Rational decomposition and orchestration for serverless computing

Deliverable D5.5

Data pipeline orchestration I

Version: 1.0

Publication Date: 21-Dec-2019

Deliverable Card

Deliverable	D5.5
Title:	Data pipeline orchestration I
Editor(s):	Chinmaya Dehury (UTR)
Contributor(s):	Chinmaya Dehury (UTR), Satish Srirama (UTR), Pelle Jakovits (UTR), Matija Cankar (XLB)
Reviewers:	Mike Long (PRQ), Vladimir Yussupov (UST)
Type:	Report
Version:	1.0
Date:	21-Dec-2019
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables/
Copyright:	RADON consortium

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825040

Executive summary

This document presents an insight view of the initial version of RADON data pipeline orchestration. The main objective of adopting the data pipeline approach is to provide an environment to the EU software industry to freely deploy, schedule and scale the pipeline tasks. The pipeline tasks can be serverless functions, or microservices.

This document describes the TOSCA templates for modeling data pipelines, the interaction of data pipeline components among themselves, realization and execution of the data pipeline using different RADON tools and ecosystem. The implementation of data pipeline thumbnail generation example is discussed along with the future work.

Glossary

DP	Data Pipeline
AWS	Amazon Web Services
IDE	Integrated Development Environment
EMR	Elastic MapReduce
GMT	Graphical Modeling Tool
CTT	Continuous Testing Tool
CDL	Constraint Definition Language

Table of contents

1. Introduction

In this deliverable, we present the initial version of data pipeline methodology and orchestration, mainly using one private cloud service provider and open source data management platform. The rest of this deliverable describes the overall data pipeline architecture and its interactions with other RADON components/tools. For better understanding, this deliverable additionally discusses the implementation of RADON Toy (thumbnail generation) example.

1.1. Deliverable objectives

The main objectives of this deliverable can be summarized as follows:

1. Define the data pipeline methodology taking existing technologies into consideration.
2. Define the first version of the overall data pipeline architecture with the basic components.
3. Describe the basic components of data pipeline architecture on high degree of granularity.
4. Creating and updating the existing TOSCA model [2] for data pipelines based on the basic components of overall architecture.
5. Outline or summarize the interaction of data pipeline with other RADON components, such as template library, graphical modeling tool, RADON orchestrator etc.
6. Implement the RADON thumbnail generation example based on the introduced data pipeline approach.

1.2. Overview of main achievements

In the process of achieving the aforementioned objectives, the consortium' contributions to this deliverable can be summarized as follows:

1. Presented the methodology that is followed to conceptualize the orchestration of data pipelines.
2. The consortium designed and formally released the first version of data pipeline architecture, with basic components that are aligned with RADON objectives and existing related technologies.
3. The data pipeline architecture is designed with dedicated components, keeping the requirement of modern serverless applications in mind as well.
4. With the new data pipeline components, it is now possible to freely compose an application combining independently deployable, schedulable, and scalable pipeline tasks, such as microservices, serverless functions, or self-contained applications.
5. The data pipeline methodology is modeled using TOSCA specification.

1.3. Structure of the document

The rest of this deliverable is structured as follows:

- **Section 2** presents the detailed survey of some existing technologies with their key features and their comparison.

- **Section 3** gives the overall architecture of data pipeline, and describes the basic components introduced in this version. In addition to this, the TOSCA extension of data pipeline based on the overall architecture is presented.
- **Section 4** presents the interactions of data pipeline with other RADON components including RADON orchestrator, template library, graphical modeling tool, RADON IDE.
- **Section 5** presents the implementation of RADON thumbnail generation example using Apache NiFi as open source data management platform and one private FaaS provider.
- **Section 6** gives the vision of the data pipeline plugin that may work with RADON orchestrator.
- **Section 7** gives the future plan to improve the current version of data pipeline architecture and related TOSCA models, and presents the concluding remarks.

2. Reference data pipeline technologies

This section presents the detailed survey of some existing technologies with their key features and their comparison. Table 2.0.1 lists the requirements in the context of RADON data pipeline is followed in the process of development, implementation, and orchestration of data pipelines.

In the context of RADON, data pipeline concept enables the ability to compose applications with independently deployable, schedulable and scalable pipeline tasks, such as microservices, serverless functions or self-contained applications.

Upon adoption of this approach, RADON customers would be able to shift from utilizing monolithic Big Data applications to more general data pipelines consisting of freely composable, portable and reusable functions and microservices for data processing (including transformation) and management. Initially, RADON supports Apache NiFi (open source data management platform) with private cloud environments as the main data pipeline solution. However, in the next phase of the project, RADON will also support Amazon data pipeline technology.

Table 2.0.1: RADON requirements list

Id	Requirement Title	Priority
R-T5.4-1	The data pipeline module of the Orchestrator must be able to orchestrate data pipelines	Must have
R-T5.4-2	The data pipeline module of the Orchestrator must support cron based scheduled data pipelines	Must have
R-T5.4-3	The data pipeline module of the Orchestrator should support event based scheduled data pipelines	Should have
R-T5.4-4	It would be useful for the data pipeline module of the Orchestrator to support logging and generating alerts on pipeline task failures.	Should have
R-T5.4-5	The data pipeline module of the Orchestrator should support deployment of data pipelines which automate movement of data between two or more clouds	Should have
R-T5.4-6	The data pipeline module must support data pipelines tasks that initiate data analytics tasks for processing data moving through the pipeline	Must have
R-T5.4-7	The data pipeline module of the Orchestrator should support configuring encryption between data pipeline tasks when data needs to be moved between systems	Should have
R-T5.4-8	The data pipeline module of the Orchestrator should support deploying data pipelines expressed using TOSCA models into the AWS data pipeline service.	Must have
R-T5.4-9	The data pipeline module of the Orchestrator should support deploying data pipelines expressed using TOSCA models into a private OpenStack cloud.	Must have

In the following subsections, we discuss two data pipeline technologies: Amazon data Pipelines and Apache Nifi.

2.1. Amazon Data Pipelines

Amazon Data Pipelines is an AWS service for orchestrating the transportation and transformation of data between different AWS cloud services [3]. Users can define data pipelines and schedule them to be triggered at specific time schedule, event, or on-demand.

The main components of Amazon Data Pipelines are DataNodes and Activities, which define the functionality of the data pipeline. Available DataNodes are:

A. DynamoDBDataNode - A DynamoDB table.
B. SqlDataNode - SQL database table (including SELECT query).
C. RedshiftDataNode - Amazon Redshift database table.
D. S3DataNode - Amazon S3 data folder or file path.

Available pipeline activities, which specify an operation to perform on the data are:

A. CopyActivity - Copies data from source DataNode to destination DataNode.
B. EmrActivity - Runs a data processing action inside an Amazon Elastic MapReduce (EMR) cluster.
C. HiveActivity - Executes a Hive query in EMR cluster.
D. HiveCopyActivity - Executes a Hive query in EMR cluster with support for advanced data filtering and delivery options.
E. PigActivity - Executes a Pig script in EMR cluster.
F. RedshiftCopyActivity - Copies data between Amazon Redshift tables and other DataNodes.
G. ShellCommandActivity - Runs a custom Linux shell command or script on an Amazon EC2 instance.
H. SqlActivity - Executes a SQL query on the input DataNode.

As there are only a few DataNode types and activities available (which cover only a small fraction of available AWS services), very often the design of the pipeline boils down to using a set of *ShellCommandActivities*, which internally use the API libraries of different AWS services.

Additional pipeline components are Schedules for specifying when data pipelines (or activities) are initiated, Preconditions for specifying conditions when pipeline tasks can be executed and Resources for specifying EC2 computing resources or other AWS services that data pipeline depends on (e.g. EC2 cloud instances or Elastic MapReduce clusters). More detailed description is available online at AWS Data Pipeline Documentation[1].

2.2. Apache NiFi

Apache NiFi [4] provides an easy-to-use web-interface for orchestrating the flow and preprocessing of data across several systems. Unlike AWS data pipelines, Apache NiFi (being an open source platform) can be extended freely. Using this, existing pipelines can be composed into complex data pipelines that contain a large number of pluggable pipeline blocks. In the case of devices with limited

[1] AWS Data Pipeline, https://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/what-is-datapipeline.html

amount of available resources, MiNiFi[2], a sister project of Apache NiFi, can be used for orchestration of data movement and processing.

The main components of Apache NiFi are:

A. *FlowFile* - Logical notation of a piece of data that moves through the pipeline, which contains a pointer to the data together with its metadata, which consists of a set of dynamic attributes (e.g. path, name, size, priority). FlowFile content data is stored separately in Content Repository or/and kept in memory. FlowFile is a virtual file, meaning the content of a single FlowFile might be a section of a larger physical file in memory or disk.
B. *Processor* - A component that performs a specific data pipeline action on a FlowFile. Processors can have multiple load balanced and concurrently running instances inside NiFi Node/cluster. Different types of Processors are:
    - Data Egress - Sending FlowFile (content) to external systems.
    - Data Ingress - Creating FlowFiles from data located outside NiFi service.
    - Routing of FlowFiles - Can be used to filter and split stream of FlowFiles between multiple branching paths in the data pipeline.
        - Based on content - based on actual data
        - Based on attributes - based on metadata/attributes of the FlowFiles
    - Split Content - Divide one FlowFile into multiple
    - Update Attributes - Attributes of data objects are modified. Attributes are metadata of data objects. E.g. mime type, size, extension.
    - Modify Content - Content of FlowFile is streamed through the processor to modify or enrich it. Produces new FlowFiles.
C. *Queue* - a queue between two NiFi Processors, can be used for configuring FlowFile priorities, back pressure and load balancing strategies.
D. *Input Port* - A NiFi component for defining which port to listen for incoming FlowFiles from external NiFi systems. Incoming FlowFiles are passed along to the pipeline.
E. *Output Port* - The NiFi component mainly to establish the data communication to the external processors that are not present within the same process group.
F. *Process Group* - Represents another data pipeline running in the same NiFi cluster. Allows to compose data pipelines into bigger pipelines and reuse existing ones.
G. *Remote Process Group* - Represents another data pipeline running on a different NiFi node/cluster. The composition of *Remote Process Group* and *InputPort* is seen as a pipe between two remote systems. It is also possible to use *PutKafka* and *GetKafka* processors to create pipelines between two remote systems running NiFi.

Scheduling in NiFi is defined separately for each processor. Available scheduling types are:

[2] https://nifi.apache.org/minifi/index.html

A. Timer driven - Default option. Processor will be executed on a regular interval. Default interval value is 0, meaning processor is triggered as often as possible as long as there are FlowFiles in the input queue.
B. Event driven - Processor is triggered as new FlowFiles enter into the input queue.
C. CRON driven - Processor is triggered at specific times.

Unlike the *DataNode* component of Amazon data pipeline, no such specific component is offered by NiFi that can represent a storage point for data. In Apache NiFi, processors directly consume the data from a local (e.g. *GetFile* get the data from local file system) or remote data source (e.g. *GetKafka* listens to a Kafka topic). Some processors are also offered to push the data to local (e.g. *PutFile* write data to local file system) or remote data source (e.g. opposite to *GetKafka*, *PutKafka* push the data to remote Kafka topic).

NiFi processors like *GetFile* can be viewed as a composition of a *DataNode* object and a pipeline Activity/Processor object, as the user must both configure the processor and also specify the information about the location of the data. In the case of the *GetFile* processor, the input folder path on the local system is the minimum required configuration.

Data (or FlowFiles) between two processors are stored in a local NiFi *Content Repository* and reference to the data object (together with its metadata) is assigned to the queue between two processors, meaning data is not transported directly through queues.

Apache NiFi data pipelines can be saved as pipeline templates (either the whole pipeline or multiple components at a time) and exported as XML files, which then can be imported as templates into other NiFi systems. The in-depth information on Apache NiFi components can be found at Apache NiFi in depth guide[3].

2.3. A comparative study

To generalize the data pipeline concepts, both aforementioned data pipeline technologies are compared and presented in Table 2.3.1.

Table 2.3.1. A comparative overview of data pipeline concepts from AWS data pipeline and NiFi [1].

General concept	Apache NiFi	AWS Data pipeline
Pipeline Ingress Point (Gateway for incoming data)	**Processor** (Ingress), **Input Port**	**DataNode**, **Resource** (EC2, EMR)
Pipeline Egress Point (Gateway for outgoing data)	**Processor** (Egress), **Output Port**	**DataNode**, **Resource** (EC2, EMR)
Intermediate location of data	**Queue**	**DataNode**
Data pipeline task	**Processor**	**Activity**

[3]Apache NiFi In Depth - https://NiFi.apache.org/docs/NiFi-docs/html/NiFi-in-depth.html

Task Schedule	**Scheduling** (internal sub-component)	**Schedule**
Triggering condition	**Processor (***RouteOnAtribute, RouteOnContent)*	**Precondition**
Reusing external Data Pipelines (Composing data pipelines)	**Process Group, Remote Process Group**	**N/A**

Based on the investigation in Table 2.3.1, it is found that a very limited number of services are offered by the AWS data pipeline. On the contrary, Apache NiFi provides a large variety of components/processors for different types of pipeline tasks. Amazon data pipeline supports the interactions of different services provided by Amazon such as DynamoDB, SQL compatible databases, RedShift and S3 buckets. Additionally, AWS Data Pipeline supports very limited types of activities, such as i) activity for duplicating data between datanodes, ii) activities for initiating data processing application inside Elastic MapReduce (EMR) cluster iii) and activity for execution of custom linux shell command within an EC2 instance. By allowing the execution of custom Linux shell commands, developer/user can handle the limitations of basic activities for data flow and processing/transformation.

Both NiFi and AWS Data Pipeline services can be composed to form a larger pipeline. NiFi supports dynamic composition of pipelines through an API command, and NiFi Input and Output Ports together with queues are used as intermediate components between pipelines. In AWS data pipeline, *DataNodes* (representing an external data storage location) can be used between two different data pipelines to create a chain of pipelines.

In reference to the mentioned data pipeline technologies, a general data pipeline model is needed that can support different underlying pipeline concepts and technologies, as discussed in the following sections.

3. Overall data pipeline architecture

In this section, we discuss the overall architecture of RADON data pipeline (DP) including the methodology of how the data pipeline should be orchestrated. Our focus in this deliverable is to describe the modeling of generic data pipeline concepts by introducing the basic components based on our study on the reference technologies and to describe the overall architecture of data pipeline with the corresponding TOSCA extension that are required to model DP-based services.

The need of an environment that offers dynamic orchestration of the entire application consisting of a number of independently deployable components mainly forces us to empower RADON framework with data pipeline capabilities/features. The introduction of data pipeline features facilitates users to create and manage the lifecycle of pipeline tasks. Pipeline task could be a microservice, serverless function, or a self-contained application, which simply accepts a set of input data, processes the input data, and returns the desired result. The designed data pipeline architecture

mainly works with RADON orchestrator (along with other RADON components) to create, deploy, and manage the lifecycle of each pipeline block.

3.1. General data pipeline concepts

In RADON data pipeline, we generalize the data pipeline concept considering the requirement of a service development. As the output of this modeling, users are offered different data pipeline components through RADON IDE (using the graphical modeling tool) to design and develop their application. In RADON DP, individual detailed actions are not implemented. For example, the functionality of *GetFile* NiFi processor is not implemented, and instead respective available processors are reused. The smallest component of RADON DP is an individual pipeline based on specific data pipeline technology. Each single data pipeline block in RADON may consist of one or more pipeline blocks.

Basic Components:

The basic components of RADON DP include i) *PipelineBlock*, ii) *InputPipe*, and iii) *OutputPipe*.

i) PipelineBlock: A *PipelineBlock* can be seen as a processor of Apache NiFi or an activity of Amazon data pipeline. This may consist of one or more pipeline tasks. Based on the requirements, a pipeline task may interact with the local file system, or remote database system, or a remote function. In this deliverable, a *PipelineBlock* represents the general pipeline for several types of actions or functionalities. For instance, the *PipelineBlock* can be used for reading local file as well as publishing data to remote file system. It is to be noted that, a *PipelineBlock* transmits the actual data rather than the artefacts or reference of actual data. As a result, in RADON thumbnail generation example, the actual images are transmitted from one system to another system.

The *PipelineBlocks* is designed to accept the scheduling information such as artifacts from the developer. The artifacts may contain the scheduling type such as time driven, event drive or CRON driven, similar to Apache NiFi. Such scheduling artifacts are further used to create and deploy the technology specific pipeline tasks. Further, the RADON DP model should offer the grouping of *PiepelineBlocks* in order to produce a single *PipelineBlock*.

ii) InputPipe: This is designed as an input port or a gateway for listening to the incoming data for any pipeline block. This is similar to the existing component such as Apache NiFi input port, Kafka stream etc. This pipeline port mainly listens other pipeline blocks or remote systems. For instance, in RADON thumbnail generation example, a pipeline block for publishing the thumbnail image files may be configured with an input pipe to listen to the result of remote function that generates the thumbnail.

iii) OutputPipe: To facilitate the connectivity of multiple pipelines, *InputPipe* and *OutputPipe* are introduced. *OutputPipe* is used to simply push/publish the output of a pipeline block and act as a gateway for outgoing data to other local or remote pipeline blocks. *InputPipe* and *OutputPipe* are necessary for connecting multiple number of pipelines.

Connecting *PipelineBlocks*:

RADON graphical modeling tool can be used to model a set of data pipelines to be deployed, the relationships (flow of data) between pipelines themselves and any relationships to external data sources/destinations, such as file system folder location, SQL table or FaaS function endpoint URL.

The current standard specification of TOSCA provides six different types of relationships: DependsOn, HostedOn, ConnectsTo, AttachesTo, and RoutesTo. DependsOn mainly refers to the dependencies of one component upon another. Whereas, HostedOn refers to the hosting relationship of two components. To represent the remote connection over a network, ConnectsTo is offered by TOSCA specification. AttachesTo represents the relationship of type attachment. For instance the attachment of storage node and compute node. For routing the network between two endpoints in different networks, RoutesTo relationship can be used. However, in the case of RADON data pipeline, to model the relationship, ConnectsTo is used to derive as a simpler form of relationship. The connection between any two data pipelines should be based on a directional relationship, where two *PipelineBlocks* can be connected to each other using the *InputPipe* of destination *PipelineBlock* and *OutputPipe* of source *PipelineBlock*.

The relationship can be created and initiated in a dynamic manner. This allows the developer to add or remove the *PipelineBlocks* in a dynamic fashion without bringing huge impact to other *PipelineBlocks*. In the context of RADON thumbnail generation example, the intermediate function for thumbnail generation can be updated with minimum impact on other storage systems. To create a relationship between two *PipelineBlocks*, the basic required information are unique ID of the *OutputPipe* of the source *PipelineBlock* and unique ID of the *InputPipe* of destination *PipelineBlock*. In the following section, a detailed architecture of RADON data pipelines along with its position with RADON architecture is presented.

3.2. Data pipeline architecture

In this subsection of the deliverable, the overall architecture of the data pipeline including the basic components is presented, as shown in Figure 3.2.1. Continuing the component description in Section 2, we present an in-depth description of each component, the interaction among them, and their role in designing the data pipeline-based applications. Figure 3.2.1. shows the basic components of the data pipeline architecture. The primary components that can be used in the design of any application, are *InputPipe*, *OutputPipe*, and *PipelineBlock*.

The *PipelineBlocks* may consist of one or more pipeline tasks, where the output of one pipeline task can act as the input to other pipeline tasks. In the Figure 3.2.1, it is shown that all the pipeline tasks are implemented and executed sequentially. However, the pipeline tasks can also be arranged in a Directed Acyclic Graph (DAG) fashion. In such a scenario, the output of a particular pipeline task can be input to multiple other pipeline tasks. Similarly, the input of one pipeline tasks can be derived by combining the output of multiple pipeline tasks. Further going in-depth in regard to the capability or the functionality of a pipeline task, RADON DP offers several types of tasks such as invoking serverless function, fetching data objects from remote storage system like from the Amazon S3 bucket, working with local and remote file system, etc.

Each *PipelineBlock* may be equipped with a buffer queue to store the input and resultant data. The advantage of this queue can be realized in the situation where the throughput of the pipeline task is less than the arrival rate. The throughput in this case can be affected by the scheduling policy. If the data arrival rate varies at a particular time and the pipeline task is scheduled to run on a periodical basis, the input queue can be used to store the data temporarily. Two queues are implemented for a *PipelineBlock*: i) *DataIngestionQueue* for input data, and ii) *DataEmissionQueue* for resultant data. It is to be noted that these two queues are different from *InputPipe* and *OutputPipe*. The queues are primarily used while connecting multiple *PipelineBlocks* - such an example shown in Figure 3.2.2. However, when two groups of *PipelineBlocks* connect to each other, the pipes are used. Additionally, when a *PipelineBlock* is needed to connect to external sources such as a *PipelineBlock* group in another remote machine, these pipes can be used.

Each *PipelineBlock* group may need two other major components in order to process any data. The *InputPipe* acts as a gateway that listens and receives the data from external *PipelineBlocks*. On the other hand, *OutputPipe* acts as a gateway, that forward the resultant data to external *PipelineBlock*s. It is not mandatory for a *PipelineBlock* group to be attached to both *InputPipe* and *OutputPipe*. However, a *PipelineBlock* group should have at least any one of them. A *PipelineBlock* group with no *InputPipe* and *OutputPipe* can be considered as an isolated pipeline group, which is not connected to any other group of *PipelineBlock*s. The exception to this can be a *PipelineBlock* group connected to two different groups of *PipelineBlocks*, or it is the source *PipelineBlock* group or destination *PipelineBlock* group. RADON DP encourage the developer to avoid such scenarios.

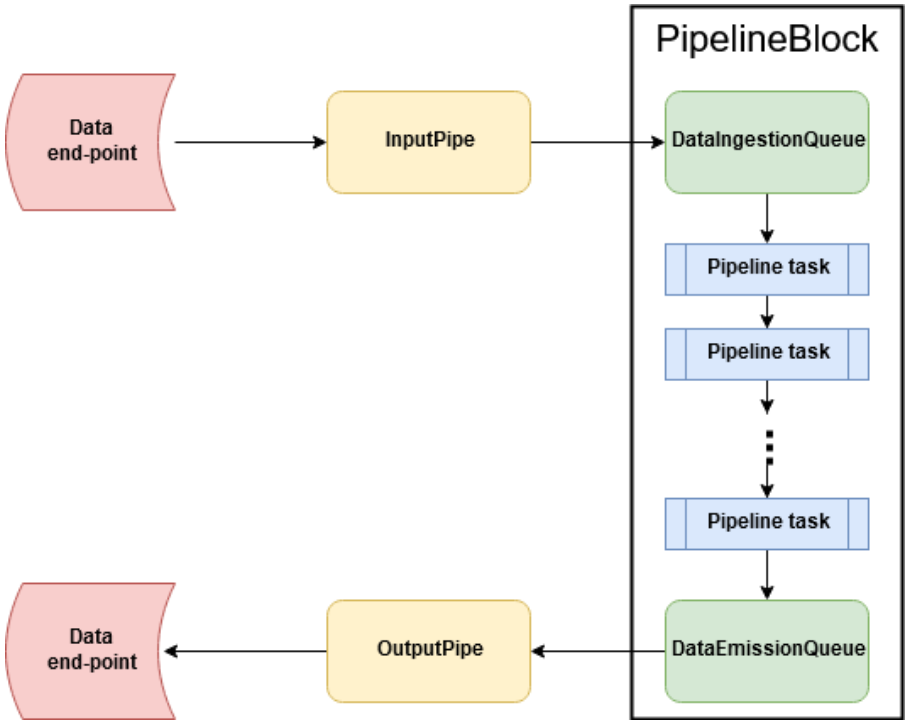

Figure 3.2.1: Overall architecture of data pipeline.

An example is provided in Figure 3.2.2., for better understanding of pipes, queues, *PipelineBlocks*, and how the *PipelineBlocks* connect to external services. The example in below figure consists of

two *PipelineBlocks*: *PipelineBlock1* and *PipelineBlock2*. *PipelineBlock1* is connected to an *InputPipe* which act as gateway between the data end-point and the *PipelineBlock*. When the data end-point is pushing the data, it is received by the *InputPipe* and passed to the *DataIngestionQueue* of *PipelineBlock1*. *PipelineBlock1* is further consisting of one pipeline task for invoking OpenFaaS function to process the input data. The process data is further forwarded to another pipeline, *PipelineBlock2*. This pipeline is consisting of one pipeline task which stores the data in Amazon S3 bucket, followed by the termination of the pipeline.

This example shows how a service can be composed by combining multiple external microservices, FaaS functions, etc. from different providers. Here, it is to be noted that, *InputPipe* is used while listening to and collecting data from external data end-point. However, no pipeline is used while connecting two *PipelineBlocks*. Both pipeline blocks are attached with pipes to store the data. If the data received by *PipelineBlock1* at a higher rate and the throughput of the OpenFaaS function is low, the *DataIngestionQueue* keeps the data temporarily.

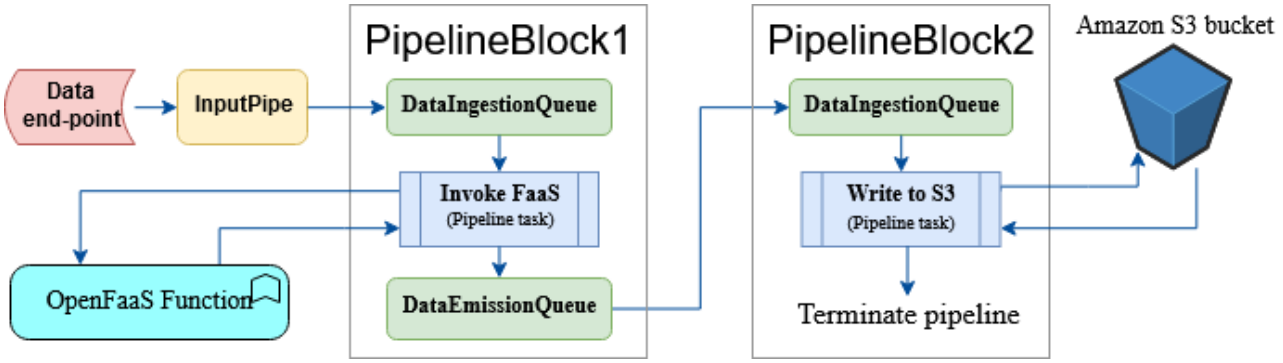

Figure 3.2.2: An example of two connected pipelines.

3.3. TOSCA extension for data pipeline

The modeling of data pipelines, as presented in Radon Deliverable 4.3 [2], is updated based on the components described in the above Section 3.2. In general, the *PipelineBlocks* can be categorized into: *SourcePB*, *DestinationPB*, *MidwayPB*, as shown in Figure 3.3.1, based on the functionalities.

SourcePB

SourcePB represents the *PipelineBlock* for reading (or consuming) the data from one data end-point. This special *PipelineBlock* is designed with no incoming-connection from another *PipelineBlock*. However, the outgoing-connections can be to another *MidwayPB* or another *DestinationPB*. The number of outgoing-connections can be more than 1. To create a TOSCA node of this type, users need to provide other pipelines information to where the consumed data are forwarded. The consumption of data can be of two types: *ConsumeRemote*, and *ConsumeLocal*, as shown in Figure 3.3.1. As the name suggests, *ConsumeRemote* is primarily for consuming the data from remote data end-points, such as Amazon S3Bucket. In case of consuming the data from a remote S3 bucket, users need to provide the basic information such as bucket name, bucket region, credentials, etc. to fetch the data objects, in the TOSCA application blueprint. Furthermore, we will be studying other ways

of consuming the data (e.g. consuming remote data over FTP, http) from other remote data end-point (e.g. remote SQL database, Kafka topic).

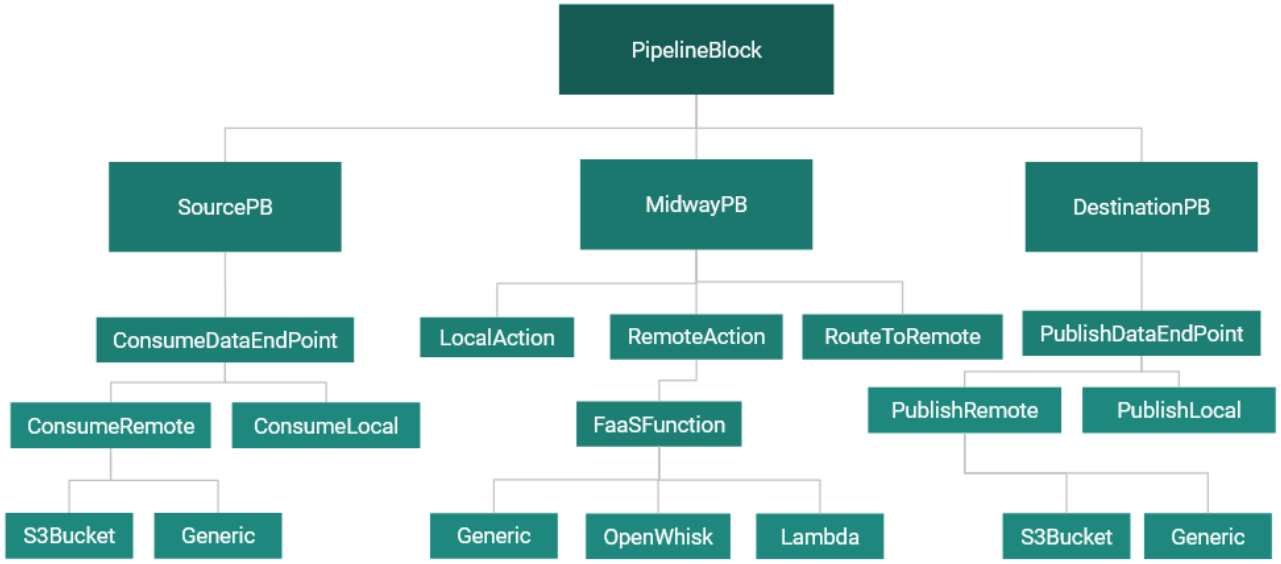

Figure 3.3.1: TOSCA extension for data pipeline.

MidwayPB

This pipeline is derived from *PipelineBlock* and is mainly used to implement the pipeline tasks (or actions) related to data processing. An action on any data can be: *LocalAction* (processing the data in local system), *RemoteAction* (invoking a FaaS function), or *RouteToRemote* (routing the data from one host to another host). *RemoteAction* mainly refers to invoking a FaaS function. We are developing the corresponding TOSCA template for private FaaS provider (i.e. Amazon lambda), and public FaaS provider (i.e. using OpenFaaS). For invoking Amazon lambda function, *PutLambda* is used. In addition to this, the TOSCA node definition for invoking a generic FaaS function will be developed, for which the corresponding NiFi processor (*InvokeHTTP*) are used. *InvokeHTTP* interacts with a configurable HTTP Endpoint.

Furthermore, from our investigation, it is concluded that an action can be related to routing the data to another host. This situation may arise when two *PipelineBlocks* are in different hosts. In such case, users need to create the node of type *RouteToRemote*. If any condition is required to implement to route the data, it has to be implemented with separate node as a *LocalAction*. For example, user may need to route the data to different *PipelineBlock* on different host based on the size and type of the data. For this, user needs to create a node of type *LocalAction* implementing the routing conditions as actions.

DestinationPB

Similarly, to write (or publish) any data to another data end-point, users need to create TOSCA node of type *DestinationPB*. For the node of this type, the number of incoming-connections can be more than 1. However, the number of outgoing-connections is restricted to 0. The data can be published to local or remote data end-points. For publishing the data to remote Amazon S3Bucket, users need to

create the node of specific type (S3Bucket node). In the context of S3Bucket, the corresponding NiFi processor (*PutS3object*) is used as the underlying tool. The remote data end-point can also be a generic one, which can be invoked using other protocols such as HTTP, FTP, etc. This node type definition needs further study and will have a detailed description in the next (or final) RADON deliverable, Data pipeline orchestration II - due on M22. For publishing the data to local file system, the corresponding NiFi processor (*PutFile*) is used as the underlying tool.

4. Data pipeline interacting with other RADON components

The outline and overall architecture of the RADON framework have been presented and described in RADON deliverable D2.3-Architecture and integration plan I [1]. Deliverable D2.3 [1] provides a detailed description of different RADON components including RADON IDE, Graphical Modeling Tool, Delivery Toolchain, Verification Tool, Continuous Testing Tool, Constraint Definition Language Tool, Template Library, and Orchestrator. The current deliverable (i.e. D5.5 - Data pipeline orchestration I) provides a preliminary architecture of data pipeline in the scope of RADON. This section presents the interaction of designed data pipeline architecture with other RADON components, as shown in Figure 4.0.1.

4.1. RADON Orchestrator

This is another primary component that orchestrates the node template and relationship template of the TOSCA-based application blueprint. The data pipelines are deployed on the infrastructure using the RADON orchestrator (xOpera). The orchestrator i) interprets the TOSCA-based data pipeline model, ii) prepares the data pipeline infrastructure on the provider, and iii) deploys and configures the given data pipeline application. The orchestrator controls the life cycle (creation, configuration, starting, stopping, and deleting) of TOSCA data pipelines nodes that eventually automates the flow of data from one end-point to others. To achieve these complex tasks a special *library* of RADON particles, will be developed and published on the RADON template library by extending specification of TOSCA language. These extensions enable the management of pre-deployed equipment (as Apache NiFi) required for processing the data pipelines and somehow provides the abstraction of the infrastructure for the data pipeline applications.

Currently the Template Library is not yet effective, and it is still in development phase and also xOpera does not support the CSAR structure which allows importing such particles. Currently xOpera can be used only with a single yaml and only deploy and undeploy playbooks that are stored separately. According to the orchestrator plans presented in the deliverable D5.1 these improvements are planned for Y2, where more complex data pipeline examples can be deployed and managed.

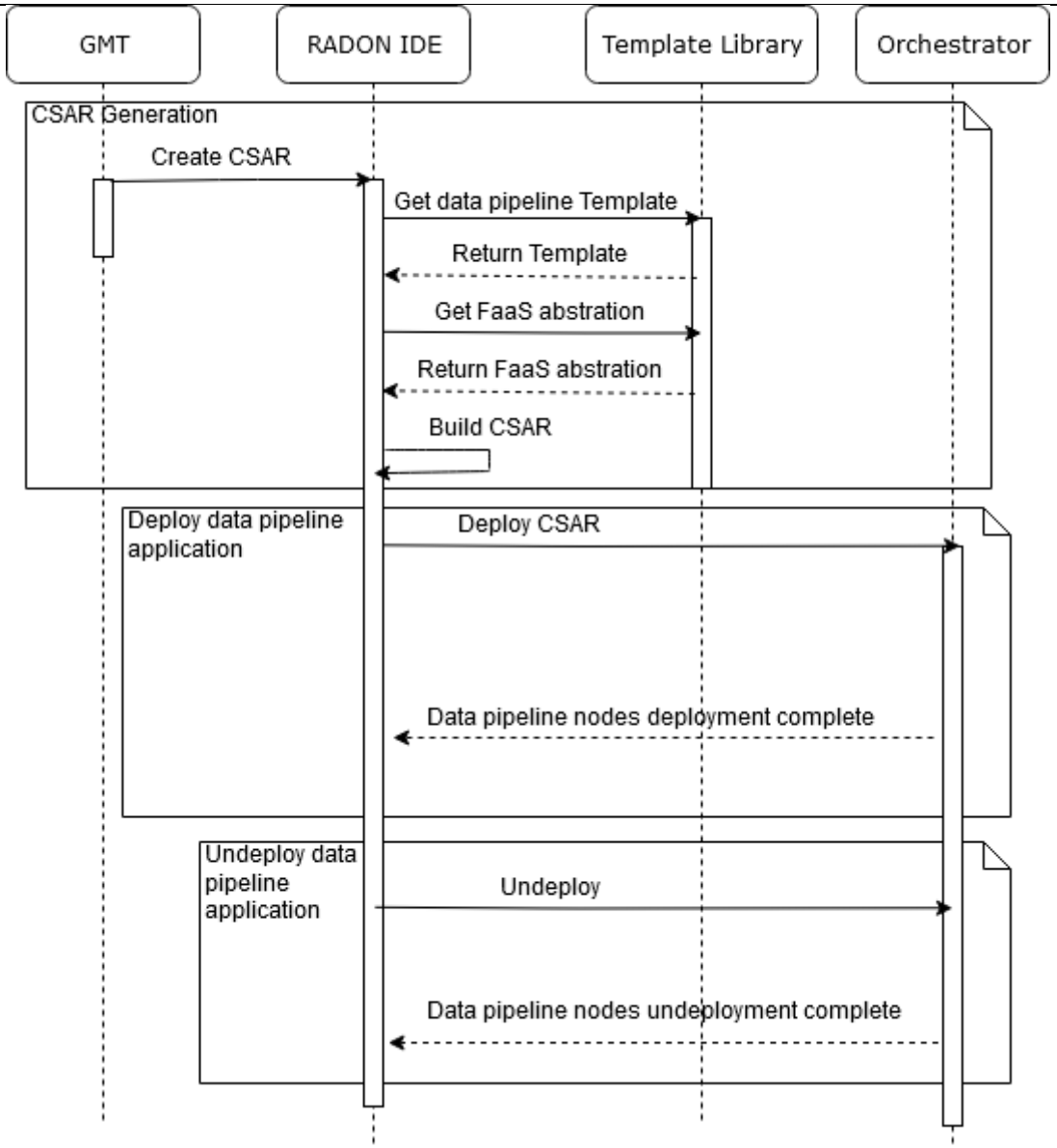

Figure 4.0.1: Interaction with other RADON tools

4.2. Template Library

Template library is a repository consisting of the definitions of application blueprint, application abstractions (including data pipeline components), high-level system abstractions, and the extension of TOSCA language. In addition to the extension of TOSCA language, the template library also provides the templates for serverless platform. This template library for serverless platform includes the definitions of platform provider, gateway, function name, language, url to the function, etc. The library includes the definition for pipeline blocks, where user/developer can provide the pipeline artifacts such as host, port, pipeline connection, etc. In deliverable D4.3 [2], the initial version of RADON model is presented, which includes the TOSCA extended model for data pipeline.

The TOSCA node definitions for data pipeline, as presented in [2], is updated keeping the functionalities of Apache NiFi and Amazon data pipeline in mind. The current updates are pushed to the template library in radon particles GitHub repository[4] after having a thorough discussion in consortium. Upon which, users are offered with mainly three types of data pipeline nodes: pipeline for source node with the functionality to read the data, pipeline for destination node with the functionality to write the data, and pipelines for processing the data locally or by invoking a FaaS function such as AWS lambda.

4.3. Graphical Modeling Tool

Graphical Modeling Tool (GMT) provides a web-based environment, where the user can graphically model their data pipeline applications in a drag-and-drop fashion. To make the process of designing the TOSCA-based application simpler, GMT provides several features, such as i) modeling application topologies, ii) modeling non-functional requirements, iii) creating and modifying reusable types, and iv) exporting orchestrator-ready TOSCA artifacts [1]. The interface would provide a set of components related to data pipeline, such as PipelineBlock, InputPipe, OutputPipe, etc. GMT allows the users to view, edit, and modify the existing data pipeline models that are created from the template library by the RADON IDE within the current local workspace. Using the application specific data pipeline models, further GMT creates the corresponding TOSCA template for the orchestrator to deploy and implements the life cycle each PipelineBlocks and other components of data pipeline. This may also allow the users to provide the application artifacts within the GUI. GMT allows the newly created data pipeline nodes to be saved and reused in different applications. This tool is used to generate the DATA pipeline CSAR, as shown in Figure 4.0.1.

4.4. RADON IDE

As discussed before, several categories of the functionalities are supported by RADON IDE, such as management of i) authorization, ii) RADON workspaces, iii) application blueprints, iv) test specification, and v) deploy application [1]. As the application blueprint may contain a number of data pipeline blocks as nodes, RADON IDE also provides the functionalities to carry-out all kinds of data pipeline related development activities, such as deployment, testing, etc. Upon successful login to the RADON IDE and creation of a new workspace, RADON Particles including the TOSCA extended model for the data pipelines are retrieved by the RADON IDE from the Template library. Further, by creating a new project, offered by IDE' workspace management, GMT would be able to show the designated icons for graphically modeling a data pipeline application. Upon modeling the application through RADON IDE, the orchestrator deploys the graphically modeled data pipeline application.

In the entire journey of modeling an application and its deployment, users can monitor the data pipeline applications through RADON monitoring system, test the data pipeline applications with

[4] https://github.com/radon-h2020/radon-particles

Continuous Testing Tool (CTT), impose application specific constraint using Constraint Definition Language (CDL) tool, and verify the application using RADON Verification Tool (VT).

5. Example of data pipeline

In this section, we explain the implementation of the Toy application example, which refers to the generation of thumbnails from original images. The example is implemented in two ways: (a) single node implementation, (b) multi-nodes implementation. A Node in this example, is referring to a virtual machine, or instance, or a host. For implementation, Apache NiFi v1.9.2 and private cloud infrastructure, OpenStack, is used.

5.1. Thumbnail generation example

Implementation of the thumbnail generation example mainly consists of three steps:

1. Reading images from one Amazon S3 bucket.
2. Invoking lambda function with the image as input and receiving the corresponding thumbnail.
3. Pushing the thumbnail to the same or different Amazon S3 bucket.

All the above three steps can be implemented within one lambda function. However, in doing so, it is hard to get the benefits of freely composable, portable and reusable microservices or serverless function. In this deliverable, we implement the thumbnail generation example following the data pipeline concepts. RADON data pipeline divides the entire problem into three sub-problems, where each sub-problem can be designed to act as independent pipeline component, with a predefined set of input and output.

5.2. Thumbnail generation on single instance

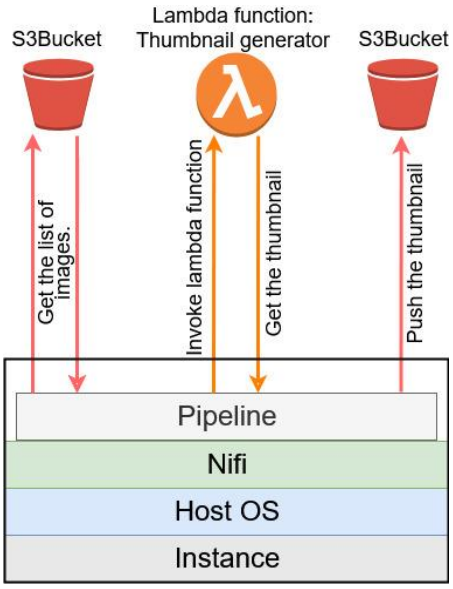

Figure 5.2.1: Thumbnail generation on single node/instance

The architectural view of the implementation of thumbnail generation on single node can be seen in Figure 5.2.1. In Figure 5.2.1, the first task is to create an instance in private cloud provider, i.e. in OpenStack environment with a host operating system. In this implementation, Centos is installed as the host OS on the OpenStack instance. Atop the host OS, Apache NiFi is installed. This provides an environment to create, deploy and manage multiple pipelines within one instance. Using Apache NiFi, a pipeline is created to handle all the three major steps involved in this example, as explained in Section 5.1.

5.2.1 TOSCA Type definition

Listing 5.2.1 shows the TOSCA node type for managing the OpenStack instances. The main properties of this TOSCA node type are the name of the instance, instance image or the OS, the flavor of the instance, OpenStack key name, etc. Upon successful creation of the instance, the runtime attributes such as `id` and `public_address`, are captured. This TOSCA node type is basically derived from `tosca.nodes.Compute` node type provided by the TOSCA standard[5]. The `id` attribute is mainly used while deleting the instance.

Listing 5.2.1. Node type for creating RADON compute instances

```
node_types:
  radon.nodes.VM.OpenStack:
    derived_from: tosca.nodes.Compute
    properties:
      name:type=string
      image:type=string
      flavor:type=string
      network:type=string
      key_name:type=string
    attributes:
      id:type=string
      public_address:type=string
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        create:
          inputs:
            vm_name: {default:{get_property:[SELF, name    ]}}
            image:   {default:{get_property:[SELF, image   ]}}
            flavor:  {default:{get_property:[SELF, flavor  ]}}
            network: {default:{get_property:[SELF, network ]}}
            key_name:{default:{get_property:[SELF, key_name]}}
          implementation: nodetypes/vm/create.yml
        delete:
          inputs:
            id:{default:{get_attribute:[SELF, id]}}
          implementation: nodetypes/vm/delete.yml
```

On each instance, Apache NiFi is installed. Listing 5.2.2 give the node type for creating such TOSCA node. This TOSCA node is derived from basic *SoftwareComponent* node type provided by TOSCA standard. The major properties are the version of the NiFi and the port number. Both the properties

[5] http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html

are mandatory for creating any node of this type. As shown in Figure 5.3.1, this software component further must have the capability to host another node of type *NifiPipeline*.

Listing 5.2.2. Node type for creating NiFi software component

```
radon.nodes.nifi.Nifi:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
      component_version:
        required: true
      port:
        required: true
        default: 8080
    capabilities:
      host:
        type: tosca.capabilities.Container
        valid_source_types:[radon.nodes.nifi.NifiPipeline]
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        inputs:
          tarball_version:{default:{get_property:[SELF, component_version]}}
        create: ...
        start: ...
        stop: ...
        delete: ...
        configure: ...
```

Atop NiFi software component, one or more data pipelines are created. The TOSCA node type for creating the data pipeline node is given in Listing 5.2.3. The basic four properties of a NiFi pipelines are the path to the template file, the name of the template, path to the credential file and the name of the object. For each pipeline, NiFi accept an XML template file that includes detailed information regarding the pipeline. The template is then uploaded to the targeted instance, which is deployed onto the local NiFi template repository. The name of the template must match with the name given in the template file. Credential file is optional and is used to get or push the images from/to the S3 buckets and invoke the lambda function. In this example, the credential file is uploaded to a temporary directory.

Listing 5.2.3. Node type for creating nodes of type NiFi pipeline

```
radon.nodes.nifi.NiFiPipeline:
    derived_from: radon.nodes.abstract.DataPipeline
    properties:
      template_file: type=string
      template_name: type=string
      cred_file_path: type=string
      object_name: type=string
    artifacts:
      pipeline_template:
        description: The pipeline template XML file
        type: tosca.artifacts.File
    attributes:
      id: type=string
      pipeline_type: type=string
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: radon.nodes.nifi.Nifi
```

```
          relationship: tosca.relationships.HostedOn
      - connect:
          capability: tosca.capabilities.Endpoint
          node: radon.nodes.nifi.NifiPipeline
          relationship: tosca.relationships.ConnectsTo
    capabilities:
      connect:
        type: tosca.capabilities.Endpoint
        description: Capability to receive data from other pipeline nodes
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        inputs:
            template_file: {default:{get_property:[SELF, template_file]}}
            template_name: {default:{get_property:[SELF, template_name]}}
            pipeline_type:{default:{get_attribute:[SELF, pipeline_type]}}
            pipeline_id:{default:{get_attribute:[SELF, id]}}
            cred_file_path:{default:{get_property:[SELF, cred_file_path]}}
            object_name:{default:{get_property:[SELF, object_name]}}
        create: ...
        start: ...
        stop: ...
        configure: ...
        delete: ...
```

5.2.2. TOSCA node definition

Based on the above TOSCA node type definitions, the current implementation of thumbnail generation example consists of three TOSCA nodes definitions: i) node for creating OpenStack Instance ii) node for Apache NiFi platform, and iii) node for deploying the pipeline, as described below.

Listing 5.2.4, gives the definition of TOSCA node for creating and implementing the lifecycle of OpenStack instances. This single instance, named `nifihost_utr` will host the Apache NiFi and the pipeline atop NiFi platform. The node `vmone` is defined with the type `radon.nodes.VM.OpenStack`, which needs five basic inputs: name, image, flavor, network, and the name of the key. These inputs are then passed to an Ansible playbook[6] to create the instance and upon successful creation, the `id`, and `public_address` are captured to the corresponding attribute variables.

Listing 5.2.4. Node for creating and managing OpenStack instance

```
vmone:
      type: radon.nodes.VM.OpenStack
      properties:
        name: nifihost_utr
        image: d3eeb4ae-9b8c-49ed-8314-123a8a808b2b
        flavor: 6b254b9e-db1c-40de-994c-07d69dd732a6
        network: provider_64_net
        key_name: rem-VM2
```

The node definition for Apache NiFi platform installation and configuration is given in Listing 5.2.5.

6 https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline/blob/master/single-node-example/nodetypes/vm/create.yml

`nifi_vmone` node of type `radon.nodes.nifi.Nifi` requires `vmone` node to be ready, and the version of NiFi, in our case version 1.9.2, as the property value. This provides an environment to upload, deploy, and manage the one or more pipeline blocks.

Listing 5.2.5. Node for installing and configuring Apache NiFi platform

```
  nifi_vmone:
    type: radon.nodes.nifi.Nifi
    requirements:
      - host: vmone
    properties:
      component_version: "1.9.2"
```

To implement the above example on single instance, one pipeline node, `pipeline1_thumbgen`, is used. This TOSCA node uploads the NiFi template file[7] from the local file system, as provided in `template_file` property value, to the NiFi environment in remote instance, `nifihost_utr`. The uploaded template file `nifi_thumbgen.xml` further contains a set of sub-pipeline blocks as shown in Figure 5.2.2. Mainly three NiFi processors: `ListS3`, `FetchS3Object`, and `PutS3Object`, and one process group, `invokeLambda_PG`, are used to implement the basic three steps of the above example. The process group `invokeLambda_PG` further consists of several NiFi processors, such as `Base64EncodeContent`, `PutLambda`, `ReplaceText`, `UpdateAttribute`, etc. `Base64EncodeContent` is used to encode and decode the jpg file to/from base64 format. `PutLambda` is used to invoke the Amazon lambda function[8]. `ReplaceText` is used to extract the JSON data from the returned value of lambda function. The uploaded NiFi template file is then deployed using REST API. In addition to this, the credential file[9] is uploaded to the `/tmp/` directory of `nifihost_utr` instance. This credential file is used to provide the `accessKey` and `secretKey` to access Amazon S3 buckets and invoke lambda function.

Listing 5.2.6. Node for deploying and starting the thumbnail generation *Pipeline*

```
pipeline1_thumbgen:
   type: radon.nodes.nifi.NiFiPipeline
   requirements:
     - host: nifi_vmone
   properties:
     template_file: "/lambda-thumbGen-TOSCA-dp/files/nifi_thumbgen.xml"
     cred_file_path: "/lambda-thumbGen-TOSCA-dp/files/credentials"
     object_name: "NifiThumbGen"
     template_name: "NifiThumbGen"
   artifacts:
     pipeline_template:
       file: "/lambda-thumbGen-TOSCA-dp/files/nifi_thumbgen.xml"
```

[7] https://github.com/radon-h2020/lambda-thumbgen-toca-datapipeline/blob/master/single-node-example/files/nifithumbgen.xml

[8] https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline/tree/master/lambdafunction

[9] https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline/blob/master/single-node-example/files/credentials

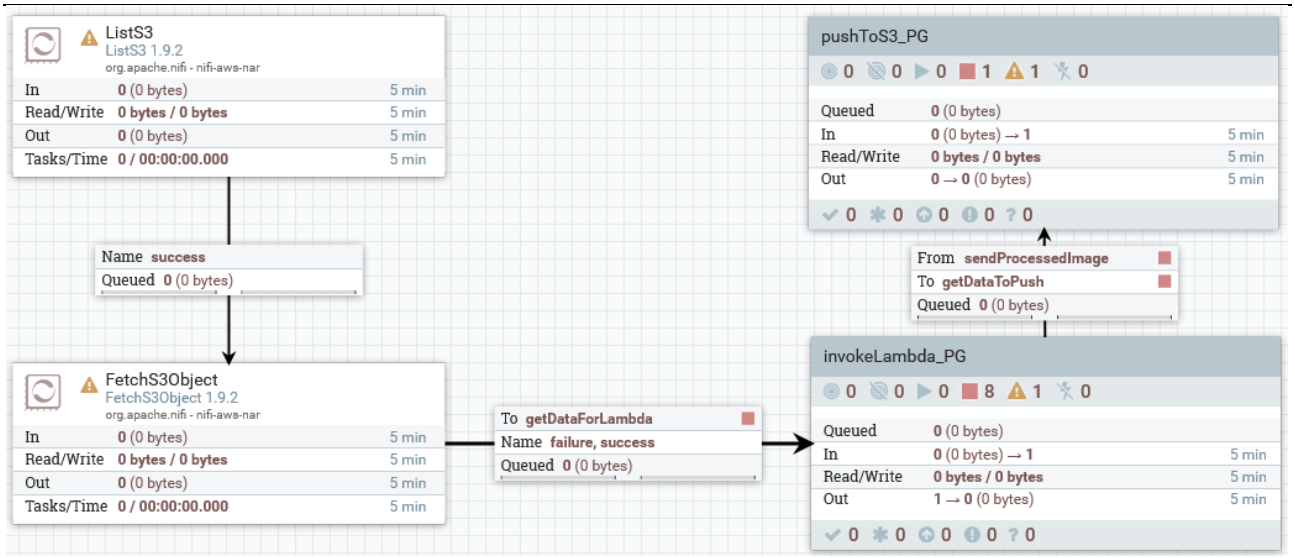

Figure 5.2.2: Detailed pipeline blocks of thumbnail generation on single instances

5.3. Thumbnail generation on multiple instances

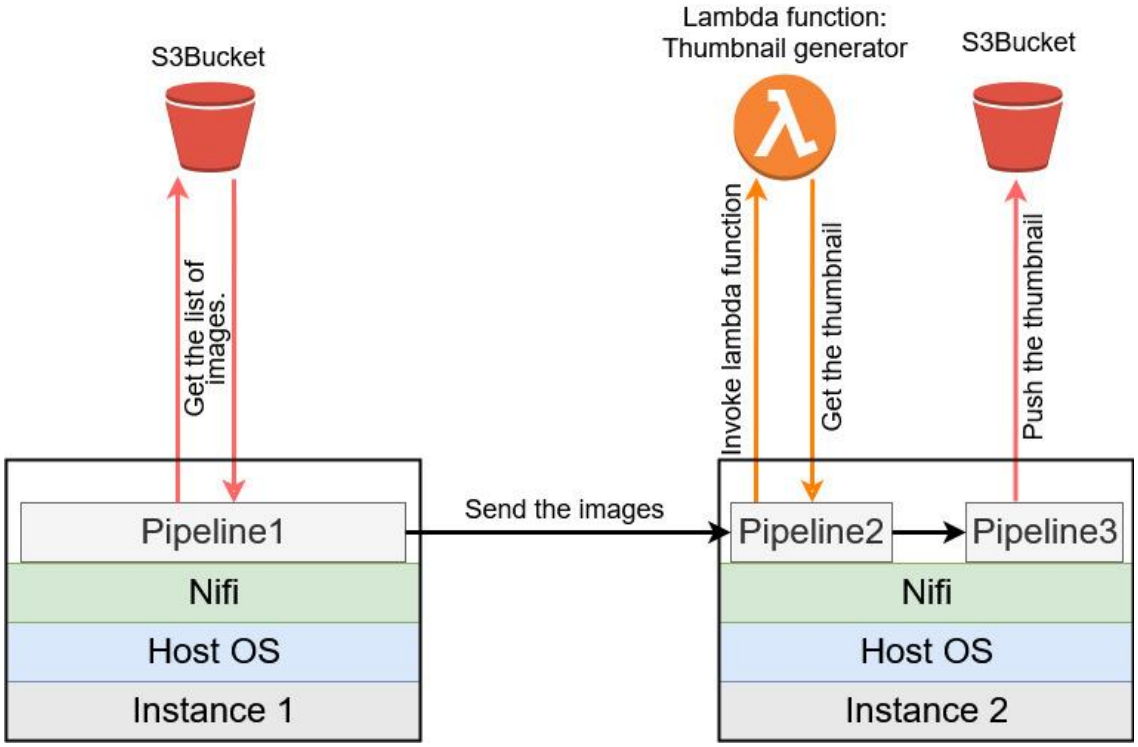

Figure 5.3.1: Thumbnail generation on multiple instances

As shown in Figure 5.3.1, The example[10] is implemented with three pipeline blocks: i) *Pipeline1* on *Instance 1* to access the images from Amazon S3 bucket, ii) *Pipeline2* on *Instance 2* to invoke the

[10] https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline

lambda function, and iii) *Pipeline3* on *Instance 2* to push the thumbnail to different Amazon S3 bucket. All the pipelines should be designed in such a way that they can be reused to achieve different bigger tasks, fulfilling the objective of reusability. The corresponding TOSCA service model is given in Listing 5.3.1 - 5.3.8.

The TOSCA template mainly consists of 5 different types of nodes: i) node type for creating the instance, ii) node type for creating, configuring and starting Apache NiFi, iii) node type for creating and managing basic pipelines, iv) node types for establishing the connection, v) node types for special types of pipeline that provides a way to sending the data from one instance to another instance.

5.3.1 TOSCA Type definition

The TOSCA node type to manage the OpenStack instance mentioned in Listing 5.2.1 is used to create two instances in this implementation. Similarly, to install and manage the NiFi platform on both instances, the type definition for NiFi platform mentioned in Listing 5.2.2 is used. Atop NiFi platform, the pipelines are deployed using NiFi template file. For this, the type definition mentioned in Listing 5.2.3 is used in this implementation. In this example, on first instance i.e. *Instance 1*, one pipeline is created and on *Instance 2* two pipelines are created, as shown in Figure 5.3.1.

In addition to the TOSCA node type definitions, provided in Section 5.2, `radon.nodes.nifi.NiFiPipelineConnection` is a special type of nodes, used to establish the connection between two TOSCA nodes sharing the same instance, as shown in Listing 5.3.1. The essential properties of any TOSCA node of this type are: `output_port_name`, `source_group_id`, `input_port_name`, and `destination_group_id`. From the source pipeline, `output_port_name` and `source_group_id` are required. On the other hand, from the destination pipeline, `input_port_name` and `destination_group_id` are required.

Listing 5.3.1. Node type for creating node of type pipeline connection

```
radon.nodes.nifi.NiFiPipelineConnection:
    derived_from: radon.nodes.abstract.DataPipeline
    properties:
      input_port_name:
        type: string
        description: input_port_name (present in the DESTINATION pipeline)
      output_port_name:
        type: string
        description: output_port_name
      source_group_id:
        type: string
        description:  source_group_id
      destination_group_id:
        type: string
        description: destination_group_id
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: radon.nodes.nifi.Nifi
          relationship: tosca.relationships.HostedOn
      - connect:
          capability: tosca.capabilities.Endpoint
          node: radon.nodes.nifi.NifiPipeline
          relationship: tosca.relationships.ConnectsTo
```

```
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        inputs:
            input_port_name:{default:{get_property:[SELF, input_port_name]}}
            output_port_name:{default:{get_property:[SELF, output_port_name]}}
            source_group_id:{default:{get_property:[SELF, source_group_id]}}
            destination_group_id:{default:{get_property:[SELF, destination_group_id]}}
        create: ...
```

The above special type of pipeline does not offer any way to establish the connection between two pipelines on different instances. The TOSCA node type definition for connecting two pipelines on different instance is given in Listing 5.3.2. This scenario occurs in thumbnail generation example, when the retrieved images on *Instance 1* are transferred to *Instance 2* for invoking lambda function. For this purpose, two primary properties are required, namely `remote_host`, `remote_port`. While deploying, it is assumed that the pipeline in remote host is already deployed and started. The default port number of the remote NiFi is 8080.

Listing 5.3.2. TOSCA Node type for creating remote pipeline nodes

```
radon.nodes.nifi.NiFiPipeline.remotePipeline:
    derived_from: radon.nodes.nifi.NiFiPipeline
    properties:
      remote_host:
        type: string
        required: true
      remote_port:
        type: string
        default: 8080
        required: true
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        create: ...
        start: ...
        stop: ...
        configure:
          inputs:
            pipeline_id:{default:{get_attribute:[SELF, id]}}
            pipeline_type:{default:{get_attribute:[SELF, pipeline_type]}}
            cred_file_path:{default:{get_property:[SELF, cred_file_path]}}
            object_name:{default:{get_property:[SELF, object_name]}}
            remote_host:{default:{get_property:[SELF, remote_host]}}
            remote_port:{default:{get_property:[SELF, remote_port]}}
          implementation:
nodetypes/radon/nodes/nifi/NiFiPipeline/files/configure_RPG.yml
        delete:  ...
```

5.3.2 TOSCA Node definition

Based on the above TOSCA node types, the implementation of thumbnail generation consists of 8 different nodes. The first two nodes: `vmone` and `vmtwo`, as shown in Listing 5.3.3, are mainly for managing the lifecycle of two instances. The inputs for these nodes include name, image or host OS, flavor of the instance, network and the access key name. It is assumed that Ansible is able to access the OpesStack cloud environment. For this, OpenStack client should be installed before executing

xOpera. Further, the key name should be available with the OpenStack client. In this implementation, `vmtwo` node depends on `vmone`.

Listing 5.3.3. Nodes for creating and managing OpenStack instances

```
vmone:
      type: radon.nodes.VM.OpenStack
      properties:
        name: nifihost_one_utr
        image: d3eeb4ae-9b8c-49ed-8314-123a8a808b2b
        flavor: 6b254b9e-db1c-40de-994c-07d69dd732a6
        network: provider_64_net
        key_name: rem-VM2
vmtwo:
      type: radon.nodes.VM.OpenStack
      properties:
        name: nifihost_two_utr
        image: d3eeb4ae-9b8c-49ed-8314-123a8a808b2b
        flavor: 6b254b9e-db1c-40de-994c-07d69dd732a6
        network: provider_64_net
        key_name: rem-VM2
      requirements:
        - dependency: vmone
```

Apache NiFi is installed top of both instances. The TOSCA nodes `nifi_vmone` and `nifi_vmtwo` require the instances `vmone` and `vmtwo`, respectively, to be ready. For each NiFi node, `create.yml` Ansible file, as in Listing 5.2.2., downloads Apache NiFi package based on the version provided, as in Listing 5.3.4, and installs the downloaded software package in `usr/local/bin/` location. This follows by the configuration of NiFi installation with the port number by invoking `configuration.yml` Ansible file and starting the NiFi service by invoking `start.yml` Ansible playbook, as provided in Listing 5.3.4.

Listing 5.3.4. Nodes for installing and configuring Apache NiFi platform on both instances

```
    nifi_vmone:
      type: radon.nodes.nifi.Nifi
      requirements:
        - host: vmone
      properties:
        component_version: "1.9.2"
    nifi_vmtwo:
      type: radon.nodes.nifi.Nifi
      requirements:
        - host: vmtwo
      properties:
        component_version: "1.9.2"
```

As soon as NiFi platform is ready on both instances, the data pipeline blocks are uploaded and deployed. For each pipeline, it is assumed that the corresponding template file `(.xml)` is present in the current directory or in any accessible directory. To satisfy the dependency and to align with current version of xOpera, we are first deploying the *Pipeline3* (in Figure 5.3.1). *Pipeline3* mainly pushes the thumbnail to different S3 bucket. `pipeline3_pushImg` TOSCA node, as shown in Listing 5.3.5, is mainly responsible for uploading, deploying, starting the *Pipeline3* on *Instance 2*. This requires the path to the pipeline template file `pushToS3_PG.xml`. It is assumed that the S3 bucket is already created. The information (including bucket name, region) of the S3 bucket is present in the NiFi template file.

Listing 5.3.5. Node for deploying and starting *Pipeline3*

```
  pipeline3_pushImg:
    type: radon.nodes.nifi.NiFiPipeline
    requirements:
      - host: nifi_vmtwo
    properties:
      template_file: DataPipelineToyExample/files/pushToS3_PG.xml
      cred_file_path: "DataPipelineToyExample/files/credentials"
      object_name: "PutS3Object"
      template_name: "pushToS3_PG"
    artifacts:
      pipeline_template:
        file: DataPipelineToyExample/files/pushToS3_PG.xml
```

`pipeline2_invokeLmabda` TOSCA node in Listing 5.3.6 is created from the TOSCA node type `radon.nodes.nifi.NiFiPipeline` (in Listing 5.2.3), to create *Pipeline2* of the example in Figure 5.3.1. *Pipeline2* receives the original images and invokes the lambda function for each image to get the thumbnail. It is assumed that the lambda function[11] is uploaded and configured to process the original image for thumbnail generation. This pipeline consists of several small pipelines to perform following primary tasks:

1. Receive the input image in JPG format.
2. Encode JPG image to base64 code.
3. Convert multi-line *base64* code to single-line *base64* code.
4. Invoke lambda function with encoded image.
5. Receive the encoded thumbnail.
6. Decode *base64* to JPG and change the MIME type to image.
7. Forward the thumbnail to next pipeline.

All the above tasks are mentioned in NiFi template file[12]. In the current configuration, lambda function is invoked once every 60 second.

Listing 5.3.6. Node for deploying and starting *Pipeline2*

```
  pipeline2_invokeLmabda:
    type: radon.nodes.nifi.NiFiPipeline
    requirements:
      - host: nifi_vmtwo
    properties:
      template_file: DataPipelineToyExample/files/invokeLambda_PG.xml
      cred_file_path: "DataPipelineToyExample/files/credentials"
      object_name: "invokeLambda"
      template_name: "invokeLambda_PG"
    artifacts:
      pipeline_template:
        file: DataPipelineToyExample/files/invokeLambda_PG.xml
```

To enable the flow of thumbnail from `pipeline2_invokeLmabda` to `pipeline3_pushImg` node, `pipelineConnection` node is created of type `radon.nodes.nifi.NiFiPipelineConnection` in Listing 5.3.7. This node requires the

[11] https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline/tree/master/lambdafunction

[12] https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline/blob/master/files/invokelambda_pg.xml

above two nodes ready to receive or send image data. In the current form of the implementation, this requires mainly specifying the following information: name of output port, id of the source pipeline group, the name of the destination port, and the id of the destination pipeline group. The ids of the source pipeline group and destination pipeline group are obtained dynamically through NiFi REST API.

Listing 5.3.7. Node for connecting *Pipeline3* and *Pipeline2*

```
pipelineConnection:
  type: radon.nodes.nifi.NiFiPipelineConnection
  requirements:
    - host: nifi_vmtwo
  properties:
    output_port_name: "sendProcessedImage"
    source_group_id: "none"
    input_port_name: "getDataToPush"
    destination_group_id: "none"
```

The TOSCA node `pipeline1_getS3Img`, in Listing 5.3.8, is created to implement the functionalities of *Pipeline1* (in Figure 4.3.1). This pipeline is of type `radon.nodes.nifi.NiFiPipeline.remotePipeline`, as defined in Listing 5.3.2. This TOSCA node uploads and deploys the `read_from_s3_RPG.xml`[13] NiFi template to *Instance 1*. The primary tasks of this pipeline are:

1. Get the list of images (*JPG* files),
2. Retrieve actual *JPG* image files based on the list,
3. Establish the connection with NiFi on *Instance 2*, port number *8080*, and
4. Send each *JPG* file to the remote instance, i.e. *Instance 2*.

It is assumed that the Amazon S3 bucket is already created manually and contains at least one JPG file to verify the result. In order to establish the communication, *Pipeline2* (`pipeline2_invokeLmabda` node) on *Instance 2* is assumed to be ready to accept any files from remote instances. The images are sent using RemoteProcessGroup NiFi processor. This special NiFi processor is mainly used to send data to remote instance. In the implementation of thumbnail generation, the image files are sent over HTTP.

Listing 5.3.8. Node for deploying and starting *Pipeline1*

```
pipeline1_getS3Img:
  type: radon.nodes.nifi.NiFiPipeline.remotePipeline
  requirements:
    - host: nifi_vmone
  properties:
    template_file: "DataPipelineToyExample/files/read_from_s3_RPG.xml"
    cred_file_path: "DataPipelineToyExample/files/credentials"
    object_name: "ListS3"
    template_name: "read_from_s3_RPG"
    remote_host: "nifihost_two_utr"
    remote_port: "8080"
  artifacts:
```

[13] https://github.com/radon-h2020/lambda-thumbgen-tosca-datapipeline/blob/master/files/read_from_s3_rpg.xml

```
        pipeline_template:
          file: "DataPipelineToyExample/files/read_from_s3_RPG.xml"
```

6. Vision for the data pipeline plugin

The deliverable in the previous sections presented the initial version of the data pipeline methodology. The DP methodology and its orchestration are demonstrated with the discussed RADON thumbnail generation example, both in single and multi-node environments. We also have discussed the respective DP TOSCA templates (initial versions), which are stored in the template library. From these templates for multi-node environment, we have observed additional requirements for the orchestration of those TOSCA DP model, at the runtime. We have studied certain scenarios, where user created TOSCA model needs to be modified without user intervention. For instance, the relationship type between two DP blocks also depends on the underlined hosts. The relationship type will be local if two DP blocks are hosted on the same machine. Whereas the relationship type will be remote, and an exchange of host addresses is involved when two DP blocks are hosted on different machines. However, developers may not consider the underlined host while establishing the relationship between two DP blocks, during the design phase. Those two DP blocks may share the same host. However, in case of different hosts, the relationship type needs to be updated in runtime before the service model is passed to the RADON orchestrator. This gives the scope for the envisioned data pipeline plugin, as given in Figure 2 in Deliverable D2.3 [1].

Our vision and interactions of the data pipeline plugin with other tools of RADON is shown in Figure 6.0.1. This will be explored further during the second year of the project. The CDL tool will be equipped with a set of DP rules that will mainly check the user's TOSCA-based service model designed using RADON GMT. The CDL tool will notify the DP plugin with required auxiliary information through RADON IDE, if it found the TOSCA-based DP service model violates one or more DP rules. The auxiliary information would mainly indicate the portion of the service model where the DP rules violation occurs. For example, CDL tool may indicate which relationship type should be updated by DP plugin. In such a scenario the DP plugin will update the TOSCA templates and CSAR, before it is actually pushed to the RADON orchestrator.

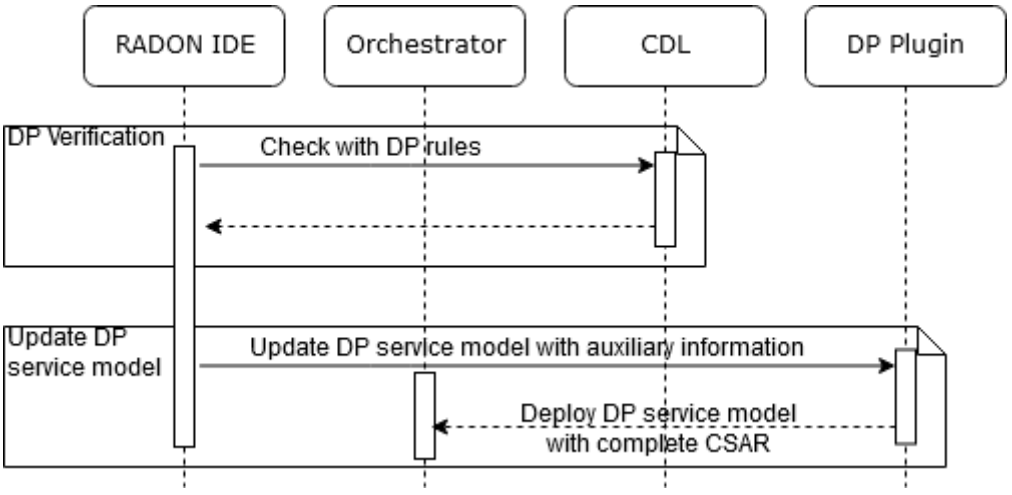

Figure 6.0.1: Envisioned DP plugin and its interaction

7. Conclusion and future work

In this deliverable, we have described the initial version of data pipeline methodology and its orchestration along with the new components. We have presented the updated version of TOSCA extension for data pipeline-based application design. This report presents the methodology and overall architecture of data pipeline, the interaction of data pipelines with other RADON components, and the implementation of RADON thumbnail generation example on single and multiple instances. In addition to this, we have presented the future plan continuing the current work.

Table 7.0.1 shows the achieved level of compliance to RADON requirements. The values in "Level of compliance" are defined as follows:

(i) ✗ (unsupported): the requirement is not fulfilled by the current version

(ii) ✔ (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version

(iii) ✔ ✔ (partially-high supported): most of the aspects of the requirement is fulfilled by the current version

(iv) ✔ ✔ ✔ (fully supported): the requirement is fulfilled by the current version.

Table 7.0.1: Achieved level of compliance to RADON requirements

Id	Requirement Title	Priority	Level of compliance
R-T5.4-1	The data pipeline module of the Orchestrator must be able to orchestrate data pipelines	Must have	✔
R-T5.4-2	The data pipeline module of the Orchestrator must support cron based scheduled data pipelines	Must have	✔
R-T5.4-3	The data pipeline module of the Orchestrator should support event based scheduled data pipelines	Should have	✔
R-T5.4-4	It would be useful for the data pipeline module of the Orchestrator to support logging and generating alerts on pipeline task failures.	Should have	✗
R-T5.4-5	The data pipeline module of the Orchestrator should support deployment of data pipelines which automate movement of data between two or more clouds	Should have	✔ ✔
R-T5.4-6	The data pipeline module must support data pipelines tasks that initiate data analytics tasks for processing data moving	Must have	✗

	through the pipeline		
R-T5.4-7	The data pipeline module of the Orchestrator should support configuring encryption between data pipeline tasks when data needs to be moved between systems	Should have	✗
R-T5.4-8	The data pipeline module of the Orchestrator should support deploying data pipelines expressed using TOSCA models into the AWS data pipeline service.	Must have	✗
R-T5.4-9	The data pipeline module of the Orchestrator should support deploying data pipelines expressed using TOSCA models into a private OpenStack cloud.	Must have	✔ ✔

7.1. Future work

The next (and final) version of this deliverable, D5.6 Data pipeline orchestration II - due on M22, will contain the detailed and final release of data pipeline architecture. The final release will also present the TOSCA extension for data pipeline. Extending the Section 3, and based on the above requirement summary, in Table 7.0.1, we will be primarily focusing on the following points:

1. Verify that the current architecture is enough for designing pipeline-based applications. If needed, we will be updating the current architecture satisfying the KPIs.
2. Extend and improve the current TOSCA-based data pipeline model, especially focusing on the implementation of *SourcePB*, *MidwayPB*, and *DestinationPB*. The Generic nodes in Figure 3.2.1 need to be defined.
3. While improving the current data pipeline model, we will also be studying, how a data pipeline-based application is different from other applications from the orchestrator point-of-view. Upon doing so, we will be answering the question “does RADON orchestrator need additional feature only to implement a data pipeline-based application topology expressed in TOSCA language?”
4. Based on the above study, the additional features will be incorporated (if required) to support orchestrating TOSCA data pipeline.
5. In the process of realizing the data pipeline concept for RADON, we will be creating several TOSCA templates and will be pushed as dedicated module to RADON Particles[14].
6. A plugin will be developed for dynamic updation of user’s TOSCA-based DP service model and a set of DP rules will be provided to the CDL tool to verify the consistency of that DP model.

[14] https://github.com/radon-h2020/radon-particles

References

[1] RADON Consortium, Deliverable 2.3 – Architecture and integration plan I, 2019 http://radon-h2020.eu/wp-content/uploads/2019/11/D2-3_Architecture-and-integration-plan-I.pdf

[2] RADON Consortium, Deliverable D4.3 - RADON Models I, 2019, http://radon-h2020.eu/wp-content/uploads/2019/11/D4.3-RADON-Models-I.pdf

[3] Amazon Web Services (AWS) - Cloud Computing Services, https://aws.amazon.com/ [Online; accessed 26-November-2019] (2019).

[4] Apache Nifi documentation, https://nifi.apache.org/, [Online; accessed 25-November-2019] (2019)

[5] Casale, G., M. Artač, W. J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long et al. "Rational Decomposition and Orchestration for Serverless Computing." In *The Symposium and Summer School on Service-Oriented Computing (SummerSoc)*. 2019.