collection	nodes->select(n | n.hardware.cores >= 4)->size() = 2	Exactly two nodes with at least 4 cores are required
size()	Returns the size of the collection it is applied to.	See above	See above
isUnique()	Enforces that the attribute is unique.	nodes->isUnique(n | n.location.geoLocation.country)	Every node needs to be placed in a different country

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

sum()	Returns the sum of the attribute across all nodes.	nodes.hardware.cores->sum() >= 15	The sum of all cores across all nodes needs to be at least 15.

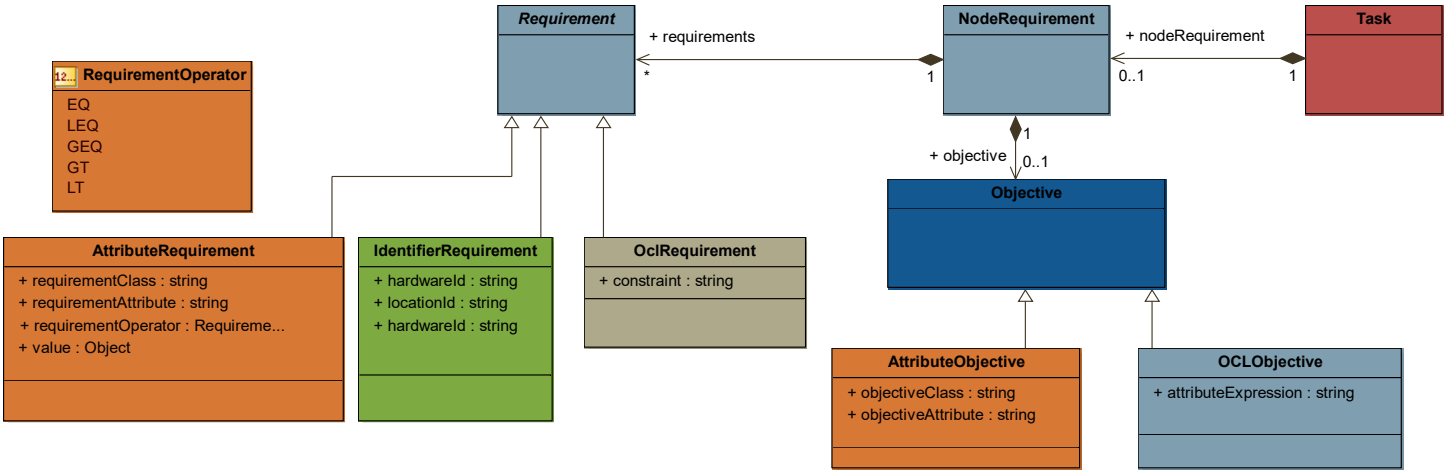

Figure 6: Requirement

3.3.2 Matchmaking / Scheduling

Matchmaking and Scheduling are tasks of Melodic's Upperware. The Upperware will pass the user expressed constraints to the node advertisement logic of the Executionware (see Section 3.3.1) and then receive back the possible node configurations as response. It will then select the best suited configuration and pass it back to the Executionware which will finally start the allocation of the resources and the deployment of the respective tasks.

For testing purposes and to allow a standalone exploitation, the Executionware features a basic matchmaking feature. It allows the user to define an objective that will be optimized. This objective can either be expressed by referring to the attribute that should be minimized/maximized (similar to the attribute requirement) or by giving an OCL expression that should be minimized/maximized (similar to the OCLRequirement). The matchmaking feature receives the possible node configurations generated by the resource advertisement step and will select the candidate minimizing/maximizing the given objective.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

A more detailed description of the requirement description and matchmaking can be found in [25].

3.3.3 Resource Allocation

The resource allocation layer receives the output of the matchmaking step which will represent a concrete cloud offer to use. By relying on the cloud provider agnostic interface depicted in Section 3.1 on page 11 it will allocate the resources from the provider. It will finally install a set of agents on the acquired node, that include a monitoring agent (cf. Section 3.5 on page 24) and the deployment agent (cf. Section 3.4 on page 23).

3.4 Deployment

The deployment feature is responsible for the deployment of the user described tasks (cf. Section 3.2 on page 18) on the resources allocated by the resource management layer (cf. Section 3.3.3 on page 23). We represent tasks that run on a resource by the concept of processes.

We differentiate between four different deployment types represented by the different deployment interfaces the Excecutionware supports: i) *LifecycleInterface*, ii) *DockerInterface*, iii) *SparkInterface* and iv) *PlatformInterface*. Currently only the LifecycleInterface is implemented and the remaining interfaces are left for future work.

The LifecycleInterface describes a task by giving executable scripts that need to be executed at specific points of the task's lifecycle, e.g., while installing, starting or stopping the task. In addition, we use detection scripts for detecting if the task has already started or stopped unexpectedly. We currently support two alternative deployment types for tasks implementing the LifecycleInterface: Docker or Plain. Docker means that the scripts will be executed inside a Docker container allowing isolation if multiple tasks run on the same virtual machine. Plain means executing the scripts directly on the underlying resources, if isolation is not required or the task does not support running inside a Docker container.

The *DockerInterface* will support the direct execution of Docker containers in constrast to the Docker mode of the *LifecycleInterface* where the container is built on-demand from the user's scripts. The *SparkInterface* will be capable of describing Apache Spark applications while the *PlatformInterface* will support PaaS-applications.

Tasks defined within the same Job may have communication dependencies expressed by using the Communication relationship as depicted in Section 3.2 on page 18, declaratively representing a deployment workflow where specific tasks need to be executed before other tasks as tasks providing communication typically need to start before those consuming it. The Executionware is capable of automatically deriving the implied workflow and ensures in-order execution of the described tasks. Currently only basic communication dependencies can be expressed, but could be extended if the need arises [26].

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.5 Monitoring

The Executionware offers a monitoring framework responsible for collecting metrics depicting the current runtime state of all tasks and resources managed by it. For this purpose, it offers a monitoring agent that is deployed on each resource managed by the Executionware.

The monitoring agent has four building blocks as depicted in Figure 7: (i) *Sensors* that collect monitoring information from the underlying resource or the running Task (process), (ii) a telnet interface provided to the running process allowing it to push monitoring data to the monitoring agent, (iii) a reporting interface used to actively publish the monitoring data to, e.g., a time-series database or a message queue and (iv) a REST interface allowing the monitoring agent to be reconfigured during runtime.

While our monitoring agent implements a set of default sensors and reporters, it also offers interfaces allowing the end user to implement different sensors and reporting interfaces. These interfaces are depicted in Figure 8. Each sensor can optionally implement the *init()* method, providing a possibility to configure the sensor to the context of the running machine, but also to some user provided sensor configuration parameters. The *measure()* method will be called at the interval configured by the user. The returned *Measurement* will be enriched with context information and passed as metric to the reporting interface.

The reporting interface is implemented to forward the collected messages to either a (time-series) database or to a message queue where it can be further processed or aggregated. In case of Melodic, the measured metrics will be passed to the event processing agent based on Esper[26] (cf. [1]) where it is aggregated and evaluated by the Upperware's reasoners and scalability rule engine. The reporting interface offers two methods for either reporting single or multiple metrics, as some reporting interfaces can achieve significantly higher throughput if multiple metrics are forwarded at once. The interval at which the reporting interface is called can be configured, and metrics will be queued internally until the next call.

An up-to-date list of supported Sensors and ReportingInterfaces can be found in the documentation of the monitoring agent on Github[27].

[26] http://www.espertech.com/esper/
[27] https://github.com/cloudiator/visor

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

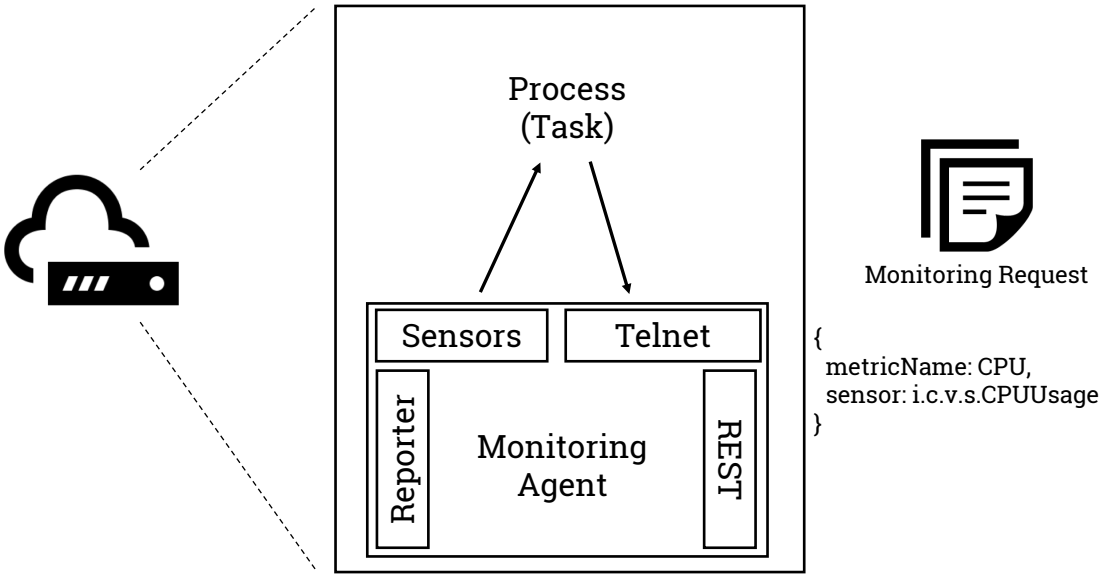

Figure 7: Monitoring Framework

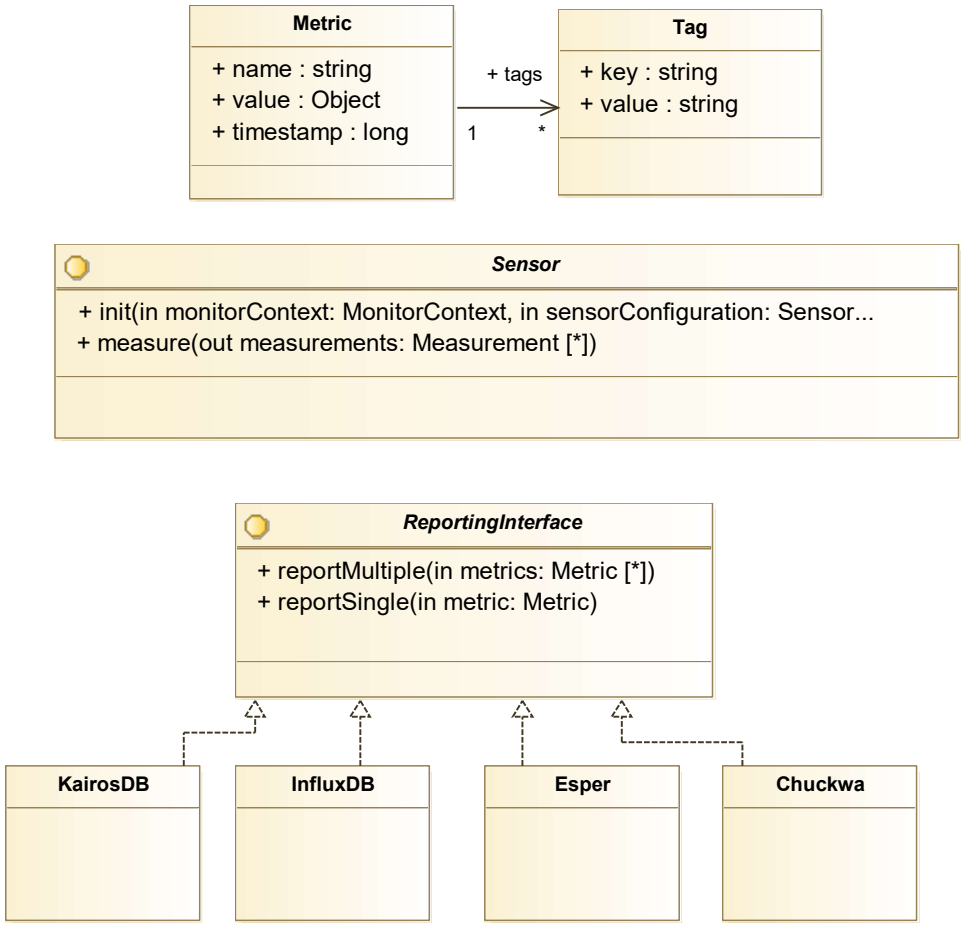

Figure 8: Monitoring Class Diagram

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.6 Adaptation

The Melodic framework implements a feedback loop, where the monitoring data collected by the Executionware is evaluated by the Upperware to check for runtime deviations. If such deviations are detected, the Upperware will derive actions to counteract the deviations. It is the task of the Executionware to enact such actions to the running application. These actions include horizontal scaling (scale-in and -out), vertical scaling (scale-down and -up) but also migration of applications to different resources.

The current implementation maps these actions to basic create or delete operations. This means that e.g. in the case of a migration, a new instance (process) of the task is spawned on a newly created resource and the old process and resources are deleted afterwards. It is left to the user to hide the involved state, meaning that the application scripts need to handle the state transparently for the Executionware, e.g. by copying the state to the newly create process.

While this approach allows to implement above actions, there may cases where better solutions exist. Taking a private cloud as example, migrations inside the cloud could be implemented by relying on the live migration capabilities cloud middleware solutions like Openstack are offering. The usage of such capabilities is left for future work (cf. Section 7.3).

4 Architecture

The introduced provider agnostic interface & mapping (cf. Section 3.1), job description (cf. Section 3.2) and resource management interface (cf. Section 3.3) provide the crucial feature set to orchestrate data-intensive applications in multi-cloud environments by the Executionware. Further, these features are required to enable the novel Resource Management Framework and the upcoming Data Processing Layer in conjunction with comprehensive monitoring and adaptation features in the Executionware. A high-level overview of the Executionware architecture is depicted in Figure 9, which was introduced in D2.2 [1]. As the Cloud orchestration tool (COT) Cloudiator provides the base of the Executionware, the presented concepts enhance Cloudiator in order to provide the required feature set introduced in D2.1 [2].

The entry point to Cloudiator is a REST interface by the *RestServer* as depicted in Figure 9. The interaction with the REST interface is enabled programmatically via API client libraries for multiple programming languages. Consequently, the programmatic interaction is exploited in the Upperware, which is interacting with the REST Server via the API client library. In addition, a Web-based *UserInterface* is provided, which interacts with the *RestServer* internally.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Cloudiator's internal architecture is built upon a message-driven architecture, following the publish-subscribe paradigm. Therefore, each call against the *RestServer* is transformed into a Cloudiator specific message and published to the *KafkaMessageQueue*. For the sake of clarity, the *KafkaMessageQueue* is depicted without the dedicated interaction between all Cloudiator components. Yet, all internal communication (behind the *RestServer*) of Cloudiator relies on messages over the *KafkaMessageQueue*.

The message-driven architecture of Cloudiator enables the smooth integration of the building blocks *Monitoring Services*, *Resource Management Framework* and *Data Processing Layer* into the Executionware, which interact with the Cloud provider via the provider agnostic interface mapping as depicted in Figure 9.

In the following, the technical implementation details of the presented architecture are outlined.

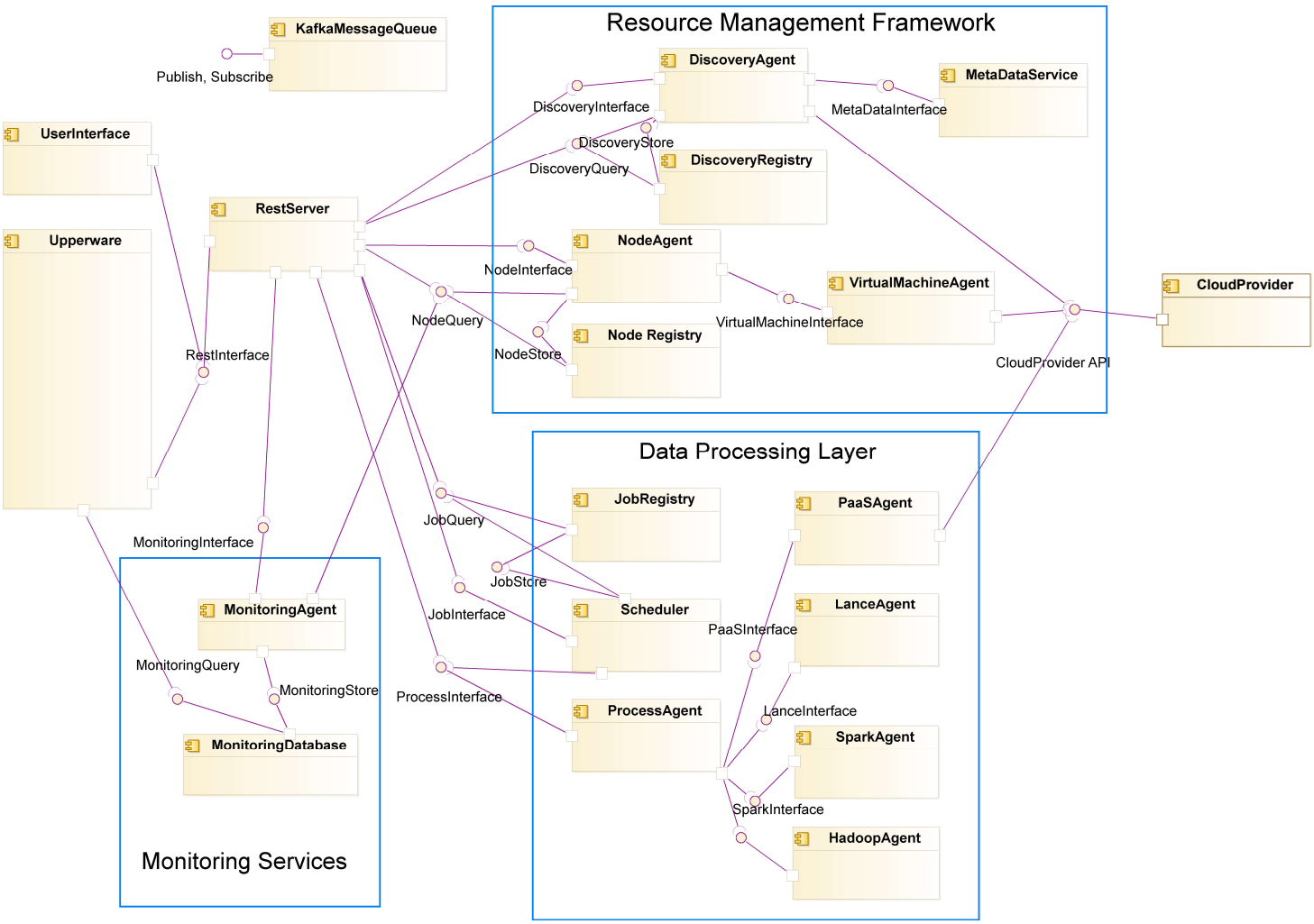

Figure 9: Cloudiator Architecture

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

5 Implementation

The Executionware is implemented using the Cloudiator Framework developed by Ulm University. It is published under the Apache License 2.0. The Cloudiator Framework is mainly developed in the Java programming language using Apache Maven[28] for managing dependencies and building. Other components can interact with Cloudiator using its RESTful web service interface. This interface is described using the OpenAPI[29] specification and the Swagger toolset[30]. This allows automatic generation of clients in many programming language and human-friendly documentation.

The Cloudiator Framework was initially developed in the PaaSage[31] and CACTOS[32] projects as multi-cloud orchestration tool being capable to deploy applications across multiple clouds. Within Melodic we extend Cloudiator to also enable the orchestration of data processing frameworks and the support of cross-level deployment meaning the access to different cloud infrastructure levels like IaaS and PaaS. As the initial Cloudiator was not designed for such a task, multiple architectural changes have been done. While the business logic of the initial Cloudiator framework resided in a single deployable component called *Colosseum* the Melodic version will rely on a micro-service architecture using event-based communication over a message queue. This allows a seaming less integration of additional logic without having to touch existing logic, making it much easier to adopt Cloudiator's functionalities to a new environment.

Furthermore, the original Cloudiator version featured a very simple resource management layer only being able to allocate virtual machines given the provider dependent ids. Additionally, it was designed having infinitely running service-type applications. The resource management layer developed in Melodic will feature a provider agnostic approach as described in Section 3.3. Additionally, it is redesigned to also support batch-type applications as they are typical in a data processing environment.

6 Integration and Documentation

A micro service architecture, as the one depicted in Section 4, increases the need for a well-defined integration strategy, to be able to easily deploy a higher number of components. Additionally, the high velocity of explorative projects requires the possibility to quickly roll out

[28] https://maven.apache.org
[29] https://www.openapis.org
[30] https://swagger.io
[31] https://paasage.ercim.eu/
[32] http://cactos-cloud.eu/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

new versions of the software. Therefore, the integration process of Cloudiator is completely automated using continuous integration to build new versions of the software and containerization for fast and easy deployment.

6.1 Integration

The general integration workflow is depicted in Figure 10. It consists of four domains: (a) the personal development environment of every developer, (b) GitHub as code repository and collaboration environment, (c) Travis CI[33] as continuous integration facility and (d) multiple artefact repositories where build results are stored and can be later retrieved for deployment.

The personal development environment consists of the Java Development Kit (JDK), Apache Maven for dependency management and the build management as well as the Swagger Framework for describing the RESTful web service interface Cloudiator offers and enabling automatic generation of client and server code. In addition, we use git[34] for software configuration management.

To allow multiple users to seamlessly cooperate while developing the Cloudiator framework, GitHub[35] is used to host the code repositories and provide a collaborative environment allowing issue reporting and code reviews.

The continuous integration platform Travis CI[36] is used to automatically build changes that are pushed to the code repositories and the build status is returned and displayed in Github. This allows to detect build errors early and automatically. In addition, Travis CI builds the deployment artefacts and publishes them at the respective repositories.

There are two main artefact repositories: Docker Hub and Maven Central. Docker Hub stores and publishes Docker images for each component of Cloudiator. Those images are later used to easily deploy the Cloudiator toolset. Maven Central is used for storing and publishing Java libraries, so that they can be shared across multiple components using Apache Maven's dependency mechanism or be used by other projects.

Sonarcloud[37] is used to derive quality metrics from the developed code. These metrics include test coverage, code quality, technical depth but also static code analysis features trying to detect common programming errors that can lead to bugs and vulnerabilities.

[33] https://travis-ci.org/
[34] https://git-scm.com/
[35] https://github.com
[36] https://travis-ci.org
[37] https://sonarcloud.io

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

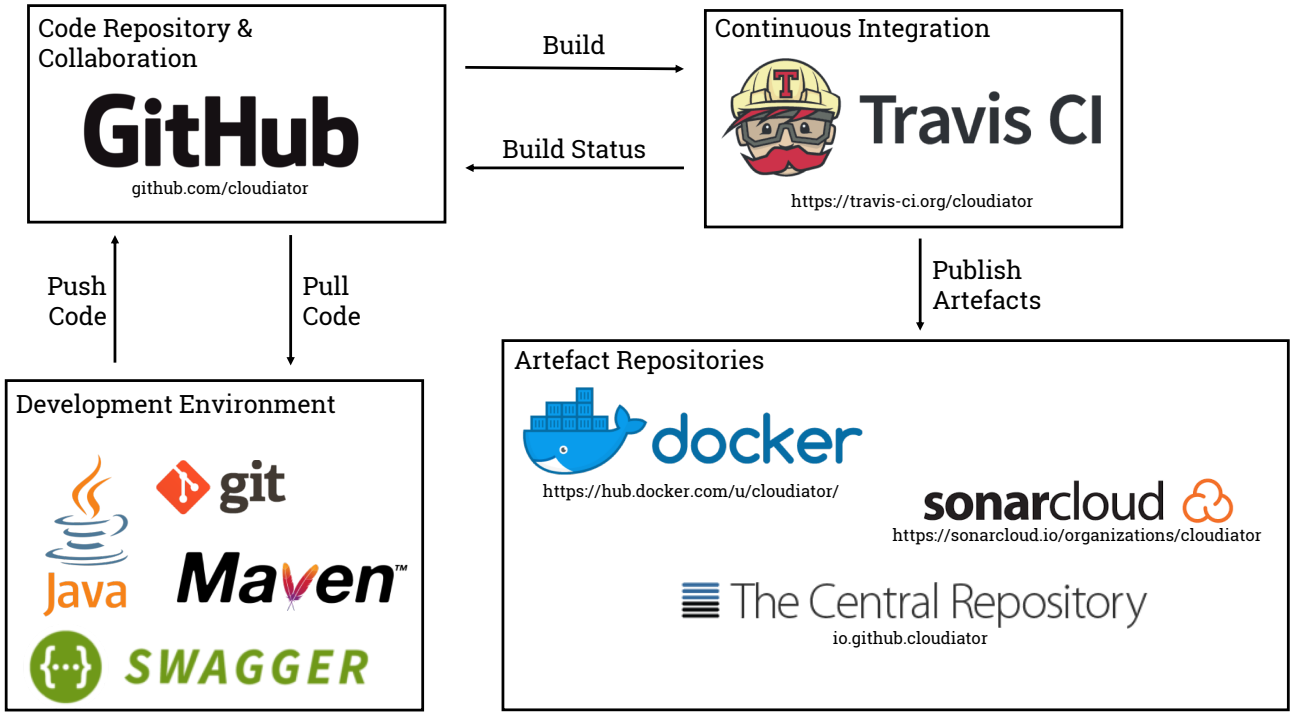

Figure 10: Cloudiator Integration Tools & Workflow

6.2 Documentation

Cloudiator has different sources of documentation, each targeting a different group of users. Table 6 gives an overview of the different documentation sources, a description and their intended target group.

Table 6: Documentation Sources

URL	**Description**	**Target Group**
cloudiator.org	Homepage of the Cloudiator project. General overview of the project, end user documentation including installation instructions, examples, relevant publications and links to other sources.	Users, Researchers
github.com/cloudiator	Organization of Cloudiator on Github. Groups individual repositories of Cloudiator	Developers
github.com/cloudiator/{tool}	Technical documentation of the tool	Developers

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

cloudiator.org/rest-swagger	Detailed documentation of the RESTful API	Developers

7 Future Work

This section briefly discusses future features planned to be added to the Executionware.

7.1 Resource Management

While the resource management layer currently already supports advertisement and allocation of on-demand cloud resources, it does not support using already existing resources. Additionally, to support the upcoming data processing layer (see Section 7.4), a scheduling solution needs to be implemented, allowing the data processing tasks to be easily scheduled and deployed on the managed resources.

With respect to management of existing resources, we plan to implement Bring your own node (BYON) support, allowing users to register nodes within the Executionware, e.g., by extending the RESTful API. Those resources could then be included in the node advertisement and be used for executing tasks. In a subsequent version, registering the nodes could be automated by relying on agents as, e.g., done by Apache Mesos.

To allow scheduling of data processing task, the resource management layer needs to be integrated with existing data processing frameworks like Apache Spark[38] or Apache Map Reduce[39]. For this task, existing scheduling solutions like Apache YARN, Apache Mesos or the Spark Job Server[40] need to be evaluated, and based on the outcome of the evaluation, be integrated into the Executionware.

7.2 Deployment

As described in the deployment section 3.4, only tasks that describe the lifecycle interface are currently supported for deployment. For the next release iteration, the support of Docker containers (described by the *DockerInterface*) is planned. While Docker is in general already supported by the lifecycle agent, the usage of already existing Docker images needs to be implemented. The support of data processing tasks, like Apache Spark, will be implemented

[38] https://spark.apache.org/
[39] http://hadoop.apache.org/
[40] https://github.com/spark-jobserver/spark-jobserver

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

closely related to the integration of the resource management layer with data processing frameworks (see Section 7.1) and the data processing layer (see Section 7.4).

7.3 Adaptation

The introduced adaptation capabilities of the Executionware (cf. Section 3.6) currently map all adaptation actions to basic create and delete actions. In future work, we want to exploit cloud middleware capabilities like live migration. For this purpose, we will develop an adaptation service implementing different strategies to scale or migrate tasks. This service will receive the current and the intended configuration and rely on the strategy pattern to derive possible adaptation strategies to adapt the current state to the new intended state. These strategies will e.g. include the usage of live migration strategies offered by cloud middleware. As fallback strategy, the current logic will remain.

7.4 Data Processing Layer

The introduced provider agnostic interface mapping of the Executionware provides the technical base to the Upperware for enabling the management of data-intensive applications in a multi-cloud environment. While the provider agnostic interface mapping enables the access to the heterogeneous Cloud resources, the *Resource Management Layer* (c.f. Section 3.3) of the Cloudiator will allocate these resources in an optimized way for the actual data-intensive applications. The *Data Processing Layer* will enhance data-intensive application by providing native support for the required Big Data Processing frameworks [2] in conjunction with typical application types such as web servers, application servers or databases.

While web servers, application servers and databases are orchestrated by the *LifecycleInterface* for IaaS or the *PlatformInterface* for PaaS resources (cf. Section 3.4), it is also necessarys to integrate the required Big Data processing framework Apache Spark as outlined in [2] and additional frameworks such as Apache Map Reduce if required. The integration of such Big Data processing frameworks will be realized via the respective *ProcessAgents* as depicted in Figure 9. The modular and event-driven architecture of the Executionware eases the integration of additional Big Data processing frameworks as well, if required. Hereby, the Executionware will orchestrate the processing framework clusters and provide an interface to the Upperware to submit the specific data-intensive processes.

8 Conclusion

In this deliverable, we have shown the initial features of the Executionware. Specifically, we have presented (i) the provider agnostic mapping of existing Cloud provider offerings, (ii) the initial

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

draft of the resource management focusing on the node advertisement logic and (iii) the monitoring framework.

Furthermore, we have depicted the concrete implementation of the Executionware and its integration procedure and referenced available documentation required for using the Executionware.

Additionally, we have depicted Future Work that is required to provide functionality needed for the final release. We plan to develop a data processing layer capable of handling user defined data processing tasks, requiring an integration of the resource management layer with data processing frameworks.

Bibliography

[1] Yiannis Verginadis *et al.*, 'D2.2 Architecture and Initial Feature Definitions', Melodic Project Deliverable, Feb. 2018.

[2] Yiannis Verginadis *et al.*, 'D2.1 System Specification', Melodic Project Deliverable, Jun. 2017.

[3] Y. Elkhatib, 'Mapping Cross-Cloud Systems: Challenges and Opportunities', in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016.

[4] J. Opara-Martins, R. Sahandi, and F. Tian, 'Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective', *J. Cloud Comput.*, vol. 5, p. 4, Apr. 2016.

[5] S. Kachele, C. Spann, F. J. Hauck, and J. Domaschka, 'Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking', in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, 2013, pp. 75–82.

[6] B. Parák and Z. Sustr, 'Challenges in Achieving IaaS Cloud Interoperability across Multiple Cloud Management Frameworks', in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 404–411.

[7] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and others, 'Cloud Orchestration Features: Are Tools Fit for Purpose?', in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 2015, pp. 95–101.

[8] D. Baur and J. Domaschka, 'Experiences from Building a Cross-cloud Orchestration Tool', in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, New York, NY, USA, 2016, pp. 4:1–4:6.

[9] A. Karmarkar, 'CAMP: a standard for managing applications on a PaaS cloud', in *Proceedings of the 2014 Workshop on Eclipse Technology eXchange*, 2014, pp. 1–2.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

[10] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, 'TOSCA: Portable Automated Deployment and Management of Cloud Applications', in *Advanced Web Services*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. New York, NY: Springer New York, 2014, pp. 527–549.

[11] J. Domaschka, D. Baur, D. Seybold, and F. Griesinger, 'Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine', in *9th Symposium and Summer School on Service-Oriented Computing*, 2015.

[12] D. Baur and J. Domaschka, 'Experiences from building a cross-cloud orchestration tool', 2016, pp. 1–6.

[13] S. Kolb and G. Wirtz, 'Towards Application Portability in Platform as a Service', in *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, 2014, pp. 218–229.

[14] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, 'PaaS-Independent Provisioning and Management of Applications in the Cloud', presented at the Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on, 2013, pp. 693–700.

[15] S. Kolb and C. Rock, 'Unified Cloud Application Management', in *Services (SERVICES), 2016 IEEE World Congress on*, 2016, pp. 1–8.

[16] A. J. Ferrer, D. G. Pérez, and R. S. González, 'Multi-cloud Platform-as-a-service Model, Functionalities and Approaches', *Procedia Comput. Sci.*, vol. 97, pp. 63–72, 2016.

[17] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, 'Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems', in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, 2013, pp. 887–894.

[18] G. Goncalves *et al.*, 'CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds', in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, pp. 399–406.

[19] J. Carrasco, J. Cubo, F. Duran, and E. Pimentel, 'Bidimensional Cross-Cloud Management with TOSCA and Brooklyn', in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, 2016, pp. 951–955.

[20] J. Carrasco, F. Durán, and E. Pimentel, 'Component-wise Application Migration in Bidimensional Cross-cloud Environments':, in *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, 2017, pp. 287–297.

[21] B. Hindman *et al.*, 'Mesos: A Platform for Fine-grained Resource Sharing in the Data Center', in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2011, pp. 295–308.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, 'Dominant Resource Fairness: Fair Allocation of Multiple Resource Types', in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2011, pp. 323–

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

336.

[23] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, 'Large-scale Cluster Management at Google with Borg', in *Proceedings of the Tenth European Conference on Computer Systems*, New York, NY, USA, 2015, pp. 18:1–18:17.

[24] S. Kolb and G. Wirtz, 'Data Governance and Semantic Recommendation Algorithms for Cloud Platform Selection', in *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, 2017, pp. 664–671.

[25] D. Baur, D. Seybold, F. Griesinger, Masata, Hynek, and J. Domaschka, 'A Provider-agnostic Approach to Multi-cloud Orchestration using a Constraint Language', presented at the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2018).

[26] J. Domaschka, F. Griesinger, D. Baur, and A. Rossini, 'Beyond Mere Application Structure Thoughts on the Future of Cloud Orchestration Tools', *Procedia Comput. Sci.*, vol. 68, pp. 151–162, 2015.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664