The Journal of Systems & Software 186 (2022) 111164

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

TOSCAdata: Modeling data pipeline applications in TOSCA☆

Chinmaya Kumar Dehury [a], Pelle Jakovits [a], Satish Narayana Srirama [b,*], Giorgos Giotis [c], Gaurav Garg [a]

[a] Mobile & Cloud Lab, Institute of Computer Science, University of Tartu, Tartu 50090, Estonia
[b] School of Computer and Information Sciences, University of Hyderabad, Hyderabad 500 046, India
[c] Athens Technology Center S.A., Chalandri 15233, Athens, Greece

ARTICLE INFO

Article history:
Received 11 January 2021
Received in revised form 5 July 2021
Accepted 23 November 2021
Available online 20 December 2021

Keywords:
Data pipeline
Data flow management
Serverless computing
Data migration
TOSCA
DevOps

ABSTRACT

The serverless platform allows a customer to effectively use cloud resources and pay for the exact amount of used resources. A number of dedicated open source and commercial cloud data management tools are available to handle the massive amount of data. Such modern cloud data management tools are not enough matured to integrate the generic cloud application with the serverless platform due to the lack of mature and stable standards. One of the most popular and mature standards, TOSCA (Topology and Orchestration Specification for Cloud Applications), mainly focuses on application and service portability and automated management of the generic cloud application components. This paper proposes the extension of the TOSCA standard, *TOSCAdata*, that focuses on the modeling of data pipeline-based cloud applications. Keeping the requirements of modern data pipeline cloud applications, TOSCAdata provides a number of TOSCA models that are independently deployable, schedulable, scalable, and re-usable, while effectively handling the flow and transformation of data in a pipeline manner. We also demonstrate the applicability of proposed TOSCAdata models by taking a web-based cloud application in the context of tourism promotion as a use case scenario.

© 2021 Elsevier Inc. All rights reserved.

1. Introduction

Orchestration of cloud services is important for companies and institutions which need to design complex cloud-native applications or to migrate their existing services to the cloud. A number of orchestration solutions exist for clouds, but many of them are designed for specific platforms and introduce vendor lock-in, meaning it may be very difficult to migrate to other platforms when requirements change and the initially chosen technologies are no longer optimal (Opara-Martins et al., 2016; Casale et al., 2020). Tools such as Chef,[1] Ansible[2] and Puppet[3] provide infrastructure-as-code (IaC) language to automate the installation and configuration of cloud applications, but are not simple to use for designing complex cloud systems consisting of tens or hundreds of components.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) (Rutkowski et al., 2020) language focuses on modeling the structure of cloud services to support their automation and orchestration. The developer can represent the structure of the services using node and relationship topology. The lifecycle of the cloud services can be managed automatically through different operations, such as create, start, configure, and stop. In addition, the relationship among those services can be managed automatically, for instance, which services to be configured before and after a relationship is created.

One of the main advantages of the TOSCA standard is that it is platform-agnostic (Orazio et al., 2021). Users can easily switch out node types representing cloud platform for other providers or even open source cloud software and enable the same cloud applications to be easily migrated between different vendors. It also is well compatible with different CI/CD technologies, which enable easier testing, re-deployment and re-engineering of applications expressed in TOSCA language (Artac et al., 2017).

1.1. Motivation and goal

However, the building blocks of TOSCA modeling language mainly focus on automatic deployment and orchestration of generic cloud applications (Wild et al., 2020) and do not deal with controlling the flow of data inside such systems (Dehury et al., 2020c). The challenge arises when data-intensive applications need to be designed and orchestrated. One of the crucial

☆ Editor: W. K. Chan.

* Corresponding author.
E-mail addresses: chinmaya.dehury@ut.ee (C.K. Dehury), Jakovits@ut.ee (P. Jakovits), satish.srirama@uohyd.ac.in (S.N. Srirama), g.giotis@atc.gr (G. Giotis), gaurav.garg@ut.ee (G. Garg).

[1] https://www.chef.io/.
[2] https://www.ansible.com/.
[3] https://puppet.com/.

https://doi.org/10.1016/j.jss.2021.111164
0164-1212/© 2021 Elsevier Inc. All rights reserved.

challenges in designing such cloud applications is migrating the data into a multi-cloud system and integrating the serverless platform. It requires orchestrating the infrastructure and software and controlling the entire life-cycle of data, from data ingestion to migration, transformation, serverless platform integration, processing, and storage.

With this motivation, in this work the TOSCA language has been extended and proposed as *TOSCAdata* with the primary focus on efficiently handling the flow and processing of the data. Our goal is to take advantage of TOSCA capabilities to enable developers to rapidly model, develop, and deploy data pipeline applications, such as designing pipelines for migrating data between systems, tracking data versions, maintaining privacy and security, transforming data on-the-fly using serverless platform, processing with data analytics platform, fusing, and merging the data.

1.2. Contributions

The main contributions of TOSCAdata to TOSCA language are summarized as below:

- TOSCAdata extends the TOSCA standard with the ability to model the flow of data across cloud services.
- It enables the data-pipelines-as-code pattern, allowing the automated deployment of data-pipelines (e.g., as part of CI/CD pipelines).
- Reduces the development effort of designing data migration and processing services by providing reusable and freely combinable data pipeline blocks.
- Reduces the effect of data lock-in by providing data pipeline blocks for different cloud providers (e.g., AWS, Azure, Google Cloud, OpenStack).
- Ensures that data is encrypted while moved across cloud platforms over the internet (e.g., between on-premise and cloud, or in multi-cloud deployments).

The proposed *TOSCAdata*, extends our previous work (Dehury et al., 2020a), where the overall concept of how to model data pipeline applications with TOSCA language is presented with the intention to take advantage of the TOSCA features to model the flow of data in a multi-cloud system. The proposed extension of the TOSCA standard is also adopted in the modeling of data pipeline applications in the RADON project (Casale et al., 2020). The major extensions made to our previous work (Dehury et al., 2020a) can be summarized below:

- The methodology for orchestrating data pipeline services with the TOSCA standard is provided.
- A detailed description of each category of the data pipeline TOSCA node types, their specific functionalities, characteristics, implementation of all node types' life-cycle operations in Ansible, etc., are presented.
- After a thorough investigation, it is concluded that previous research does not provide a way to develop a specific type of TOSCA node type suitable for standalone deployment. As a result, we have leveraged the TOSCAdata with a new *Standalone* category of TOSCA node types.
- To ensure the deployability of the TOSCA service template, *TOSCAdata Verifier* is designed.
- The extended work also focuses on design and development of the essential features, such as event and cron-based scheduling, of TOSCAdata that may be required while migrating the data across a multi-cloud environments.
- The application of TOSCAdata in real-world tourism promotion applications is explained and demonstrated using the Viarota application.
- In addition to a real-world application, TOSCAdata is also demonstrated by implementing an image data migration and processing application described in Section 5.2. The demonstration example shows the movement across four different private and public clouds and the integration of serverless platforms to process those data.

Further to demonstrate the purpose and capabilities of the proposed work that extends the TOSCA language, we have applied the extended works on Viarota (ATC, 2021), a mobile and web-based cloud application in the context of tourism promotion, as a use case. In this use case, the data needs to be synchronized between the storage system available in three different public clouds: AWS cloud, Google cloud and Azure cloud. A detailed description, including the shortcoming of existing TOSCA, benefits of adopting data pipeline nodes in the use case implementation, is presented in Section 5.

The rest of the paper is organized as follows: Section 2 presents the technical background and the related works about the data pipeline and TOSCA. Section 3 outlines the methodology of using TOSCA for modeling data pipeline services. Section 4 describes the extensions introduced to TOSCA language. Section 5 explains the use case scenarios followed by the concluding remarks in Section 6.

2. Background and related works

This section discusses the technical background on TOSCA and the recent related works on data pipeline modeling.

2.1. Data pipeline-based cloud applications

Data pipeline mainly refers to automatizing the basic three operations: Extract, Transform, and Load in a pipeline manner. A data pipeline application consists of a large number of independent pipeline blocks[4] connected sequentially. The output of one pipeline block is the input for another pipeline block. Each block is composed of one or more microservices, serverless functions, or self-contained applications designed to perform a specific task for data transformation, storage, and processing. Such blocks are freely composable, portable, and reusable; designed in such a way that they are independently deployable, schedulable, and scalable in the cloud environment. For example, a block could be to read the list of files present in a remote file server. Simultaneously, another block could be designed to get the files one after another based on the list. Another example of a pipeline block could be to read the images from the AWS S3 bucket and store them in a local directory. Between each pair of pipeline blocks, temporary buffer storage is used to balance the throughput of pipeline blocks at both ends.

Pervaiz et al. (2019) investigated the challenges that are faced in the phase of data cleaning and data processing in development data. It is observed that, even with modern technology, it is difficult to maintain data consistency during the transition of data between data collection, data cleansing, and data analytics. The data pipeline approach is also applied in the evaluation of supervised and unsupervised machine learning algorithms in smart transportation (Howard et al., 2018) systems. The algorithms are designed to execute on a distributed system, employing different AWS cloud services, such as S3 bucket and MongoDB for data storage, EC2 and EMR cluster for providing computational resources, and Spark for big data processing platform.

One of the main hurdles in cloud adoption of data-intensive applications is the absence of mature data management solution

[4] Henceforth, the term *pipeline block* and *pipeline* are used interchangeably.

that addresses vendor lock-in issue (Casale et al., 2020; Opara-Martins et al., 2016) despite of several open-source and commercial data management solutions available in the market, such as Apache NiFi (Apache, 2019), AWS data pipeline (Amazon, 2012), Google dataflow (Google, 2021), and Azure data factory (Microsoft, 2021). Mainly one open-source (Apache NiFi) and one commercial data management solutions (AWS data pipeline) are taken into account in this work.

Apache NiFi (Apache, 2019), an open-source data management solution, focuses on smooth and efficient flow and processing of data through a large number of components called processors. To connect one processor with others, input ports and output ports are used. Apache NiFi internally creates the intermediate temporary storage and implements a queuing system between adjacent connected processors. The processors are developed for different purposes, such as processors for interacting with AWS storage, Google Storage, Microsoft storage system, different serverless platforms, the transformation of text data, invoking local system commands, and many more.

On the other hand, AWS data pipeline (DP) service (Amazon, 2012) is a service that especially focuses on the flow and processing of data within different services provided by AWS cloud. Data in the AWS cloud can be transformed and moved from one AWS cloud service to another using different activities. These activities act as basic pipeline components/blocks that apply certain operations on the data nodes. Different data nodes that AWS DP supports are S3DataNode, SqlDataNode, DynamoDBDataNode, and RedshiftDataNode. In general, it is observed that, AWS DP does not provide enough stable mechanisms for a smooth and efficient flow of data in and out AWS cloud platform. To move the data in and out of the AWS cloud ecosystem, user needs to issue a shell command using ShellActivity data pipeline. There is no specific data pipeline to migrate the data from/to any AWS storage service (Amazon, 2012). This introduces a data lock-in issue while developing large data-intensive cloud applications.

The security breach is another major issue that may bring huge monetary loss to the business (Byrne and Jacobs, 2020). This may occur due to the lack of platform-specific expertise and data mishandling in the public cloud. Another research challenge in handling the data in a multi-cloud data pipeline platform is the recovery of the data from its failure, as this involves different storage units from different cloud providers, as discussed in Wang et al. (2019). Another reason that makes this research challenge more complex is due to the diverse backup and recovery services provided by multiple cloud platforms and the lack of co-operations between those services.

2.2. TOSCA language for modeling cloud applications

On the other hand, Topology and Orchestration Specification for Cloud Applications (TOSCA) (Rutkowski et al., 2020) is a recently developed OASIS standard focusing on the portability and interoperability of cloud-based applications. In TOSCA language, a service blueprint describes the structure of the whole cloud application along with the management aspect (i.e., deployment, operation, termination) of each component. The service blueprint consists of a set of *nodes* (to represent software components) and *edges* (to represent the relationship among software components). A data pipeline block is called a node in TOSCA language. These set of nodes and the edges form a *topology template*. Here each node represents a single independent or dependent software component, and an edge represents the relationship (i.e. *HostedOn*, *ConnectsTo*, *DependsOn*, etc. (Binz et al., 2013; Binz et al., 2012)) between two software components. An example of the TOSCA topology is presented in Fig. 1 that copies the data from AWS S3 bucket to Google cloud Storage bucket using NiFi. The topology consists of eight nodes/pipeline blocks: *AWSPlatform*, *OpenStackPlatform*, *EC2_VM*, *OpenStack_VM*, two *Nifi_Platform* nodes, *ConsumeS3Bucket*, and *PublishGoogleBucket*. This also demonstrates *HostedOn* and *ConnectsTo* relationships. TOSCA service blueprints follow the YAML syntax, and all the related definitions and artifacts are encapsulated in a CSAR (Cloud Service Archive) file, which is a standardized packaging format.

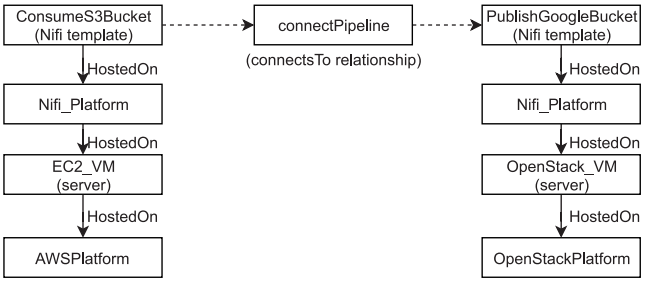


Fig. 1. An example of TOSCA topology.

TOSCA Simple Profile (Rutkowski et al., 2020) provides a set of normative node types, relationship types, capabilities types, data types, different types of interfaces, etc. Such normative types can be used to extend and define desired nodes, relationships, capabilities types based on the requirement of application modeling. Each node type definition consists of a set of properties, attributes, requirements, capabilities, implementations and other related information. Some examples of normative node types are *Compute*, *SoftwareComponent*, *WebServer*, *WebApplication*, *DBMS*, *Database*, and *ObjectStorage*.

Requirements and *capabilities* of a TOSCA node type are counterpart to each other. For any relationship between two nodes, first node should have the requirement to connect to the second node and similarly the second node should have capability to accept the connection from the first node. The *requirements* of a node type describe the needs of that node, which can be in terms of hosting environment, computing or storage resource, or any software component. On the other hand, the required node type should have the corresponding capability to fulfill the demand of counterpart TOSCA node. For instance, in Fig. 1, the requirement of *EC2_VM* is a AWS hosting platform of type *AWSPlatform*. On the other hand, *AWSPlatform* should have the capability to host a *EC2_VM* type TOSCA node. Similarly, the requirements of *ConsumeS3Bucket* are *Nifi_Platform* and *PublishGoogleBucket*. At the same, *Nifi_Platform* must have the capability to host *ConsumeS3Bucket* TOSCA node and *PublishGoogleBucket* must have the capability to accept connection from *ConsumeS3Bucket*.

Each node type can have one or more capabilities, such as *Compute*, *Network*, *Storage*, *Container*, and *Endpoint*. On the contrary, a node type definition can have more than one requirements depending on the node type this needs to be connected to. When two node types are connected, a relationship between them needs to be defined. The normative relationship types provided by TOSCA Simple Profile are *HostedOn*, *ConnectsTo*, *DependsOn*, *AttachesTo*, and *RoutesTo*.

The lifecycle of each node is implemented using *create*, *configure*, *start*, *stop*, and *delete* operations. Depending on the TOSCA orchestration platform, the implementation file(s) for each operation can be provided using Ansible script, Python script, or other scripting languages. A number of commercial and open-source TOSCA orchestration platforms are developed, such as Cloudify (Cloudify, 2021), and xopera (XLAB, 2021). It is to be noted that TOSCA only provides a high-level description of cloud

applications. The high-level description contains the properties, attributes, requirements, and capabilities of each respective component. Further, with each component, a standard interface is attached that describes the lifecycle of the component. It is the responsibility of the orchestrator to understand and implement each node along with the associated relationships. The orchestrator provides the necessary runtime environment to invoke the implementation file for each lifecycle operation. An implementation file provides information that is enough to make automatic deployment and un-deployment of the applications, provisioning of the resources, and manage the lifecycle of the application (Kopp et al., 2013).

2.3. Literature survey on TOSCA

Along with commercial and non-profit organizations, TOSCA has been widely adopted in other technologies, such as the Internet of Things (IoT) (da Silva et al., 2016; Franco da Silva et al., 2017; Li et al., 2013), Network Function Virtualization (NFV) (Antonenko et al., 2017; de Brito et al., 2017; Hung et al., 2017), quantum computing (Wild et al., 2020), fog and mobile edge computing (de Brito et al., 2017). TOSCA is mainly designed and developed to improve the portability and interoperability of the cloud technology (Di Martino et al., 2020). This has also been extended to support the scaling (Cankar et al., 2020), load balancing, monitoring, and other aspects of cloud computing. Dehury et al. in Dehury et al. (2020c,a) developed the TOSCA profile for serverless and data pipeline-based cloud applications.

The adoption of TOSCA Simple Profile in the field of IoT can be found in early 2013 by Li et al. (2013), where a set of node type definitions, such as *Controller*, *Gateway*, and *Driver*, along with the modeling of required relationship types are derived keeping the requirement of building automation system in mind. Similarly, Franco da Silva et al. (2017) adopt the TOSCA Simple Profile for automatic deployment and setup of IoT environments. This addresses the major challenges due to the heterogeneous nature of all the components of the IoT environment. The authors here proposed TOSCA profiles for defining and setting up hardware components, deployment of middlewares, and deployment of IoT applications atop the IoT middlewares. For exchange of messages among the IoT devices, authors in da Silva et al. (2016) propose the required TOSCA profiles for deployment and configuration of MQTT brokers.

TOSCA is further extended to handle several cloud deployment challenges leading to its multi-dimensional extensions. Kehrer and Blochinger (2018) took advantage of the TOSCA to address the lock-in problem in the container management system. The authors propose a two-phase deployment method for Mesos to integrate cloud service orchestration based on TOSCA and their automation using container-based artifacts. However, this extension of TOSCA entirely focuses on the orchestration and automation aspect of the cloud services. Further, on efficient management of complex cloud services across the heterogeneous platform, Brogi et al. (2018) extended the TOSCA standard for orchestrating and lifecycle management of the Docker-based software components. Furthermore, the TOSCA standard is also extended to the Kubernetes for the cloud-based enterprise applications (Bogo et al., 2020). The above extensions mainly focus on the cloud application's deployability and not the data migration and processing.

To handle the above issue, TOSCA is also adopted to design and orchestrate big data architecture and services (Guerriero et al., 2016). Keeping that in focus, the authors have developed a set of TOSCA profiles for the data source, storage, computation of data, data specification, and others. However, the developed TOSCA models are most suitable for big-data applications and do not consider data migration and processing pipeline characteristics. Further, there is no focus on the integration of the serverless platform with the big data application. To handle the data exchange in the IoT environment and their customization and provisioning of the complex event processing (CEP), da Silva et al. (2018) took the capability of TOSCA standard and proposed an approach that efficiently handles the CEP systems, especially in an IoT environment. However, the proposed system does not take the lifecycle of the data flow into consideration.

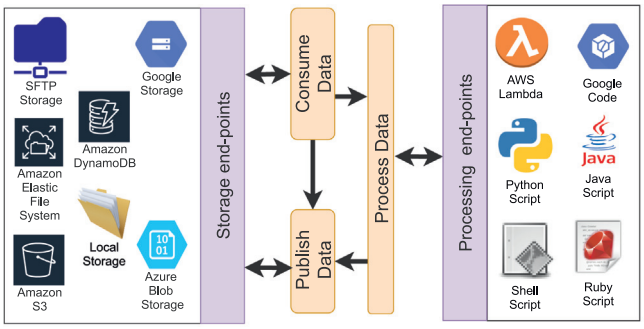


Fig. 2. Basic building blocks of data-focused cloud services.

Broadening the applicability of the TOSCA from just cloud service management, Tsagkaropoulos et al. (2021) extended the TOSCA standard for providing support to edge and fog deployment of the services. The authors have developed a set of TOSCA node types, policies, relationships, capabilities, suitable for deployment of software components in fog environment taking the resource constraints into consideration. The proposed extensions are developed mainly considering the Fog-assisted surveillance application and applicability of such improvement in general situations is not clear.

3. TOSCAdata: Data pipeline modeling using TOSCA

This section presents the proposed *TOSCAdata*, the extended version of the TOSCA standard for cloud-based data pipeline applications. Considering the motivation and goal, as mentioned in Section 1.1, this section provides a detailed explanation on the features, capabilities and functionalities that *TOSCAdata* facilitates to the TOSCA standard. The extension to TOSCA standard includes design and development of essential TOSCA profiles for consuming the data from the source, publishing the result to sink and processing the intermediate data by enabling serverless platform integration.

As discussed before, data pipeline applications mainly focus on extraction, transformation, and migration of the data. Such applications are composed of a large number of pipeline blocks that focus on different data-related operations. Further, each block is independent, freely composable, portable, and reusable. This section discusses on methodology to develop such independent pipeline blocks using TOSCA.

Any data flow-based cloud service can be generalized as a combination of three basic building blocks: (a) consuming the data, (b) processing the consumed data, and (c) publishing the processed or consumed data, as shown in Fig. 2. *Consuming* and *Publishing* data refer to reading and writing the data from/to local/remote storage end-point. The local storage end-point refers to the local directory structure of the same host machine where the service component is running. The remote storage end-point refers to the storage bucket provided by other Cloud Service Providers (CSPs). The remote storage end-point can also be an SFTP server. On the other hand, *Processing* of the consumed data

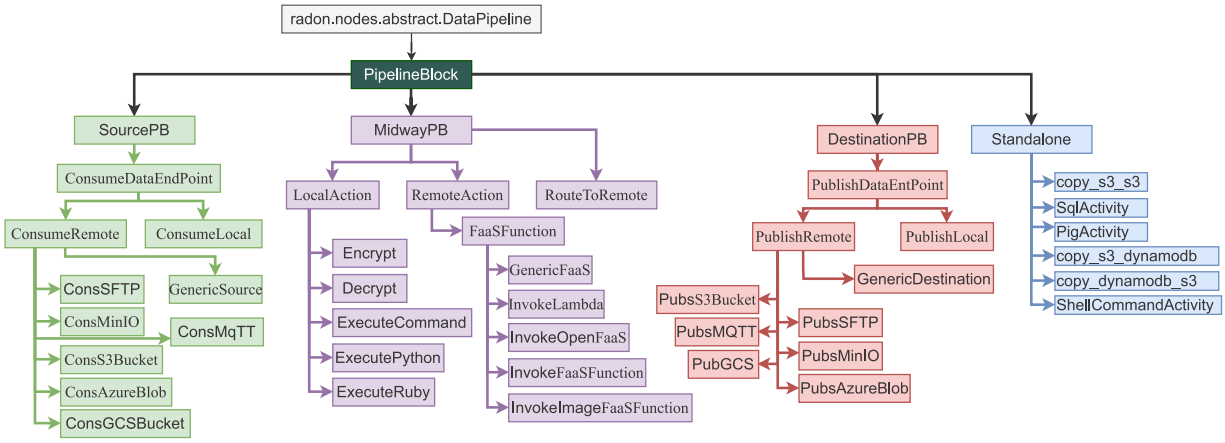


Fig. 3. TOSCA data pipeline models hierarchy.

can be achieved by invoking the remote serverless functions, such as Amazon Lambda, Google function, Azure function, and OpenFaaS function. The processing can also be done in the local machine by using shell script, Python script, Ruby script, or any other programming language script.

Based on the above basic data pipeline building blocks, the proposed model hierarchy provides a set of TOSCA-based data pipeline models to fulfill the requirements of each building blocks that are used to develop modern data flow-based cloud services. The TOSCA models are mainly classified into three categories: *SourcePB*, *MidwayPB*, *DestinationPB*, which are derived from an upper level of TOSCA node type *PipelineBlock* (PB), as shown in Fig. 3. The detailed description of those TOSCA node types are given in Section 4.

As shown in Fig. 3, *PipelineBlock* (PB) TOSCA node type is derived from the abstract node type, *radon.nodes.abstract.DataPipeline*. The definition of *PipelineBlock* TOSCA node type contains the definitions of properties, attributes, and other artifacts that are common to all the derived pipeline blocks, as in Listing 1.

```
1  tosca_definitions_version: tosca_simple_yaml_1_3
2  node_types:
3    radon.nodes.datapipeline.PipelineBlock:
4      derived_from: radon.nodes.abstract.DataPipeline
5      attributes:
6        id:
7          type: string
8      properties:
9        schedulingStrategy:
10         type: string
11         default: "EVENT_DRIVEN"
12       schedulingPeriodCRON:
13         type: string
14         default: "* * * * * ?"
15       name:
16         type: string
```

Listing 1: *PipelineBlock* TOSCA node type.

The *PipelineBlock* in Listing 1 also contains the definition of *schedulingStrategy*, Line 9, and *schedulingPeriodCRON*, Line 12. These two common properties allow the developers to schedule the pipelines that are triggered based on specific events or time. Time-driven scheduling is implemented using the CRON scheduler. The detailed description of the scheduling properties of the pipelines and their implementation are described in the Section 3.3.

3.1. TOSCAdata properties & attributes

The properties and attributes are specific to each TOSCA DP node types. However, it is found that some properties and attributes can be common to all the DP node types, such as the name of the pipeline block, and its scheduling strategy, and are defined in *PipelineBlock* node type, as shown in Listing 1. Some of the common properties are the *name* of the pipeline, which is of *string* data type, as in Line-15. For each pipeline, a user needs to mention the name of the pipeline in the service template. It is possible to assign two pipelines with the same name without any runtime conflict that may occur during the orchestration of the service template. This is due to the fact that each pipeline is assigned with a unique ID to the *id* attribute (say in Listing 1, Line 6) of the corresponding pipeline, and such unique IDs are used for future reference. The *id* of a pipeline is mainly generated by the underlined pipeline technology used by the implementation file. If the pipeline is based on Apache NiFi, the NiFi engine will generate a unique ID. However, if the pipeline is based on ADP, the unique id would be generated by the corresponding AWS service (Amazon, 2012). Hence, the generation of unique IDs of the pipelines is very specific to and dependent on the underlined pipeline technology.

3.2. TOSCAdata requirements, relationships & capabilities

The developed TOSCA data pipeline nodes/blocks have three requirements: (a) requirement to connect to a local pipeline (*connectToPipeline*), (b) requirement to connect to a remote pipeline (*connectToPipelineRemote*), and (c) requirement of a hosting environment (*host*), based on the functionality of the node. *connectToPipeline* and *connectToPipelineRemote* indicates the requirement to connect to other pipeline, whereas *host* indicates the requirement of an hosting platform, which in turn requires an hosting environment, as shown in Fig. 4. The relationship between two local or remote pipelines can be of type *ConnectNifiLocal* or *ConnectNifiRemote*, which are derived from the *tosca.relationships.ConnectsTo* normative relationship type available in TOSCA simple profile v1.3.[5] Both the relationship types can be differentiated by means of their implementations only. Two NiFi-based pipelines are connected by connecting the output port of the source pipeline to the input port of the destination pipeline. However, if the pipelines are on different host machines, we create a Remote Process Group (RPG) on the host machine where the source pipeline is deployed. The implementation file for performing this job is created using Ansible.

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

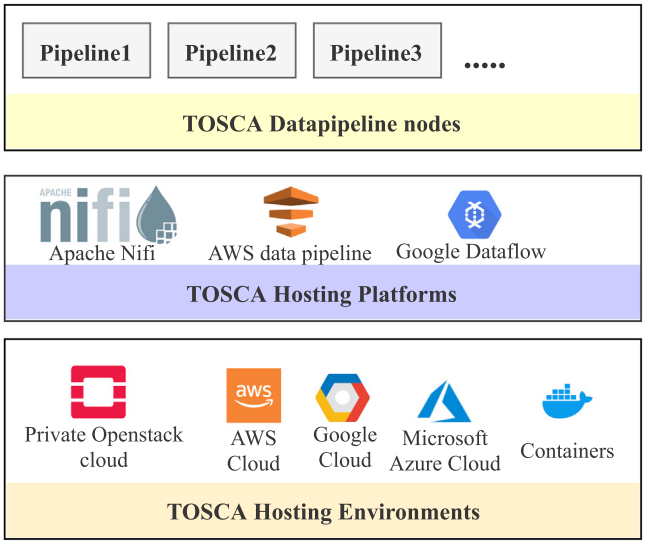


Fig. 4. Hosting hierarchy of TOSCA pipelines, platforms, and environments.

Further, as mentioned above, the developed DP node types require a hosting environment. The current development of TOSCA data pipeline node types supports the Apache NiFi and AWS data pipeline as the hosting platforms. However, depending upon the node type, the compatible hosting platform needs to be chosen. Apache NiFi hosting platforms can be deployed on a virtual machine or a container on either a private cloud or public cloud, as shown in Fig. 4. The TOSCA node type for Apache NiFi, *NiFi*,[6] hosting platform is given in Listing 2. *NiFi* node type has the properties *port* number (to access NiFi's web interface), in Line 5–1, and the *component_version* (to specify the version of the Apache NiFi), in Line 8–9. This hosting platform has the capability to host *UNBOUNDED* number of NiFi-based pipelines, in Line 17–20, which indicates no limit to the number of data pipeline nodes that can be hosted on. Further, this hosting platform requires a hosting environment, as in Line 11–15, with normative *tosca.capabilities.Compute* capability. As shown in Fig. 4, the hosting environment can be OpenStack-based private cloud or public clouds, such as AWS cloud, Google cloud, and, Microsoft Azure cloud.

```
1  node_types:
2    radon.nodes.nifi.Nifi:
3      derived_from: tosca.nodes.SoftwareComponent
4      properties:
5        port:
6          type: string
7          default: 8080
8        component_version:
9          type: string
10     requirements:
11       - host:
12           capability: tosca.capabilities.Compute
13           node: tosca.nodes.Compute
14           relationship: tosca.relationships.HostedOn
15           occurrences: [ 1, 1 ]
16     capabilities:
17       host:
18         occurrences: [ 1, UNBOUNDED ]
19         valid_source_types: [ radon.nodes.abstract.DataPipeline ]
20         type: tosca.capabilities.Container
```

Listing 2: TOSCA node type for Apache NiFi hosting platform.

The developed TOSCA DP node types have the capabilities to accept the data through the incoming connection from other pipelines to either publish or process the data. For this, ConnectToPipeline capability type is developed, which is derived from *tosca.capabilities.Endpoint* TOSCA normative capability type, as shown in Listing 3. The data may arrive from one or more local or remote pipelines.

```
1 tosca_definitions_version: tosca_simple_yaml_1_3
2 capability_types:
3   radon.capabilities.datapipeline.ConnectToPipeline:
4     derived_from: tosca.capabilities.Endpoint
5     metadata:
6       targetNamespace: "radon.capabilities.datapipeline"
7       abstract: "false"
8       final: "false"
```

Listing 3: TOSCA data pipeline capability type.

3.3. TOSCAdata functionalities

With the current development of TOSCAdata, a number of functionalities and features are provided. Some of the major functionalities are supported for scheduling the pipeline-based on event-driven or time-driven data flow across multiple private and public clouds by addressing data lock-in issue, and incorporation of the serverless platform, as discussed below.

3.3.1. Scheduling of data pipelines

The scheduling definition is implemented in *PipelineBlock* TOSCA node, as shown in Listing 1, Line 9–14. Two properties *schedulingStrategy* (Listing 1, Line 9–11) and *schedulingPeriodCRON* (Listing 1, Line 12–14) are introduced. *schedulingStrategy* property allows the user to decide whether the pipeline should be triggered based on events or a specific time. The default scheduling strategy is set to be event-driven. The scheduling strategy of Apache NiFi is used to schedule the NiFi-based pipelines. For timer driven, users need to provide time interval using CRON syntax into *schedulingPeriodCRON* properties. However, for the AWS related TOSCA pipelines (as discussed in detail in later sections), only CRON-based scheduling is supported, and hence the above two properties *schedulingStrategy* and *schedulingPeriodCRON* are not made available to AWS related TOSCA pipelines.

3.3.2. Cross cloud data flow

With the developed set of TOSCA DP node types, it is easy to allow the data movement across different clouds. This is demonstrated in the Use case section (Section 5), where, with the minimal design time effort, the data are migrated from one cloud storage (e.g. AWS S3 Bucket) to the other (e.g. Google Cloud Storage). TOSCAdata provides an essential set of models to consume and publish the data from and to several storage systems. As it can be seen in Fig. 3, a storage system can be in local directory structure (*ConsumeLocal* node type), or from AWS S3 bucket and DynamoDB (*ConsS3Bucket*, *PubsS3Bucket* & *copy_dynamodb_s3* node types), Google cloud storage (*ConsGCSBucket* & *PubGCS* node types), Azure blob storage (*ConsAzureBlob* & *PubsAzureBlob* node types), or on-premise MinIO server (*ConsMinIO* & *PubsMinIO* node types). It is also possible to consume the data from edge devices or resource constraint devices over lightweight MQTT protocol (*ConsMqTT* & *PubsMQTT* node types) and the external data sources over SFTP protocol (*ConsSFTP* & *PubsSFTP* node types). TOSCAdata also provides a set of node types to integrate the serverless platform that enables the data-on-fly to be processed by user-defined FaaS functions. Such node types are: *InvokeLambda* to invoke AWS lambda function, *InvokeOpenFaaS* to invoke a FaaS function deployed in OpenFaaS environment, *InvokeFaaSFunction* to invoke any FaaS function over HTTP request, and *InvokeImageFaaSFunction* to invoke generic FaaS function only with image data as arguments. The detailed descriptions are provided in Section 4.

[6] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.nifi/Nifi.

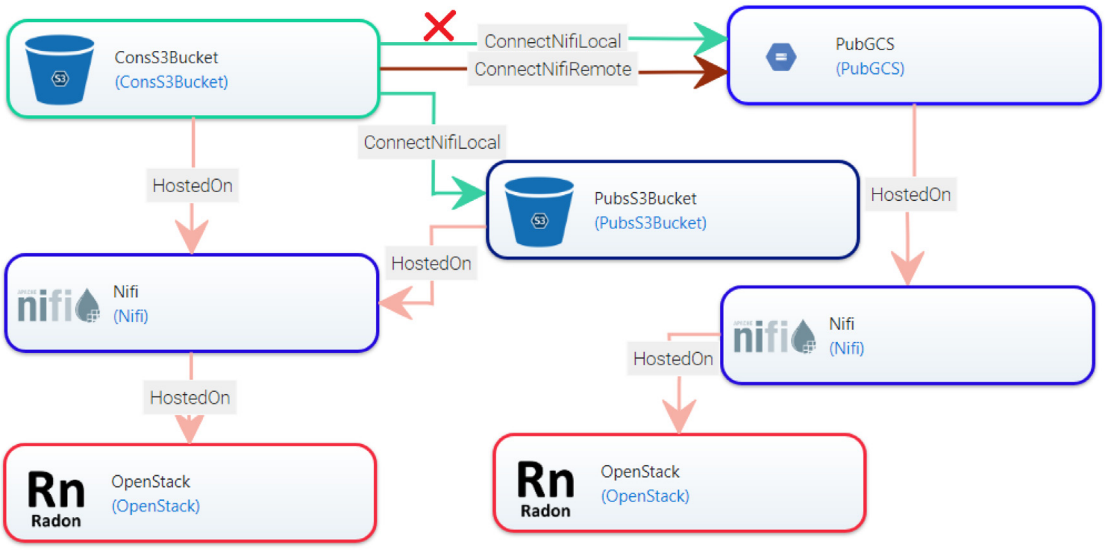


Fig. 5. An example of erroneous TOSCA service Template.

3.3.3. TOSCAdata DP verifier

In the design time, the TOSCA service blueprint may contain inconsistency/bugs related to inappropriate connections among nodes. The orchestration tool (such as TOSCA orchestration or xopera (XLAB, 2021)) may generate unexpected errors during the deployment of nodes, or the user may get undesired result during the runtime if such erroneous TOSCA service templates are used for the deployment of the cloud services. To overcome such issues, the *TOSCAdata Verifier*[7] tool is developed that ensures the workability of the TOSCA service template. The verifier parses through the nodes defined in the service blueprint and verifies the relationship, capabilities, and requirements. In case any mismatch is detected, it resolves the error and generates a verified service blueprint.

Fig. 5 shows an example of an erroneous TOSCA service template. In this design, the node *ConsS3Bucket* is hosted on one *NiFi*, while node *PubGCS* is hosted on another *NiFi* platform on different virtual machine. The defined connection type between them should be remote *ConnectNifiRemote*. However, it may happen that there exist two connections of different relationship types between the same pair of TOSCA nodes, as shown in Fig. 5. In this figure two different connections: *ConnectNifiRemote* and *ConnectNifiLocal*, exist between *ConsS3Bucket* and *PubGCS*. This is a good example of an inconsistent TOSCA service template, which causes the TOSCA orchestrator to generate errors while trying to deploy the nodes. Such potential errors can be eliminated by parsing the blueprint through the TOSCA DP verifier.

Apart from finding and fixing the design-time errors in the service template, the DP verifier also checks for the properties of *Encrypt* and *Decrypt* node types. These two node types are designed to provide a way to encrypt the content while sending the data from one machine to another and decrypt the received encrypted content. While doing so, it is essential to ensure that both the nodes at different machines have the same passphrase for encryption and decryption purposes. In case of the presence of such passphrase mismatch, the verifier will generate a new passphrase for both the nodes pair and accordingly update the service template. The DP verifier also ensures that for each *Encrypt* type node in the service template, there exists a *Decrypt* type node and vise versa.

The current version of the *TOSCAdata Verifier* can verify and update the relationships among the data pipelines and the encryption configuration for the secure transmission of the data. However, the verifier can further be extended to handle several runtime configurations of pipelines. Some of the features that are under development and will be incorporated into the *TOSCAdata Verifier* are: ensuring that the right set of artifacts are provided to each data pipelines, the TOSCA definition for each pipeline is available in the service template, and ensuring that the properties values are of valid type.

4. TOSCAdata models

This section discussed the developed models that are introduced in Section 3, for *TOSCAdata* using the TOSCA language. For the implementation of the lifecycle of developed TOSCA models, Ansible[8] is used, which provides a set of modules for software provisioning, configuration management, and application deployment. As discussed before, the developed TOSCA data pipeline nodes are based on Apache NiFi and AWS data pipeline. To be more precise, the node types that are developed under *SourcePB*, *MidwayPB*, and *DestinationPB* categories to consume, process and publish the data, respectively, are based on the Apache Nifi technology. All the node types that are developed on *Standalone* data pipeline category are based on the AWS data pipeline technology. During the orchestration of NiFi-based TOSCA nodes, the corresponding pre-designed NiFi template is uploaded to the NiFi hosting platform. Such NiFi templates are exported in XML format and may compose of one or more NiFi processors. The corresponding REST requests are issued for uploading the NiFi template to the remote NiFi platform, deployment of the NiFi templates, and activating the processors. *uri*[9] Ansible module is used to issue those REST requests.

4.1. SourcePB

This subsection discusses a specific category of the TOSCA node types (including their requirements and capabilities) specifically developed to consume the data from several external sources,

[7] https://github.com/radon-h2020/radon-datapipeline-plugin.

[8] https://docs.ansible.com/.

[9] https://docs.ansible.com/ansible/latest/collections/ansible/builtin/uri_module.html.

such as FTP server, Google storage bucket, AWS S3 bucket, MinIO servers, MQTT broker, and Azure storage system.

For any data flow-based cloud application consuming the data from any local or remote location is one of the basic building blocks. To address this *SourcePB*[10] TOSCA node type is defined that is derived from *PipelineBlock* TOSCA node type. *SourcePB* node type is further used to derive *ConsumeDataEndPoint* node type. *SourcePB* node type defines the *requirements* that are common to any type of data end-point. The data or storage endpoint may refer to SFTP storage server, Google storage, Amazon Elastic File System, Amazon S3, Azure storage or a local storage, as shown in Fig. 2. The TOSCA nodes that are of type *SourcePB* or of type derived from *SourcePB* may have three requirements, as shown in Listing 4: (a) *connectToPipeline* requirement, in Line 2–6, (b) *connectToPipelineRemote* requirement, in Line 7–11, and (c) *host*, in Line 12–16.

```
1  requirements:
2    - connectToPipeline:
3        capability: radon.capabilities.datapipeline.ConnectToPipeline
4        node: radon.nodes.abstract.DataPipeline
5        relationship:
         ↪  radon.relationships.datapipeline.ConnectNifiLocal
6        occurrences: [ 1, UNBOUNDED ]
7    - connectToPipelineRemote:
8        capability: radon.capabilities.datapipeline.ConnectToPipeline
9        node: radon.nodes.abstract.DataPipeline
10       relationship:
         ↪  radon.relationships.datapipeline.ConnectNifiRemote
11       occurrences: [ 1, UNBOUNDED ]
12   - host:
13       capability: tosca.capabilities.Container
14       node: radon.nodes.nifi.Nifi
15       relationship: tosca.relationships.HostedOn
16       occurrences: [ 1, 1 ]
```

Listing 4: List of requirements for *SourcePB* and all derived node types

Such TOSCA nodes require to send the consumed data to other pipelines of type, either *MidwayPB* or *DestinationPB*. *connectToPipeline* and *connectToPipelineRemote* requirements are used to connect to the other end of the pipelines deployed on the same and different virtual machines or the containers, respectively. From its functionalities, it is obvious that such TOSCA nodes do not need any capabilities to receive the data from other pipelines. Hence, all the TOSCA nodes of type *SourcePB* cannot send the data to another *SourcePB* TOSCA node. In other words, the consumed data by *SourcePB* TOSCA nodes cannot be treated as the data source for another *SourcePB* TOSCA node.

All the *SourcePB*-derived TOSCA node types are based on Apache NiFi data management technology. Hence such TOSCA nodes require an Apache NiFi hosting platform. As discussed above, the Apache NiFi hosting platform can be deployed either in the AWS cloud platform or Openstack-based private cloud platform.

Based on the storage location, two TOSCA node types are created: *ConsumeRemote* and *ConsumeLocal*. *ConsumeRemote* TOSCA node type is further used to derived a number of other TOSCA node types for consuming data from remote storage servers. When the source of data is other than the local machine where the pipeline is deployed, *ConsumeRemote* TOSCA node type is used. For different data sources, such as, *ConsFTP*, *ConsGCSBucket*, and *ConsS3Bucket* TOSCA node types are developed.

ConsFTP is used to consume the data from a FTP server. Similarly, *ConsGCSBucket* (in Listing 5) and *ConsS3Bucket* (in Line 6) TOSCA node types are developed to consume the data from Google cloud storage bucket and AWS S3 bucket. For *ConsGCSBucket* TOSCA node type has three main properties: *bucket* (Listing 5, Line 5–8) for providing the name of the bucket, *project_id* (Line 9–11) for the id of the project, and *credential_JSON_file* (Line 12–14) for path the file that contains login credentials, as shown in Listing 5. Similarly, *ConsS3Bucket* TOSCA node type has three main properties: *BucketName* (Line 5–6) for name of the S3 bucket, *cred_file_path* (Line 7–8) for path the file that contains login credentials, and *Region* (Line 9–10) of the S3 bucket, as shown in Listing 6. On the contrary to the remote data consumption, *ConsumeLocal* TOSCA node type is developed that would allow the user to consume the data from the local file system. The local file system refers to the file system where the pipeline is deployed.

```
1  node_types:
2    radon.nodes.datapipeline.source.ConsGCSBucket:
3      derived_from: radon.nodes.datapipeline.source.ConsumeRemote
4      properties:
5        bucket:
6          type: string
7          description: Name of the bucket
8          required: false
9        project_ID:
10         type: string
11         description: ID of the project.
12       credential_JSON_file:
13         type: string
14         description: Path of the credentials JSON file
```

Listing 5: Properties of *ConsGCSBucket* TOSCA node type.

```
1  node_types:
2    radon.nodes.datapipeline.source.ConsS3Bucket:
3      derived_from: radon.nodes.datapipeline.source.ConsumeRemote
4      properties:
5        BucketName:
6          type: string
7        cred_file_path:
8          type: string
9        Region:
10         type: string
```

Listing 6: Properties of *ConsS3Bucket* TOSCA node type.

It is to be observed that all the TOSCA node types derived from SourcePB are based on NiFi technology, and hence in the implementation of each such node type may contain one or more NiFi processors.

4.2. MidwayPB

This subsection discusses the category of TOSCA data pipeline node types specifically developed to process the data while migrating from one system to another. This set of node types allows the developers to transform the data either locally or by invoking remote serverless functions.

Processing the consumed data before publishing it is another basic building block to develop data flow-based cloud applications. To fulfill such requirements, *MidwayPB*[11] TOSCA node type is developed. *MidwayPB* TOSCA node type is developed to provide a parent node type for all potential data processing related TOSCA node types. *MidwayPB* TOSCA node type is derived from *PipelineBlock*.

Similar to *SourcePB* TOSCA node type, *MidwayPB* (in Listing 7) also has three requirements: (a) *connectToPipeline*: requirement to connect to a local pipeline, in Line 6–10, (b) *connectToPipelineRemote*: requirement to connect to a remote pipeline, in Line 16–20, and (c) *host*: requirement of an hosting environment, in Line 11–15. The requirements of *MidwayPB* and *SourcePB* TOSCA node type have same purpose. However, unlike *SourcePB*, *MidwayPB* has the capabilities to receive the data from other local pipelines

10 https://github.com/radon-h2020/radon-particles/nodetypes/radon.nodes.datapipeline.source.

11 https://github.com/chinmaya-dehury/radon-particles/nodetypes/radon.nodes.datapipeline.process.

(*ConnectToPipeline*, in Line 26–29) and remote pipelines (*ConnectToPipelineRemote*, in Line 22–25), as shown in Listing 7. Both the capabilities are of types *radon.capabilities.datapipeline.ConnectToPipeline* and have the valid source type of *SourcePB* and *MidwayPB*. It is possible that the data can be consumed from multiple data sources using multiple *SourcePB* type TOSCA nodes. Moreover, the output of one *MidwayPB* type TOSCA node can be input to other node of type *MidwayPB*. To facilitate such feature, the upper bound of receiving data from other local and remote pipelines is set to *UNBOUNDED*, as shown in Listing 7.

```
1  tosca_definitions_version: tosca_simple_yaml_1_3
2  node_types:
3    radon.nodes.datapipeline.MidwayPB:
4      derived_from: radon.nodes.datapipeline.PipelineBlock
5      requirements:
6        - ConnectToPipeline:
7            capability:
               ↪ radon.capabilities.datapipeline.ConnectToPipeline
8            node: radon.nodes.datapipeline.PipelineBlock
9            relationship:
               ↪ radon.relationships.datapipeline.ConnectNifiLocal
10           occurrences: [ 1, UNBOUNDED ]
11       - host:
12           capability: tosca.capabilities.Container
13           node: radon.nodes.nifi.Nifi
14           relationship: tosca.relationships.HostedOn
15           occurrences: [ 1, 1 ]
16       - ConnectToPipelineRemote:
17           capability:
               ↪ radon.capabilities.datapipeline.ConnectToPipeline
18           node: radon.nodes.datapipeline.PipelineBlock
19           relationship:
               ↪ radon.relationships.datapipeline.ConnectNifiRemote
20           occurrences: [ 1, UNBOUNDED ]
21     capabilities:
22       ConnectToPipelineRemote:
23         occurrences: [ 1, UNBOUNDED ]
24         valid_source_types: [ radon.nodes.datapipeline.SourcePB,
             ↪ radon.nodes.datapipeline.MidwayPB ]
25         type: radon.capabilities.datapipeline.ConnectToPipeline
26       ConnectToPipeline:
27         occurrences: [ 1, UNBOUNDED ]
28         valid_source_types: [ radon.nodes.datapipeline.SourcePB,
             ↪ radon.nodes.datapipeline.MidwayPB ]
29         type: radon.capabilities.datapipeline.ConnectToPipeline
```

Listing 7: List of requirements and capabilities for *MidwayPB* TOSCA node type.

Based on the processing location and the functionality, three different node types are further created: *LocalAction* with the purpose to process the data on the local machine, *RemoteAction* to invoke a remote serverless function to process the data, and *RouteToRemote* to route the data to other local or remote pipelines based on the specific condition. Handling the data on the local machine is suitable in case of data fusion, data integration, and data cleaning. Further, this is suitable for applying analytic tasks on large scale data, where invoking the serverless function would not give the desired performance due to the size of the task or due to the nature of the data analysis job. To provide such functionalities, a number of TOSCA node types are developed that are derived from *LocalAction* TOSCA node type. *ExecuteCommand*, *ExecutePython*, and *ExecuteRuby* TOSCA nodes, as shown in Fig. 3, are developed allowing the user to invoke and execute the data analytics task in shell terminal, Python engine, and in Ruby engine, respectively. With this, users need to provide the path to the scripting code or provide the scripting code to process and analyze the input data. Moreover, user can also encrypt the raw data with *Encrypt* TOSCA node and decrypt the encrypted data using *Decrypt* TOSCA node on the local machine.

The modern data-flow based cloud applications rely highly on a serverless platform as well, which not only allows the developer to detach themselves from infrastructure and platform management responsibilities, including resource provisioning, scaling, and maintenance, but also minimize the service cost and application development time. To fulfill this requirement of such modern serverless cloud applications, a number of TOSCA node types are developed that are derived from *RemoteAction*. The developed *InvokeLambda* and *InvokeOpenFaaS* TOSCA node types can be used to invoke AWS Lambda serverless function and OpenFaaS serverless function, respectively. While invoking such remote serverless function, the received data are sent in JSON format as a part of a body of the HTTP request. In general, the route of the data is mostly decided by the developer in the design time of the cloud application. In some exceptional situations, the data can be routed based on conditions defined in dedicated data routing programs installed on local machines. However, to make the data routing job more dynamic, we have envisioned to develop a TOSCA node type, *RouteToRemote*, that handles routing of data in a dynamic manner. Dynamic routing refers to deciding the next destination of data on its arrival. The next destination can be another *MidwayPB* TOSCA node or a TOSCA node to publish the data. Apache NiFi is used as the underlined open-source data management tool to develop all the above TOSCA node types that are derived from *MidwayPB*.

4.3. DestinationPB

This subsection discusses the developed TOSCA node types that are used to publish the data to external storage systems such as FTP server, Google storage bucket, AWS S3 bucket, MinIO servers, MQTT broker, and Azure storage system.

DestinationPB[12] TOSCA node type, as shown in Listing 8, is created as the counterpart of *SourcePB* TOSCA node type. Unlike *SourcePB*, *DestinationPB* mainly focuses on publishing the data/result to local or remote storage server provided by private or public cloud provider. As shown in Fig. 2, multiple nodes in TOSCA service template that are of type *SourcePB* and *DestinationPB* may refer same storage location as both data source and data destination. The storage server can be located in the local machine or in the remote machine. For this purpose, *PublishRemote* and *PublishLocal* TOSCA node types are developed that are derived from *DestinationPB*. For each TOSCA node type developed under *SourcePB*, the corresponding counterpart TOSCA node type is developed under *DestinationPB*, which can be seen in Fig. 3.

All the TOSCA node types that are derived from *DestinationPB*, as in Fig. 3, have the capabilities to accept the connections from other pipelines of type either *SourcePB* or *MidwayPB* and not from the nodes of the same type. The requirement and the capability *DestinationPB* node type is given in Listing 8.

```
1  tosca_definitions_version: tosca_simple_yaml_1_3
2  node_types:
3    radon.nodes.datapipeline.DestinationPB:
4      derived_from: radon.nodes.datapipeline.PipelineBlock
5      requirements:
6        - host:
7            capability: tosca.capabilities.Container
8            node: radon.nodes.nifi.Nifi
9            relationship: tosca.relationships.HostedOn
10           occurrences: [ 1, 1 ]
11     capabilities:
12       ConnectToPipelineRemote:
13         occurrences: [ 1, UNBOUNDED ]
14         valid_source_types: [ radon.nodes.datapipeline.SourcePB,
             ↪ radon.nodes.datapipeline.MidwayPB ]
15         type: radon.capabilities.datapipeline.ConnectToPipeline
16       ConnectToPipeline:
17         occurrences: [ 1, UNBOUNDED ]
18         valid_source_types: [ radon.nodes.datapipeline.MidwayPB,
             ↪ radon.nodes.datapipeline.SourcePB ]
19         type: radon.capabilities.datapipeline.ConnectToPipeline
```

Listing 8: List of requirements and capabilities for *DestinationPB* TOSCA node type.

[12] https://github.com/radon-h2020/radon-particles/nodetypes/radon.nodes.datapipeline.destination.

4.4. Standalone

This section discusses another category of the developed TOSCA node types that act as standalone data pipelines and do not need any input/output from/to another pipeline block.

Along with the basic building blocks discussed above, another building block, *Standalone*,[13] is created focusing on performing a very specific task, such as synchronizing two AWS S3 buckets, and taking data backup from AWS DynamoDB to AWS S3 bucket, as shown in Fig. 3. Each TOSCA node type under the Standalone pipeline block can be very vendor-specific. In other words, these TOSCA node types focus on the flow and processing of data using the storage and computational service of a single cloud provider. The *Standalone* TOSCA node type does not have any specific requirement or capabilities. Requirement and capabilities vary with the derived node types. As the name suggests, these node types do not send any data to other pipelines, nor do they have any capabilities to receive data from other pipelines. Such node types are developed by combining all the three basic building blocks. For example, the TOSCA node type, *AWSCopyS3ToS3*, as in Fig. 3, copies the data from one S3 bucket to another S3 bucket by combining the functionalities of *SourcePB* and *DestinationPB* node types.

We have developed a number of standalone TOSCA data pipeline node types using the services provided by Amazon cloud. *AWSCopyS3ToS3* node type is developed to synchronize two S3 buckets. This node type requires *AWSPlatform* as a hosting platform. Some of the basic properties that the users need to provide are the name of the source and destination S3 bucket, the credential information, and the S3 bucket name to store the logs. It is also possible that the source and destination S3 bucket names are the same. In such a scenario, the source and destination directories need to be different. Similarly, *AWSCopyDynamodbToS3* and *AWSCopyS3ToDynamodb* TOSCA node types are developed to copy the data between Amazon cloud provided DynamoDB and S3 buckets. *AWSShellCommand* TOSCA node type is developed to execute a command or a script. Similarly, to execute an SQL command on a database, *AWSSqlActivity* TOSCA node type is developed. All the above-mentioned AWS-related TOSCA node types required *AWSPlatform* as a hosting platform. These nodes cannot be hosted on a private cloud or any other public cloud instances.

The above TOSCA node types provide the basic functionalities to handle smooth flow and transformation of the data by incorporating multiple cloud provider and their serverless platform. To realize the applicability and the advantage of TOSCAdata, the developed TOSCA node types are used to develop a cloud application, as discussed in a further section.

5. Application of TOSCAdata

In previous sections, the proposed TOSCAdata is explained, including the detailed description of the TOSCA node types, requirements, capabilities, relationships and others. TOSCAdata provides a number of necessary TOSCA node type definitions for building data pipeline-based cloud applications. It is to be noted that no additional orchestration or modeling tool is required other than the tools that provide the support for TOSCA to model a data pipeline-based cloud service. The steps required to develop any TOSCA-based cloud application can be followed to develop a data pipeline application for all the stakeholders. To demonstrate the functionalities, the proposed TOSCAdata is implemented in Viarota, a real-world mobile and web-based tourism promotion cloud application currently under development at Athens Technology Center (ATC), Greece (ATC, 2021). In addition to this use case, we also have demonstrated the image data movement across four different cloud providers through *image data migration and processing* examples.

5.1. Use case: Viarota application

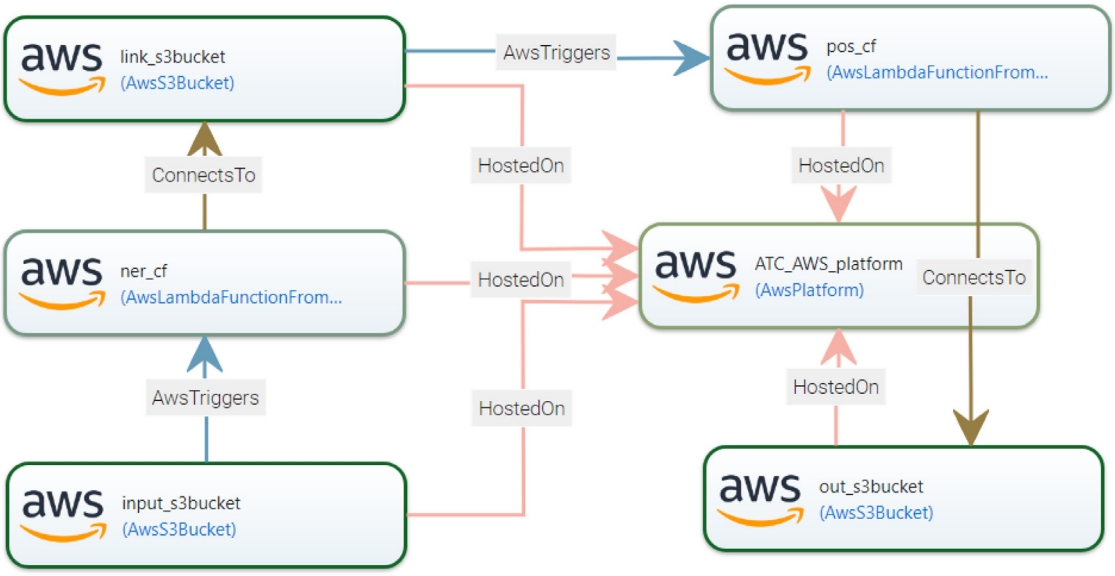
In this section, a web-based tourism promotion cloud application is considered as a use case scenario and discussed how the proposed data pipeline-based TOSCA node types can be used to efficiently handle the data flow and transformation without much worrying about the vendor lock-in issue while migrating data from one cloud to other.

Viarota (ATC, 2021) is a mobile as well as a web-based cloud application in the context of tourism promotion. Viarota enhances a visitor's travel experience by providing optimal loyalty-based personalized city tour planning, developed within the RADON project (Casale et al., 2020). A tour plan is composed of Point of Interests (POIs) that follow the taxonomy concepts for the places of a visit being included in the tours proposed for a touristic destination. Viarota crawls related content from different social media sources like Twitter and YouTube as well as from tour related website RSS feeds. The crawled data are then processed and stored in a database for providing aggregated reviews on the visits placed in the proposed tours. The raw text, extracted from the crawled social media items, processing consists of a set of Natural Language Processing (NLP) functions, a sentiment analysis function and a hate speech detection function. Each function is modeled and implemented as a FaaS in the cloud platforms used, namely the Amazon Web Service (AWS), the Google Cloud Platform (GCP) and the Azure platform. To synchronize the data between these two different cloud storages, the data pipeline approach can be applied by providing a data link between the NLP/ML functions running on different cloud platforms as a single processing layer.

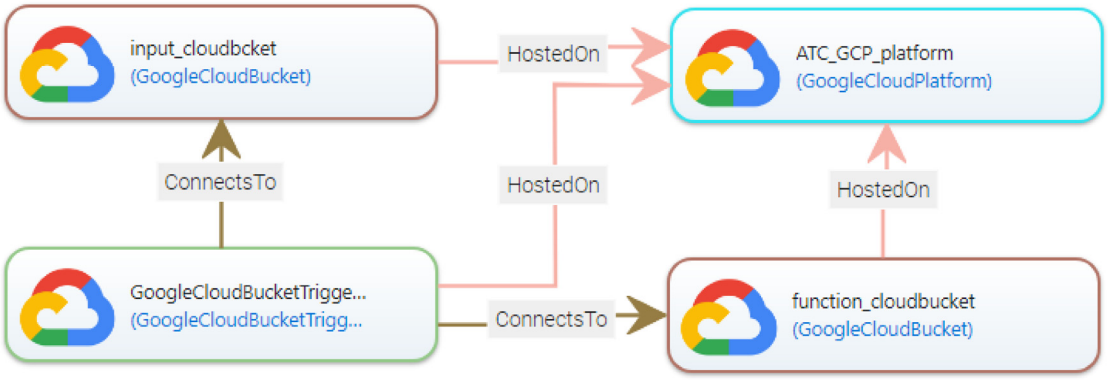
The topology of the Viarota NLP/ML pipeline is modeled using Eclipse Winery, a graphical modeling tool for TOSCA-based cloud application (Kopp et al., 2013; Eclipse, 2021), following the serverless execution model where the Named-entity recognition and the Part-of-speech tagging functions are hosted on AWS Lambda, Amazon's FaaS platform, as shown in Fig. 6(a), the sentiment analysis function is hosted on GCP Cloud Function, Google's FaaS platform, as shown in Fig. 6(b) and the hate speech detection function is hosted on Azure Functions, Azure's FaaS platform, as shown in Fig. 6(c). The function deployment is modeled using the corresponding TOSCA node type definitions developed in the context of the RADON project. Following an event-driven programming model, cloud storage services are used to trigger the Viarota NLP/ML functions, acting as event sources. The Viarota social media crawlers continuously feed the NLP/ML functions with events in the form of new social media item insertions in the configured cloud storage input buckets. By configuring the output bucket of one NLP/ML function as the input bucket of another NLP/ML function, the social media items can traverse the whole processing pipeline without further intervention. However, the TOSCAdata approach is needed to be introduced in order to allow the stream of social media items and the corresponding analysis results to be transmitted between the different cloud platforms.

The TOSCA data pipeline nodes allow establishing a multi-cloud integration layer that links the inputs and outputs of different cloud platforms. Specifically, the Viarota NLP functions (hosted on AWS) need to communicate the analysis results to the ML functions, the sentiment analysis (hosted on GCP), and the hate speech detection (hosted on Azure), in order to be taken into

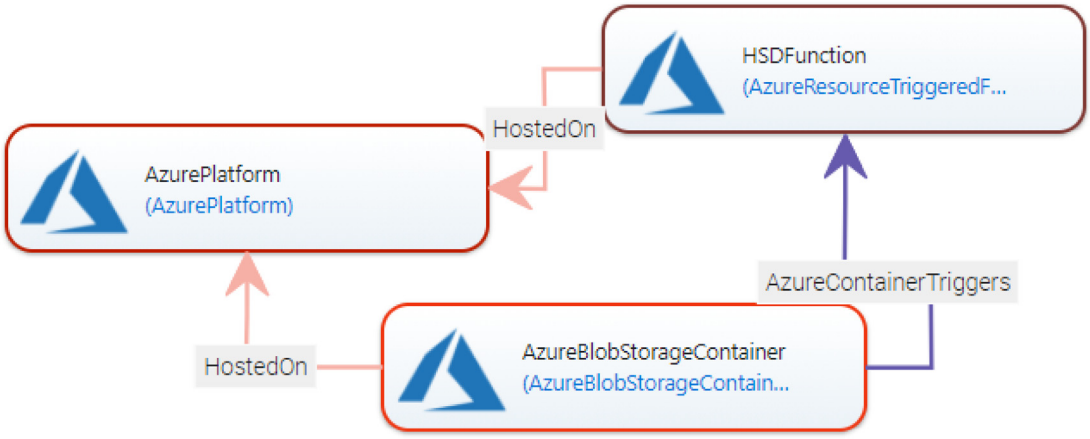
[13] https://github.com/chinmaya-dehury/radon-particles/tree/dp_tmplt/nodetypes/radon.nodes.datapipeline.Standalone.



(a) First part of the whole Viarota NLP pipeline.



(b) Second part of the whole Viarota NLP pipeline.



(c) Third part of the whole Viarota NLP pipeline.

Fig. 6. Viarota NLP pipeline topology.

account in the processing tasks (i.e. words tagged as adjectives seems to capture strong sentiment polarity). There is the need to configure two cross-cloud data flows that would link the respective cloud storage of each platform, allowing the synchronization of data without further interventions. The first data flow (AWS - GCP) is modeled using the TOSCAdata nodes *ConsS3Bucket* and *PubsGCS*, while the second data flow configuration (GCP - Azure) consists of the TOSCAdata nodes *ConsGCSBucket* and *PubsAzureBlob*, as shown in Fig. 7. The TOSCAdata nodes modeled refer to the storage buckets configured in the Viarota pipeline topology (Fig. 6). The node *ConsS3Bucket* consumes events from an AWS S3 bucket and in conjunction with the *PubsGCS* node, it replicates the

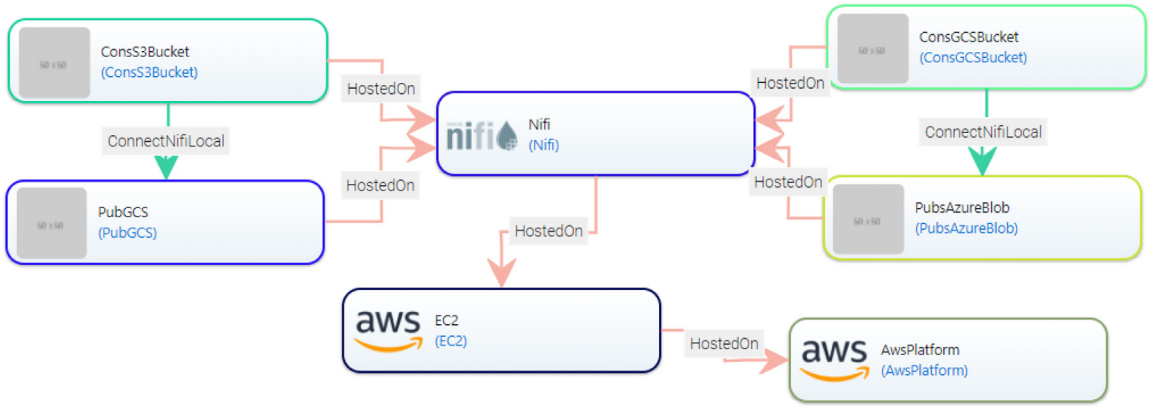


Fig. 7. Viarota NLP data pipeline.

storage events in a GCP cloud storage bucket while the *ConsGCSBucket* node consumes from a GCP cloud bucket and it replicates the events to Azure blob storage (through node *PubsAzureBlob*).

This way, it is possible to synchronize data flows across different vendor storage buckets that would otherwise require to invoke vendor-specific technologies. Thus, the data pipeline layer that spans all cloud platforms (AWS, GCP and Azure) promotes data lock-in avoidance in the architecture of the Viarota application. Although, existing solutions such as Apache Airflow[14] and Node-RED,[15] can address the data lock-in issue, the TOSCAdata tool is still considered advantageous since it provides native integration with the Viarota models developed based on TOSCA language. The TOSCA specification addresses the lack of standardization and contributes to the portability of the implemented Viarota functions. Moreover, the Apache Airflow tool requires writing code for implementing tasks like storage replication, while the TOSCAdata tool allows to drag and drop the YAML based data pipeline node definitions in a more declarative and user-friendly way while designing the topology.

5.2. Image data migration and processing application

In addition to implementing a real-world cloud application above, an *image data migration and processing* application are implemented using TOSCAdata. Fig. 8 depicts the flow of data from an on-premise MinIO database server to OpenStack private cloud, Google cloud storage, and Azure storage system. As shown in Fig. 8, four cloud environments are used: OpenStack private cloud, AWS cloud, Google cloud, and Azure cloud. The automatic data migration takes place upon manual uploading of the image data by the user to the local MinIO server. The uploaded data are automatically passed through OpenStack cloud, processed by AWS serverless platform, and then stored in Google cloud storage and compressed by Azure function, which eventually are stored in Azure blob storage.

Atop OpenStack VM, NiFi software component is installed and configured using *NiFi* TOSCA node. Atop *NiFi* node *ConsMinIO*[16] data pipeline node is deployed with the properties as mentioned in Listing 9. This data pipeline node is used to consume the data from the local MinIO database server.

```
ConsMinIO_0:
  type: radon.nodes.datapipeline.source.ConsMinIO
  properties:
    BucketName: "firstbucket"
    cred_file_path: "{ get_artifact: [SELF, credentials]}"
    MinIO_Endpoint: "http://172.17.25.36:8089"
    schedulingStrategy: "EVENT_DRIVEN"
    schedulingPeriodCRON: "* * * * * ?"
```

Listing 9: *ConsMinIO* data pipeline node.

The *ConsMinIO* node forwards the image data to the *InvokeLambda*[17] data pipeline node, which invokes an image processing function deployed on AWS lambda serverless platform to grayscale the original image. The properties of this node are provided in Listing 10. The grayscaled image is further forwarded to another *InvokeLambda* data pipeline node that invokes an image processing Lambda function to blur the grayscaled image. The properties of this node are provided in Listing 11. Both the *InvokeLambda* data pipeline nodes are deployed on a NiFi software component hosted on AWS EC2 instance, as shown in 8 and 9.

```
InvokeLambda_0:
  type: radon.nodes.datapipeline.process.InvokeLambda
  properties:
    cred_file_path: "{ get_artifact: [SELF, credFile]}"
    schedulingStrategy: "EVENT_DRIVEN"
    function_name: "img-grayscale-nifi"
    schedulingPeriodCRON: "* * * * * ?"
    region: "eu-west-1"
```

Listing 10: Properties of *InvokeLambda* data pipeline node for image grayscale.

```
InvokeLambda_1:
  type: radon.nodes.datapipeline.process.InvokeLambda
  properties:
    cred_file_path: "{ get_artifact: [SELF, credFile]}"
    schedulingStrategy: "EVENT_DRIVEN"
    function_name: "img-blur-nifi"
    schedulingPeriodCRON: "* * * * * ?"
    region: "eu-west-1"
```

Listing 11: Properties of *InvokeLambda* data pipeline node for image blurring.

The second *InvokeLambda* data pipeline node forwards the blurred image to two data pipeline nodes: *PubGCS*[18] and *InvokeImageFaaSFunction*.[19] The *PubGCS* data pipeline node (configured with the properties shown in Listing 12) store the blurred

14 https://airflow.apache.org/.

15 https://nodered.org/.

16 https://github.com/chinmaya-dehury/radon-particles/tree/dp_tmplt_part3/nodetypes/radon.nodes.datapipeline.source/ConsMinIO.

17 https://github.com/chinmaya-dehury/radon-particles/tree/dp_tmplt_part3/nodetypes/radon.nodes.datapipeline.process/InvokeLambda.

18 https://github.com/chinmaya-dehury/radon-particles/tree/dp_tmplt_part3/nodetypes/radon.nodes.datapipeline.destination/PubGCS.

19 https://github.com/chinmaya-dehury/radon-particles/tree/dp_tmplt_part3/nodetypes/radon.nodes.datapipeline.process/InvokeImageFaaSFunction.

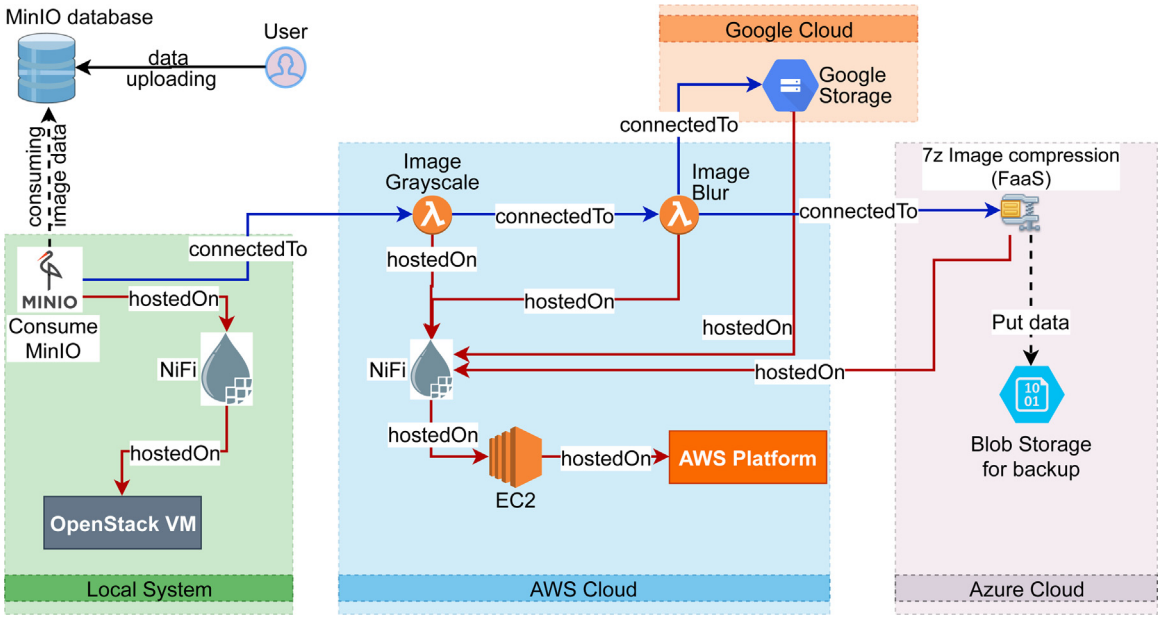


Fig. 8. Image data migration and processing using TOSCAdata.

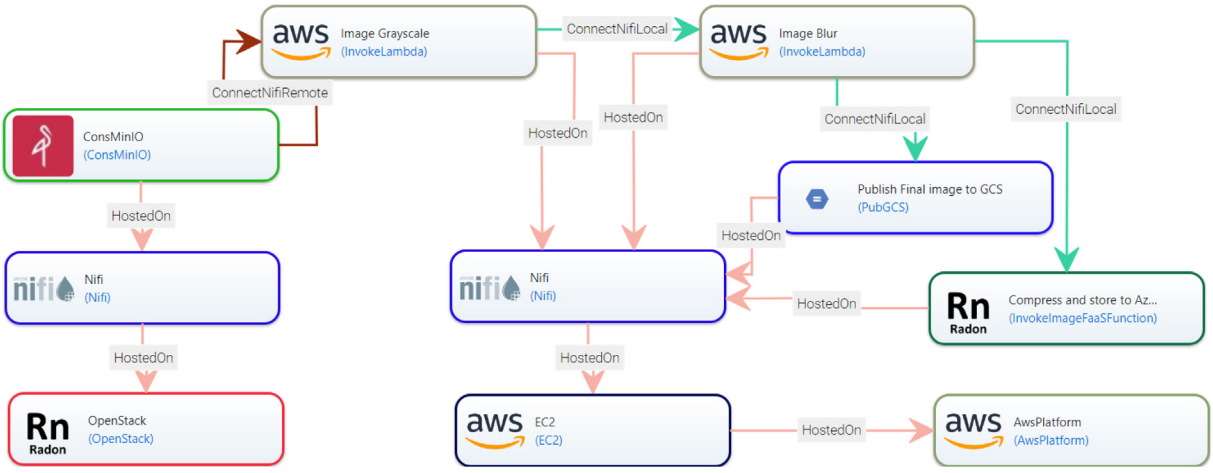


Fig. 9. Modeling of data pipeline example (given in Fig. 8) using RADON graphical modeling tool.

image to Google Cloud Storage. On the other hand, the *InvokeImageFaaSFunction* data pipeline node is used to invoke a function with image data as arguments deployed on the Azure serverless platform. The *InvokeImageFaaSFunction* data pipeline node is configured with the properties given in Listing 13. The Azure function is developed to compress the received blurred image and store it in an Azure storage container, as shown in Fig. 8. The detailed source code of this implementation can be found in GitHub repository (Dehury, 2021).

```
PubGCS_0:
  type: radon.nodes.datapipeline.destination.PubGCS
  properties:
    BucketName: "radongcs"
    cred_file_path: "{ get_artifact: [SELF, credFile ] }"
    schedulingStrategy: "EVENT_DRIVEN"
    ProjectID: "radon-825040-utr"
    schedulingPeriodCRON: "* * * * * ?"
```

Listing 12: Properties of *PubGCS* data pipeline node for storing the blur image to Google storage.

```
InvokeImageFaaSFunction_0:
  type: radon.nodes.datapipeline.process.InvokeImageFaaSFunction
  properties:
    function_URL: "AWS_function_endpoint_here"
    schedulingStrategy: "EVENT_DRIVEN"
    schedulingPeriodCRON: "* * * * * ?"
    HTTP_method: "POST"
```

Listing 13: Properties of *InvokeImageFaaSFunction* data pipeline node.

5.3. Discussion

In the above sections, we have demonstrated how to apply the TOSCAdata on real-world use cases to implement real-time data migration services across multi-cloud and on-premise environments. In the case of the Viarota application, it enabled real time data synchronization between AWS, Azure and Google Cloud, supporting a serverless application where FaaS functions are deployed strategically in different clouds. In the case of image processing applications, we also demonstrated how to integrate on-premise systems deployed in OpenStack, how to chain Serverless functions and how to utilize data pipelines deployed across multiple NiFi deployments.

As a result of providing TOSCA data pipeline blocks that users can simply drag and drop into a TOSCA graphical modeling tool/editor, it reduces the development effort of designing data integration and processing services. As a result of each data pipeline block being designed as an individual software service with its own life cycle control scripts (implemented in Ansible), TOSCAdata enables the data-pipelines-as-code pattern, allowing the automated deployment of data pipelines. For example, as part of CI/CD pipelines. TOSCAdata also reduces the effect of data lock-in when dealing with data migration and processing service, as the data pipeline blocks can be deployed on any of the supported public cloud providers (e.g. AWS, Azure, Google Cloud) and other platforms like OpenStack. This approach also hides any low-level, platform-specific complexities of designing multi-cloud data synchronization services.

6. Conclusions and future works

TOSCA standard focuses on portability and interoperability of the cloud application through a service blueprint that organizes the service components and the relationship among themselves in a graph structure. In this work, we have extended the capability of TOSCA and introduces TOSCAdata, focusing on the modeling of data pipeline-based cloud application. TOSCAdata not only provides a set of TOSCA models for designing a provider and technology-specific cloud application but also empowers the developers to integrate the multiple services provided by multiple cloud providers, different serverless platform, and microservices without compromising the smooth flow and transformation of data. With the developed TOSCA models, it becomes easy to migrate the data across multiple private and public cloud environments through rapid modeling, development, and deployment, leveraging the capabilities of modern data-intensive cloud applications, as discussed in Section 3.3. TOSCAdata is developed to provide a unified solution to the software development community that works atop one open-source (Apache NiFi) and one commercial data management solution (AWS Datapipeline), which further can be extended by providing the support for a number of other commercial data management platforms.

It should be noted that the current development of TOSCAdata does not provide all possible TOSCA models that could be essential to build a data pipeline-based cloud applications, and hence there are rooms for its further development. Currently, TOSCAdata supports one open-source and one commercial data management solutions. To support more number of commercial data management platform from different cloud providers is a part of our future work. Scalability and a higher level of security of data is another development direction that we would like to improve TOSCAdata. Further, our future works will also include the realization of the performance of TOSCA data pipeline-based cloud applications by experimenting with a large scale multi-cloud environment.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially funded by the European Union's Horizon 2020 research and innovation project RADON (825040). We also thank financial support to UoH-IoE by MHRD (F11/9/2019-U3(A)).

References

Amazon, 2012. AWS Data pipeline - developer guide. AWS Data Pipeline - Developer Guide, https://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/datapipeline-dg.pdf.

Antonenko, V., Smeliansky, R., Ermilov, A., Plakunov, A., Pinaeva, N., Romanov, A., 2017. C2: General Purpose Cloud Platform with NFV Life-Cycle Management. In: 2017 IEEE International Conference on Cloud Computing Technology and Science, (CloudCom), pp. 353–356.

Apache, 2019. Apache NiFi. https://nifi.apache.org/, [Online (Accessed 21 October 2019)].

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., Tamburri, D.A., 2017. Devops: introducing infrastructure-as-code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion. ICSE-C, IEEE, pp. 497–498.

ATC, 2021. Viarota Enhance the Travel Experience of Your Visitors, https://viarota.com/.

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S., 2013. Opentosca – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (Eds.), Service-Oriented Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 692–695.

Binz, T., Breiter, G., Leyman, F., Spatzier, T., 2012. Portable cloud services using TOSCA. IEEE Internet Comput. 16 (3), 80–85.

Bogo, M., Soldani, J., Neri, D., Brogi, A., 2020. Component-aware orchestration of cloud-based enterprise applications, from TOSCA to docker and kubernetes. Softw. - Pract. Exp. 50 (9), 1793–1821.

de Brito, M.S., Hoque, S., Magedanz, T., Steinke, R., Willner, A., Nehls, D., Keils, O., Schreiner, F., 2017. A Service Orchestration Architecture for Fog-Enabled Infrastructures. In: 2017 Second International Conference on Fog and Mobile Edge Computing, (FMEC), pp. 127–132.

Brogi, A., Rinaldi, L., Soldani, J., 2018. Tosker: a synergy between TOSCA and docker for orchestrating multicomponent applications. Softw. - Pract. Exp. 48 (11), 2061–2079.

Byrne, R., Jacobs, D., 2020. Development of a high throughput cloud-based data pipeline for 21 cm cosmology. arXiv:2009.10223 [Astro-Ph].

Cankar, M., Luzar, A., Tamburri, D.A., 2020. Auto-scaling using TOSCA infrastructure as code. In: Muccini, H., Avgeriou, P., Buhnova, B., Camara, J., Caporuscio, M., Franzago, M., Koziolek, A., Scandurra, P., Trubiani, C., Weyns, D., Zdun, U. (Eds.), Software Architecture. In: Communications in Computer and Information Science, Springer International Publishing, Cham, pp. 260–268.

Casale, G., Artač, M., van den Heuvel, W.-J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A., Srirama, S.N., Tamburri, D.A., Wurster, M., Zhu, L., 2020. RADON: Rational decomposition and orchestration for serverless computing. SICS Softw.-Intens. Cyber-Phys. Syst. 35 (1), 77–87.

Cloudify, 2021. Cloudify Orchestration Platform –Multi Cloud, Cloud Native & Edge, Cloudify, https://cloudify.co/.

da Silva, A.C.F., Breitenbücher, U., Képes, K., Kopp, O., Leymann, F., 2016. Opentosca for IoT: automating the deployment of IoT applications based on the mosquitto message broker. In: Proceedings of the 6th International Conference on the Internet of Things. In: IoT'16, Association for Computing Machinery, New York, NY, USA, pp. 181–182.

da Silva, A.C.F., Hirmer, P., Breitenbücher, U., Kopp, O., Mitschang, B., 2018. Customization and provisioning of complex event processing using TOSCA. Comput. Sci. Res. Dev. 33 (3), 317–327.

Dehury, C., RADON Data Pipeline Webinar, https://github.com/chinmaya-dehury/radon-datapipeline-webinar, (Accessed: 7 June 2021).

Dehury, C., Jakovits, P., Srirama, S.N., Tountopoulos, V., Giotis, G., 2020a. Data pipeline architecture for serverless platform. In: Muccini, H., Avgeriou, P., Buhnova, B., Camara, J., Caporuscio, M., Franzago, M., Koziolek, A., Scandurra, P., Trubiani, C., Weyns, D., Zdun, U. (Eds.), Software Architecture. In: Communications in Computer and Information Science, Springer International Publishing, Cham, pp. 241–246.

Dehury, C.K., Srirama, S.N., Chhetri, T.R., 2020c. CCoDaMiC: A framework for coherent coordination of data migration and computation platforms. Future Gener. Comput. Syst. 109, 1–16.

Di Martino, B., Esposito, A., Nacchia, S., Maisto, S.A., Breitenbücher, U., 2020. An ontology for OASIS tosca. In: Barolli, L., Amato, F., Moscato, F., Enokido, T., Takizawa, M. (Eds.), Web, Artificial Intelligence and Network Applications. In: Advances in Intelligent Systems and Computing, Springer International Publishing, Cham, pp. 709–719.

Eclipse, 2021. Eclipse Winery — Eclipse Winery Documentation, https://winery.readthedocs.io/en/latest/.

Franco da Silva, A.C., Breitenbücher, U., Hirmer, P., Képes, K., Kopp, O., Leymann, F., Mitschang, B., Steinke, R., 2017. Internet of things out of the box: using TOSCA for automating the deployment of IoT environments:. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, Porto, Portugal, pp. 358–367.

Google, 2021. Dataflow Documentation | Google Cloud, https://cloud.google.com/dataflow/docs.

Guerriero, M., Tajfar, S., Tamburri, D.A., Di Nitto, E., 2016. Towards a model-driven design tool for big data architectures. In: Proceedings of the 2nd International Workshop on BIG Data Software Engineering. In: BIGDSE '16, Association for Computing Machinery, New York, NY, USA, pp. 37–43.

Howard, A.J., Lee, T., Mahar, S., Intrevado, P., Woodbridge, D.M.-K., Distributed Data Analytics Framework for Smart Transportation, in: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2018, pp. 1374–1380.

Hung, Y., Chien, S., Hsu, Y.-Y., 2017. Orchestration of NFV Virtual Applications Based on TOSCA Data Models. In: 2017 19th Asia-Pacific Network Operations and Management Symposium, (APNOMS), pp. 219–222.

Kehrer, S., Blochinger, W., 2018. TOSCA-based container orchestration on mesos. Comput. Sci. Res. Dev. 33 (3), 305–316.

Kopp, O., Binz, T., Breitenbücher, U., Leymann, F., 2013. Winery–a modeling tool for TOSCA-based cloud applications. In: International Conference on Service-Oriented Computing. Springer, pp. 700–704.

Li, F., Vögler, M., Claeßens, M., Dustdar, S., 2013. Towards Automated IoT Application Deployment by a Cloud-Based Approach. In: 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, pp. 61–68.

Microsoft, 2021. Azure Data Factory Documentation - Azure Data Factory, https://docs.microsoft.com/en-us/azure/data-factory/.

Opara-Martins, J., Sahandi, R., Tian, F., 2016. Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective. J. Cloud Comput. 5 (1), 4.

Orazio, T., Domenico, C., Pietro, M., et al., 2021. TORCH: a TOSCA-based orchestrator of multi-cloud containerised applications. J. Grid Comput. 19 (1).

Pervaiz, F., Vashistha, A., Anderson, R., 2019. Examining the challenges in development data pipeline. In: Proceedings of the 2nd ACM SIGCAS Conference on Computing and Sustainable Societies. In: COMPASS, vol. 19, Association for Computing Machinery, New York, NY, USA, pp. 13–21.

Rutkowski, M., Lauwers, C., Noshpitz, C., Curescu, C., 2020. TOSCA Simple profile in YAML version 1.3. https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html.

Tsagkaropoulos, A., Verginadis, Y., Compastié, M., Apostolou, D., Mentzas, G., 2021. Extending TOSCA for edge and fog deployment support. Electronics 10 (6), 737.

Wang, L., Ramasamy, H., Salapura, V., Arnold, R., Wang, X., Bakthavachalam, S., Coulthard, P., Suprenant, L., Timm, J., Ricard, D., Harper, R., Gupta, A., 2019. System Restore in a Multi-Cloud Data Pipeline Platform, In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track. pp. 21–24.

Wild, K., Breitenbücher, U., Harzenetter, L., Leymann, F., Vietz, D., Zimmermann, M., 2020. TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications. In: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference, EDOC, pp. 125–134.

XLAB, 2021. Welcome to xOpera's Documentation! – RADON Documentation, https://xlab-si.github.io/xopera-opera/.

Chinmaya Kumar Dehury received a bachelor's degree from Sambalpur University, India, in June 2009 and an MCA degree from Biju Pattnaik University of Technology, India, in June 2013. He received a Ph.D. Degree in the Department of Computer Science and Information Engineering, Chang Gung University, Taiwan. He is currently a Assistant Professor of Distributed System, member of Mobile & Cloud Lab in the Institute of Computer Science, University of Tartu, Estonia. His research interests include scheduling, resource management and fault tolerance problems of Cloud and fog Computing, the application of artificial intelligence in cloud management, edge intelligence, Internet of Things, and data management frameworks. His research results are published by top-tier journals and transactions such as IEEE TCC, JSAC, TPDS, FGCS, etc. He is a member of IEEE and ACM India. He is also serving as a PC member of several conferences and reviewer to several journals and conferences, such as IEEE TPDS, IEEE JSAC, IEEE TCC, IEEE TNNLS, Wiley Software: Practice and Experience, etc.

Pelle Jakovits received his Ph.D. in computer science from University of Tartu in March 2017 on topic "Adapting Scientific Computing Algorithms to Distributed Computing Frameworks". His main research interests are algorithm parallelization, high-performance computing in the cloud and efficiency of real-time stream data analytics. He has contributed to several EU funded projects, such as H2020 RADON (http://radon-h2020.eu/), FP7 REMICS (http://remics.eu/) and I4 W Robo M.D. He has significant experience in setting up various scale distributed computing environments in the cloud from Hadoop (Cloudera CDH or custom) ecosystems to IoT platforms like Cumulicity using OpenStack, Docker, Ansible, Chef and shell scripts. He is teaching courses related to cloud computing and large scale data processing.

Satish Narayana Srirama is an Associate Professor at the School of Computer and Information Sciences, University of Hyderabad, India. He is also a Visiting Professor and the honorary head of the Mobile & Cloud Lab at the Institute of Computer Science, University of Tartu, Estonia, which he led as a Research Professor until June 2020. He received his Ph.D. in computer science from RWTH Aachen University, Germany in 2008. His current research focuses on cloud computing, mobile web services, mobile cloud, Internet of Things, fog computing, migrating scientific computing and enterprise applications to the cloud and large-scale data analytics on the cloud. He is IEEE Senior Member, an Editor of Wiley Software: Practice and Experience, a 52 year old Journal, was an Associate Editor of IEEE Transactions in Cloud Computing and a program committee member of several international conferences and workshops. Dr. Srirama has co-authored over 150 refereed scientific publications in international conferences and journals. For further information of Prof. Srirama, please visit: http://kodu.ut.ee/~srirama/.

Giorgos Giotis Giotis holds a B.Sc. in Informatics and Telecommunications from the National and Kapodistrian University of Athens and a M.Sc. in Information Systems from Athens University of Economics and Business. He has been working as a software engineer in the R&D department of ATC since 2011. His research interests include data mining, big data analytics and distributed applications.

Gaurav Garg received a bachelor's degree from Punjab Technical University, India, in June 2011. He has worked as an application engineer with FANUC for more than 4 years, where he worked on IoT and control system projects. Currently, he is pursuing a Master's degree in Robotics and Computer Engineering and working as a research assistant in the Mobile & Cloud Lab, Institute of Computer Science, University of Tartu, Estonia.