Melodic
Big data cloud

Title:

Data Processing Layer Prototype

Executive summary:

This deliverable presents the prototype of the data processing layer of the Executionware. The data processing layer enhances the Executionware with concepts for enabling the native deployment and orchestration of Big Data processing frameworks as well as Function-as-a-Service (FaaS) and container-based applications. These concepts are implemented within this prototype for the Big Data Processing framework Apache Spark, the AWS' FaaS service Lambda and for general Docker-based applications.

This document provides an overview of the implementation details of the prototype, describes its functionality, documentation and integration. In order to evaluate the first prototype, a qualitative evaluation against the use case requirements has been carried out.

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:
D4.5

Date:
31 January 2019

Responsible partner:
UULM

Editor(s):
Daniel Seybold

Author(s):
Daniel Seybold, Daniel Baur, Florian Held, Pawel Skrzypek

Approved by:
Jörg Domaschka

ISBN number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D4.5 Data Processing Layer Prototype.pdf

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

Document	
Period Covered	M8-24
Deliverable No.	D4.5
Deliverable Title	Data processing layer prototype
Editor(s)	Daniel Seybold
Author(s)	Daniel Seybold, Daniel Baur, Florian Held, Pawel Skrzypek
Reviewer(s)	Sebastian Schork, Jörg Domaschka
Work Package No.	4
Work Package Title	Executionware
Lead Beneficiary	Ulm University
Distribution	PU
Version	1.0
Draft/Final	Final
Total No. of Pages	26

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

List of Listings

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

1 Introduction

This document describes the first iteration of the data processing layer of the Executionware prototype. The main purpose of the data processing layer is to *i)* offer an extensible interface [1] to the Melodic Upperware for executing Big Data processing jobs on top of data processing frameworks , *ii)* to automatically orchestrate the data processing frameworks, including the resource allocation, processing framework deployment and configuration and *iii)* monitor the usage of the data processing framework resources as well as job-specific execution metrics.

In order to enable this functionality, the Executionware relies on the Cloudiator framework[1] [3], [4] and its modular and extensible architecture [1], [5]. The extensibility of Cloudiator enables that all functionality of the data processing layer can be implemented within the Cloudiator framework itself. This is done by building on the established unified mapping interface described in D4.1 "Provider agnostic interface definition & mapping cycle" [1] and the resource management layer described in D4.3 "Resource management framework prototype" [2].

In order to validate the concepts of Cloudiator to support the orchestration of data processing frameworks [3], this prototype implements support for Apache Spark[2]. The selection of Apache Spark as the first Big Data framework supported by Cloudiator is driven by the Melodic use cases [5] and their requirements.

Support for the orchestration Function-as-a-Service (FaaS) has been included, as FaaS is getting more attention for Big Data processing [6] [7]. AWS Lambda[3].was selected based on the results of a survey over current FaaS technologies conducted within Melodic [8].

In addition, the existing deployment capabilities of Cloudiator are extended by an enhanced management of the lifecycle of arbitrary non-containerized and containerized applications.

The following sections of this document discuss the implementation results within Cloudiator and its usage within Melodic.

[1] http://cloudiator.org/
[2] https://spark.apache.org/
[3] https://aws.amazon.com/de/lambda/features/

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

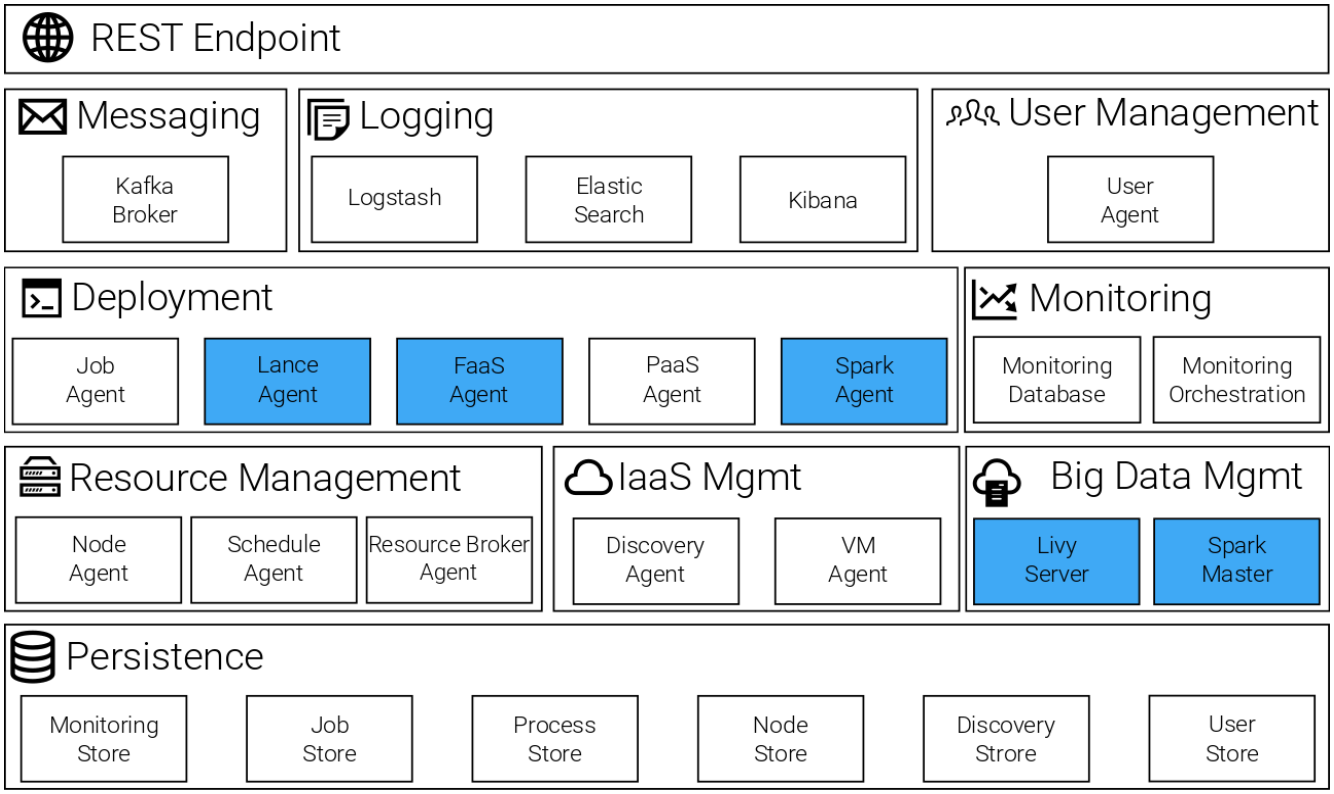

Figure 1: Cloudiator Architecture with Data Processing Layer

1.1 Scope of the document

This document is intended for audience interested in the development process, functionality and usage of Melodic's Executionware and the data processing layer. The work presented here builds on the concepts presented in D4.1 [1] and relies on the resource management features from D4.3 [2].

In order to ease the understanding of the data processing layer extension to the Cloudiator framework, a brief overview of Cloudiator is provided in the following. The interested reader can find more details in D2.2 [9], D4.1 [1] and D4.3 [2].

The modular architecture of Cloudiator is depicted in Figure 1 where the internal components are grouped by their logical scope of functions. The terminology of the Cloudiator framework specifies each Cloudiator component as an *Agent*. All Agents and additional services depicted in Figure 1 build the *Cloudiator Home Domain,* which is typically operated on a dedicated host. Allocated nodes (e.g. Virtual Machines) and optional agents such as the monitoring agent represent the *Cloudiator Remote Domain*.

The central interface to interact with Cloudiator is the *REST Endpoint*. Incoming requests are transformed into messages and distributed via the *Kafka Message Broker* to the responsible Agents which execute the requested resource management, IaaS management and deployment

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

operations. This architecture enables the data processing extensions by implementing new agents.

New agents implemented in the context of the data processing layer are highlighted as blue boxes in Figure 1. Within the Cloudiator framework the data processing agents are part of the Deployment logic. The Spark Agent enables the orchestration of Apache Spark by allocating resources, setting up the cluster and submitting data processing jobs. For that purpose, it requires the additional Big Data Management services Livy Server and Spark Master as depicted in Figure 1. The functionality of the *Spark Agent* is introduced in Section 2.1. The extended Docker support of the *Lance Agent* is described in Section 2.2 and the FaaS support provided by the *FaaS Agent* is presented in Section 2.3. These agents realise the data processing layer and enable the Upperware, to deploy Big Data processing jobs on multiple data processing frameworks in a transparent manner.

1.2 Structure of the document

The remainder of the document is structured as follows: Section 2 describes the new functionality introduced and the overall functionality covered by the prototype. Afterwards, Section 3 gives an overview of the code that was implemented to achieve this functionality. Section 4 describes the usage of the prototype, whereas Section 5 presents the integration of the data processing layer into the Cloudiator framework. Section 6 provides a technical, use case driven evaluation of the data processing layer before Section 7 concludes.

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

2 Functionality

In the following, the functionality of the data processing layer, namely the Apache Spark support by the Spark Agent, the native Docker Support by the Lance Agent and the FaaS support by the FaaS Agent, are introduced. Implantation details follow in Section 3 while the usage of the agents is described Section 4.

2.1 Apache Spark Support

In order to support the data processing requirements derived from the Melodic use cases [5], we decide to support Apache's Spark framework for the first iteration of the data processing layer. Apache Spark provides a unified data processing framework for batch, SQL, graph and machine learning related jobs [10]. The Apache Spark architecture comprises one master node and an arbitrary number of worker nodes to execute the data processing jobs.

In this context the Spark Agent provides two main functionalities: (i) the automated deployment and orchestration of an Apache Spark cluster and (ii) the submission of Apache Spark jobs on the Apache Spark cluster.

2.1.1 Cluster Orchestration

As depicted in Figure 1 a Spark Master resides in the home domain of Cloudiator, which acts as a central master for an arbitrary number of Spark Worker nodes. Hereby, the Spark Master can reside on the same host as Cloudiator or on a dedicated host, depending on the decision of the Cloudiator operator[4].

The Spark Worker nodes are orchestrated via the Spark Agent by allocating the required nodes specified by the Upperware. The Spark Agent uses the Resource Management layer for the node allocation. In addition, the Spark Agent manages the installation and configuration of the Spark Worker service on each allocated node. The Spark Workers running in the Remote Domain and the Spark Master running in the Home Domain build the operative Apache Spark cluster for submitting data processing job.

2.1.2 Job Submission

In order to execute data processing jobs on an Apache Spark cluster, a programmatic API is required enabling the job submission. As Apache Spark does not provide such an API by default,

[4] http://cloudiator.org/docs/installation.html

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

an additional service is required, which offers an API on top of Apache Spark. In this context, two projects have been identified that offer a REST-based interface on top of Cloudiator, namely the Spark Job Server[5] and the Livy Server[6]. While both projects provide the required features, i.e. a REST-based API to deploy Apache Spark jobs and the comprehensive support of job related configuration options, the Livy Server is selected for the data processing layer within the Executionware due to the following reasons: (1) both projects are community driven and the Livy Server has recently reached the Apache Incubator state, which promises smooth integration into the Apache ecosystem and (2) while the Job Server offers additional features, its configuration is rather complex and the additional features are not required by the Executionware. On the other hand the Livy Server is more lightweight while providing all required features.

Consequently, the Spark Agent implements the client logic to deploy Spark jobs via the Livy Server after the Apache Spark cluster is operative. Additionally, the Spark Agent offers multiple job deployment semantics, which enable mapping of one data processing job to one Spark Worker node or one data processing job to Spark Worker nodes.

2.2 Native Docker Support (Lance Agent v2)

The Lance Agent is responsible for deploying general purpose applications and controlling their lifecycle states and the dependencies between the application components.

2.2.1 Lance Agent v1

In order to deploy applications, the Lance Agent governs the application components through different (lifecycle) states of a state machine. In particular, the following states are run through: *NEW, CREATED, BOOTSTRAPPED, READY, DESTROYED*. In parallel, another state machine is run, which handles actions that are associated with these transitions. For example, between the *BOOTSTRAPED* and *READY* transition, the second state machine traverses among others the states: *PRE_INSTALL*, *INSTALL* and *POST_INSTALL*. The important thing here is that the user can enforce the Lance Agent to execute certain shell commands or scripts during these transitions that e.g. configure and install the application component in a certain way. If e.g. the application component should reside on a machine that runs a Linux operating system with a bash[7] shell available, the user could tell the Lance Agent to download certain install scripts during the PRE_INSTALL state from the web and run the install scripts during the *INSTALL* state. Furthermore, the Lance Agent offers a comprehensive error handling in that the state machines

[5] https://github.com/spark-jobserver/spark-jobserver
[6] https://livy.incubator.apache.org/
[7] https://www.gnu.org/software/bash/

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

can reach certain error states if something went wrong during the traverse of the states. Information about the states of the application components and the overall topology of the application, i.e. the communication dependencies between the components and network information (e.g. public IPs and provided ports of the components), is stored in a global etcd[8]-registry by the Lance Agent. Depending on the communication requirements, the Lance Agent eventually ensures that the components are started in the correct order. All in all, the Lance Agent is responsible for the deployment and the correct functioning of the application. In addition, the Lance Agent gives detailed information about the overall state of the application, e.g. its healthiness.

2.2.2 Docker Support

To show how the Lance Agent integrates with Docker[9], a short explanation of Docker has to be given. Docker is a technology for building software containers by giving a developer the opportunity to bundle an application with all of its required parts. The whole Docker ecosystem has reached a huge market share and by now is the de facto standard for building, running and shipping distributed applications in the form of lightweight containers. To ease up the deployment of containers, Docker offers the developers templates, called images, that can be used to build the software containers, which package and isolate an application part and all its associated dependencies like libraries and configuration files. Containers can be seen as lightweight virtual machines, which run upon and are administered by the Docker daemon. The Docker daemon is part of the ecosystem and runs on top of the operating system, e.g. a virtualized operating system on a machine in a cloud.

As it is the job of the Lance Agent to deploy the components of an application, the advantages of Docker can be incorporated by bundling the components into Docker containers. Apart from circumventing problems like incompatible library versions when deploying the application components, the sometimes cumbersome task of writing the shell commands/scripts for the lifecycle transitions can be omitted now as all application dependencies get resolved by means of the container image.

2.2.3 Lance Agent v2 Extension

In versions prior to Melodic 2.0., the Lance Agent already provided the feature to deploy Docker containers. However, it was just possible to deploy Docker containers built from a native "Ubuntu 14.04" image. This meant that the actual application component had to be manually built inside the container through the said commands/scripts during the lifecycle transitions. The Lance Agent v2 of Melodic 2.0 now breaks these restrictions by enabling the deployment of containers

[8] https://coreos.com/etcd/
[9] https://www.docker.com/

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

built from images (i) residing at https://hub.docker.com or (ii) residing at a private registry on a host, which the user can authenticate against. Again, part of this new functionality was already there before v2 of the Lance Agent. For example, the basic functionality of authenticating against a private registry or the base code structure to pull/retrieve images from arbitrary registries was available. However, to provide these features to the user of the Lance Agent, the said functionalities had to be implemented or extended. Another technical problem that had to be solved in the implementation of the Lance Agent v2 was the correct injection of environment variables that become available during the lifecycle transitions into the containers. This was accomplished by redeploying the containers each time a difference in the values of the environment variables occurred. In a nutshell, the discussed new features add several advantages for the user of Melodic:

i. The user isn't restricted to only use base layer containers anymore.
ii. The user isn't obliged anymore to go through the process of providing shell commands and scripts to build his application inside the container.
iii. The user can rely on high-quality images for his application found at https://hub.docker.com
iv. For all other required application images, that were e.g. built by himself or his company and which shouldn't be publicly available, the user can use images from a private registry.

Putting it all together, with the advent of Melodic 2.0, the user is able to use features that make Docker popular. That is, deploying the application in a portable, maintainable and flexible manner.

2.3 FaaS Support

The idea of FaaS is to provide the possibility to run code directly in the Cloud, without the need to provision any VMs. It is the next step after Containers of abstracting the application from the underlying infrastructure and operating systems and allows for better cloud resources utilization: the code runs for a short time and frees the resources afterwards.

Numerous components of Cloudiator were extended to make the FaaS deployment possible. Before Melodic 2.0, the Discovery Service was only able to collect information about Providers' Offerings for Virtual Machines. Now, also Function Offers are being gathered and included in the common offers registry. Those are then being exploited to build node candidates of type FaaS. Also, the realization of node candidate into provider-specific nodes is being handled in a totally different way than other types of deployments provided by Cloudiator. While Docker, Spark or Lance interfaces rely on the creation of virtual machines before installation of the application itself, the FaaS components is deployed directly into the Cloud. That is why a new virtual type of Node is introduced.

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

The target provisioning of FaaS components to the Cloud is handled by the FaaS Agent. This component orchestrates the deployment process which includes function code uploading, access rights, logging and function triggering configuration.

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

3 Implementation

This section discusses the implementation of the functionality presented in Section 2. It gives an overview of the licenses, the major third-party dependencies of the new Cloudiator agents and the structure of the code repositories where Cloudiator's code resides.

3.1 License

As previous releases of Cloudiator all developed code is released in the public domain under the Apache License 2.0[10].

3.2 Main dependencies

Cloudiator's data processing layer relies on multiple open source projects. The following Table 1 describes the required dependencies in addition to the resource management layer dependencies described in D4.3 [2].

Name	Description	Link
Spark Agent		
Apache Livy	Provides a service that allows for easy interaction with a Spark cluster over a REST interface by enabling easy submission of Spark Jobs or snippets of Spark code, synchronous or asynchronous result retrieval, as well as Spark Context management, all via a simple REST interface.	https://livy.incubator.apache.org/
Apache Spark	Apache Spark is an open-source distributed general-purpose cluster-computing framework that support SQL, streaming and analytical jobs.	https://spark.apache.org/

[10] https://www.apache.org/licenses/LICENSE-2.0.html

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

Apache HttpComponents	The Apache HttpComponents™ project is responsible for creating and maintaining a toolset of low-level Java components focused on HTTP and associated protocols.	https://hc.apache.org/
Lance Agent v2		
Docker (ecosystem)	Docker is a container engine, making it easier to build, run and ship applications. The Executionware uses it for two main purposes: a) the Executionware itself is distributed using Docker and b) the Executionware optionally uses it to deploy the users' applications.	https://www.docker.com/
FaaS Agent		
AWS SDK for Java	The AWS SDK for Java enables Java developers to easily work with Amazon Web Services and build scalable solutions with Amazon S3, Amazon DynamoDB, Amazon Glacier, and more.	https://github.com/aws/aws-sdk-java/

Table 1: Main dependencies of Cloudiator

3.3 Source Code Repositories

All source code of Cloudiator is hosted on GitHub under the organization Cloudiator[11]. The functionality of the data processing layer prototype is located within the deployment repository[12]. Table 2 provides an overview of the individual data processing agents within the deployment repository.

[11] https://github.com/cloudiator
[12] https://github.com/cloudiator/deployment

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

Repository	Description
deployment/deployment-common	Contains commonly used classes across all deployment agents with respect to messaging and persistence.
deployment/spark-agent	The spark agent of Cloudiator that enables the provisioning and orchestration of the Apache Spark cluster as well as the job deployment via the Livy Server.
deployment/lance-agent	Sets up information about the deployment topology in the global etcd-registry and installs a Lifecycle Agent on all used nodes. The Lifecycle Agents execute tasks on the nodes, e.g. run Docker containers, corresponding to the overall deployment of the application. Furthermore, the Agent ensures that the processes of the tasks are executed in the correct order and the communication between them functions properly.
deployment/faas-agent	FaaS Agent allows deployment and orchestration of functions on different serverless service providers.
livy-server-docker	A preconfigured Docker container of the Livy Server that exposes the required configuration options to interact with the Spark Agent.
spark-master-docker	A preconfigured Docker container of the Spark Master that exposes the required configuration options to interact with the Livy Server and the orchestrated Spark Workers.
spark-worker-docker	A preconfigured Docker container of the Spark Worker that is used by the Spark Agent to deploy the Spark Worker services on the provisioned nodes. The Docker container exposes the required configuration options to interact with the Spark Master that resides in the Cloudiator home domain.

Table 2: Source code repositories

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

4 Documentation

All documentation can be found on the Cloudiator website[13].

4.1 Installation and Packaging

To ease the installation process, Cloudiator uses Docker for each of its components. In combination with docker-compose[14], this allows to deploy all components using a single command. The docker-compose description and usage instructions can be found at: https://github.com/cloudiator/docker.

4.2 Usage

This section describes the usage of the data processing layer via Cloudiator by providing basic examples for each data processing agent. More examples and documentation details can be found on the Cloudiator website[13].

In the following, exemplary job descriptions for the Spark Agent, Lance Agent V2 and FaaS Agent are presented and the main attributes of the respective jobs are presented. In the Cloudiator terminology a *Job entity* groups multiple independent, executable *Task entities* that require execution. One execution of a task is termed *Process entity*. A Task entity comprises the following entities: *interfaces, communications* and *requirements*. Interfaces define which Deployment Agent (see Figure 1) is required to execute the Task and contain interface-specific details. Communications represent a communication requirement between two tasks by mapping a required port of a task to a provided port of another task. Requirements contain the information to allocate the required nodes.

4.2.1 Spark Agent

The following example in Listing 1 shows a deployment description the Spark Agent requires to execute data processing jobs. The example calculates π on an Apache Spark Cluster with two Spark Worker nodes. The Job entity contains one Task entity using the SparkInterface. The SparkInterface entity contains all required details to execute the Task, i.e. to start a Process. The number of Spark Workers are specified within the Requirements entity and the actual values are derived by the Upperware. In this example it contains two entities with the required details to allocate the nodes for the Spark Workers. As soon as the cluster is operational, the Spark Agent

[13] http://cloudiator.org
[14] https://docs.docker.com/compose/

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

executes the Task by creating an internal Process of the π-calculation that triggers the submission of the π-calculation on the Apache Spark cluster via the Livy Server.

More extensive Apache Spark examples can be found in the examples section of the Cloudiator homepage[15].

```
{
    "name": "spark-pi-job",
    "tasks": [
    {
      "name": "pi-calculation",
      "executionEnvironment": "SPARK",
      "taskType": "BATCH,
      "ports": [

      ],
      "interfaces": [
        {
          "type": "SparkInterface",
          "file": " https://raw.githubusercontent.com/cloudiator/livy-server-
docker/master/examples/pi.py",
          "className": "",
          "arguments": [10],
          "sparkArguments":{
             "driverCores" : "2",
             "numExecutors": "10"
          },
          "sparkConfiguration":{}

        }
       ],
       "requirements": [
       {
        "hardwareId": "1",
        "locationId": "2",
        "imageId": "3",
        "type": "IdentifierRequirement"
       },
      {
        "hardwareId": "1",
        "locationId": "2",
        "imageId": "3",
        "type": "IdentifierRequirement"
       }]
    }
  ],
  "communications": [ …
  ]
}
```

Listing 1 - Spark Job Example

[15] http://cloudiator.org/docs/sparkInterface_examples.html

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

4.2.2 Lance Agent v2

To show how the Lance Agent's 'native Docker support' works with Cloudiator's REST interface, a basic example is shown in the following Figure 2.

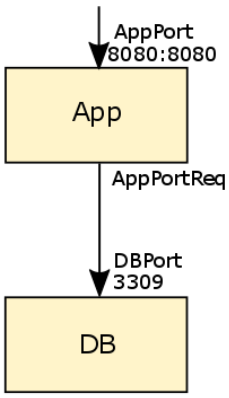

Figure 2: Exemplary application topology

The topology implies that a Docker container named 'App' is dependent on a Docker container named 'DB'. The communication is handled via the ports 'AppPortReq' and 'DBPort'. 'DBPort' provides the port number 3309 and 'AppPort' provides the port number 8080 and makes it available on the host. In the following we focus only on the DB component to highlight the extensions carried out in the context of the Lance Agent v2. A complete Job description, including the DB and the App as well as more complex examples can be found in the examples section of the Cloudiator homepage[16].

The following Listing 2 shows an excerpt of a Job description containing a Task with the LanceInterface with the containerType NATIVE. By using this interface, the DB is deployed directly on the specified node in the requirements entity. Therefore, a set of lifecycle actions need to be specified and the required Shell scripts need to be provided from an accessible endpoint. In order to deploy the DB all specified lifecycle agents are executed on the allocated nodes in the remote domain.

```
"tasks": [
    {
      "name": "Lance-v1-DB",
      "executionEnvironment": "LANCE",
      "taskType": "SERVICE,
      "ports": [
          "port":3309,
           "type":"PortProvided",
           "name":"DBPort"

      ],
```

[16] http://cloudiator.org/docs/lanceInterface_examples.html

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

```
      "interfaces": [{
        "type": "LanceInterface",
        "containerType": "NATIVE",
        "init": " apt-get update -y",
        "download": " wget http://somerepo.com/db-script.sh ",
        "install": " ./db-script.sh install",
        "start": " ./db-script.sh start ",
      "requirements": [
       {
        "hardwareId": "1",
        "locationId": "2",
        "imageId": "3",
        "type": "IdentifierRequirement"
       }

    ]
    }
...
]
```

Listing 2 - Lance Agent v1 Job Example

As the implementation of such scripts can become complex and error prone, the support for existing containers is desirable. In addition, as many applications already exist in public Docker repositories, using or extending these containers eases the integration of applications into Melodic. Consequently, Listing 3 shows the Task description using the new DockerInterface. In this job description, the DB image is pulled from a public registry and configured with a set of provided environment variables. With environment entity, certain variables which are set inside the containers are specified. These set variables are generally required by the container's application to function properly. All other fields will get mapped into the containers' environments.

```
"tasks": [
    {
      "name": "Lance-v2-DB",
      "executionEnvironment": "DOCKER",
      "taskType": "SERVICE,
      "ports": [
          "port":3309,
          "type":"PortProvided",
          "name":"DBPort"

      ],
      "interfaces": [{
        "type": "DockerInterface",
        "dockerImage": " registry.hub.docker.com/library/mysql:latest",
        "environment":
            {
                  "port": "3309",
                  "MYSQL_ROOT_PASSWORD": "admin",
                 "MYSQL_USER": "melodic",
```

```
                    "MYSQL_PASSWORD": "topsecret",
                    "MYSQL_DATABASE": "app",
              }
              }],
       "requirements": [
        {
         "hardwareId": "1",
         "locationId": "2",
         "imageId": "3",
         "type": "IdentifierRequirement"
        }

    ]
    }
...
]
```

Listing 3 - Lance Agent v2 Job Example

4.2.3 FaaS Agent

The following job, shown in Listing 4, deploys a 'Hello World' function. The code of the function is loaded from the URL provided in the 'sourceCodeUrl' field. Starting point is defined by 'handler'. The code in the example is written in JavaScript: In this case 'code.helloWorld' means that the function is in the code.js file and the function we want to call is named helloWorld. Triggers defines events in which the function should be invoked. HttpTrigger will setup a HTTP endpoint to the function. Timeout is the timeframe within the function must finish its execution. Function environment is a set of key and value pairs available to the function.

Requirements assures that the chosen node will support FaaS, the language in which the function is written and that it provides enough memory. More extensive FaaS examples can be found in the examples section of the Cloudiator homepage[17].

```
{
    "name": "HelloWorld",
    "tasks": [
    {
      "name": "helloLambda",
      "taskType": "BATCH,
      "ports": [],
      "interfaces": [
        {
          "type": "FaasInterface",
          "functionName": "hello",
          "sourceCodeUrl": "https://s3-eu-west-1.amazonaws.com/cloudiator-faas-
code/code.zip",
          "handler": "code.helloWorld",
```

[17] http://cloudiator.org/docs/faasInterface_examples.html

```
          "triggers": [
            {
              "type": "HttpTrigger",
              "httpMethod": "GET",
              "httpPath": "hello/world"
            {
          ],
          "timeout": 3,
          "functionEnvironment": {
            "key": "value"
          }
        }
      ],
      "requirements": [
        {
          "constraint": "nodes->forAll(type = NodeType::FAAS)",
          "type": "OclRequirement"
        },
        {
          "constraint": "nodes->forAll(environment.runtime = Runtime::NODEJS)",
          "type": "OclRequirement"
        },
        {
          "constraint": "nodes->forAll(hardware.ram = 1024)",
          "type": "OclRequirement"
        }
    ]
    }
  ],
  "communications": []
}
```

Listing 4 - FaaS Job Example

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

5 Integration

The data processing layer is integrated into Cloudiator's fully-featured integration process, comprising building, continuous integration and deployment features as depicted in Figure 2. More details of this process are described in D4.3 [2].

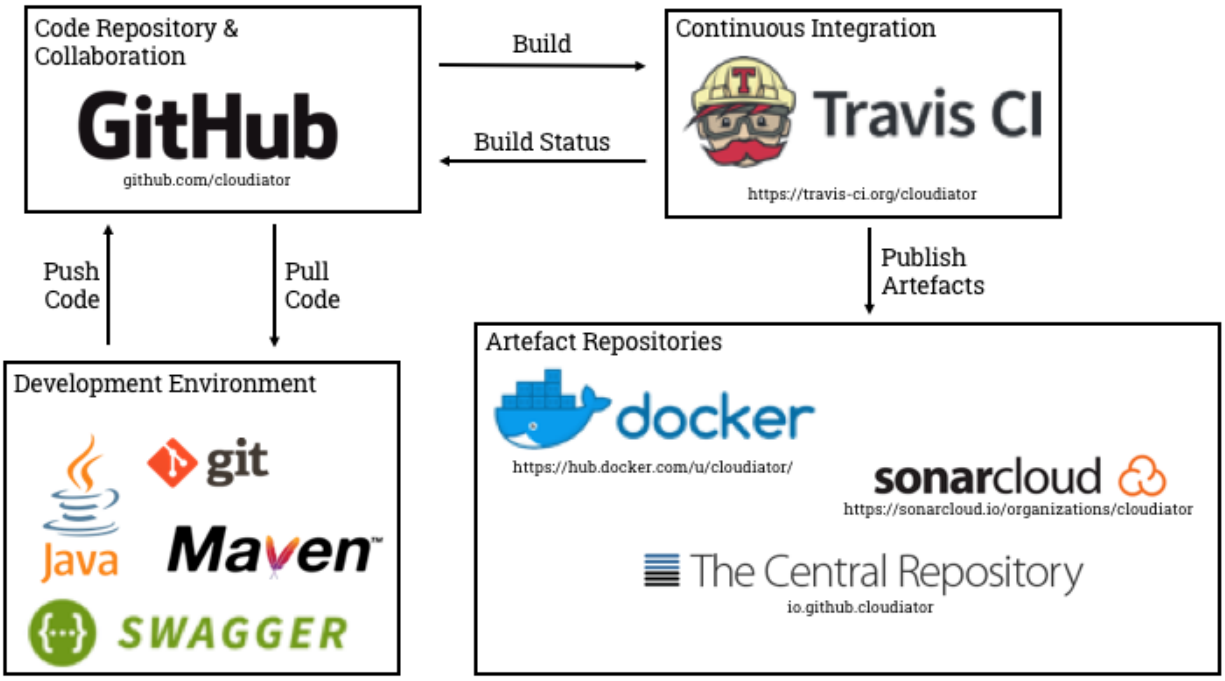

Figure 3: Integration tools and process

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

6 Data Processing Layer Prototype Evaluation

In this section, a qualitative evaluation of the data processing layer prototype is carried out against the deployment and Big Data management related use case requirements that were derived from D6.2 [5]. Hereby, the required features are listed in Table 3. The *Prototype Version* column indicates the current state of the required feature which is qualified by *supported* ✓, *partially supported* (✓), or *not yet supported* ✖. The *Final Version* column indicates the planned improvements to enable the required feature for the upcoming final data processing layer (deliverable D4.6) if not already supported by the data processing layer prototype.

Technical Use Case Requirement Description	Prototype Version	Final Version
Installation and deployment of an N-component application in Docker containers on M different Cloud Providers	✓	✓ (no further extensions required)
Installation and deployment of an N-component application, where X components are installed in a Docker container and Y on a normal VM on M different Cloud Providers	(✓)	The communication between Lance and Docker tasks is already supported, the need for additional task communications will be reviewed
Installation and deployment of an N-component application on M different Cloud Providers with a more advanced set of requirements, like non-functional ones.	(✓)	the supported non-functional features will be enhanced
Big data application deployment execution	✓	As additional data processing framework, Hadoop MapReduce will be integrated. The FaaS support will be extended for Upperware and the support of Azure functions is planned.

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

Big data application monitoring and reconfiguration	(✓)	reconfiguration of data processing clusters will be enabled based on historical execution data
Big data application deployment optimization	✘	The Executionware interface will be extended to provide optimization actions to the Upperware.

Table 3: Data Processing Layer Evaluation

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

7 Summary

This deliverable presents the functionality, the technical details and a first technical evaluation of the data processing layer of Melodic's Executionware, which is implemented via the Cloudiator framework. The data processing layer is an extension to the deployment features of Cloudiator, that builds upon the provider agnostic interface mapping [1] and the resource management layer of Cloudiator [2], that enhances the deployment features of general purpose applications with the native support for Big Data processing frameworks, Function as a Service (FaaS), as well as the usage of existing Docker containers. The first version of the data processing layer enables the support for the Apache Spark framework and Docker containers from arbitrary Docker registries. In addition, the FaaS support provided is now available for AWS Lambda. Consequently, the job deployment API of Cloudiator offers a more feature-rich interface to the Upperware in order to enhance the execution of data-intensive applications in a multi-cloud context.

The data processing layer prototype has been evaluated against the first iteration of the deployment and Big Data processing requirements derived from the Melodic use cases. The upcoming deliverable D4.6 will enhance the current version of the data processing layer with the required, but not yet fully supported, features. Therefore, the final data processing layer will build upon the extended interface mapping features of the upcoming deliverable D4.2 and the extended resource management features of the upcoming deliverable D4.4. The final data processing layer will be thoroughly evaluated against the extended use case requirements derived from upcoming D6.3. The integration of the data lifecycle management within the data processing layer will be described in the upcoming deliverable D3.2.

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

8 References

[1] Daniel Baur and Daniel Seybold, "D4.1 Provider agnostic interface definition & mapping cycle," Melodic Project Deliverable, Sep. 2019.

[2] Daniel Baur and Daniel Seybold, "D4.3 Resource Management Layer Prototype," Melodic Project Deliverable, Aug. 2018.

[3] D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka, "A Provider-Agnostic Approach to Multi-cloud Orchestration Using a Constraint Language," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Washington, DC, USA, 2018, pp. 173–182.

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

[5] Sébastien Kicin, Sebastian Schork, Antonia Schwichtenberg, Pawel Gora, Tomasz Przeździęk, and Michal Semczuk, "D6.2 Use Cases Implementation and Feedback (1st Iteration)," Melodic Project Deliverable, May 2018.

[6] Bhaskar Das, "Can Serverless computing reshape big data and data science?," *Can Serverless computing reshape big data and data science?*, 16-Jan-2019. .

[7] Sunil Mallya, "Ad Hoc Big Data Processing Made Simple with Serverless MapReduce," *Ad Hoc Big Data Processing Made Simple with Serverless MapReduce*, 16-Jan-2019. .

[8] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Zurich, Switzerland, 2018, pp. 161–168.

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

[10] M. Zaharia *et al.*, "Apache Spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

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