

### 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 Number  
731664

Duration

1 December 2016 –  
30 November 2019

[www.melodic.cloud](http://www.melodic.cloud)

Deliverable reference  
D4.4

Date

30 November 2019

Responsible partner  
UULM

Editor(s)  
Daniel Baur

Author(s)  
Daniel Baur, Volker Foth, Florian Held, Florian Lappe, Łukasz Szymański

Reviewers  
Ernst Gunnar Gran, Gregoris Mentzas

Distribution  
Public

Approved by  
Gregoris Mentzas

ISBN number  
N/A

Document URL

<http://www.melodic.cloud/deliverables/D4.4 Resource Management Framework.pdf>

#### Executive summary

The purpose of this deliverable is to show the evolvement of the prototype of the resource management framework to the final product. For this purpose, it depicts the feedback received by the testers and use cases of the project and depicts the taken steps to address this feedback. It thus presents the newly developed features integrated to address this feedback, and in addition maintenance effort done to improve existing features.



Document	
Period Covered	M20-32
Deliverable No.	D4.4
Deliverable Title	Resource Management Framework
Editor(s)	Daniel Baur
Author(s)	Volker Foth, Florian Held, Florian Lappe, Łukasz Szymański
Reviewer(s)	Ernst Gunnar Gran, Gregoris Mentzas
Work Package No.	4
Work Package Title	Executionware
Lead Beneficiary	Ulm University
Distribution	PU
Version	1.0
Draft/Final	Final
Total No. of Pages	29

## Table of Contents

1	Introduction .....	4
1.1	Scope of this document .....	4
1.2	Structure of this document .....	4
2	Feedback to the prototype .....	6
3	Features .....	7
3.1	Monitoring .....	7
3.1.1	Monitoring Model .....	9
3.1.2	Monitoring Orchestration .....	10
3.1.3	Scaling Engine .....	11
3.2	Bring Your Own Node (BYON) .....	12
3.2.1	Functionality .....	12
3.2.2	Usage .....	13
3.3	User Interface .....	14
4	Implementation .....	18
4.1	License .....	18
4.2	Main Dependencies .....	18
4.3	Source Code Repositories .....	18
5	Maintenance .....	19
5.1	Updates to the previous prototype .....	19
5.1.1	Refined Docker Interface .....	19
5.1.2	Parallelization .....	23
5.1.3	Matchmaking .....	23
5.1.4	Cost Discovery .....	24
5.2	Other Improvements .....	25
6	Documentation .....	26
7	Integration .....	27
8	Summary .....	28
9	References .....	29

# 1 Introduction

## 1.1 Scope of this document

This document describes the final product of the resource management layer of the Executionware work package. As it describes the evolution of the previous prototype it builds upon the deliverable D4.3 [1]. Moreover, this deliverable and part of the described technical functionalities will serve as a basis for D4.6 Data processing layer.

## 1.2 Structure of this document

In Section 2, this document will first give an overview of the received feedback about the prototype described in deliverables D4.1 [2], D4.3 [1] and D4.5 [4]. It includes a) feedback given by the use case providers adapting their applications using the resource framework and b) feedback given by the testing team. The remainder of the deliverable will discuss actions that were executed to address the feedback given by those users. First, Section 3 will discuss new major features of the resource management layer that were introduced to address feature requests by the end-users. Section 4 will then elaborate on formal and technical dependencies of the Cloudiator Framework like the used software licence and used third-party software. Afterwards, Section 5 will discuss general maintenance effort executed to address the feedback, including smaller improvements to existing components.

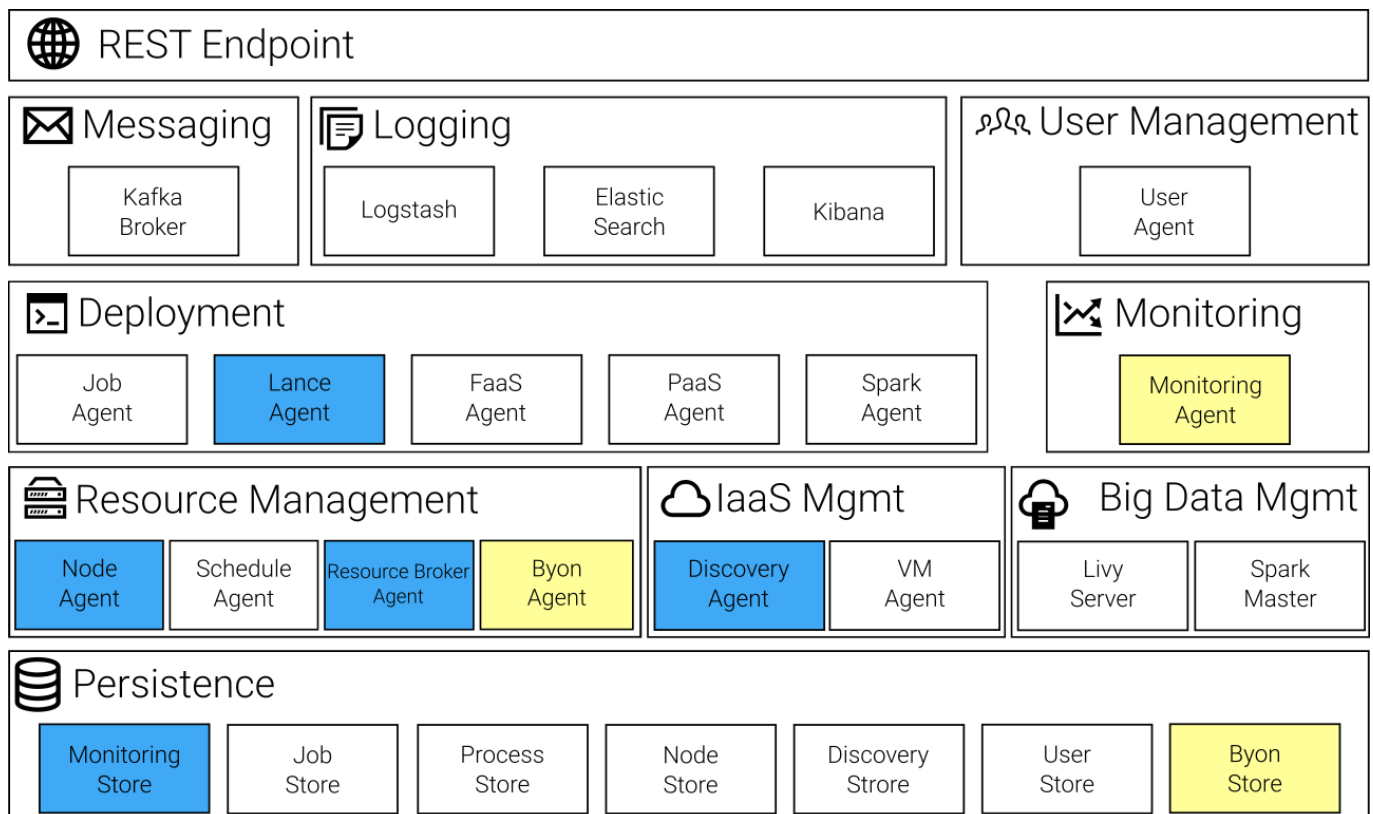


Figure 1: Revised Cloudiator Architecture

Figure 1 shows the corresponding changes in the overall Cloudiator. architecture. Yellow boxes are associated with newly developed features, whereas blue boxes imply components that underwent the mentioned maintenance process. The new Monitoring Agent gives the Cloudiator user the opportunity to configure the monitoring settings for the deployed application by exposing appropriate interfaces to the REST Endpoint. The Byon Agent makes it possible for the user to register his or her own nodes, e.g. a virtual server, that don't necessarily stem from a cloud provider, in the system. Maintenance of the Lance Agent included improvements in communication capabilities of the associated Docker Support feature. The Node Agent now is able to allocate resources in parallel and the Resource Broker Agent now finds resources for an optimal deployment in much shorter time. Moreover, the newly developed Cost Discovery mechanism associated with the Discovery Agent, makes it now possible to gather cloud-specific pricing-lists in an agnostic, consistent way. All these mentioned new implementations, improvements and maintenances of the components are discussed in depth in Section 3 through 5.

## 2 Feedback to the prototype

In Table 1, feedback to the previous prototype, presented in D4.3 [1], that was either delivered by a single use-case partner, the use-case partners as a whole or the developers of the prototype itself, is listed. These feedbacks were considered and the resulting requests for new feature implementations or improvements were eventually implemented in the new prototype and will be discussed in Section 3 and 5. The first column in the table lists feedback IDs and associated reference names, whereas the second column gives a brief description of the feedback.

*Table 1: Feedback to the Prototype*

ID	Description
F1 – Monitoring	The monitoring framework that only existed as a draft needed implementation.
F2 – Docker Communication	The Docker Support, first described in D4.5 [4], needed to be expanded with respect to communication capabilities for the deployed containers. These capabilities were required in the 'CRM and App Store' use case originally described as Use Case 1 in D6.1 [5].
F3 –Deployment Parallelization	Parallelization of the resource allocation and deployment to speed up the overall deployment process.
F4 – Resource Offering Speed	The time required to perform the resource offering and the internal matchmaking needs to be improved.
F5 – Bring Your Own Node (BYON)	In some cases, the use case partners wanted to reuse some already allocated nodes with the Cloudiator Framework. For this purpose the BYON feature was developed.
F6 – User Interface	To ease the testing and usage of the Framework, a user interface to perform basic actions was requested.
F7 – Stable Release	The integration needed to be changed to provide an additional, more stable, release version.

## 3 Features

This section will discuss new features introduced with the final version of the resource management framework. Those features mainly include: i) the final implementation of the monitoring layer responsible for monitoring the allocated resources (F1), ii) the Bring-Your-Own-Node features that allows the usage of Cloudiator with already existing nodes (e.g. residing at unsupported providers or physical node not managed by a cloud software) (F5) and iii) a user interface allowing manual interaction with Cloudiator's API (F6), e.g. for error handling and debugging.

### 3.1 Monitoring

This section describes the monitor-handling and -orchestration of Cloudiator. The complete scientific approach of the monitoring is expounded and published in the corresponding paper [3]. The monitor-handling is part of the complete orchestration structure of Cloudiator shown in Figure 2, which visualizes the orchestrational interactions inside Cloudiator. The monitoring part mainly takes place between the Monitor-Orchestration-Service and the Monitoring-Agent called Visor. Visor is a lightweight software client for monitoring installed on a virtual machine. The Event Processing Agent (EP-Agent) handles the outgoing communication of the nodes. This includes the aggregated data of Visor as well as the orchestrational feedback of a node and triggers events based on the deployment rules. EP-Agent is not in scope of monitoring and is described in paper [3] and deliverable D3.4 [6]. Cloudiator offers the possibility to orchestrate monitors with two types of sensors called Push- and Pullsensor. Pullsensors are the equivalent to traditional hardware related Systemsensors monitoring metrics like CPU-Usage or Memory-Usage. Pushsensors are application specific, custom metrics handled over a specific port. This port can be specified in the monitor request or is randomly chosen by Visor.

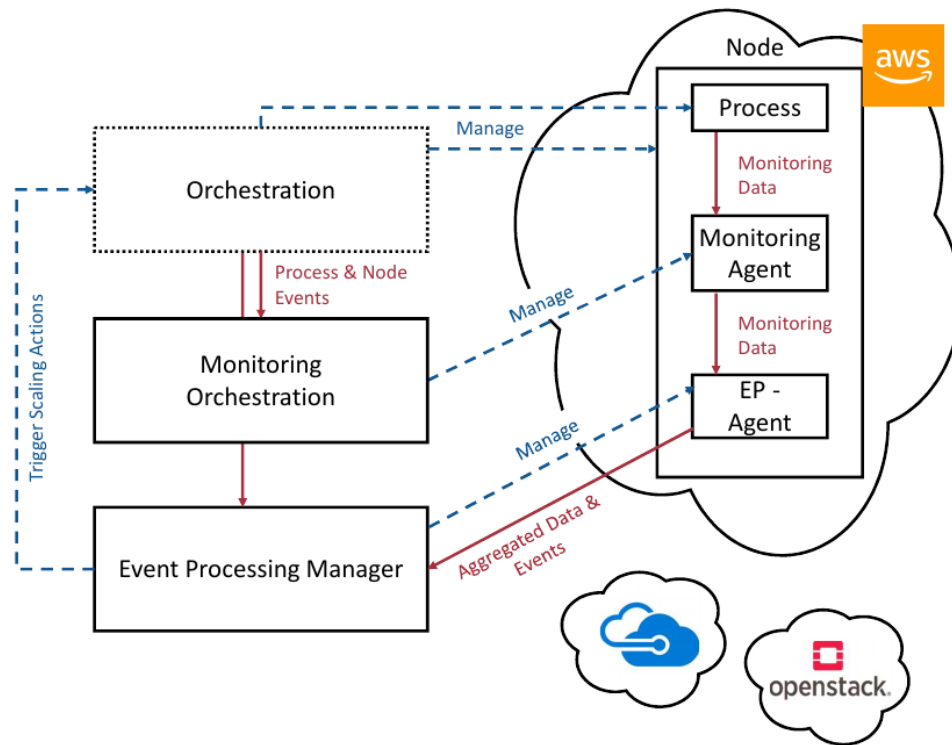


Figure 2: Monitoring architecture [3]



### 3.1.1 Monitoring Model

The fundamental structure of a Cloudiator monitor is defined in a YAML-file which is also used for the REST-API<sup>1</sup> definition. This guarantees a common structure for all interfaces interacting with Cloudiator monitors. Using Cloudiator REST-API to make a monitor request results in a json description of the requested monitor whose structure is shown in Listing 1.

```
{
  "metric": "string",
  "targets": [
    {
      "type": ["JOB", "TASK", "PROCESS", "NODE"]
      "identifier": "string"
    }
  ],
  "sensor": {
    "type": ["PushSensor", "PullSensor"]
    "configuration": [polymorph type specific]
  },
  "sinks": [
    {
      "type": ["KAIROS_DB", "INFLUX_DB", "JMS", "CLI"]
      "configuration": {
        "keystring1": "valuestring1",
        "keystring2": "valuestring2"
      }
    }
  ],
  "tags": {
    "keystring1": "valuestring1",
    "keystring2": "valuestring2"
  }
}
```

*Listing 1: Structure of Cloudiator monitors*

In contrast to enumeration types of 'targets' and 'sinks', in the monitor request the enumeration at the 'sensor' is a discriminator for polymorphism. The sensor configuration is type specific, so the configuration attributes differ depending on the sensor type. The 'datasinkconfiguration' and 'tags' are designed as a stringmap, thus any number of tags or configuration properties can be added to each monitor. For internal handling a monitor instance is extended by some private attributes like 'monitorstatus' or 'Visor-Uuid' before it is stored in the monitoring database. The 'Visor-Uuid' is a unique identifier for a sensor in Visor and will be added to the associated instance on node level. The status attribute and its handling will be explained in the following section.

<sup>1</sup> <http://cloudiator.org/rest-swagger/>

### 3.1.2 Monitoring Orchestration

The monitoring as part of the Cloudiator Framework interacts with other agents via its internal messaging service. Furthermore, there are two direct connections. On the one hand to the Store-Agent which contains all Cloudiator databases, thus also the monitoring database, on the other hand the interaction with Visor which is handled via an internal REST-Interface.

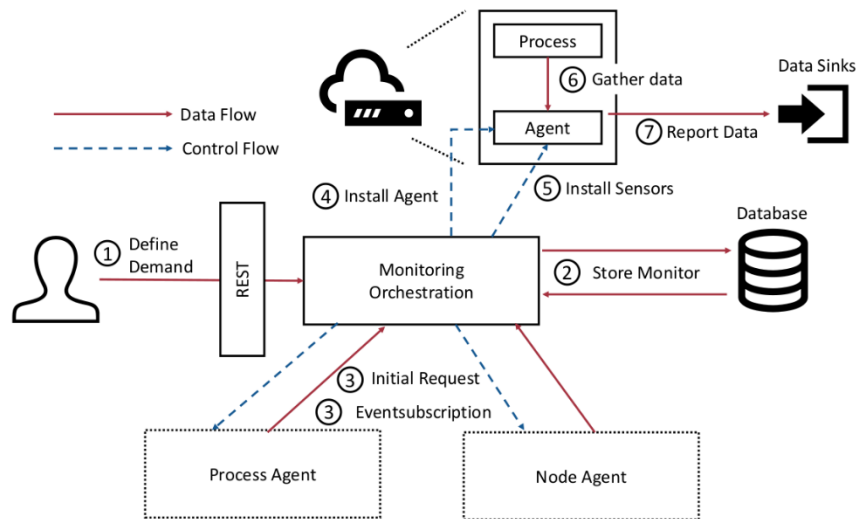


Figure 3: Monitor Orchestration Workflow

The basic monitoring workflow is shown in Figure 3. Melodic Upperware uses the REST-API to define a monitor request (1). According to this Request a monitor model is created and stored in the monitoring database (2). The affected virtual machine will be requested and the related Events subscribed (3). After identification of the virtual machine all needed agents (4) and sensors (5) will be installed. Finally, the requested data can be gathered (6) and sent to the monitor data sink (7).

According to the specified targets of a monitor request, one or more internal monitor instances will be created. How many instances are generated depends on the hierarchical level of the specified targets.

The hierarchical structure of Cloudiator monitors can contain multiple instances of the next lower level down to the lowest level, i.e. the node level, see Figure 4. From a monitoring point of view, a job can contain multiple tasks which can contain multiple processes. A process can also contain multiple nodes if it is a cluster process.

According to the monitor target, all monitor instances will be initiated to set up the demanded sensor on all related nodes. During the creation of a monitor, an initial search request in Cloudiator is performed to identify all selected targets and their statuses. After the first initial request any further changes of the targets will be transmitted by the Cloudiator internal event-messaging system.

Thereby, changes in the backend will be noticed and the affected monitors can be updated. If a status of a node or process changes, the event listener checks the monitoring database for related monitor instances.

On node- and process level, the monitoring status is a direct copy of its targets' status. On job- and task level, the status is either the status of all sublevel instances which have in common or it's 'ERROR' if one of the related instances differ. The handling of all monitor instances on one node is done by the Monitor-Orchestration-Service to keep Visor as lightweight as possible. Furthermore, the monitoring is also responsible for installing all needed agents on the nodes. These agents will only be installed if needed, so the installation is triggered by the first monitor request on the respective node. The monitor installation on every node is handled by a dedicated queue to which all requested monitors are added. Each queue has a separate queue worker responsible for installing the monitors in the queue. All existing queues are listed in the monitoring. When a monitor request is delivered, it checks if already a queue for the requested node exists. If so, the requested monitor is added to the existing queue. Otherwise a new queue and its associated worker will be created. If a queue worker has emptied its queue, it finally removes the queue from the list of existing queues and terminates itself.

### 3.1.3 Scaling Engine

The Scaling Engine is responsible for enacting scaling requests issued either manually by the user or automatically by the Melodic Upperware. A scaling request includes the following information: the process (i.e. the running application component) that needs to be scaled, the scaling direction and either an explicit set of resources that should be used for scaling or an implicit target node size.

As scaling direction, the scaling engine supports horizontal scaling in both direction: *i)* scale-out meaning that additional resources are added to the running process and *ii)* scale-in meaning that resources are removed. Using an explicit resource description, the user or the Upperware needs to previously acquire the nodes using the API. Using the implicit resource description, the user only needs to express the amount of resources needed and the internal matchmaking of Cloudiator will perform the required matchmaking and it will allocate the resources needed.

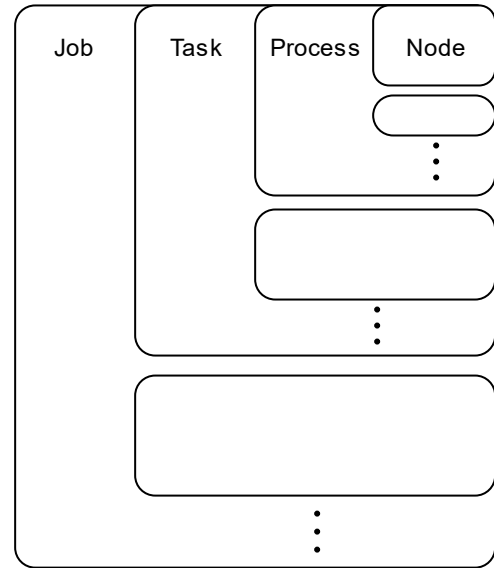


Figure 4 hierarchical monitor structure

## 3.2 Bring Your Own Node (BYON)

This section describes the functioning and the usage of the BYON (Bring-Your-Own-Node) feature that was integrated into the Resource Framework of Cloudiator as part of Melodic R2.5.

### 3.2.1 Functionality

The BYON feature was developed as an extension to the resource management framework described in D4.3 [1]. The basic idea behind the BYON support is to give a user the possibility to provision his 'own physical nodes' with the help of the Executionware, or more precisely, the Cloudiator Framework. By using his own node (e.g. a virtual server), the user can have several advantages over the usual method of provisioning a node with the help of a cloud provider. For example, if the node is a dedicated server, which is owned by the user or the company he or she is employed at, the user will be in the position of full control over the application component that is hosted on this node during the lifecycle of the application. This implies complete ownership of the data residing on the node or the direct enforcing of security measures or policies. Moreover, if the user constructs his 'own node' by means of the API of a cloud provider, he can create a tailor-made node for his requirements by exploiting every configuration detail of the specific provider, which might not be the case if he indirectly creates his node with the generic multi-cloud interface of the Melodic platform. On the other hand, if the user or his company are the owner of the node, they would be obliged to administer the node by themselves and they also would have to pay bills for electricity or maintenance. Both things would be avoided if the node was provisioned by a cloud provider. In this case, where the user builds his node with the help of a cloud API, he would have to execute certain work steps, which in the regular case the Melodic platform would relieve him of. To summarize, the BYON Support gives the Melodic platform more flexibility in the resource provisioning process and enlarges the portfolio of possible use-cases. In the following, the term 'BYON' is used to describe nodes that are associated with the BYON Support feature.

On the technical side, specific information needs to be delivered to Cloudiator to make it aware of a single BYON. That is e.g. the public IP address, login credentials or the Operating System that was installed on the node. It should be mentioned that the Operating System doesn't need to be installed on a Virtual Machine in a virtualized environment but can also be installed on 'bare-metal' on the BYON. Internally, the interactions with a registered BYON are as follows. After being registered in the system, the node will pop up when a list of possible node candidates is queried. If specific requirements of an application component later comply with the offered properties of the BYON, the reasoning process of the Melodic Upperware will select the BYON for the deployment of an instance of the component.

The central part of the BYON implementation is the Byon-Agent. This agent saves the internal states of the BYONs and handles all interactions with them. That is adding a BYON, querying all BYONs and deleting a BYON in the system. Concerning the state, two possibilities exist: unallocated and allocated. The initial state is unallocated, and it switches to allocated if and only if an application component instance is deployed on a BYON. In the time of being allocated, the BYON cannot be deleted from the system and will not pop up when querying possible node candidates. Both will be the case again if Cloudiator removes the application component from the BYON, while at the same time the state will transition back from allocated to unallocated.

### 3.2.2 Usage

The following example in Listing 2 shows the json description the Byon-Agent requires to register a node in the system. The description must be delivered by the Cloudiator user, e.g. the Melodic Upperware. The example is for the most part self-explanatory. However, it should be noted that fields which do not describe a functional property, e.g. the 'geolocation', are used to create a unique internal id for the node. Specific units for fields can be found in the description of the Cloudiator REST-API<sup>2</sup>. The field "disk" e.g. requires a Gigabyte value.

```
{
  "name": "byon-test-node",
  "loginCredential": {
    "username": "ubuntu",
    "password": "testpw"
  },
  "ipAddresses": [
    {
      "IpAddressType": "PUBLIC_IP",
      "IpVersion": "V4",
      "value": "134.60.64.1"
    }
  ],
  "nodeProperties": {
    "providerId": "e12a32e4b62be36596e0882886a552a0",
    "numberOfCores": 8,
    "memory": 4096,
    "disk": 1.0,
    "operatingSystem": {
      "operatingSystemFamily": "UBUNTU",
      "operatingSystemArchitecture": "AMD64",
      "operatingSystemVersion": 1604
    }
  },
  "geoLocation": {
    "city": "Ulm",
    "country": "DE",
    "latitude": 48.4010822,
    "longitude": 9.9876076
  }
}
```

*Listing 2 – BYON description Example*

<sup>2</sup> <http://cloudiator.org/rest-swagger/>

### 3.3 User Interface

The initial prototype did not feature a user interface but instead relied on the exposed API<sup>3</sup> for main interaction with the other components of Melodic. For manual testing a collection of API calls were provided with the POSTMAN<sup>4</sup> collaboration platform for API development. However, it became apparent that a more user-friendly way of accessing the base functionality of the Cloudiator Framework is needed. For this purpose, a high-level graphical user interface was developed that encapsulates the main interaction endpoints with the framework: *i)* registering a new cloud provider, *ii)* running a new application and *iii)* providing an overview of a running application.

The cloud management overview depicted in Figure 5 allows the user to add/update/delete a new cloud provider which will cause the automatic detection of its offers and their addition to the resource offering mechanism. In addition, the discovered entities can be viewed.

The textual editor shown in Figure 6 allows the user to deploy an application using a newly developed YAML syntax for describing the application. A graphical feedback shows a graph of the components of the application and their dependencies, while automatically checking the provided description of syntactic and semantic errors.

Finally, the overview of the deployed application (see Figure 7) gives feedback on all running deployments. It gives quick access to vital information like the endpoints of the running applications while using a traffic-light style indication (green: running, yellow: pending, red: error), allowing the operator an overview of the deployment status.

The Cloudiator web user interface is built using the JavaScript framework Angular<sup>5</sup>. For UI design, the CSS library Bulma<sup>6</sup> was combined with Angular CDK<sup>7</sup>, a toolkit to build Angular components. In consideration of the use case it was designed desktop first, but it still has full functional support on mobile devices. To simplify internal state management, NgRx<sup>8</sup> was employed to realize a redux pattern. As the Cloudiator API is designed with the Swagger framework, Swagger's code generation tool was used to generate interfacing services. Furthermore, a Jasmine<sup>9</sup>/Karma<sup>10</sup> stack was used for testing, as it has a good default integration with Angular.

---

<sup>3</sup> <http://cloudiator.org/rest-swagger/>

<sup>4</sup> <https://www.getpostman.com/>

<sup>5</sup> <https://angular.io/>

<sup>6</sup> <https://bulma.io/>

<sup>7</sup> <https://material.angular.io/cdk>


<sup>8</sup> <https://ngrx.io>


<sup>9</sup> <https://jasmine.github.io/>


<sup>10</sup> <https://karma-runner.github.io/latest/index.html>

Cloudiator Clouds Resources Schedules Editor john.doe@example.com

Clouds


**openstack4j**  
OpenstackUser


**aws-ec2**  
AWSUser


**google-compute-engine**  
GCPUser

+

Cloudiator Clouds Resources Schedules Editor john.doe@example.com

aws-ec2

**Hardware**

Name	Cores	Ram	Disk
c5d.xlarge	4	8192	10
c3.large	2	3750	10
c4.large	2	3840	0
t2.micro	1	1024	0
m5.xlarge	4	16384	0
m1.medium	1	3750	10

**Images**

Name	OS
ami-ubuntu-18.04-1.11.3-00-1537835827	UBUNTU

**Locations**

Name	Country	City
aws-ec2		
sa-east-1	BR	Sao Paulo
sa-east-1c		
sa-east-1b		
sa-east-1a		
us-east-2	US	Columbus

Figure 5: Cloud Management

Cloudiator
Clouds
Resources
Schedules
Editor
john.doe@example.com

job\_bw.yaml

validate

submit

```

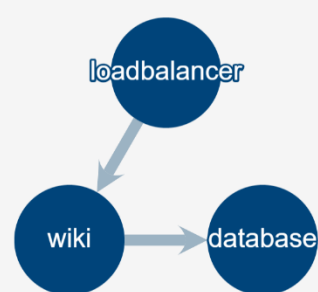
1 ---
2 job:
3   name: mediawiki_bw
4   tasks:
5     - name: loadbalancer
6       behaviour:
7         type: ServiceBehaviour
8         restart: true
9       ports:
10        - type: PortProvided
11          name: LBPROV
12          port: 80
13        - type: PortRequired
14          name: LOADBALANCERREQWIKI
15          isMandatory: 'false'
16      interfaces:
17        - containerType: NATIVE
18          type: LanceInterface
19          preInstall: sudo apt-get -y update && sudo apt-get -y install git
20            && git clone
21              https://github.com/dbaur/mediawiki-tutorial.git
22          install: './mediawiki-tutorial/scripts/lance/nginx.sh install'
23          start: './mediawiki-tutorial/scripts/lance/nginx.sh start'
24          updateAction: './mediawiki-tutorial/scripts/lance/nginx.sh
25            configure'
26      requirements:
27        - constraint: nodes->forAll(hardware.name = 'm1.nano')
28          type: OclRequirement
29        - constraint: nodes->forAll(image.name = 'Ubuntu 18.04')
30          type: OclRequirement
31    - name: wiki
32      behaviour:
33        type: ServiceBehaviour
34        restart: true
35      ports:
36        - type: PortRequired
37          name: WIKIREQMARIADB
38          isMandatory: 'true'
39        - type: PortProvided
40          name: WIKIPROV
41          port: 80
42      interfaces:
43        - type: LanceInterface
44          containerType: NATIVE
45          preInstall: sudo apt-get -y update && sudo apt-get -y install git
46            && git clone
47              https://github.com/dbaur/mediawiki-tutorial.git
48          install: './mediawiki-tutorial/scripts/lance/mediawiki.sh install'
49          postInstall: './mediawiki-tutorial/scripts/lance/mediawiki.sh
50            configure'
51          start: './mediawiki-tutorial/scripts/lance/mediawiki.sh start'
52      requirements:
53        - constraint: nodes->forAll(hardware.name = 'm1.nano')
54          type: OclRequirement
55        - constraint: nodes->forAll(image.name = 'Ubuntu 18.04')
56          type: OclRequirement
57    - name: database
58      behaviour:
59        type: ServiceBehaviour

```

mediawiki\_bw

Job

Schedule



```

graph TD
  loadbalancer --> wiki
  wiki --> database

```

Figure 6: Textual Editor using YAML format



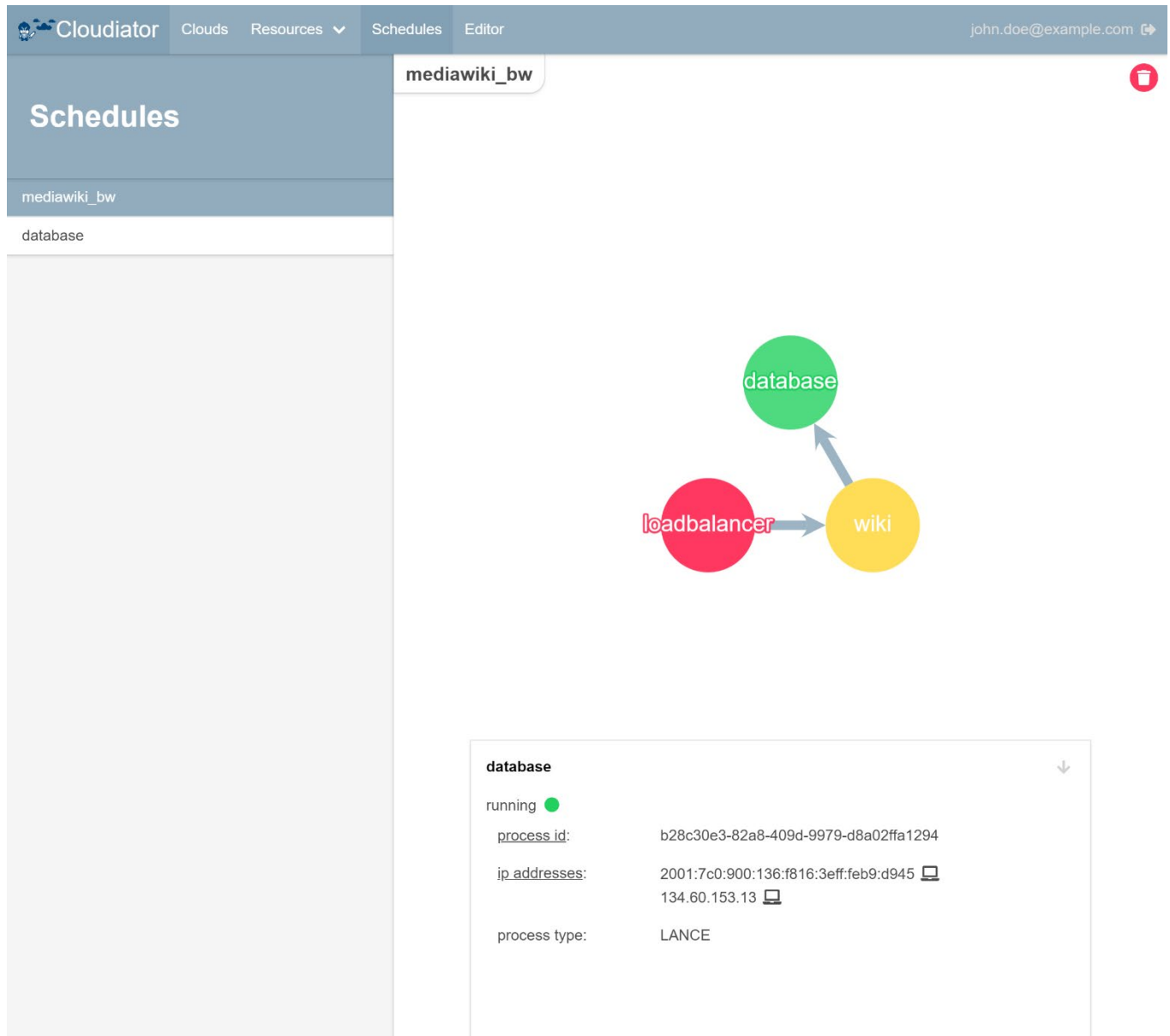


Figure 7: Overview of running Application

## 4 Implementation

This section describes the implementation of the Cloudiator Framework and will list major additions to the previous implementation section found in D4.3.

### 4.1 License

As for the prototype, all software belonging to the Cloudiator Framework is released under Apache License 2.0<sup>11</sup>.

### 4.2 Main Dependencies

During the development of the final release, multiple open source dependencies were added to the Cloudiator project, which are listed in Table 2.

Table 2: Main Dependencies

Name	Description	Link
Google Jib	Builds efficient containers from java projects. Used to improve the integration (see Section 7).	<a href="https://github.com/GoogleContainerTools/jib">https://github.com/GoogleContainerTools/jib</a>
AngularJS	A popular javascript framework used for developing the user interface.	<a href="https://angularjs.org/">https://angularjs.org/</a>
CMPL (<Coliop   Coin> Mathematical Programming Language)	A mathematical programming language for describing integer linear optimization problems.	<a href="http://www.coliop.org/">http://www.coliop.org/</a>
CBC (COIN-OR Branch-and-Cut solver)	A branch-and-cut solver for integer linear optimization problems.	<a href="https://github.com/coin-or/Cbc">https://github.com/coin-or/Cbc</a>

### 4.3 Source Code Repositories

All source code repositories of Cloudiator are hosted on Github under the organization Cloudiator<sup>12</sup>. As for the new prototype, a new repository was added, named user-interface, which contains the source code for the newly developed user interface.

<sup>11</sup> <http://www.apache.org/licenses/LICENSE-2.0>

<sup>12</sup> <https://github.com/cloudiator>

## 5 Maintenance

This section gives an overview for the maintenance effort required to leverage the Cloudiator Framework from a prototype to the final product. This maintenance effort mainly includes reacting to the feedback given by the use cases but also includes daily work like responding to bug requests, analysing software bugs and providing fixed versions. The first section will give an overview about updates done to the previous prototype, if possible, referencing feedback given by the use case providers. The second section will summarize smaller maintenance efforts not large enough warranting an individual section.

### 5.1 Updates to the previous prototype

This section presents updates to the previous prototype based on the provided feedback. In contrast to the feature section (see Section 3) which presents new features, this part will present updates to the already existing feature set.

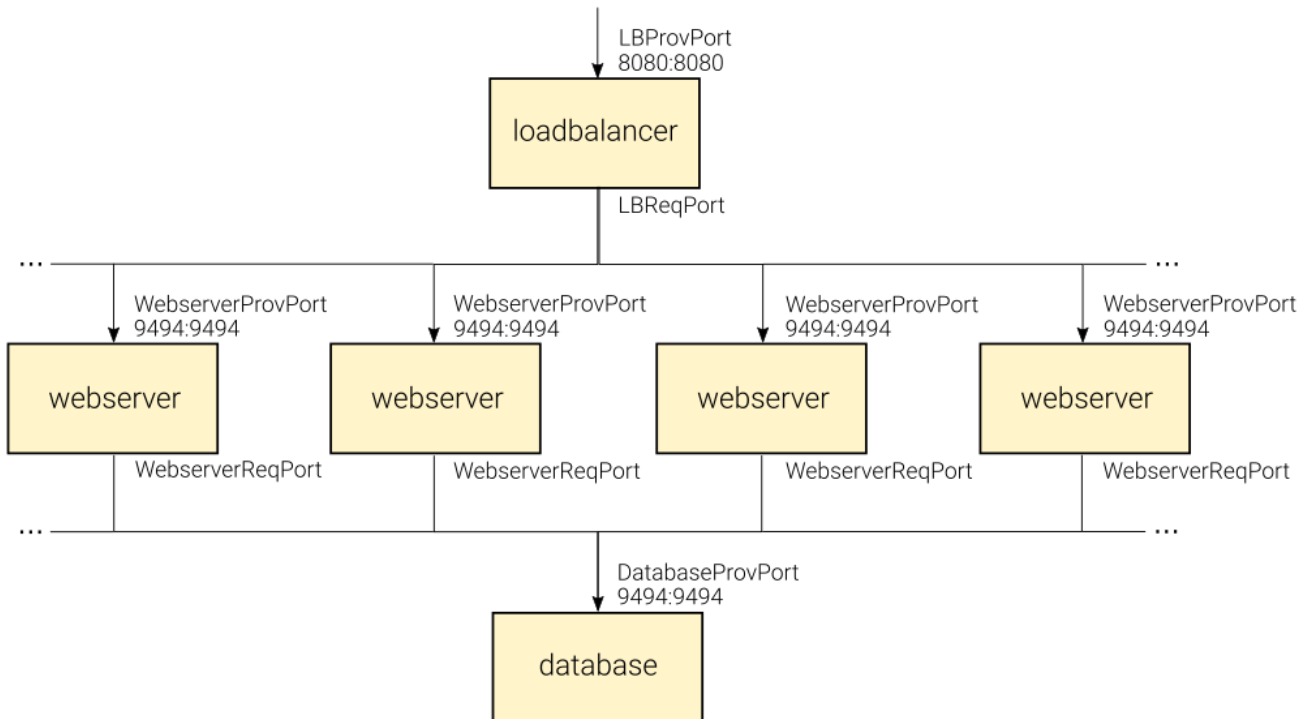
#### 5.1.1 Refined Docker Interface

The 'Native Docker Support' as described in D4.5 [4] was further refined to support a broader range of application use-cases. The main focus of the refinement lays in extending the interaction capabilities of Docker containers (F2). As an example, a load balancer (e.g. HAProxy<sup>13</sup>) must know its downstream application component instances to reconfigure itself with the goal to balance the incoming load. A generic approach was used by dividing the respective components into two groups: The first one, in which its members, respectively containers, announce their public IP addresses and ports (i.e. sockets) after they had successfully bootstrapped and started. The second one in which its members process these events. The first group is named 'dynamicgroup' and the second one 'dynamichandler'. In order to process these events (i.e. announced sockets), the application owner can inject a custom shell-script into the containers of the 'dynamichandler' group by means of the respective Dockerfile. With the help of the formerly described mechanism, this script can then process the announced socket values to reconfigure the load balancer associated with the container. The communication means for the announcements of the events is the etcd<sup>14</sup> registry. From there, the 'dynamichandler' component instances can query the socket values that were written into the registry by bootstrapped and started members of the 'dynamicgroup'.

---

<sup>13</sup> <http://www.haproxy.org/>

<sup>14</sup> <https://coreos.com/etcd/>



*Figure 8: Docker Application Example*

To illustrate this functionality, an application (see Figure 8) is described, that consists of three components: A load balancer, a webserver and a database. The load balancer's job is to evenly spread the load among instances of the second component, the webserver. The webserver is a component, that provides instances which can be scaled out horizontally and which access a single database, which forms the third component. Listing 3 shows an example of a Cloudiator job-description, that is compliant with the given example application. The first json-object under the 'tasks' array describes the load balancer. All relevant configuration parameters are mapped into the environment, of which the 'dynamichandler' and the 'updatescript' parameters are the relevant ones for the refined Docker Interface. As discussed at the beginning of this section, the load balancer instance will execute the script defined as the value of the parameter 'updatescript', with a socket value as its input-parameter, at the moment the corresponding downstream component instance (webserver) with an appropriate 'dynamicgroup' value has bootstrapped and has become ready. The second object is the webserver, which should provide the Cloudiator user, e.g. the Melodic Upperware, the possibility to scale it horizontally. For that purpose, it has to get registered at the load balancer instance with the help of the 'dynamicgroup' parameter. To make the communication between the webserver instances and the load balancer instance work, the 'dynamichandler' and 'dynamicgroup' parameters in the respective components must have the same value. In the example case, this value is equal to 'dynamicgroup1'. The third component is the database, which was chosen to be a mariadb<sup>15</sup>. In contrast to the other components, this component is pulled from the public

<sup>15</sup> <https://mariadb.org/>

docker registry<sup>16</sup>, which also implies that there are no user credentials set in the environment. The communications block in Listing 3 specifies the dependencies between the components and thus enforces a certain start order.

```
{
  "name": "EXAMPLE_APP",
  "tasks": [
    {
      "name": "loadbalancer",
      "behaviour": { "type": "ServiceBehaviour", "restart": true },
      "executionEnvironment": "DOCKER",
      "taskType": "SERVICE",
      "ports": [
        {
          "port": "8080",
          "type": "PortProvided",
          "name": "LBProvPort"
        },
        {
          "isMandatory": true,
          "type": "PortRequired",
          "name": "LBReqPort"
        }
      ],
      "interfaces": [
        {
          "type": "DockerInterface",
          "dockerImage": "omi-registry.e-technik.uni-ulm.de:443/melodic/sample-
loadbalancer:latest",
          "environment": {
            "username": "test.user@example.com",
            "password": "topsecret",
            "port": "8080:8080",
            "dynamichandler": "group1",
            "updatescript": "/etc/updateEndpoints.sh"
          }
        }
      ],
      "optimization": null,
      "requirements": []
    },
    {
      "name": "webserver",
      "behaviour": { "type": "ServiceBehaviour", "restart": true },
      "executionEnvironment": "DOCKER",
      "taskType": "SERVICE",
      "ports": [
        {
          "port": "9494",
          "type": "PortProvided",
          "name": "WebserverProvPort"
        }
      ],
    }
  ]
}
```

<sup>16</sup> <https://hub.docker.com/>

```

    {
      "isMandatory": true,
      "type": "PortRequired",
      "name": "WebserverReqPort"
    }
  ],
  "interfaces": [
    {
      "type": "DockerInterface",
      "dockerImage": "omi-registry.e-technik.uni-ulm.de:443/melodic/sample-
webserver:latest",
      "environment": {
        "username": "test.user@example.com",
        "password": "topsecret",
        "DB_USER": "melodic",
        "DB_PASSWORD": "testpwd",
        "DB_HOST": "$PUBLIC_WebserverReqPort",
        "DB_NAME": "testdb",
        "port": "9494:9494",
        "dynamicgroup": "group1"
      }
    }
  ],
  "optimization": null,
  "requirements": []
},
{
  "name": "database",
  "behaviour": { "type": "ServiceBehaviour", "restart": true },
  "executionEnvironment": "DOCKER",
  "taskType": "SERVICE",
  "ports": [
    {
      "type": "PortProvided",
      "name": "DatabaseProvPort",
      "port": 3306
    }
  ],
  "interfaces": [
    {
      "type": "DockerInterface",
      "dockerImage": "mariadb",
      "environment": {
        "MYSQL_DATABASE": "testdb",
        "MYSQL_PASSWORD": "testpwd",
        "port": "3306:3306",
        "MYSQL_USER": "melodic",
        "MYSQL_ROOT_PASSWORD": "admin"
      }
    }
  ],
  "optimization": null,
  "requirements": [],
}
],

```

```
"communications": [  
  {  
    "portRequired": "LBReqPort",  
    "portProvided": "WebserverProvPort"  
  },  
  {  
    "portRequired": "WebserverReqPort",  
    "portProvided": "DatabaseProvPort"  
  }  
]  
}
```

*Listing 3 – Docker job-description*

With this job description, it is ensured that whenever a new component instance of the webserver is added to the system after the initial deployment, the load balancer instance gets notified and can take appropriate actions in the form of e.g. reconfiguration.

### 5.1.2 Parallelization

The prototype was not able to allocate resources in parallel (F3). This caused high deployment times especially for use cases that required to allocate a large amount of virtual machines (VMs) at the same time, e.g. 100 VMs in the CE-Traffic use case. While Cloudiator's architecture consisting of micro services connected by a distributed message queue was already setup for high amounts of parallelization, we discovered some issues implementing this feature. First of all, the implementation used for the abstraction layer (Apache jClouds<sup>17</sup>) was not fully thread-safe, meaning that an allocation of multiple nodes at the same time could cause race-conditions, also being the reason the prototype did not offer parallelization in the first place. This issue could however be solved by an in-depth analysis of the problem and synchronizing the problematic parts. However, a fully parallelized startup is still hindered by request quota limitations from the providers, meaning that only a specific number of requests is possible within a defined time frame. Therefore, we are currently limiting the number of concurrent requests by a configurable threshold. Nevertheless, we were able to speed up the deployment time by a significant amount.

### 5.1.3 Matchmaking

The resource management layer prototype features a resource offering and matchmaking mechanism. However, during testing, especially with a large provider set, it became clear that the existing implementation was too slow, requiring up to multiple minutes for providing a solution.

We therefore improved the performance of the approach significantly by replacing the existing approach using the Choco Solver<sup>18</sup> with a combination of the <Coliop|Coin> Mathematical Programming Language (CMPL) and the solvers supported by it (mainly the COIN-OR Branch-and-Cut solver). Thus, instead of translating the Object Constraint Language Constraints expressed by

<sup>17</sup> <https://jclouds.apache.org/>

<sup>18</sup> <http://www.choco-solver.org/>

the user into a logical constraint problem expressed with the Choco Solver, we transform the expressed constraints to a mathematical representation of the Integer Linear Program (ILP) using the CMPL. We then use the above branch-and-cut solver to solve the newly represented problem. Our evaluation shows that for some example constraints, this can reduce the runtime of the matchmaking and resource offering from minutes to milliseconds.

In addition, the new resource offering and matchmaking system is now capable of handling resource quotas issued by the provider. We differentiate between two quota approaches: *AttributeQuotas* and *OfferQuotas*. *AttributeQuotas* restrict the amount a user can have of a specific resource type, while *OfferQuotas* restrict the amount the user can have of a specific offer. Openstack e.g. mainly uses *AttributeQuotas* restricting e.g. the total amount of CPUs a user may allocate, while Amazon EC2 uses *OfferQuotas* restricting the amount of a specific hardware flavour, e.g. t2.medium. We acquire the quotas from the provider's API and then consider them during the resource offering and matchmaking processes ensuring that the generated solution is viable and is not restricted due to quota allocation.

#### 5.1.4 Cost Discovery

The Cost Discovery mechanism is based on *MultiCloudPricingService*, Cloudiator's internal service class, which, as its name implies, is specially designed to collect price lists from multiple Cloud Service Providers (CSPs). This service uses a pricing supplier factory which creates specialized suppliers for each CSP that is being offered. Every pricing supplier needs to follow a specified interface to hook up into the discovery mechanism, which then persists the discovered price lists to the database. From there, pricing information can be used in other platform components through a pricing domain repository (e.g. attaching prices to node candidates).

For the mechanism to kick in, first an object of type *Cloud*, which represents a specific CSP along with details such as account credentials required to access its resources, needs to be registered. If a particular CSP requires authorization to retrieve pricing, a pricing supplier requests the pricing information using the appropriate authorization method. The method is specific to the CSP in question and based on credentials provided during the registration. At the time of writing, the Amazon Web Services (AWS) Price List Service API is being used to query for pricing information. AWS provides a Java SDK<sup>19</sup> to serve this purpose. AWS uses its own data format (JSON-based) for the returned pricing information as is the case with other CSPs – everyone uses a different schema. That is why the Cost Discovery mechanism internally translates particular CSP-specific data schemas to a canonical data model. This model, in other words, is a generalised pricing model employed by major CSPs. It comprises of three types of entities structured in a hierarchy. At the very top there is *PricingModel* which comprises of common properties such as:

- Cloud Service Provider Name – e.g. AWS
- Instance Name – e.g. m5.xlarge
- Operating System Family/Architecture – e.g. RHEL/AMD64

---

<sup>19</sup> <https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/pricing/package-summary.html>



- Location – e.g. eu-west-1
- Product Family – e.g. Compute Instance (bare metal)
- Tenancy – e.g. Shared
- License Model – e.g. No License required
- Currency – e.g. USD

One level below, there is PricingTermsModel, which holds a relation to PricingModel and information about the terms of pricing of specific compute instance such as:

- Lease Contract Length – e.g. 3 years
- Purchase Option – e.g. No Upfront
- Terms Type – e.g. Reserved

The last level is occupied by PricingPriceDimensionsModel, which holds a relation to PricingTermsModel and information about the price dimensions of specific terms of pricing such as:

- Price Per Unit – e.g. 0.0912
- Unit – e.g. Hours
- Begin Range – e.g. 0
- End Range – e.g. Infinity
- Description – e.g. \$0.861 per On Demand SUSE m3.2xlarge Instance Hour

Thanks to this approach, new pricing suppliers can be easily added to the discovery mechanism and use the existing infrastructure (e.g. persistence services).

## 5.2 Other Improvements

The major part of the effort was put into supporting the use case partners and the testing team in locating and fixing smaller issues that not only includes bug fixing, but also smaller ease of use requirements. A total of 70 reported issues were analysed, fixed and closed.

## 6 Documentation

As for the prototype the main documentation resides on the 'Cloudiator web page'<sup>20</sup> The installation and the usage guide were updated with the new feature set. The step-by-step tutorial was updated with an example for an Apache Spark job and an Amazon Lambda FaaS application.

---

<sup>20</sup> <http://cloudiator.org/>



## 7 Integration

As described in the prototype deliverable, Cloudiator has a fully featured integration process that proved valuable during release testing, as bug fixes and new features could be automatically provided to the testers and use cases. To even further increase the speed at which the changes could be provided, we integrated Jib<sup>21</sup> into the build process which builds optimized Docker containers for all Cloudiator components. This not only reduced the build time significantly, but also decreased the overhead of a running Cloudiator installation by providing more efficient containers.

In addition, a new release management was introduced. While the previously described integration process (see D4.3 [1]) was designed to be able to respond quickly to bug fixes and thus push changes fast to the released version, it became apparent that for use case testing a more stable release process is required ensuring that changes are only pushed to the stable version if they previously passed the defined test cases. For this purpose, the integration process was virtually duplicated, providing two releases of Cloudiator: *i*) a nightly build directly built from the latest code and *ii*) a stable build built from a separate branch. In addition, policies ensure that only code passing the test cases is merged from the night builds to the stable version.

---

<sup>21</sup> <https://github.com/GoogleContainerTools/jib>

## 8 Summary

In this deliverable we have presented the evolution from prototype to final product for the resource management layer of Cloudiator. First, we presented the collected feedback from the users of the prototype and then addressed each individual feedback by either describing a new feature addressing the feedback item or improving existing features during the maintenance task. In parallel, we have addressed issues reported by users or the testing team.

In the follow-up deliverable D4.6, we will present the final product of the Data Processing Layer in addition to a validation of the Resource Management Framework, which will be executed by said layer.



## 9 References

- [1] Daniel Baur and Daniel Seybold, "D4.3 Resource Management Layer Prototype", Melodic Project Deliverable, Aug. 2018.
- [2] D. Baur and D. Seybold, "D4.1 Provider agnostic interface definition & mapping cycle", Melodic Project Deliverable, Apr. 2019.
- [3] D. Baur, F. Griesinger, Y. Verginadis, V. Stefanidis, and I. Patiniotakis, "A Model Driven Engineering Approach for Flexible and Distributed Monitoring of Cross-Cloud Applications," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, Jan. 2019, pp. 31–40.
- [4] D. Baur, D. Seybold, F. Held and P. Skrzypek, "D4.5 Data Processing Layer Prototype", Melodic Project Deliverable, Jan. 2019.
- [5] S. Schork et al., "D6.1 Evaluation Framework and Use Case Planning", Melodic Project Deliverable, Feb. 2018.
- [6] Y. Verginadis et al., "D3.4 Workload optimisation recommendation and adaptation enactment", Melodic Project Deliverable, Jan. 2019.