eFlows4HPC Documentation

Eflows4HPC Consortium

Last updated : March, 2022

Online version available at eFlows4HPC - ReadTheDocs

Table of contents

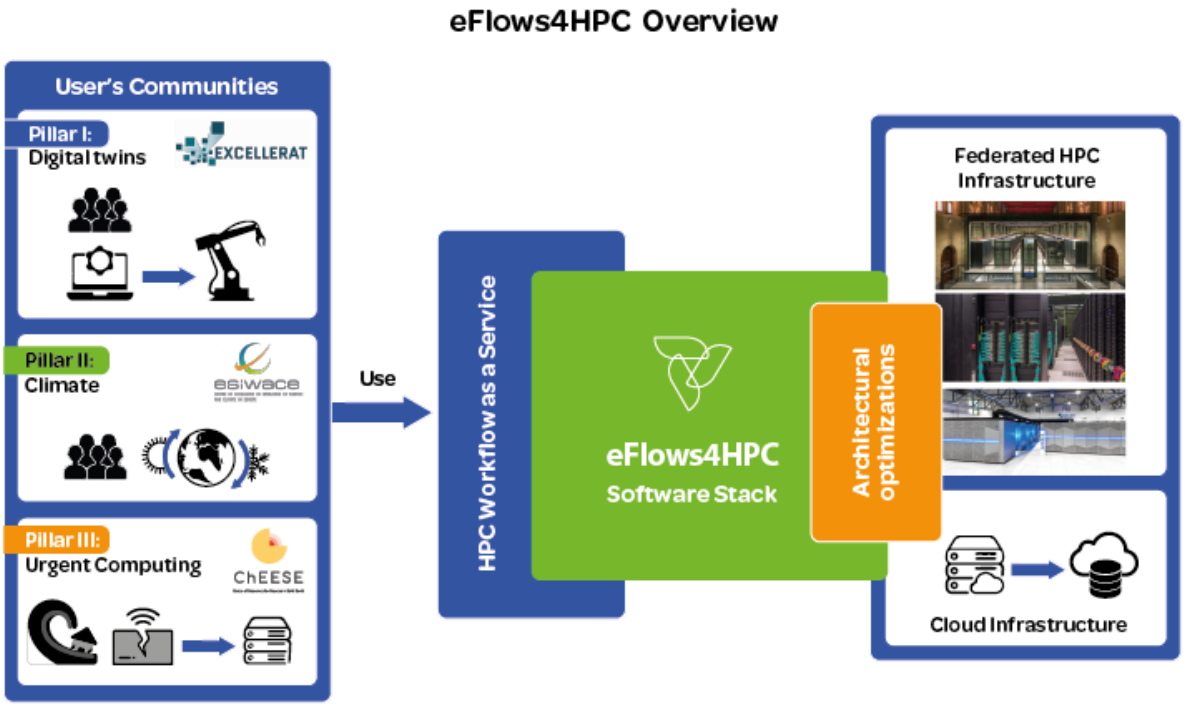
List of figures

List of tables

Welcome to the documentation page of the eFlows4HPC Software Stack. It is organized in the following sections:

Chapter 1

eFlows4HPC Overview



eFlows4HPC aims at designing and implementing a European workflow platform that enables the design of complex applications that integrate HPC processes, data analytics and artificial intelligence, making use of the HPC resources in an easy, efficient and responsible way as well as enabling the accessibility and reusability of applications to reduce the time to solution.

As the main outcome, the project is delivering the eFlows4HPC software stack which integrates different components to provide an overall workflow management system. One of the core functionalities of the software stack is the definition of the complex workflows that combine HPC, HPDA and ML frameworks and the integration of large volumes of data from different sources and locations.

On top of this software stack, the project builds an HPC Workflow as a Service (HPCWaaS) platform to facilitate the reusability of these complex workflows in federated HPC infrastructure. The goal is to provide methodologies and tools that enable sharing and reuse of existing workflows and that assist when adapting workflow templates to create new workflow instances.

The HPCWaaS platform and the eFlows4HPC software stack will be validated by use cases organised in three pillars which represent the main sectors that the project targets.

1.1 More information:

- Project website: https://www.eflows4hpc.eu
- Github organization: https://github.com/eflows4hpc

1.2 Acknowledgements

The eFlows4HPC project is founded by the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway

SPONSORED BY THE

Chapter 2

Software Stack

eFlows4HPC software stack integrates different components to provide an overall workflow management system. Figure 1 shows the components included in the eFlows4HPC Software Stack according to their functionality. On the top, we can find the programming models used for the definition of the complex workflows that combine HPC, HPDA and ML frameworks and the integration of large volumes of data from different sources and locations. Below this part, we can find the components to facilitate the accessibility and re-usability of workflows, Finally in the bottom part of the stack we can see the different components for deployment, execution and data management. Components in gray refer to components to be developed or integrated in the stack in the future releases.

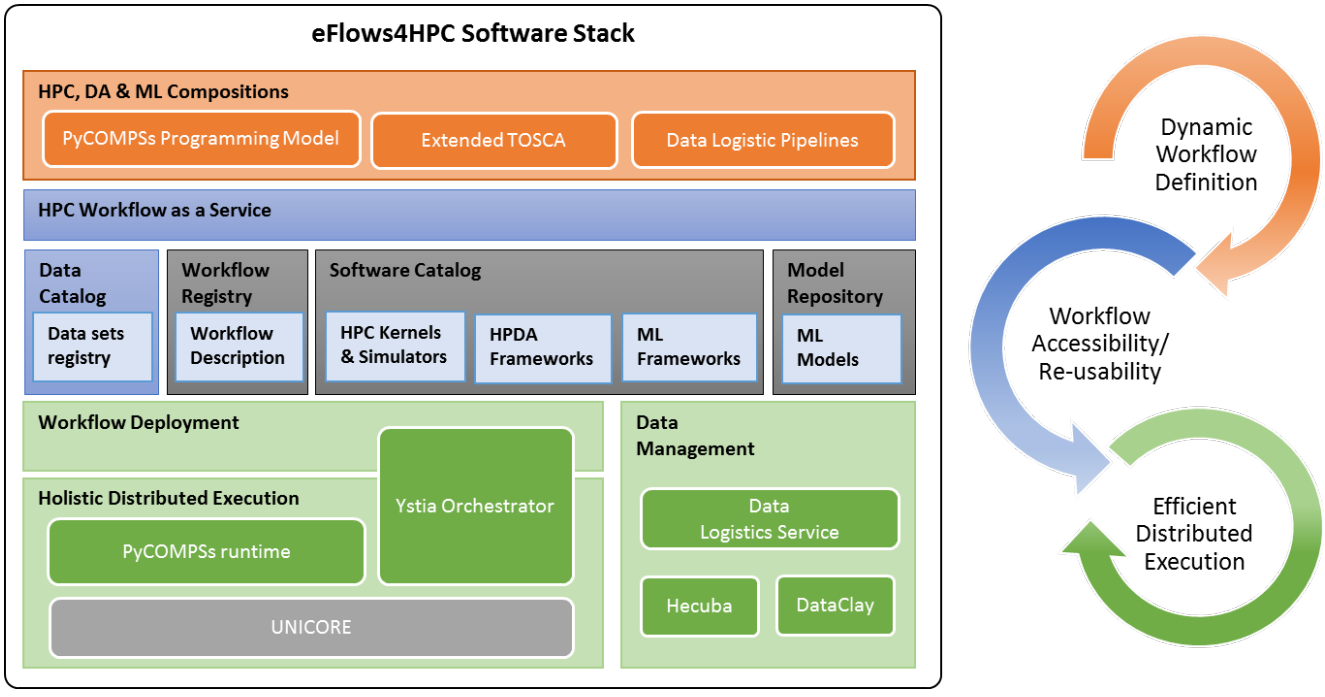


Figure 1: Software Stack release overview.

The different components of the stack can be also are grouped according to their deployment and usage as depicted in Figure 2. The Gateway Services are the components which are deployed outside the computing interface which are used to provide the HPC Workflow as a Service capabilities (Alien4Cloud and Execution API), orchestrate the deployment, execution and data movement of the overall workflow (Ystia Orchestrator and Data Logistics Service). The Runtime components are deployed in the computing infrastructure to perform the parallel execution and data management of the workflow inside the assigned computing nodes. Finally, the HPDA/ML Frameworks are the software components which are used inside the workflows to implement the Machine Learning and Data Analytic algorithms.

Next sections provide an overview of the software stack components as well as the links to the open source

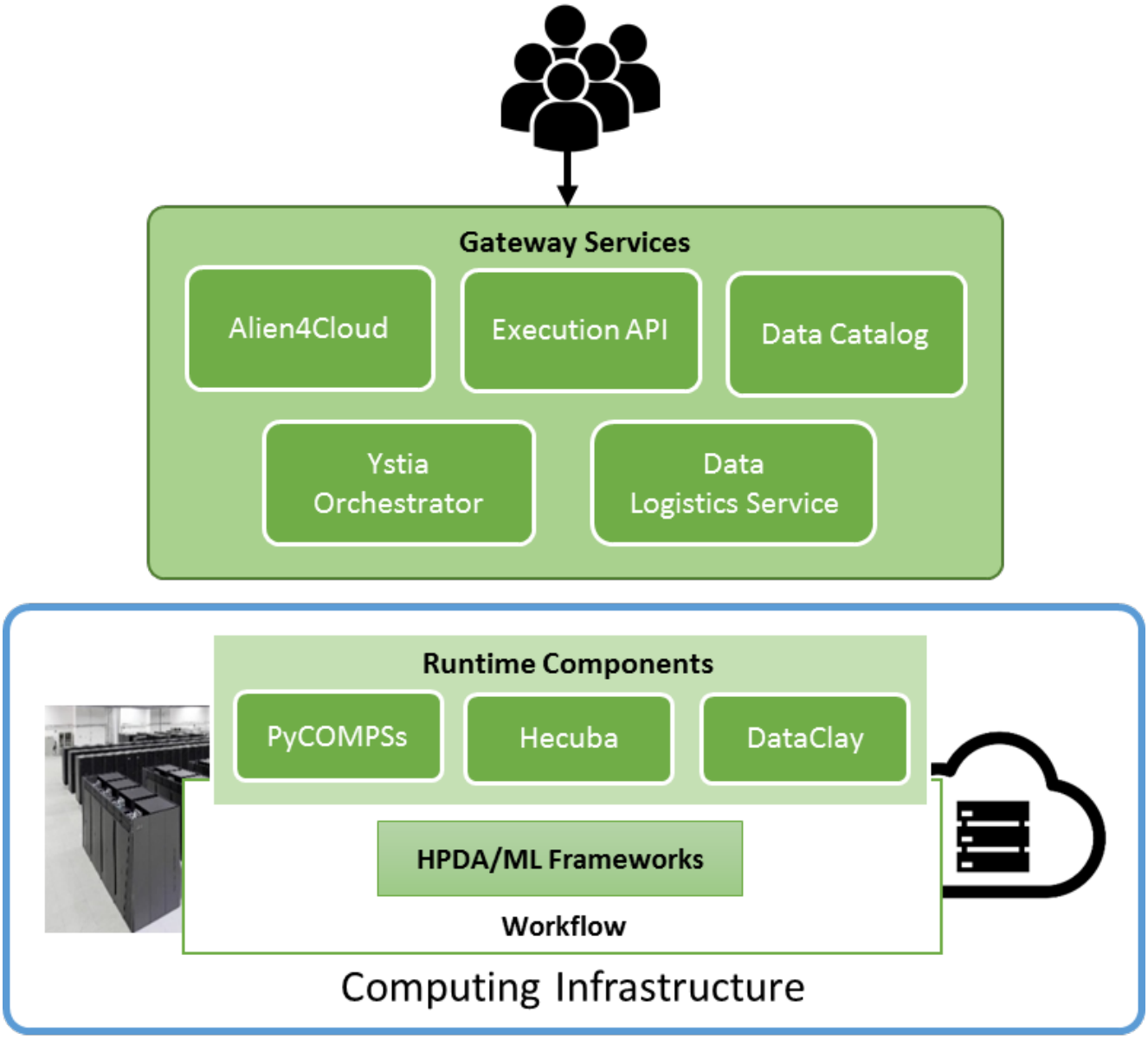


Figure 2: Deployment view of the different Software Stack components.

repositories, installation and usage guides.

2.1 Gateway Services

2.1.1 Data Catalog

The following describes the architecture of the eFlows4HPC Data Catalog. The service will provide information about data sets used in the project. The catalog will store information about locations, schemas, and additional metadata.

Main features:

- keep track of data sources used in the project (by workflows)
- enable registration of new data sources
- provide user-view as well as simple API to access the information

The Data Catalog is mainly developed at FZJ. The source code for stable versions can be found in this Repository. A description of the architecture can be found here.

The running istance with content is hosted on the HDF Cloud and can be accessed at this Address.

The Data Catalog offers an API to access and manipulate its content.

2.1.2 Data Logistics Service

The Data Logistics Service is responsible for data movements part of the workflows developed in the project.

The service is based on Apache Airflow. The project specific extensions and data pipelines formalizing the data movements can be found in the project repository.

From the user perspective, the most important part of the service are the definitions of data movements (pipelines). Some examples (e.g. minimal workflow) of those are provided in the repository. A good starting point for defining own pipelines is the original documentation. Please note that the pipelines are defined in Python programming language and can execute shell scripts. That means that if the users already have their own solution for data movements which are based on scripts or Python programs they can easily be moved to the Data Logistics Service to obtain a running environment with monitoring, retires upon failure, etc.

There is a testing instance of the data logistics service hosted in HDF could which can be accessed.

2.1.3 Alien4Cloud

Alien4Cloud is an REST API and a Graphical User Interface that allows to store, design and deploy complex applications made of reusable components thanks to the TOSCA specification.

In the context of eFlows4HPC, Alien4Cloud will be used by a workflow developer to design and deploy applicative workflows. End users will not interact directly with Alien4Cloud but with a simplified REST interface called the HPC Workflow as a Service (HPCWaaS) API. The HPCWaaS API will in turn interact with Alien4Cloud REST API to execute the workflows.

Alien4Cloud relies on the Yorc orchestration engine to actually execute the workflows.

Alien4Cloud is an open source project developed by Atos. The source code can be found in the project repository and the documentation is available online.

2.1.4 Ystia Orchestrator

Yorc is a TOSCA orchestration engine. It is designed to execute workflows on hybrid (Cloud / HPC / CaaS / ...) infrastructures.

In the context of eFlows4HPC, Yorc will be driven by Alien4Cloud. Developers and end users do not directly interact with Yorc.

Yorc is an open source project developed by Atos. The source code can be found in the project repository and the documentation is available online.

2.1.5 Workflow Execution Service

The Workflow Execution Service is a RESTful web service that provides a way for end users to execute workflows. This component is developed specifically for the eFlows4HPC project.

This service will interact with Alien4Cloud list and trigger applicative workflows and with Hashicorp Vault to manage users access credentials.

The source code can be found in the project repository.

2.1.5.1 Installation

The easiest way to install this service is to use docker. A docker image is automatically published with latest changes under the name `ghcr.io/eflows4hpc/hpcwaas-api:main`.

At press time there is no released version of this service yet. We will follow semantic versioning to tag our releases and containers. All the containers will be available in the project docker registry.

2.1.5.2 Running the service using docker

Please refer to the help of the hpcwaas-api container to know how to run it.

```
docker run ghcr.io/eflows4hpc/hpcwaas-api:main --help
```

2.2 Runtime Components

2.2.1 PyCOMPSs

COMPSs is a task-based programming model which provides parallel execution of applications on distributed systems. Its model abstracts the application from the underlying distributed infrastructure, allowing it to be portable between infrastructures with diverse characteristics. PyCOMPSs is the Python binding of COMPSs.

When developing with PyCOMPSs, distribution of the data, task scheduling, data dependency between tasks, and fault tolerance issues are hidden to the user and are the responsibilities of the COMPSs Runtime. The COMPSs Runtime is also able to react to task failures and exceptions in order to adapt the behaviour accordingly.

Programs written in a sequential way can be converted to PyCOMPSs applications simply by adding 'task' decorators to the functions that can be executed in parallel with other tasks. These sample applications show how to tag tasks to-be-parallelized.

Tasks in PyCOMPSs can be of different granularity, from fine grain tasks with short duration to invocation to external binaries (including MPI applications) that last longer time. This flexibility enables PyCOMPSs to support the development on workflows with heterogeneous task types.

Some useful links for more detailed information:

1. Source code.
2. Installation.

3. PyCOMPSs Tutorials.
4. PyCOMPSs Syntax Reference.

2.2.2 dataClay

dataClay is a distributed object store with active capabilities. It is designed to hide distribution details while taking advantage of the underlying infrastructure, be it an HPC cluster or a highly distributed environment such as edge-to-cloud. Objects in dataClay are enriched with semantics, giving them a structure as well as the possibility to attach arbitrary user code to them. In this way, dataClay enables applications to store and access objects in the same format they have in memory (Python or Java objects), also allowing them to execute object methods within the store to exploit data locality. This active capability minimizes data transfers, as only the results of the computation are transferred to the application, instead of the whole object.

dataClay implements the Storage Runtime Interface that PyCOMPSs can use to enhance data locality of parallalel and distributed applications. This implementation hints the runtime scheduler to assign tasks that access data managed by dataClay to the nodes containing that data, and allows to avoid the cost of serializing this data when it is accessed from several tasks.

Some useful links for more detailed information:

1. Source code: https://github.com/bsc-dom
2. Examples: https://github.com/bsc-dom/dataclay-demos
3. User manual (see Chapter 7 for installation instructions): https://www.bsc.es/research-and-development/software-and-apps/software-list/dataclay/documentation
4. Docker Hub repository: https://hub.docker.com/u/bscdataclay/

2.2.3 Hecuba

Hecuba is a set of tools and interfaces that implement a simple and efficient access to data stores for big data applications. One of the goals of Hecuba is to provide programmers with an easy and portable interface to access data. This interface is independent of the type of system and storage used to keep data, enhancing the portability of the applications. Using Hecuba, the applications can access data like regular objects stored in memory and Hecuba translates the code at runtime into the proper code, according to the backing storage used in each scenario. The current implementation of Hecuba implements this interface for Python applications that store data in memory or Apache Cassandra. Our next release will also include the implementation of an interface for C/C++ applications.

Hecuba also implements the Storage Runtime Interface that PyCOMPSs can use to enhance data locality of parallalel and distributed applications. This implementation hints the runtime scheduler to assign tasks that access data managed by Hecuba to the nodes containing that data, and allows to avoid the cost of serializing this data when it is accessed from several tasks.

Some useful links for more detailed information:

1. Source code and installation instructions: https://github.com/bsc-dd/hecuba
2. Manual: https://github.com/bsc-dd/hecuba/wiki/1:-User-Manual

2.3 ML and DA Frameworks

2.3.1 dislib

The Distributed Computing Library (dislib) is a library that provides various distributed machine-learning algorithms. It has been implemented on top of PyCOMPSs, with the goal of facilitating the execution of big data analytics algorithms in distributed platforms, such as clusters, clouds, and supercomputers.

Dislib comes with two primary programming interfaces: an API to manage data in a distributed way and an estimator-based interface to work with different machine learning models.

Dislib main data structure is the distributed array (ds-array) that enables to distribute the data sets in multiple nodes of a computing infrastructure. The typical workflow in dislib consists of the following steps:

- Reading input data into a ds-array.
- Creating an estimator object.
- Fitting the estimator with the input data.
- Getting information from the model's estimator or applying the model to new data.

Some useful links for more detailed information:

1. Source code.
2. Installation.
3. Tutorial.

2.3.2 EDDL

EDDL is an open-source software for deployment of neural network models on different target devices. EDDL allows the instantiation of many of the current neural network topologies, including CNNs, MLP, and Recurrent networks, performing training and inference. Training can be deployed in an HPC system by the use of COMPSs and MPI/NCCL. For this, a distributed training algorithm is used.

Inside EDDL, a Tensor class is provided with all required tensor manipulation functions needed in neural networks. Currently, EDDL runs on CPU systems, GPU (NVIDIA devices) systems and FPGAs (Xilinx devices). EDDL allows a transparent use of devices.

EDDL is written in C++. A python wrapper is available. EDDL is available on github.

Complete documentation (description, usage, API, examples) is available.

2.3.2.1 Installation

EDDL allows different methods for installation. The simplest one is by using conda:

```
conta install -c deephealth eddl-cpu
```

More information and alternatives are available in the installation section of the documentation page.

2.3.2.2 Usage

When EDDL is installed basic and advanced examples are compiled and build. Therefore, the user can practice with these exameples in order to get experience with the library and how can be used. On the documentation page video tutorials are provided aswell.

2.3.3 HeAT

HeAT is a flexible and seamless open-source software for high performance data analytics and machine learning. It provides highly optimized algorithms and data structures for tensor computations using CPUs, GPUs and distributed cluster systems on top of MPI. The goal of Heat is to fill the gap between data analytics and machine learning libraries with a strong focus on single-node performance, and traditional high-performance computing (HPC). Heat's generic Python-first programming interface integrates seamlessly with the existing data science ecosystem and makes it as effortless as using numpy to write scalable scientific and data science applications.

HeAT allows you to tackle your actual Big Data challenges that go beyond the computational and memory needs of your laptop and desktop.

2.3.3.1 Installation

The simplest way of installing HeAT is to use pip:

```
pip install heat[hdf5,netcdf]
```

More information can be found in project's git repository.

2.3.3.2 Usage

HeAT main features are:

- support for high-performance n-dimensional tensors
- efficient CPU, GPU and distributed computation using MPI
- powerful data analytics and machine learning methods
- abstracted communication via split tensors
- easy to grasp Python API

There are many usage examples in the git repository and documentation. A good starting point for initial exploration is also the tutorial.

2.3.4 Ophidia

Ophidia is a CMCC Foundation research effort addressing Big Data challenges for eScience. The Ophidia framework represents an open source solution for the analysis of scientific multi-dimensional data, joining HPC paradigms and Big Data approaches. It provides an environment targeting High Performance Data Analytics (HPDA) through *parallel* and *in-memory data processing, data-driven task scheduling* and *server-side analysis*. The framework exploits an array-based storage model, leveraging the datacube abstraction from OLAP systems, and a hierarchical storage organisation to partition and distribute large multi-dimensional scientific datasets over multiple nodes. Ophidia is primarily used in the climate change domain, although it has also been successfully exploited in other scientific domains.

Software license: GPLv3.

2.3.4.1 Installation

The framework is composed by different software components. The source code for the various components is available on GitHub.

The installation guide is available in the documentation.

For the client side, Ophidia also provides the Python bindings, called PyOphidia. To install PyOphidia:

```
pip install pyophidia
```

or to install in a *Conda* environment:

```
conda install -c conda-forge pyophidia
```

2.3.4.2 Usage

Ophidia provides features for data management and analysis, such as:

- data reduction and subsetting
- data intercomparison
- array processing
- time series analysis
- statistical and mathematical operations
- data manipulation and transformation
- interactive data exploration

The user guide documents all the available Ophidia features.

2.3.5 ParSoDA

ParSoDA (Parallel Social Data Analytics) is a high-level library for developing parallel data mining applications based on the extraction of useful knowledge from large data set gathered from social media. The library aims at reducing the programming skills needed for implementing scalable social data analysis applications.

The main idea behind ParSoDA is to simplify the creation of data analysis applications, making some aspects of development transparent to the programmer. The main effort for developing ParSoDA was to create a set of interfaces, abstract classes and concrete classes that could be reused several times and in a modular way for composing scalable and distributed data analysis workflows. The first prototype of ParSoDA was built on Apache Hadoop. Another version of ParSoDA based on Spark has been implemented. The Spark version has proven to offer several performance benefits compared to the Hadoop-based version. We are now working on a new version based on PyCOMPSs.

2.3.5.1 Source code

The source code of ParSoDA is available here.

The current version of the library (v. 1.3.0 dated October 25, 2018) contains more than forty predefined functions organized in seven packages, corresponding to the seven ParSoDA steps.

2.3.5.2 Installation and use guide

The software requirements of ParSoDA are:

- Java JDK 1.8 or higher
- Maven as dependency manager and build automation tool. We used Maven for our convenience, but it doesn't mean that other valid solutions, such as Gradle, can't be used.
- GIT as versioning tool

The current version of ParSoDA has been tested with Hadoop 2.7.4, but we are working on addressing some minor issues to make it work with Hadoop version 3.

On the ParSoDA project available on GitHub, you can find a dedicated branch containing a docker-compose file that can be used to quickly deploy a Hadoop cluster with only 1 node, which can be used to test ParSoDA applications.

1) Clone the master branch of the ParSoDA's project from GitHub:

```
git clone --branch master https://github.com/SCAlabUnical/ParSoDA.git
```

2) After cloning the project, you have to launch the following command to download and install all the project dependencies:

```
mvn install
```

3) If required, add to the Maven project any external libraries you need. For example, the sample applications presented today required two external JAR libraries. In particular, we used **SPMF**, which is an open-source data mining library written in Java, specialized in pattern mining. We also used a Hadoop implementation of the well-known PrefixSpan algorithm, called **MGFSM**, to extract frequent sequential patterns. To import these libraries, you can run the following commands:

```
mvn install:install-file -Dfile=./libs/spmf.jar -DgroupId=ca.pfv.spmf -DartifactId=spmf -
↪Dversion=1.0.0 -Dpackaging=jar

mvn install:install-file -Dfile=./libs/mgfsm-hadoop.jar  -DgroupId=de.mpii.fsm -
↪DartifactId=mgfsm-hadoop -Dversion=1.0.0 -Dpackaging=jar
```

4) Finally, to build an executable JAR you can use the following command:

```
mvn package.
```

The library code has been organized into packages, which follow the 7 main steps that compose the execution flow of ParSoDA: acquisition, filtering, mapping, reduction, partitioning, analysis, and visualization. It is organized in packages among which we find the followings:

- The package "*app*" contains some runnable example of data analysis applications based on ParSoDA;
- The package "*common*" contains the core classes of ParSoDA, including data models, intefaces, abstract classes, and so on;
- The package "*acquisition*" contains the classes of some data crawlers that can be used for collecting data from social media platforms. Currently, it contains 2 crawlers for social media platforms (i.e., Twitter and Flickr), plus a dummy crawler (called *FileReaderCrawler*) that allow to load data from local filesystem or HDFS filesystem;
- All other packages contains some pre-built functions for each corresponding block of a ParSoDA application.

Chapter 3

Programming Interfaces for integrating HPC and DA/ML workflows

The evolution of High-Performance Computing (HPC) platforms enables the design and execution of progressively more complex and larger workflow applications in these systems. The complexity comes not only from the number of elements that compose a workflow but also from the type of computations performed. While traditional HPC workflows include simulations and modeling tasks, current needs require in addition Data Analytic (DA) and artificial intelligence (AI) tasks. However, the development of these workflows is hampered by the lack of proper programming models and environments that support the integration of HPC, DA, and AI. Each of these workflow phases are developed using dedicated frameworks for the specific problem to solve. However, to implement the overall workflow, developers have to deal with programming large glue code to integrate the execution of the different frameworks executions in a single workflow.

eFlows4HPC proposes a programming interface to try to reduce the effort required to integrate different frameworks in a single workflow. This integration can be divided in two parts:

- **Software Invocation Management:** It includes the actions required to execute an application with a certain framework. This can be invoking just a single binary, a MPI application or a model training with a certain ML framework.
- **Data Integration:** In includes the transformations that the data generated by a framework has to be applied to be used by another framework. This can include transformations like transpositions, filtering or data distribution.

The proposed interface aims at declaring the different software invocations required in a workflow as simple python functions. This functions will be annotated by two decorators :

- **@software** to describe the type of execution to be performed when the function is invoked from the main code
- **@data_transformation** to indicate the required data transformations that a parameter of the invocation has to apply to be compatible with the input of expected execution.

For this first iteration, we have defined the software invocation descriptions and we have extended the PyCOMPSs programming model and runtime. Next versions of the eFlows4HPC framework will include the definition of the data transformations and their implementation.

3.1 Software Invocation Description

The idea behind the 'Software' invocation description is to define a common way in which multiple software components can be integrated in single workflow. The definition is composed of a decorator and a configuration file with the necessary parameters.

3.1.1 Software decorator

@software decorator is used to indicate that a certain python function represents the invocation of and external HPC or DA programs in a single workflow. This decorator must be combined with the 'task' decorator to indicate de directionality of the parameters and allow the runtime to detect the dependencies with the rest of workflow tasks. When a *task* with the *@software* decorator is called, an external program is executed respecting the configuration defined in its configuration file. The goal of this decorator is to describe the execution of whatever external programs included in a workflow from simple binary executable to complex MPI applications.

3.1.2 Configuration File

A configuration file must have two mandatory keys; *type* and *properties*. 'Type' is needed to specify exactly what type of program the user wants to execute (e.g: "mpi", "binary"). And the *properties* will contain the customizable properties of a type of execution such as binary path, number of processes, etc.

3.1.3 Examples

As an example, the following code snippets show how an MPI application execution can be defined using the @software decorator. Users only have to add the software decorator on top of the task function, and provide a 'config_file' parameter where the configuration details are defined:

```
from pycompss.api.software import software
from pycompss.api.task import task

@software(config_file="mpi_config.json")
@task()
def task_python_mpi():
    pass
```

And inside the configuration file the type of program, and its properties are explicitly set. For example, if the user wants to run an MPI job with two processes using 'mpirun' command, the configuration file (**"mpi_config.json"** in this example) should look like as follows:

```
{
  "type":"mpi",
  "properties":{
      "runner": "mpirun",
      "binary": "app_mpi.bin",
      "processes": 2
  }
}
```

Finally, call to the task function from the main program:

```
task_python_mpi()
```

It is also possible to refer to task parameters from the configuration file. Properties such as *working_dir* and *params* ('params' strings are command line arguments to be passed to the 'binary') can contain this kind of references. In this case, the task parameters should be surrounded by curly braces. For example, in the following

example, 'work_dir' and 'param_d' parameters of the python task are used in the 'working_dir' and 'params' strings, respectively . Moreover, the number of computing units is added as a constraint, to indicate that every MPI process will have this requirement (run with 2 threads):

Task definition:

```
from pycompss.api.software import software
from pycompss.api.task import task

@software(config_file="mpi_w_params.json")
@task()
def task_mpi_w_params(work_dir, param_d):
     pass
```

Configuration file ("mpi_w_params.json"):

```
{
  "type":"mpi",
  "properties":{
      "runner": "mpirun",
      "binary": "parse_params.bin",
      "working_dir": "/tmp/{{work_dir}}",
      "params": "-d {{param_d}}"
  },
  "constraints":{
      "computing_units": 2
  }
}
```

Call to the task function:

```
task_mpi_w_params('my_folder', 'hello_world')
```

Another example can be when the external program is expected to run within a container. For that, the user can add the *container* configuration to the JSON file by specifying its 'engine' and the 'image'. At the time of execution, the Runtime will execute the given program within the container. For example, in order to run a simple 'grep' command that searches for a pattern (e.g. an 'error') in the input file within a Docker container, the task definition and the configuration file should be similar to the examples below:

Task definition:

```
from pycompss.api.parameter import FILE_IN
from pycompss.api.software import software
from pycompss.api.task import task

@software(config_file="container_config.json")
@task(in_file=FILE_IN)
def task_container(in_file, expression):
     pass
```

Configuration file ("container_config.json"):

```
{
  "type":"binary",
  "properties":{
      "binary": "grep",
      "params": "{{expression}} {{in_file}}"
  },
  "container":{
```

(continues on next page)

(continued from previous page)

```
                "engine": "DOCKER",
                "image": "compss/compss"
        }
}
```

Call to the task function:

```
task_container('some_file.txt', 'error')
```

Warning: Limitation: Currently it is not possible to run MPI jobs within containers.

For more detailed information about the @software decorator of PyCOMPSs please see the documentation.

Chapter 4

HPCWaaS Methodology

The eFlows4HPC proposes the HPC Workflow as a Service (HPCWaaS) methodology which tries to apply the usage model of the Functions as a Service (FaaS) in Cloud environments to the workflows for HPC systems. In this model, two main roles are identified. From one side, the function developer is in charge of developing and registering the function in the FaaS platform, which transparently deploys the function in the cloud infrastructure. On the other side, the final user executes the deployed function using a REST API. In the case of running workflows in HPC systems, we can find similar roles. First, we can find the workflow developer, which is charge of developing and deploying the workflow in the computing infrastructure, and the users' communities which are usually scientist who want to execute the workflow and collect their results to advance in their scientific goals.

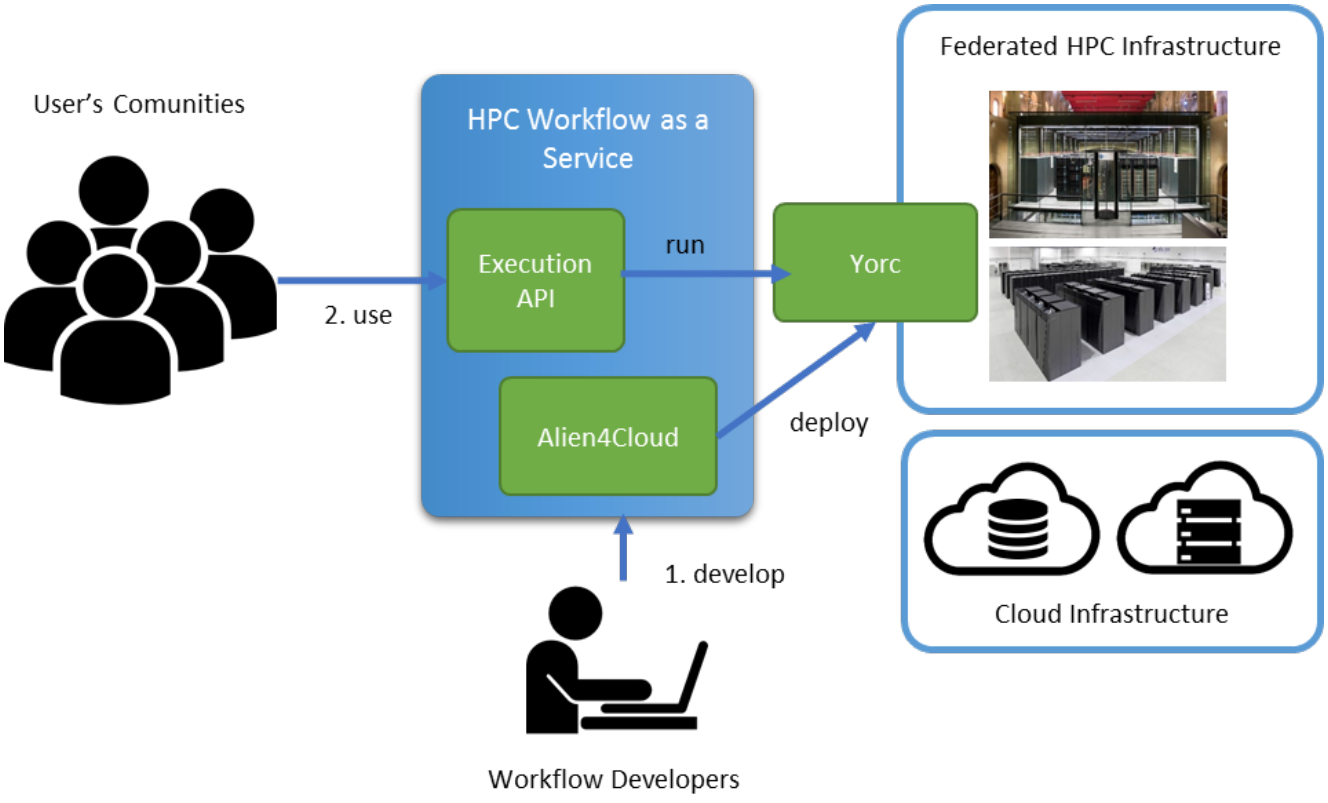


Figure 3: HPC Workfow as a Service overview

Figure 3 shows how these two roles interacts with the proposed HPCWaaS methodology. Workflow developers implement and describe the workflow in a way that allows the eFlows4HPC Gateway services to automatically deploy and orchestrate the workflow execution. This is done interacting with the Development Interface offered by the Alien4Cloud tool to describe workflows as a TOSCA application. Once the workflow is deployed users' communities can invoke this workflow using the Execution API.

Next sections provide more details about these interfaces. A simple workflow example can be found *here*.

4.1 Development Interface

4.1.1 Setup

4.1.1.1 Alien4Cloud & Yorc

Please refer to the documentation of the Alien4Cloud & Yorc project for more information.

An instance of Alien4Cloud and Yorc is available on Juelich cloud, ask to the project to obtain access

4.1.1.2 Importing required components into Alien4Cloud

Some TOSCA components and topology templates need to be imported into Alien4Cloud. If you are using the instance on Juelich cloud, this is already done and you can move to the next paragraph.

You should first move to the `Catalog` tab and then the `Manage archives` tab, finally click on `Git import` to add components as shown in Figure 4.

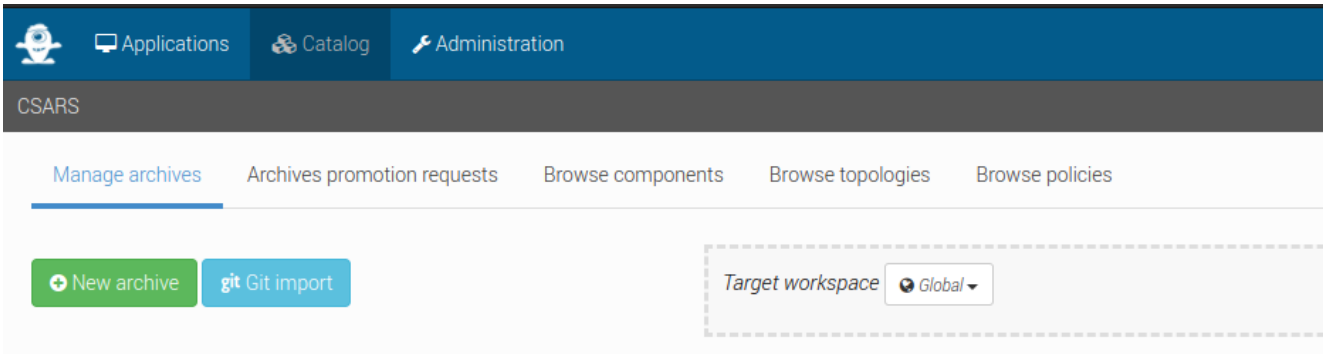


Figure 4: Click on `Git import` to add components

You should have at least the three repositories defined as shown in Figure 5:

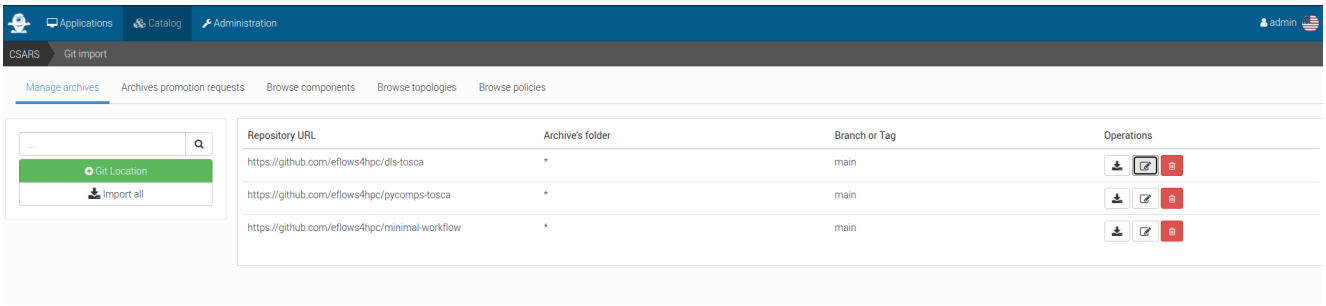


Figure 5: Click on `Git location` to define imports from a git repository

Click on `Git location` to define imports from a git repository as shown in Figure 6

Once done you can click on `Import all`.

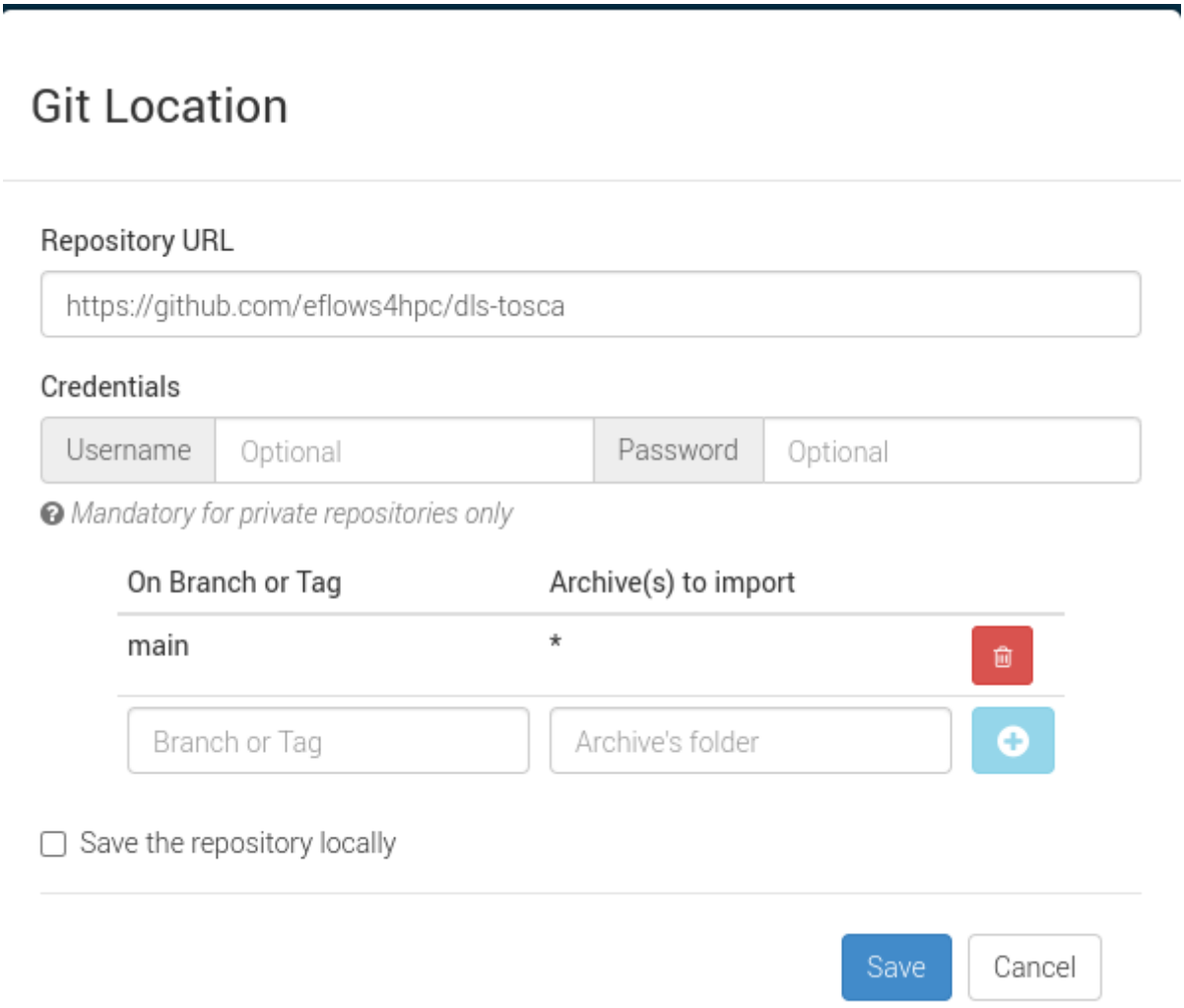


Figure 6: Alien4Cloud setup a catalog git repository

4.1.2 Creating an application based on the minimal workflow example

Move to the `Applications` tab and click on `New application` as shown in Figure 7.

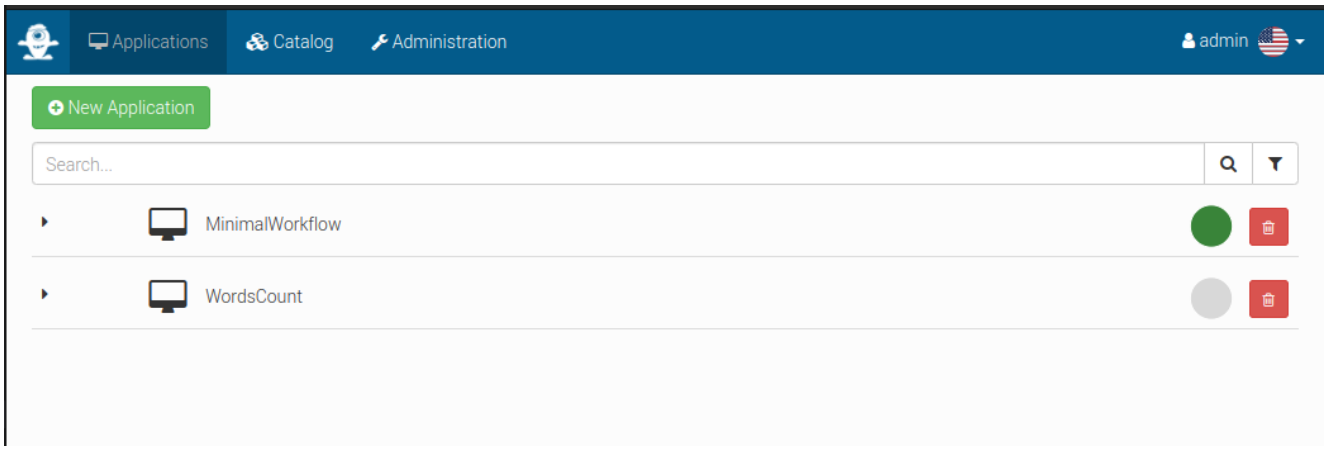


Figure 7: Manage applications in Alien4Cloud

Then create a new application based on the minimal workflow template as shown in Figure 8

Edit the topology to fit your needs as shown in Figure 9.

Then click on `Deploy` to deploy the application as shown in Figure 10.

4.1.3 Make your workflow available to end-users using the HPCWaaS API

In order for the HPCWaaS API to know which workflow to allow users to use, you should add a specific tag to your Alien4Cloud application. Move to your application main panel and under the `Tags` section add a tag named `hpcwaas-workflows` as shown in Figure 11. The tag value should be a coma-separated list of workflow names that could be called through the HPC HPCWaaS API. In the minimal workflow example, this tag value should be `exec_job`.

4.2 Execution API

The execution API is still under active development and is subject to changes. Please refer to the repository documentation for a detailed description of the current status of the different endpoints of this API.

A Command Line Interface (CLI) allows to interact with the service. It is available as a container. Please refer to the help of the `waas` container to know how to run it.

```
docker run ghcr.io/eflows4hpc/hpcwaas-api:main-cli --help
```

The API can also be accessed directly through its HTTP interface with tools like `curl` or any programming language.

There is a running instance on Juelich cloud, ask to the team for access to the API.

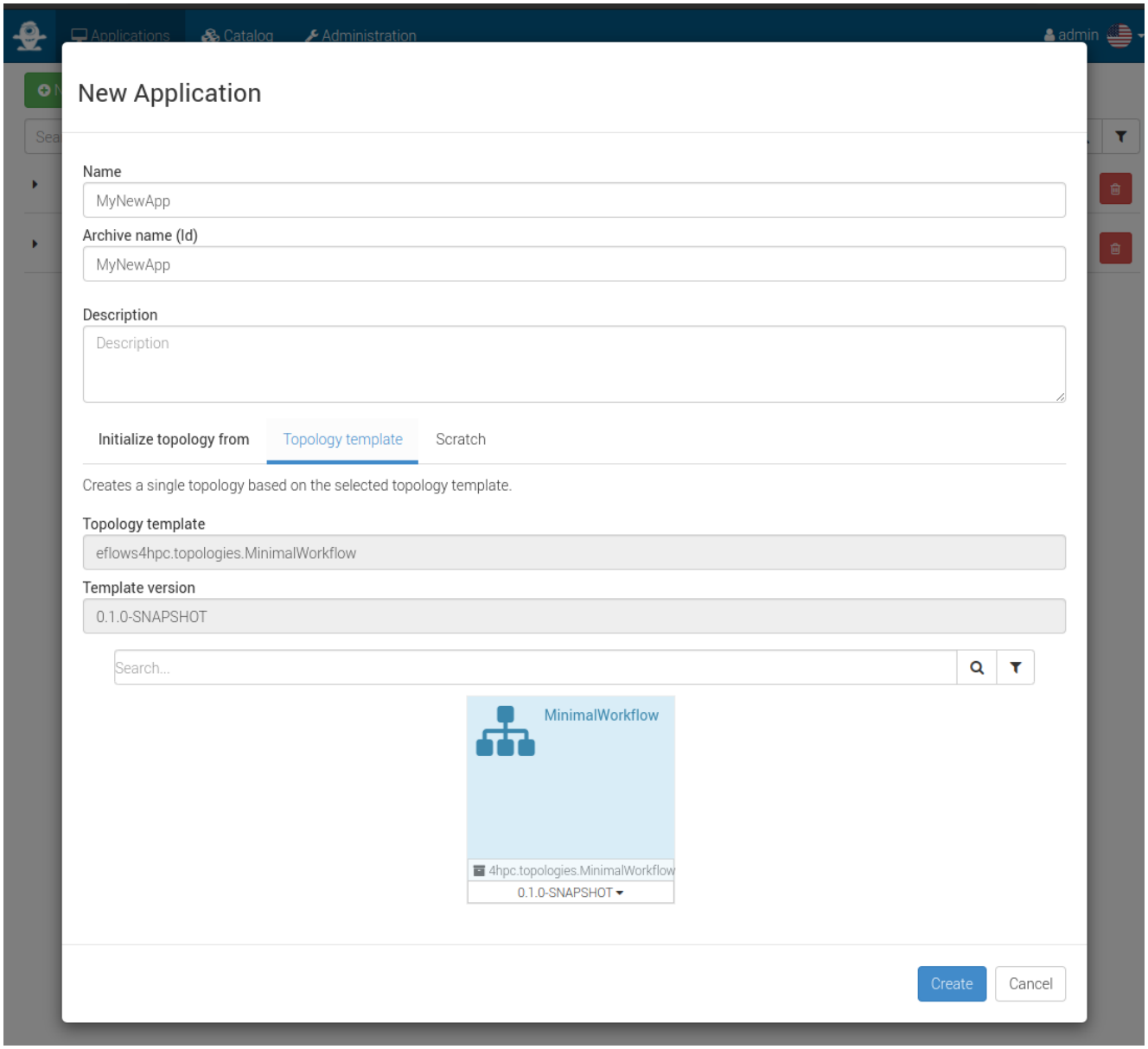


Figure 8: Alien4Cloud create a template based application

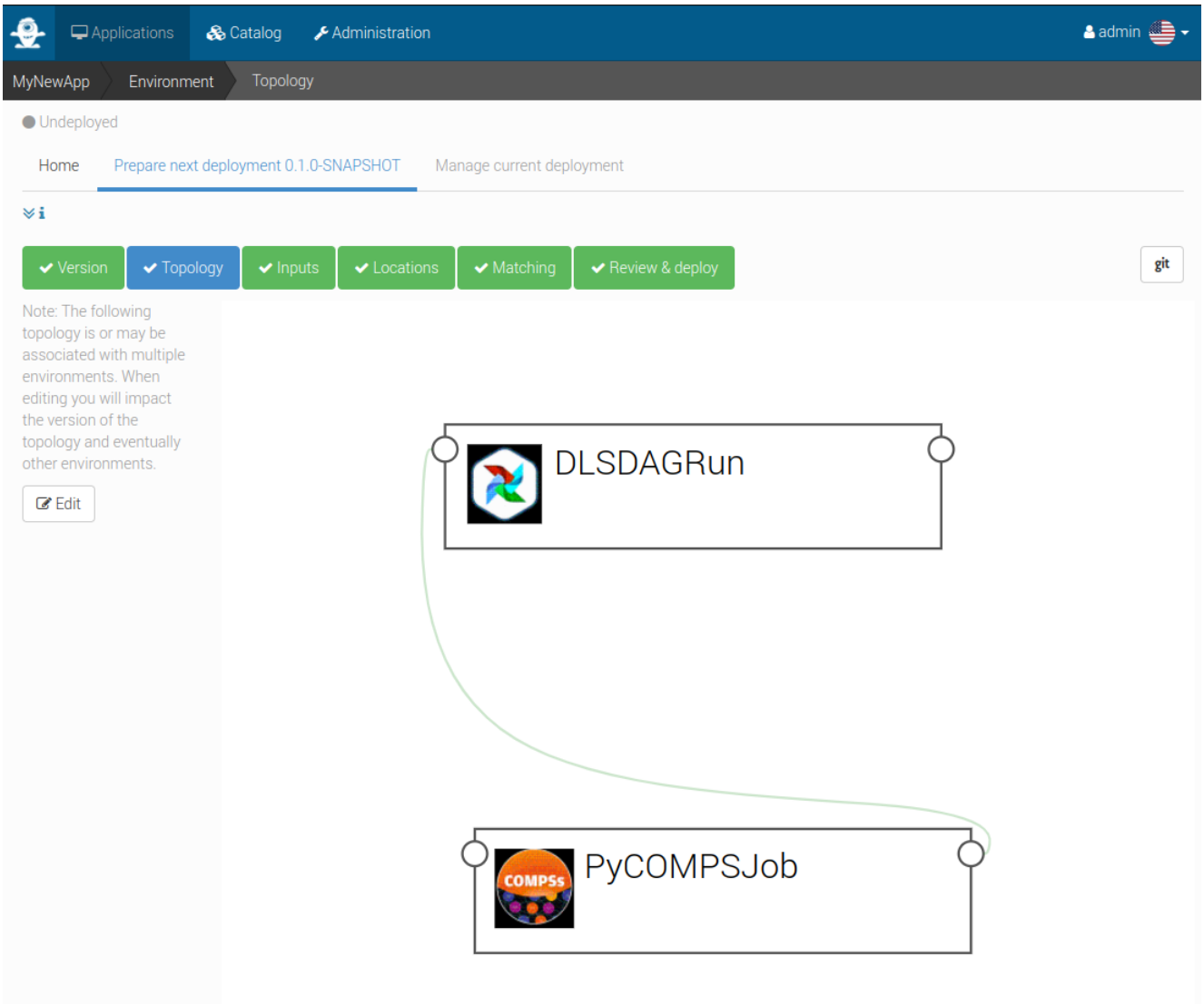


Figure 9: Alien4Cloud minimal workflow topology

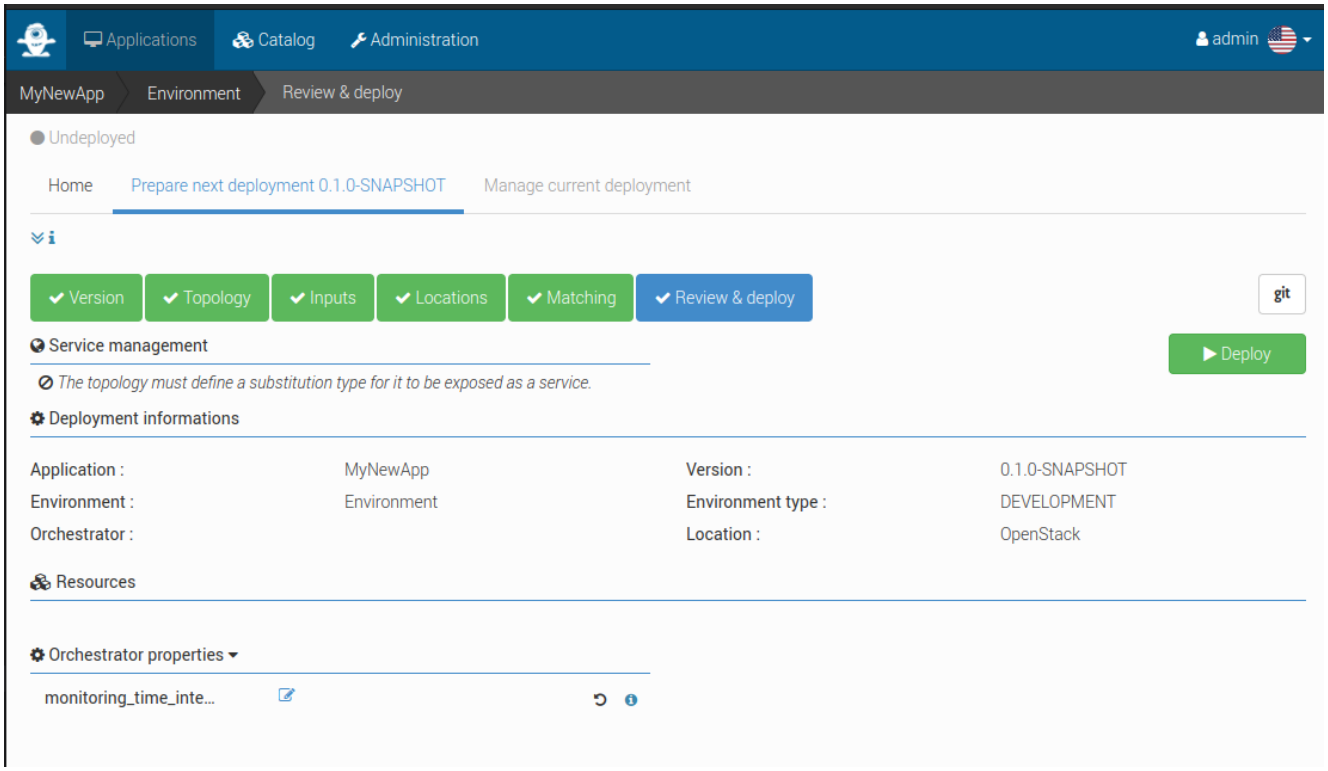


Figure 10: Alien4Cloud deploy an application

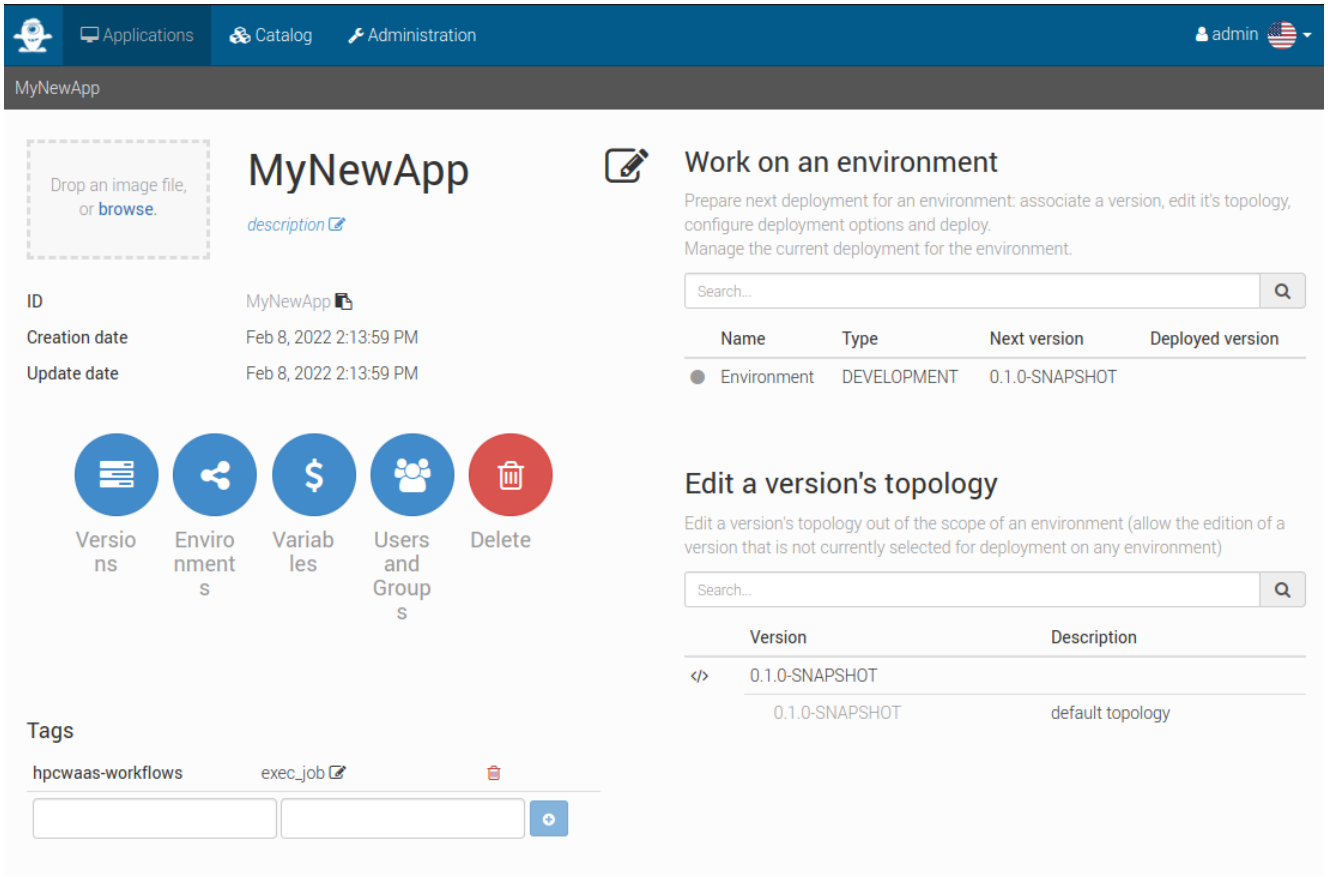


Figure 11: Alien4Cloud add tags to an application

4.2.1 Basic usage

First you need to setup your SSH credentials using the *Create an SSH Key Pair for a given user endpoint <https://github.com/eflows4hpc/hpcwaas-api/blob/main/docs/rest-api.md#create-an-ssh-key-pair-for-a-given-user>_*. By calling this endpoint the API will create a new SSH key pair and store it into a vault you will receive in return of this call the public key. You will never get or even see the private key. Add this public key as an authorized key for your HPC user account in order to let transfer data to your user account and run PyCOMPS jobs for you in an automated way.

Then you can use the list available workflows endpoint to get the list of endpoints you can access.

You can then trigger a workflow execution.

And finally monitor the workflow execution.

Chapter 5

Usage Example

This section describe a minimal usage example on how to implement, deploy and execute a workflow using the eFlows4HPC Software Stack. This example workflow consists of two main steps:

- a data logistic pipeline, where the input data is moved from an EU Data repository to the parallel file system of a supercomputer where it will be processed in the second step.
- a PyCOMPSs workflow, where an word-count computation is parallelized across the nodes of an HPC facility using a task-based programming model .

The deployment and execution of these two steps are described as a TOSCA application using the HPCWaaS methodology.

In this first version of the workflow, we have assumed that the required software and the access credentials are already deployed in the infrastructure. Next versions of this document will include how to do it with the eFlows4HPC Software Stack.

Next sections provide more details about how the different steps are implemented.

5.1 Implementing Data Logistics Pipeline

Data movements in eFlows4HPC minimal workflow are orchestrated by the Data Logistics Service and defined within it as Airflow Pipelines. The pipelines formally are Direct Acyclic Graphs (DAGs) and are defined programmatically with Python.

Each DAG definition is comprised of set of tasks and additional metadata. The metadata can be used to, e.g., orchestrate periodic data movements. The tasks are then executed by Airflow workers. The most common type of tasks are Operators. Airflow provides a wide range of Operators to interact with different data services and storages. It is also possible to create own operators.

Following will provide a short introduction to Data Logistics Pipelines based on the example of the eFlows4HPC minimal workflow. The complete source code of the minimal workflow pipeline can be found in repository. The workflow is build following the principle of Extract Transform Load (ETL) and uses Airflow taskflow API to define a DAG.

5.1.1 DAG Definition

The structure of DAG is defined as follows:

```
@dag(default_args=default_args, schedule_interval=None, start_date=days_ago(2), tags=['example
↪'])
def taskflow_example():

    @task
    def setup(**kwargs):
        ....

    @task(multiple_outputs=True)
    def extract(conn_id, **kwargs):
        ....

    ....

    conn_id = setup()
    data = extract(conn_id)
    files = transform(data)
    ucid = load(connection_id = conn_id, files=files)
    remove(conn_id=ucid)

dag = taskflow_example()
```

The DAG is defined as a Python annotated function `taskflow_example`. The submethods annotated with `@task` are Operators, finally the dependencies between tasks emerge from the order of function calls in the `taskflow_example` method.

5.1.2 Data Movement Tasks

The minimal workflow encompasses following data movements:

- download from B2SHARE repository,
- upload to target system with SCP/SFTP,
- upload computation results to B2SHARE repository.

The code for accessing and downloading from B2SHARE can be seen in repository. Objects in B2SHARE comprise of an id, set of metadata and list of files. To identify which object needs to be downloaded, the object id needs to be passed to the DAG as an `oid` parameter. The workflow will then locate the object in the B2SHARE, retrieve the list of its files (`extract` task), and download the files to the local storage with the `transform` task. There is also an example of streaming pipeline (which does not download to local storage but rather directly to target location), it can be found in repository.

Next step in data movement is to use SCP to upload the files from B2SHARE to the target system. This is done in the `load` task. The task uses functionality provided by Airflow to access SSH/SCP systems.

5.1.3 Connection setup

The credentials needed to access storages are passed to the DAG. Based on their content a temporary Airflow connection is created, used by Data Movement Tasks and removed subsequently. The connection management is taken care of by `setup` and `remove` tasks.

5.1.4 Closing remarks

Please review the examples in the repository to gain understanding how the data movements are realized. There are examples of upload/download to remote repository, streaming, accessing storages through SCP/SFTP or HTTP.

The repository also includes a set of tests and mocked tests to verify the correctness of the pipelines.

For local testing, you can use airflow standalone setup. Please refer to Airflow documentation for that.

5.2 PyCOMPSs Workflow

PyCOMPSs is a task-based programming model which allow to define parallel workflows as simple sequential python scripts. To implement a PyCOMPSs application, developers has to identify what parts of an application are the candidates to be a task. They are usually python methods with a certain computation granularity (larger than hundred milisecons) that can potentially run concurrently with other parts of the application. Those methods have to be annotated with the *@task* decorator to indicate the directionality of they parameters.

Code 1 shows how to program a PyCOMPSs workflow for counting the words in a folder. It can be found in the application repository. The code is similar to what a developer will write in a sequential python code. Two methods are defined in the application: the *wordcount* to count the words of a file; and the *merge_ dicts* to merge the results of the separate *wordcount* tasks. On top of these methods, we have added the *@task* decorator to convert it to a PyCOMPSs task, indicating the directionality of the parameters and returns. Based on these annotations, the COMPSs runtime will detect that all *wordcount* invocations are independent and the *merge_ dicts* ones will depend to the *wordcount* task of the same iteration and the *merge_ dicts* of the previous one.

Code 1: PyCOMPSs wordcount example

```
@task(file_path=FILE_IN, returns=dict)
def wordCount(file_path):
    """ Construct a frequency word dictorionary from a list of words.
    :file_path: file to count words
    :return: a dictionary where key=word and value=#appearances
    """
    partialResult = {}
    with open(file_path, 'r') as f:
        for line in f:
            data = line.split()
            for entry in data:
                if entry in partialResult:
                    partialResult[entry] += 1
                else:
                    partialResult[entry] = 1
    return partialResult


@task(returns=dict, priority=True)
def merge_dicts(dic1, dic2):
    """ Update a dictionary with another dictionary.
    :param dic1: first dictionary
    :param dic2: second dictionary
    :return: dic1+=dic2
```

(continues on next page)

(continued from previous page)

```
    """
    for k in dic2:
        if k in dic1:
            dic1[k] += dic2[k]
        else:
            dic1[k] = dic2[k]
    return dic1


if __name__ == "__main__":
    from pycompss.api.api import compss_wait_on

    # Get the dataset path
    pathDataset = sys.argv[1]

    # Read file's content execute a wordcount on each of them
    partialResult = []
    for fileName in os.listdir(pathDataset):
        path = os.path.join(pathDataset, fileName))
        partialResult.append(wordCount(path))

    # Accumulate the partial results to get the final result.
    result = {}
    for partial in partialResult:
        result = merge_dicts(result, partial)

    # Synchronize remote result
    result = compss_wait_on(result)

    # Print the results and elapsed time
    print("Word appearances:")
    from pprint import pprint
    pprint(result)
```

A part from python methods, developers can integrate executions of other software in PyCOMPSs workflows by means of the *@software* decorator described in the *Software invocation description* section.

5.3 Integration in TOSCA

To support the integration of this usage example we defined a set of new TOSCA components.

First we defined new types for the Data Logistics Service and PyCOMPSs workflows. Then we defined a TOSCA topology template called the "minimal workflow" that compose these two previous components into a TOSCA application that allows to run workflows that first transfer data from the Data Catalog to an HPC cluster and then run a PyCOMPSs workflow.

5.3.1 Data Logistics Service TOSCA component

The source code of this component is available in the dls-tosca github repository in the eFlows4HPC organization.

This component interacts with the Airflow RESTful API to trigger and monitor the execution of an airflow pipeline. It was designed to be as generic as possible in order to support different kind of pipelines.

Code 2 is a simplified (for the sake of clarity) version of the TOSCA type definition of the Data Logistics Service that shows the configurable properties that can be set for this component.

Code 2: Extract of the TOSCA definition for DLS

```
dls.ansible.nodes.DLSDAGRun:
  derived_from: org.alien4cloud.nodes.Job
  metadata:
    icon: airflow-icon.png
  properties:
    dls_api_url:
      type: string
      required: true
    dls_api_username:
      type: string
      required: true
    dls_api_password:
      type: string
      required: true
    dag_id:
      type: string
      required: true
    oid:
      type: string
      description: Transferred Object ID
      required: true
    target_host:
      type: string
      description: the remote host
      required: true
    target_path:
      type: string
      description: path of the file on the remote host
      required: true
    extra_conf:
      type: map
      required: false
      entry_schema:
        description: map of key/value to pass to the dag as inputs
        type: string
```

5.3.2 PyCOMPSs TOSCA component

The source code of this component is available in the pycomps-tosca github repository in the eFlows4HPC organization.

This component connects to an HPC cluster using SSH and then run and monitor a PyCOMPSs workflow. Again, this component was designed to be as generic as possible in order to support different kind of workflows.

Code 3 is a simplified (for the sake of clarity) version of the TOSCA type definition of the PyCOMPSs workflow that shows the configurable properties that can be set for this component.

Code 3: Extract of the TOSCA definition for PyCOMPSs

```
pycomps.ansible.nodes.PyCOMPSJob:
  derived_from: org.alien4cloud.nodes.Job
  metadata:
    icon: COMPSs-logo.png
  properties:
    pycomps_endpoint:
      type: string
      description: The endpoint of the PyCOMPSs server
      required: true
    num_nodes:
      type: integer
      required: false
      default: 1
    data_path:
      type: string
      required: false
      default: ""
    command:
      type: string
      required: true
    arguments:
      type: list
      required: false
      entry_schema:
        description: list of arguments
        type: string
```

5.3.3 Minimal workflow TOSCA topology template

The source code of this template is available in the minimal-workflow github repository in the eFlows4HPC organization.

This topology template composes the DLS and PyCOMPSs components into a TOSCA application that allows to run a workflow which first transfer a data from the Data Catalog to an HPC cluster and then run an PyCOMPSs workflow.

Code 4 shows how are defined the components and how they are connected together in order to run in sequence. Figure 12 shows the same topology in a graphical way.

Code 4: Extract of the TOSCA topology template for the minimal workflow

```
topology_template:
  inputs:
    dls_api_username:
      type: string
```

(continues on next page)

(continued from previous page)

```
      required: true
    dls_api_password:
      type: string
      required: true
  node_templates:
    DLSDAGRun:
      metadata:
        a4c_edit_x: 231
        a4c_edit_y: "-339"
      type: dls.ansible.nodes.DLSDAGRun
      properties:
        dls_api_url: "http://134.94.199.73:7001/api/v1"
        dls_api_username: { get_input: dls_api_username }
        dls_api_password: { get_input: dls_api_password }
        dag_id: "taskflow_example"
        oid: dba52935c7e444d198b377876b4fe0a8
        target_host: "amdlogin.bsc.es"
        target_path: "/home/bsc44/bsc44070/dls_transfert/data/"
    PyCOMPSJob:
      metadata:
        a4c_edit_x: 243
        a4c_edit_y: "-176"
      type: pycomps.ansible.nodes.PyCOMPSJob
      properties:
        pycomps_endpoint: "amdlogin.bsc.es"
        num_nodes: 2
        data_path: "/home/bsc44/bsc44070/dls_transfert/data/"
        command: "~/wordcount_blocks/src/wordcount_blocks.py"
        arguments:
          - "${DATA_PATH}/data.txt"
          - "${DATA_PATH}/result.txt"
          - 3000
      requirements:
        - dependsOnDlsdagRunFeature:
            type_requirement: dependency
            node: DLSDAGRun
            capability: tosca.capabilities.Node
            relationship: tosca.relationships.DependsOn
```

Code 5 shows inputs that are required to run the workflow.

Code 5: Extract of the TOSCA definition for PyCOMPSs

```
workflows:
  exec_job:
    inputs:
      user_id:
        type: string
        required: true
      oid:
        type: string
        required: true
      target_path:
        type: string
        required: true
      num_nodes:
```

(continues on next page)

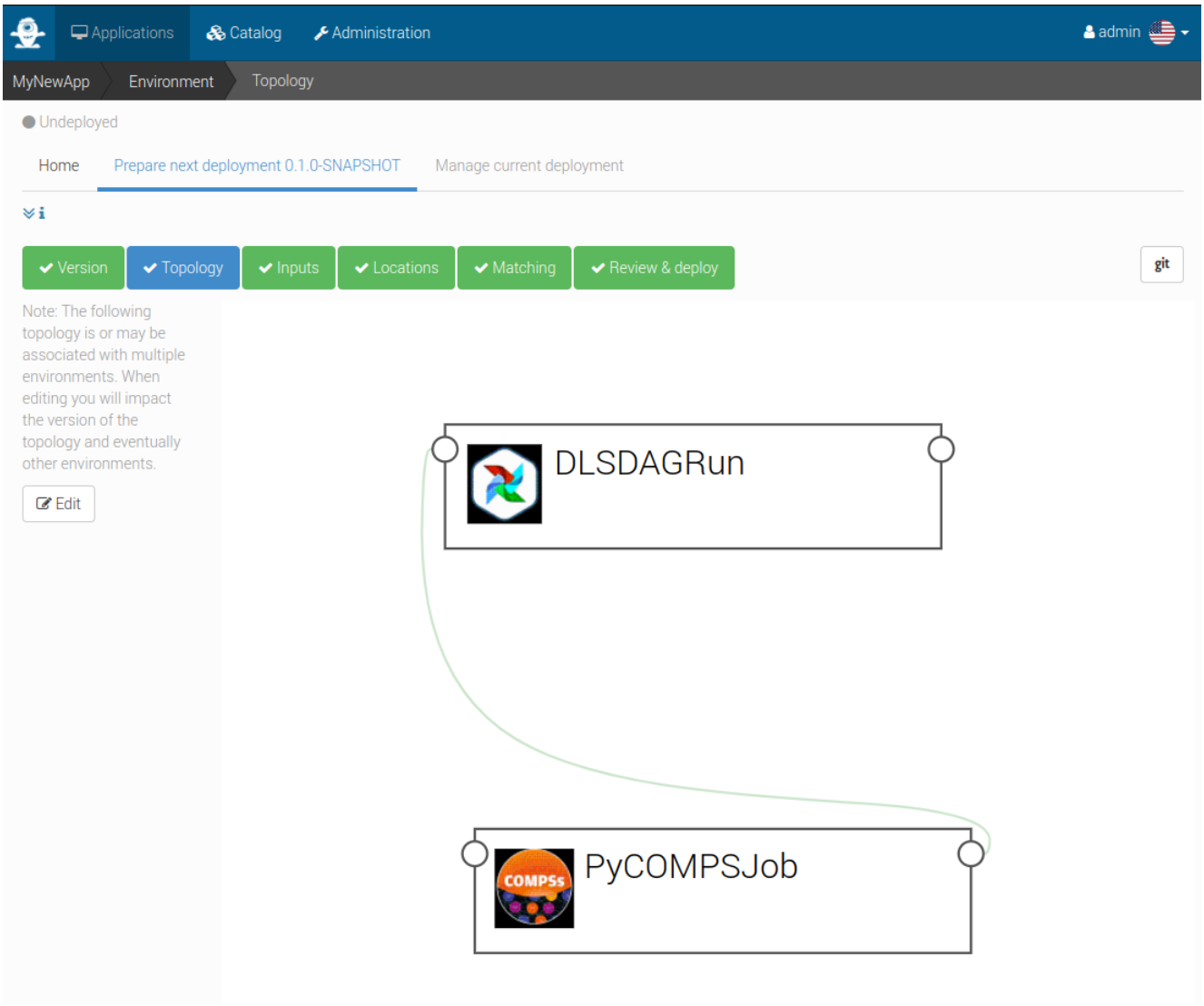


Figure 12: Alien4Cloud minimal workflow topology

(continued from previous page)

```
        type: integer
        required: false
        default: 1
```