eFlows4HPC Documentation

eFlows4HPC Consortium

Last updated : February, 2023

Online version available at eFlows4HPC - ReadTheDocs

Table of contents

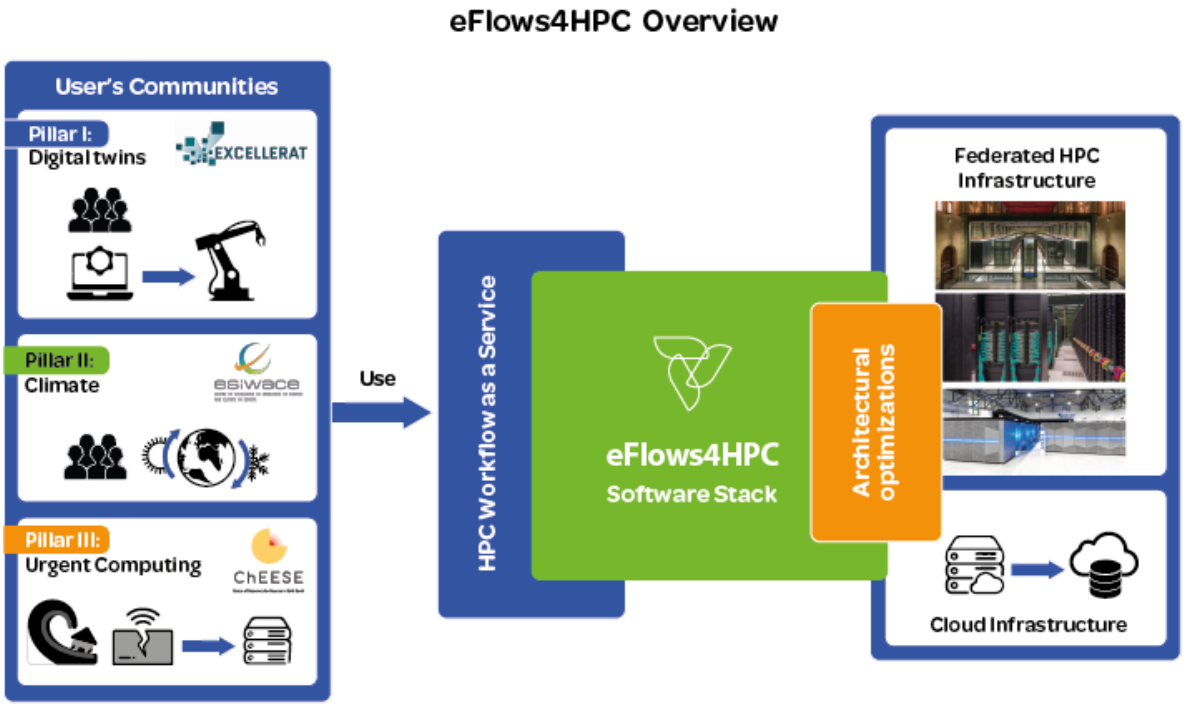
List of figures

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

Chapter 2

Software Stack

The 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.

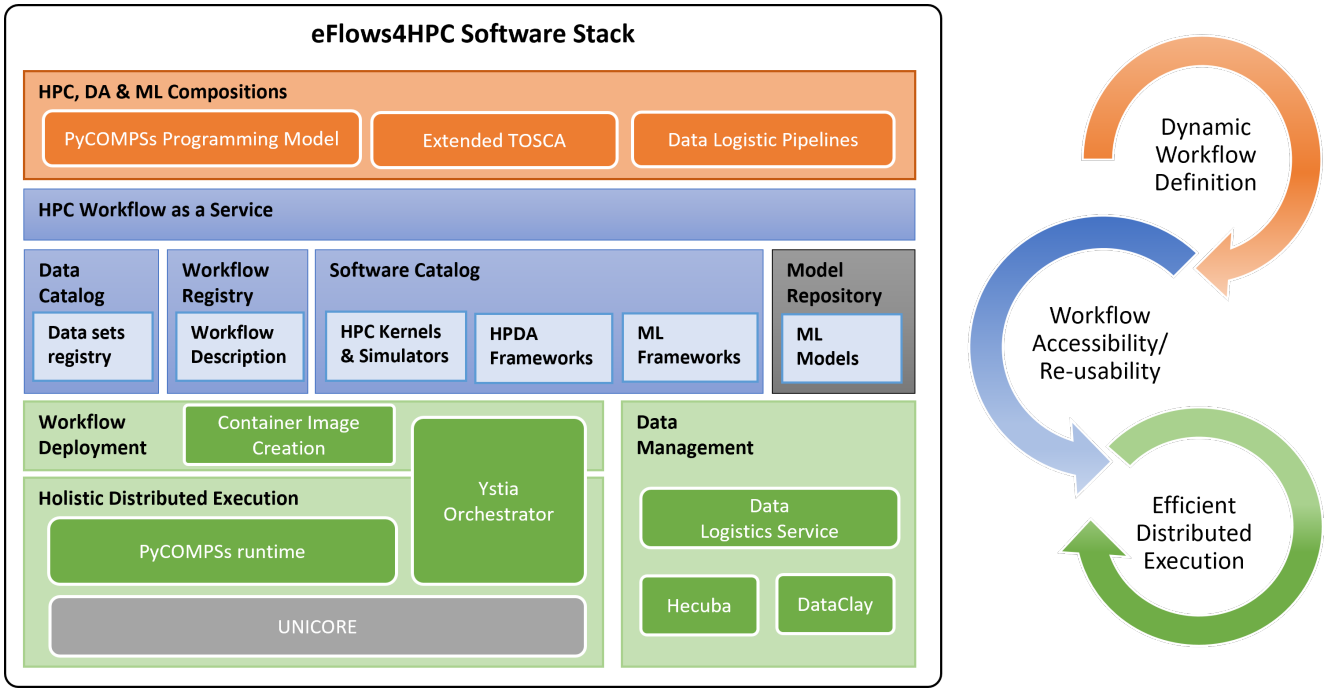


Figure 1: Software Stack release overview.

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

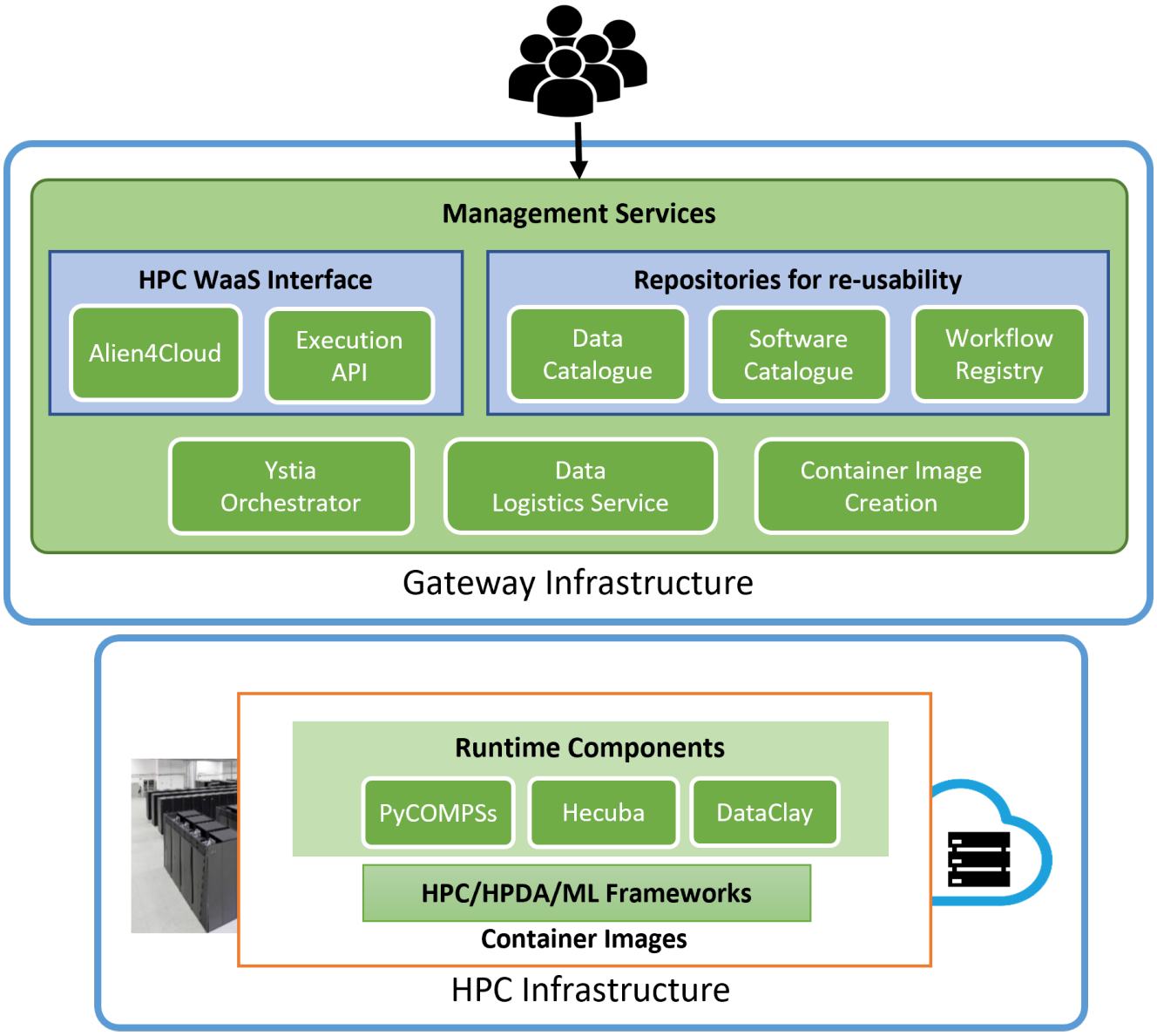


Figure 2: Deployment view of the different Software Stack components.

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 (DLS) is responsible for data movements part of the workflows developed in the project.

The service is based on Apache Airflow. The project specific extensions can be found in the project repository.

From the user perspective, the most important part of the service is the definition of data movements (pipelines). Some examples (e.g. minimal workflow) of these are provided in the dagrepo. The pipelines defined in this repository are automatically deployed to the production instances of DLS.

A good starting point for defining your own pipelines is the original documentation. Note that the pipelines are defined in the Python programming language and can execute shell scripts. This means that if the users already have their data movement solution based on scripts or Python programs, they can easily be migrated to the Data Logistics Service to obtain a running environment with monitoring, retires upon failure, etc.

There is an 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
```

2.1.6 Container Image Creation

This component allow to create HPC ready container images for eFlows4HPC platform for an specific workflow step and a target machine. Source code of this service can be found in this repository.

The following paragraph provide how to install and use this component

2.1.6.1 Requirements

This service requires to have Docker buildx system in the computer where running the service python > 3.7. Once, these tools have been installed, install the python modules described in requirements.txt file.

```
$ pip install -r requirements.txt
```

Finally, clone the workflow registry and software catalog repositories

```
$ git clone https://github.com/eflows4hpc/workflow-registry.git
$ git clone https://github.com/eflows4hpc/software-catalog.git
```

2.1.6.2 Installation and configuration

Once you have installed the requirements clone the Container Image Creation repository

```
$ git clone https://github.com/eflows4hpc/image_creation.git
```

Modify the image creation configuration, provinding the information for accessing the container registry and the loaction where the workflow registry or the software catalog has been donwloaded

```
$ cd image_creation
$ vim config/configuration.py
```

Finally, start the service with the following command

```
$ python3 builder_service.py
```

2.1.6.3 API

The Container Image Creation service offers a REST API to manage the creation of container images. The following paragraphs shows how this API works.

Trigger an image creation

This API endpoint allows the *end-user* to trigger the image creation with HTTP POST request. This request must include the description of the machine, indicating the system platform, processor architecture and the supported container engine. Optionally, it can also include the MPI version and GPU runtime version if the image requires access to MPI and GPU fabrics.

Request

```
`POST /build/`

{
  "machine": {
    "platform": "linux/amd64",
    "architecture": "rome",
    "container_engine": "singularity"
    "mpi": "openmpi@4"
    "gpu": "cuda@10" },
  "workflow":"minimal_workflow",
  "step_id" :"wordcount",
  "force": False
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "<creation_id>"
}
```

Check status of an image creation

This API endpoint allows the *end-user* to check the status of an image creation.

Request

```
GET /build/<creation_id>
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "< PENDING | STARTED | BUILDING | CONVERTING | FINISHED | FAILED >",
  "message": "< Error message in case of failure >",
  "image_id": "< Generated docker image id >",
  "filename": "< Generated singularity image filename >"
}
```

Download image

This API endpoint allows the *end-user* to download the created image.

Request

```
GET /images/download/<Generated singularity image filename>
```

Response

```
HTTP/1.1 200 OK
Content-Disposition: attachment
Content-Type: application/binary
```

2.1.6.4 Client

A simple BASH client has been implemented in `cic_cli`. This is the usage of this client.

```
cic_cli <user> <passwd> <image_creation_service_url> <"build"|"status"|"download"> <json_
↪file|build_id|image_name>
```

The following lines show an example of the different commands.

```
$ image_creation> ./cic_cli user pass https://bscgrid20.bsc.es build test_request.json
Response:
{"id":"f1f4699b-9048-4ecc-aff3-1c689b855adc"}

$ image_creation> ./cic_cli user pass https://bscgrid20.bsc.es status f1f4699b-9048-4ecc-aff3-
↪1c689b855adc
Response:
{"filename":"reduce_order_model_sandybridge.sif","image_id":"ghcr.io/eflows4hpc/reduce_order_
↪model_sandybridge","message":null,"status":"FINISHED"}

$ image_creation> ./cic_cli user pass https://bscgrid20.bsc.es download reduce_order_model_
↪sandybridge.sif
```

(continues on next page)

(continued from previous page)

```
--2022-05-24 16:01:28--  https://bscgrid20.bsc.es/image_creation/images/download/reduce_order_
↪model_sandybridge.sif
Resolving bscgrid20.bsc.es (bscgrid20.bsc.es)... 84.88.52.251
Connecting to bscgrid20.bsc.es (bscgrid20.bsc.es)|84.88.52.251|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2339000320 (2.2G) [application/octet-stream]
Saving to: ‘reduce_order_model_sandybridge.sif’

reduce_order_model_sandybridge.sif        0%[                          ]   4.35M   550KB/s    ␣
↪eta 79m 0s
```

2.1.7 Software Catalog

The Software Catalog is a git repository to store the description of the software to be used in computational HPC workflows using the eFlows4HPC methodology. The eFlows4HPC Software Catalog can be found in this repository.

2.1.7.1 Repository structure

Software descriptions have to be included inside this repository according to the following structure. The different software descriptions are located as a subfolder of the `packages` directory. This includes the installation description as a Spack package description and the *Software invocation description*.

```
software-catalog
  |- packages
  |    |- software_1
  |    |    |- package.py                 # Installation description following the Spack␣
↪package format
  |    |    |- invocation.json  # Description of the software invocation
  |    |       ...
  |    |- software_2
  |          ....
  |- cfg                                    # Spack configuration used by the Image Creation␣
↪Service
  |
  |- repo.yaml                           # Spack description of this repository
```

2.1.7.2 Including new software

To include new software in the repository, create a fork of the repository. Inside the packages folder create a new folder with the name of the software. This folder should contain the description of the new software including at least the Spack package description and *Software invocation description*.

Finally, create a create pull request with the branch of the newly added software. This pull request will be reviewed and merged to the repository.

2.1.8 Workflow Registry

The Workflow Registry is a git repository that stores the Workflow descriptions using the eFlows4HPC methodology. This description consist of at least the TOSCA description of the workflow, the code of the their different steps and their required software per step. The eFlows4HPC Workflow Registry can be found in this repository.

2.1.8.1 Repository structure

Workflow descriptions have to be included inside this repository according to the following structure. Each workflow description should contain a `tosca` folder with the TOSCA topology with the relationship of the PyCOMPSs executions and the required image creations for the different steps, data pipelines and HPC environments and one or several folders for PyCOMPSs application as workflow step.

```
workflow-registry
  |- workflow_1
  |    |- tosca
  |    |    |- types.yml          # TOSCA description of the different components involved in␣
↪the workflow
  |    |       ...
  |    |- step_1
  |    |    |- spack.yml          # Software requirements for this workflow step as a Spack␣
↪environment specification
  |    |    |- src                # PyCOMPSs code of the workflow step
  |    |       ...
  |    |- step_2
  |         ....
  |- workflow_2
  |   ...
```

2.1.8.2 Including new Workflows

To include new workflows in the repository, first create a new fork of the repository. Inside the forked repository, create a new directory with the name of your workflow. This directory should include the workflow description with a sub-folder for the TOSCA description and the different workflow steps. Each workflow step correspond to a PyCOMPSs code which must be executed in a HPC cluster. The description of the steps should include the software requirements as a Spack environment and the PyCOMPSs code.

Finally, create a pull request with the new workflow description. This pull request will be reviewed and included in the repository.

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 parallel and distributed applications. This implementation hints the runtime scheduler to assign tasks on the same nodes where dataClay stores the needed 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 and C/C++ applications that store data in memory or Apache Cassandra.

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 Hecuba-managed data to the nodes that contain 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/eflows4hpc/hecuba
2. Manual: https://github.com/eflows4hpc/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.

Ophidia can also be installed through the Spack package manager.

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. During the last months we have implemented a new version based on PyCOMPSs which is described later in this document.

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.

2.3.5.3 Parsoda-PyCOMPSs integration

ParSoDA has been ported to Python to support the use of the Python libraries ecosystem. ParSoDA has been extended to support multiple execution runtimes. Specifically, according to the bridge design pattern, we defined the ParsodaDriver interface (i.e., the implementor of the bridge pattern) that allows a developer to implement adapters for different execution systems. A valid instance of ParsodaDriver must invoke some function that exploits some parallel pattern, such as Map, Filter, ReduceByKey and SortByKey. The SocialDataApp class is the abstraction of the bridge pattern and is designed to use these parallel patterns efficiently for running ParSoDA applications. It is worth noting that the execution flow of an application remains unchanged even by changing the execution runtime, which makes the porting of a ParSoDA application to new execution runtimes.

In particular, we included four execution drivers into ParSoDA-Python:

- ParsodaSingleCoreDriver, a driver that implements parallel patterns as simple sequential algorithms to be run on a single core, on the local machine. It is useful for verifying the correctness of a new ParSoDA Driver during its construction.
- ParsodaMultiCoreDriver, which runs the application in parallel on multiple cores, on the local machine, using Python's thread pools.
- ParsodaPySparkDriver, which runs the application on a Spark cluster. It is based on the PySpark library and requires the initialization of a SparkConf object.
- ParsodaPyCompssDriver, which runs the application on a COMPSs cluster. It relies on the PyCOMPSs binding to gain access to the COMPSs runtime.

Source Code

The code of ParSoDA-Python library is available in this repository.

Installation and use

The ParSoDA library requires Python 3.8 or above. To install the current version of ParSoDA on a Python environment you just need to put the ParSoDA package into some directory, then it can be used in a new application that can be run on the local environment. To use ParSoDA on top of PyCOMPSs or PySpark, you need to install and correctly configure one or both these two environments. At that point the application can be run through the ParsodaPyCompssDriver or the ParsodaPySparkDriver classes. The current experimental version of ParSoDA comes with two example applications, Trajectory Mining and Emoji Polarization, which requires the following python packages to be installed:

```
emoji==1.7.0
fastkml==0.12
geopy==2.2.0
shapely==1.8.1
```

The ParSoDA package contains a file "requirements.txt" which can be used with pip to install the application requirements, executing the following command in the root directory of ParSoDA:

```
python3 -m pip install -r requirements.txt
```

The following example shows the Trajectory Mining application written with ParSoDA on Python:

```
driver = ParsodaPyCompssDriver()

app = SocialDataApp("Trajectory Mining", driver, num_partitions=args.partitions, chunk_
↪size=args.chunk_size)

app.set_crawlers([
    LocalFileCrawler('/root/dataset/FlickrRome2017.json', FlickrParser())
    LocalFileCrawler('/root/dataset/TwitterRome2017.json', TwitterParser())
])
app.set_filters([
    IsInRoI("./resources/input/RomeRoIs.kml")
])
app.set_mapper(FindPoI("./resources/input/RomeRoIs.kml"))
app.set_secondary_sort_key(lambda x: x[0])
app.set_reducer(ReduceByTrajectories(3))
app.set_analyzer(GapBIDE(1, 0, 10))
app.set_visualizer(
    SortGapBIDE(
        "./resources/output/trajectory_mining.txt",
        'support',
        mode='descending',
        min_length=3
    )
)

app.execute()
```

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 modelling 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 is developed using dedicated frameworks for the specific problem to solve. Nevertheless, 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.

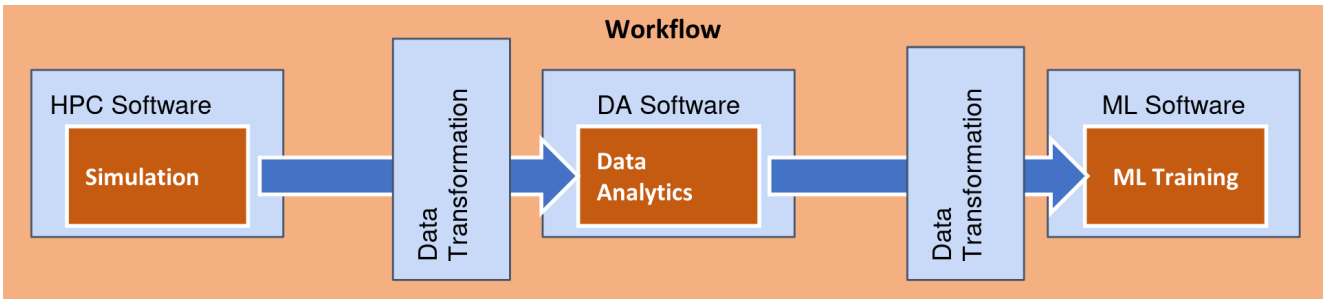


Figure 3: Overview of the programming interfaces to integrate HPC/DA/ML.

As we can see in Figure 3, when we have to include computations that are using different frameworks in the same application, developers have to deal with the execution of the different frameworks and how to convert the data generated by one framework to the model required by the other framework. 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. These 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 needs to be applied to a parameter of the invocation to be compatible with the input of expected execution.

Code 1 shows an overview of how the programming interfaces are used to implement a workflow. These decorators are declared on top of a Python function which represents the execution of the software we want to integrate into the workflow. Then, the execution of the software can be included in a workflow as a call of a standard Python method, and the runtime will convert this call to the remote invocation of the described software and the implicit data transformations.

Code 1: Overview of a workflow implementation using the programming interfaces.

```
@data_tranformation("input_data", "transformation method")
@software("invocation description")
def data_analytics (input_data, result):
   pass

#Worflow
simulation(input_cfg, sim_out)
data_analytics(sim_out, analysis_result)
ml_training(analysis_result, ml_model)
```

During the first project iteration, we defined the software invocation descriptions and extended the PyCOMPSs programming model and runtime. In the second iteration of the eFlows4HPC framework, we have included the definition of the data transformations and their implementations.

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 of the workflow.

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. When a function with the *@software* decorator is called, an (external) program is executed keeping the configuration defined in its configuration file untouched. The goal of this decorator is to support the execution of external programs in a workflow, from simple binary executable to complex MPI applications.

3.1.2 Configuration File

Configuration files can contain different key-values depending on the user's needs. We use JSON format for the configuration files and the next table provides details of some of the supported keys:

Key	Description
execution	(Mandatory) Contains all the software execution details such as "type", "binary", "args", etc..
execution.type	(Mandatory) Type of the software invocation. Supported values are 'task', 'workflow', 'mpi', 'binary', 'mpmd_mpi', 'multinode', 'http', and 'compss'.
parameters	A dictionary containing *task* parameters.
prolog	A dictionary containing *prolog* parameters.
epilog	A dictionary containing *epilog* parameters.
constraints	Parameters regarding constraints of the software execution.
container	Container parameters if the external software is meant to be executed inside a container.

Details of the configuration of the software execution can be defined in the value of the "execution" key. There the user can define the "type" of the execution and other necessary configuration parameters the *software* requires.

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 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")
def mpi_execution():
     pass

def main():
    mpi_execution()
```

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

```
{
  "execution" : {
    "type":"mpi",
    "runner": "mpirun",
    "binary":"my_mpi_app.bin",
    "processes": 8,
    }
}
```

It is also possible to refer to task parameters and environment variables from the configuration file. Properties such as *working_dir* and *args* ('args' strings are command line arguments to be passed to the 'binary') can contain this type 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 'args' strings respectively. And the number of MPI processes are obtained from the environment variable "MPI_PROCS". Moreover, epilog and prolog definitions, as well as 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 and invocation:

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

@software(config_file="mpi_w_args.json")
def mpi_with_args(work_dir, param_d, out_tgz):
     pass

def main():
working_dir = "/tmp/mpi_working_dir/"
arg_value = 1001
mpi_with_args(working_dir, ar_value, "output.tgz")
```

Configuration file ("mpi_w_args.json"):

```
{
  "execution" : {
    "type":"mpi",
    "runner": "mpirun",
    "processes" : "$MPI_PROCS",
    "binary":"my_binary.bin",
    "working_dir": "{{work_dir}}",
    "args": "-d {{param_d}}"
  },
  "parameters" : {
    "param_d": "IN",
    "work_dir": "DIRECTORY_OUT",
    "out_tgz": "FILE_OUT"
  }
  "prolog": {
    "binary": "mkdir",
    "args": "{{work_dir}}"
  },
  "epilog": {
    "binary":"tar",
    "args":"zcvf {{out_tgz}} {{work_dir}}"
  },
  "constraints":{
    "computing_units": 2
  }

}
```

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 directory recursively 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")
def task_container(in_directory, expression):
     pass
```

(continues on next page)

(continued from previous page)

```
def main():
   task_container('/tmp/my_logs/', 'Error')
```

Configuration file ("container_config.json"):

```
{
  "execution" : {
    "type":"binary",
    "binary": "grep",
    "args": "{{expression}} {{in_directory}} -ir"
    },
  "parameters":{
    "in_directory": "DIRECTORY_IN",
    "expression": "IN"
  },
  "container":{
    "engine": "DOCKER",
    "image": "ubuntu:20.04"
  }
}
```

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

3.2 Data Transformation

The *@data_transformation* (or just *@dt*) decorator is used for the execution of a data transformation function that should be applied to a given 'PyCOMPSs task' parameter. By specifying the parameter name and a Python function, users can assure that the parameter will go through a transformation process by the given function before the task execution. The result of the data transformation function will be used in the task instead of the initial value of the parameter.

The Data transformation decorator has a simple order for the definition. The first argument of the decorator is a string name of the parameter we want to transform. The second argument is the data transformation function (NOT as a string, but actual reference) that expects at least one input to which the transformation will be applied to. If the transformation function needs more parameters, they can be added to the *@dt* definition as `'kwargs'`. Moreover, if the user wants to use a workflow as a data transformation function and thus avoid the intermediate task creation, PyCOMPSs provides an optional keyword argument `is_workflow` to do so (by default *False*). This gives the flexibility of importing workflows from different libraries.

Code 2: Arguments Data Transformation decorator.

```
@dt("<parameter_name>", "<dt_function>", "<is_workflow_value>","<kwargs_of_dt_function>")
@software("example.json")
def task_func(...):
    ...
```

Important: Please note that data transformation definitions should be on top of the *@software* and/or *@task* decorator.

Adding data transformation on top of the `@software` or `@task` decorator allows the PyCOMPSs Runtime generate an intermediate task. This task method applies the given DT to the given input and the output is sent to the *original* task as the input. The following code snippet is an example of basic usage of the *@dt* decorator:

Code 3: Basic Data Transformation code example.

```
import numpy as np
from pycompss.api.data_transformation import dt
from pycompss.api.software import software
from pycompss.api.api import compss_wait_on


@software(config_file="simulation.json")
def simulation():
    ...
    return a

def reshape(A, new_x, new_y):
    return A.reshape((new_x, new_y))

@dt("input_data", reshape, new_x=10, new_y=100)
@software("data_analysis.json")
def data_analysis(input_data):
    ...
    return result

def main():
    A = simulation()
    result = data_analysis(A)
    result = compss_wait_on(result)
    print(result)
```

As we can see in the example, the result of the "simulation" function is assigned to the parameter A. However, this data is formatted in columns where the input of "data_analysis" must be shaped in blocks. Thus, before the task execution, parameter A will go through the "reshape" function where "new_x" and "new_y" will be 10 and 100 respectively. Once the execution of the Data Transformation task is finished, the transformed data will be passed to the "data_analysis" as input in the required format.

PyCOMPSs also supports inter-types data transformations which allows the conversion of the input data to another object type. For example, if the user wants to use a object's serialized file as an input for a task, but the task function expects the object itself, then `@dt` can take care of it. So far PyCOMPSs supports this kind of data transformations only for the `FILE`, `OBJECT` and `COLLECTION` types.

For the cases where type conversions happen, there are some mandatory and optional parameters:

Parameter	Description
target	(Mandatory) Name of the input parameter that DT will be applied to.
function	(Mandatory) The data transformation function.
type	(Mandatory) Type of the DT (e.g. FILE_TO_OBJECT)
destination	If the output of the DT is a file, then output file name can be specified as "destination".
size	(Mandatory only if the output of the DT is a COLLECTION) Size of the output COLLECTION.

In the example below we can see a code snippet where the Data Transformation task deserializes a file and assigns it to the input parameter. That is why its *type* is `FILE_TO_OBJECT`:

Code 4: Data Transformation with type conversion.

```
from pycompss.api.data_transformation import *
```

(continues on next page)

(continued from previous page)

```
from pycompss.api.task import task
from pycompss.api.parameter import FILE_OUT
from pycompss.api.api import compss_wait_on

@task(result_file=FILE_OUT)
def generate(result_file):
    ...

def deserialize(some_file):
    # deserialize the file
    ...
    return deserialized_object

@dt(target="input", function=deserialize, type=FILE_TO_OBJECT)
@software("example.json")
def simulation(input):
    # 'input' is deserialized object from its initial file path
    ...

def main(self):
    some_file = "src/some_file"
    generate(some_file)
    result = simulation(some_file)
    result = compss_wait_on(result)
```

It is possible to define multiple data transformations for the same parameter, as well as for multiple parameters of the same task. In both cases each data transformation with "is_workflow=False" will take place in a different task (in the order of the definition from top to bottom):

Code 5: Multiple data transformations on top of a @software function.

```
import dislib as ds
from pycompss.api.data_transformation import *
from pycompss.api.task import task
from pycompss.api.software import software
from pycompss.api.api import compss_wait_on

def load_w_dislib(file_path, blocK_size=10):
    obj = ds.load_txt_file(file_path, block_size)
    ...
    return obj

def extract_columns(input):
    # modifies input
    ...
    return input

def scale_by_x(input, rate=100):
    # modifies input
    ...
    return input

@dt(target="A", function=load_w_dislib, type=FILE_TO_OBJECT, is_workflow=True)
@dt("A", extract_columns, is_workflow=False)
@dt(target="B", function=load_w_dislib, type=FILE_TO_OBJECT, is_workflow=True)
```

(continues on next page)

(continued from previous page)

```
@dt("B", scale_by_x, rate=5)
@software("workflow.json")
def run_simulation(A, B):
    # A and B are both loaded from text files using "dislib" and modified
    ...

def main():
    first_file = "src/file_A"
    second_file = "src/file_B"
    run_simulation(first_file, second_file)
    ...
```

For more detailed information about the @dt 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, who is in charge of developing and deploying the workflow in the computing infrastructure, and then the users' communities, which are usually scientist who want to execute the workflow and collect their results to advance in their scientific goals.

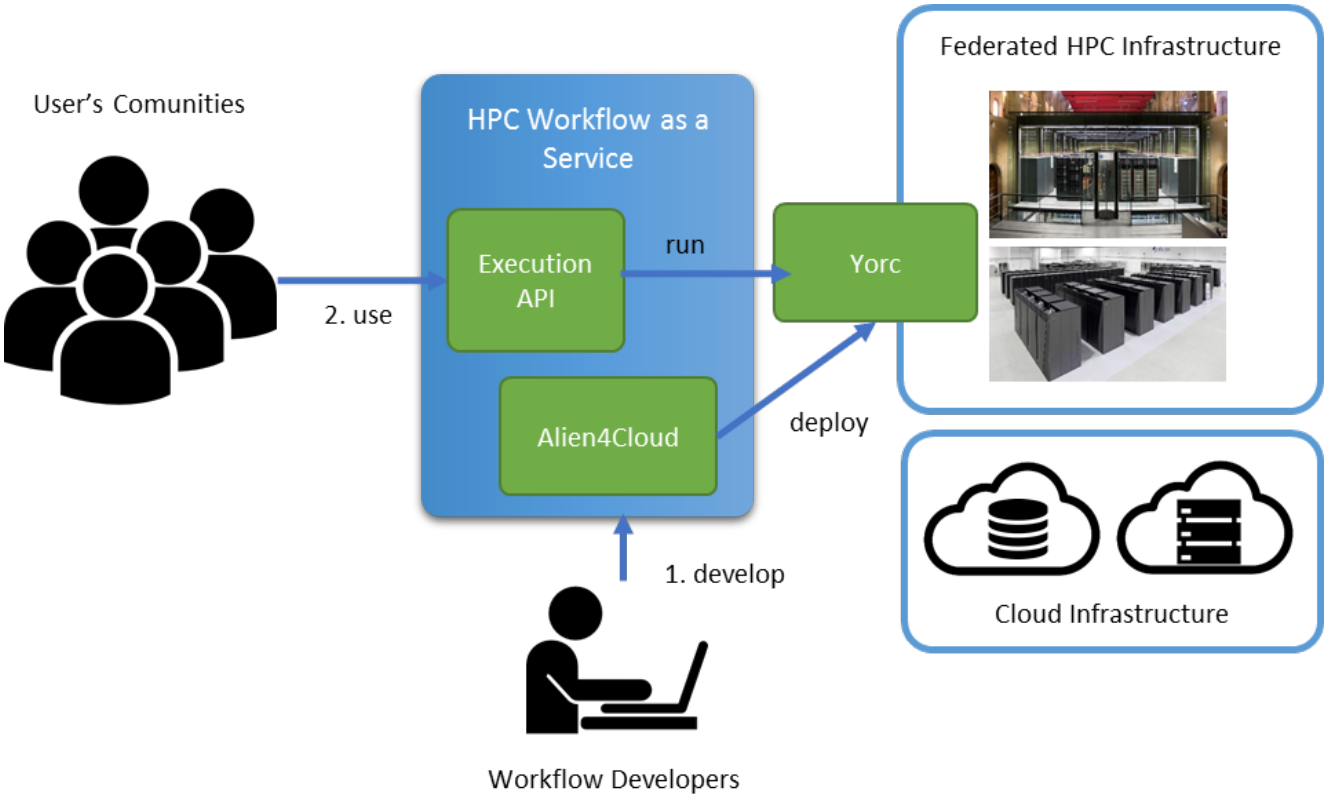


Figure 4: HPC Workfow as a Service overview

Figure 4 shows how these two roles interact 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 achieved by 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.

Two instances of Alien4Cloud and Yorc are deployed for the eFlows4HPC project. One is hosted on Juelich cloud, this instance is used for testing and integration of the software stack. The second instance is hosted on BSC cloud and is used to develop pillars use cases. Ask to the project (eflows4hpc@bsc.es) 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 one of the instances deployed for the eFlows4HPC project 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 5.

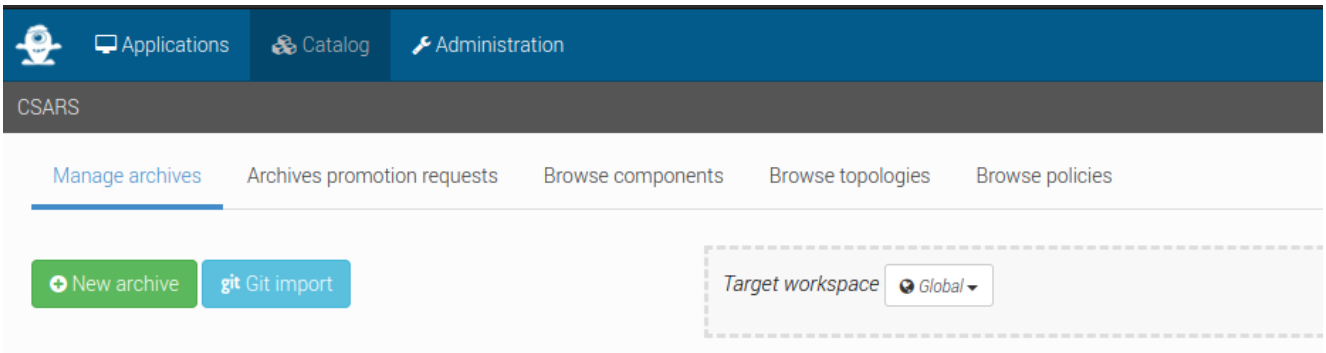


Figure 5: Click on `Git import` to add components

You should have at least the three repositories defined as shown in Figure 6:

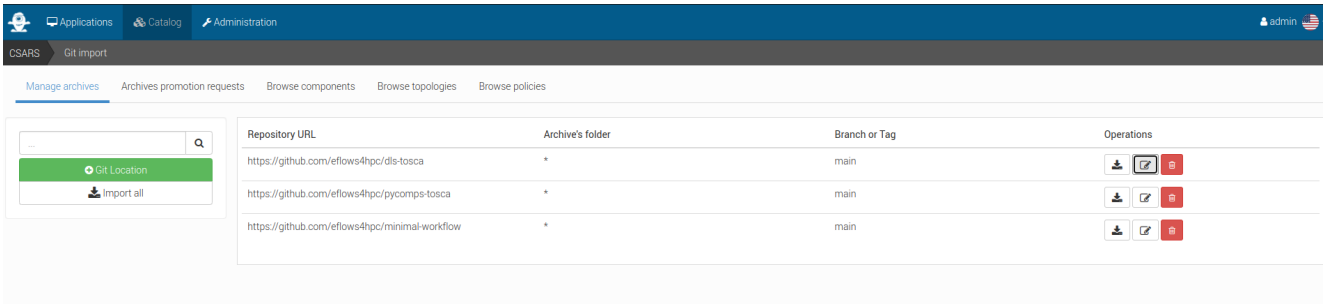


Figure 6: Click on `Git location` to define imports from a git repository

Click on `Git location` to define imports from a git repository as shown in Figure 7

Once done you can click on `Import all`.

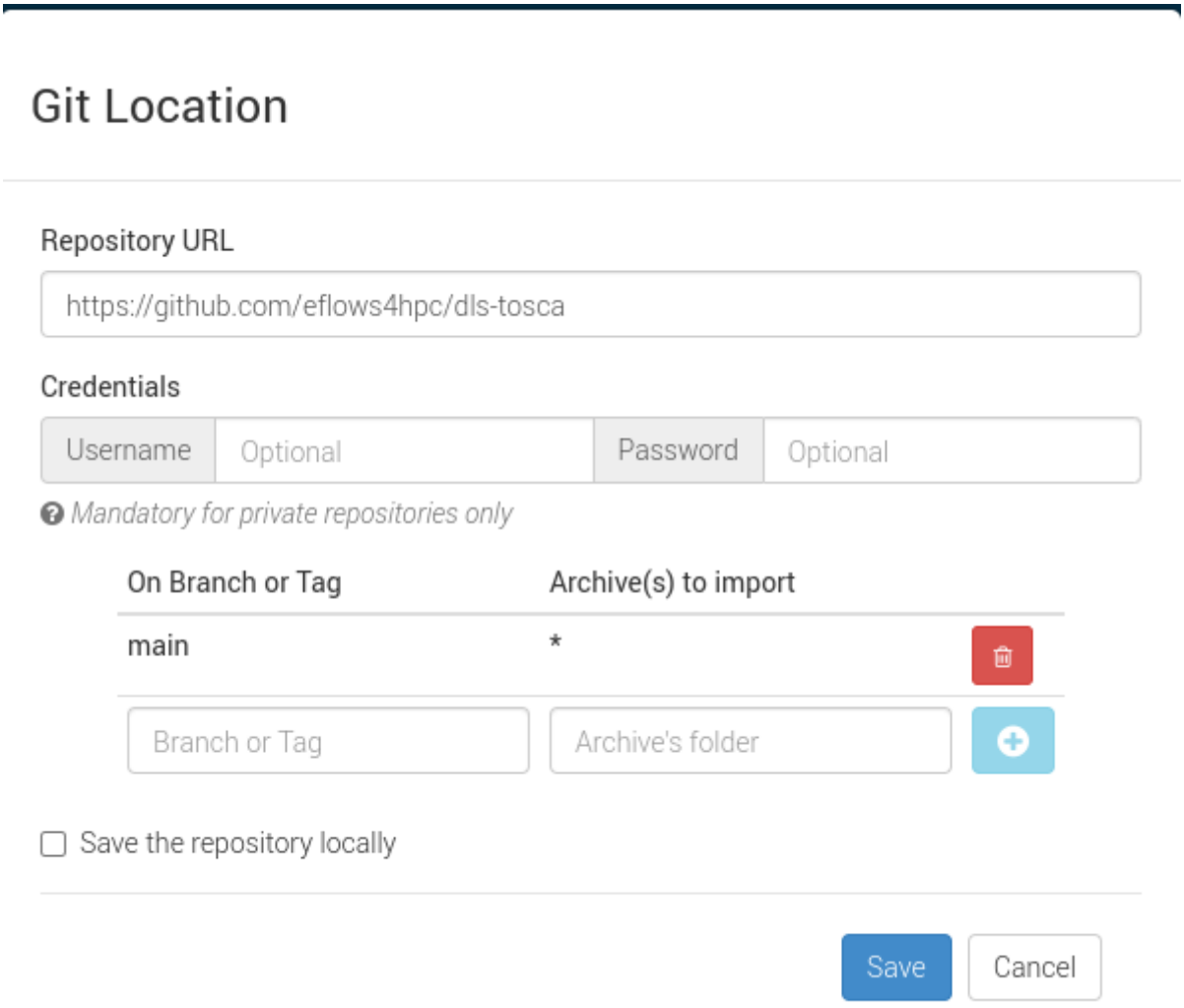


Figure 7: Alien4Cloud setup a catalog git repository

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 8.

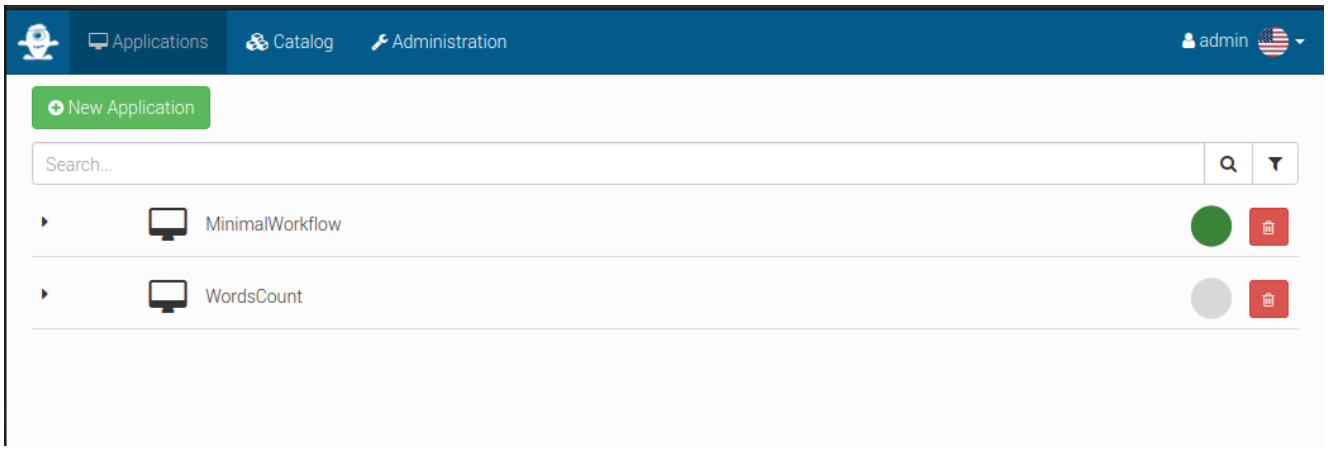


Figure 8: Manage applications in Alien4Cloud

Then create a new application based on the minimal workflow template as shown in Figure 9

Edit the topology to fit your needs as shown in Figure 10.

Then click on `Deploy` to deploy the application as shown in Figure 11.

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 12. The tag value should be a comma-separated list of workflow names that could be called through the HPCWaaS API. In the minimal workflow example, this tag value should be `exec_job`.

4.1.4 eFlows4HPC TOSCA Components

eFlows4HPC uses TOSCA to describe the high-level execution lifecycle of a workflow, enabling the orchestration of tasks with diverse nature.

To support eFlows4HPC use cases, we have defined the following TOSCA components:

- Image Creation Service TOSCA component to build container images.
- Data Logistics Service TOSCA components to manage data movement.
- PyCOMPSs execution TOSCA component to launch and monitor PyCOMPSs jobs.
- Environment TOSCA component to hold properties of an HPC cluster.

In following sections you will find a detailed description of each of these components and their configurable properties.

Section *ROM Pillar I topology template* describes how these components are assembled together in a TOSCA topology template to implement the ROM Pillar I use case. More specifically you can refer to Code 21 to see how properties of the TOSCA components are used in this particular context.

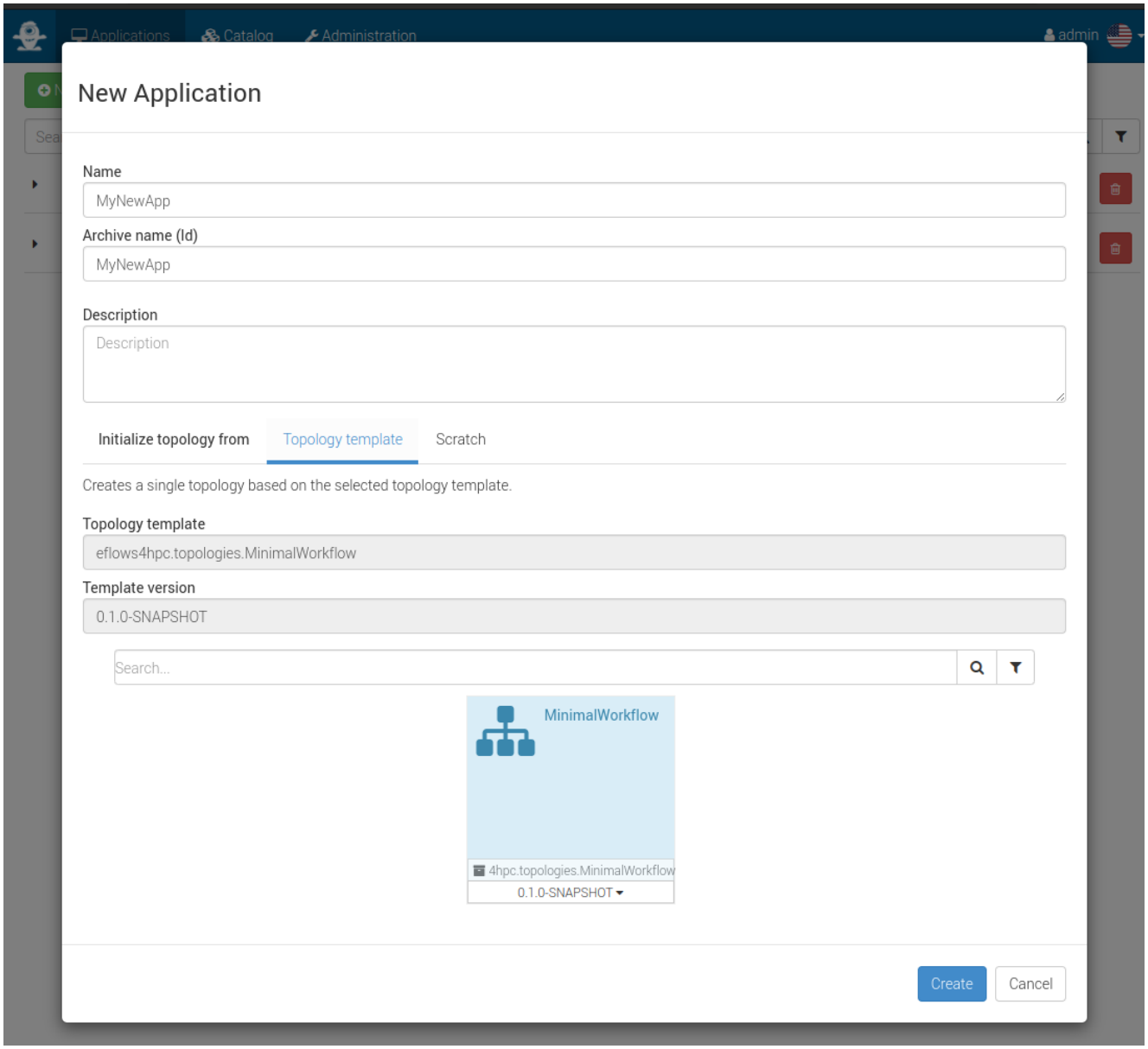


Figure 9: Alien4Cloud create a template based application

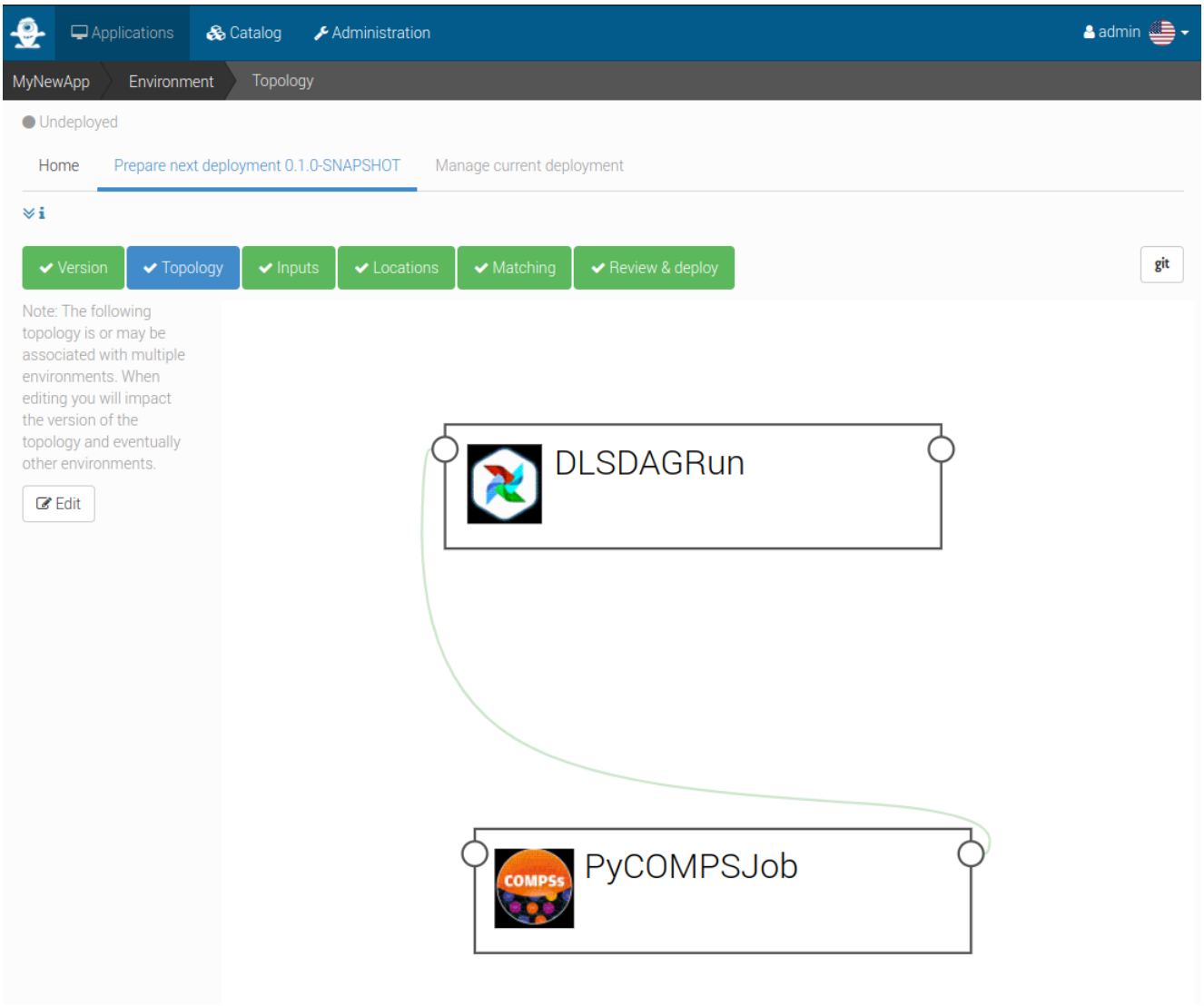


Figure 10: Alien4Cloud minimal workflow topology

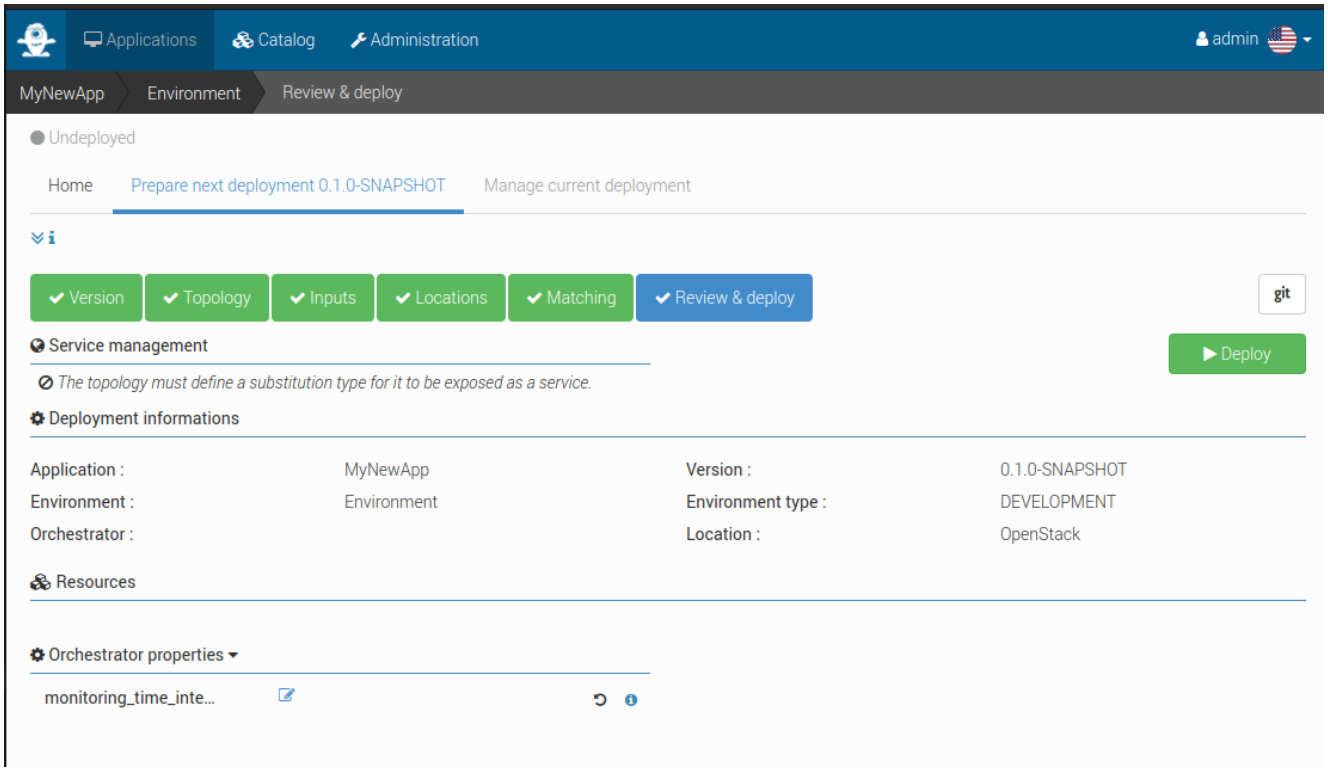


Figure 11: Alien4Cloud deploy an application

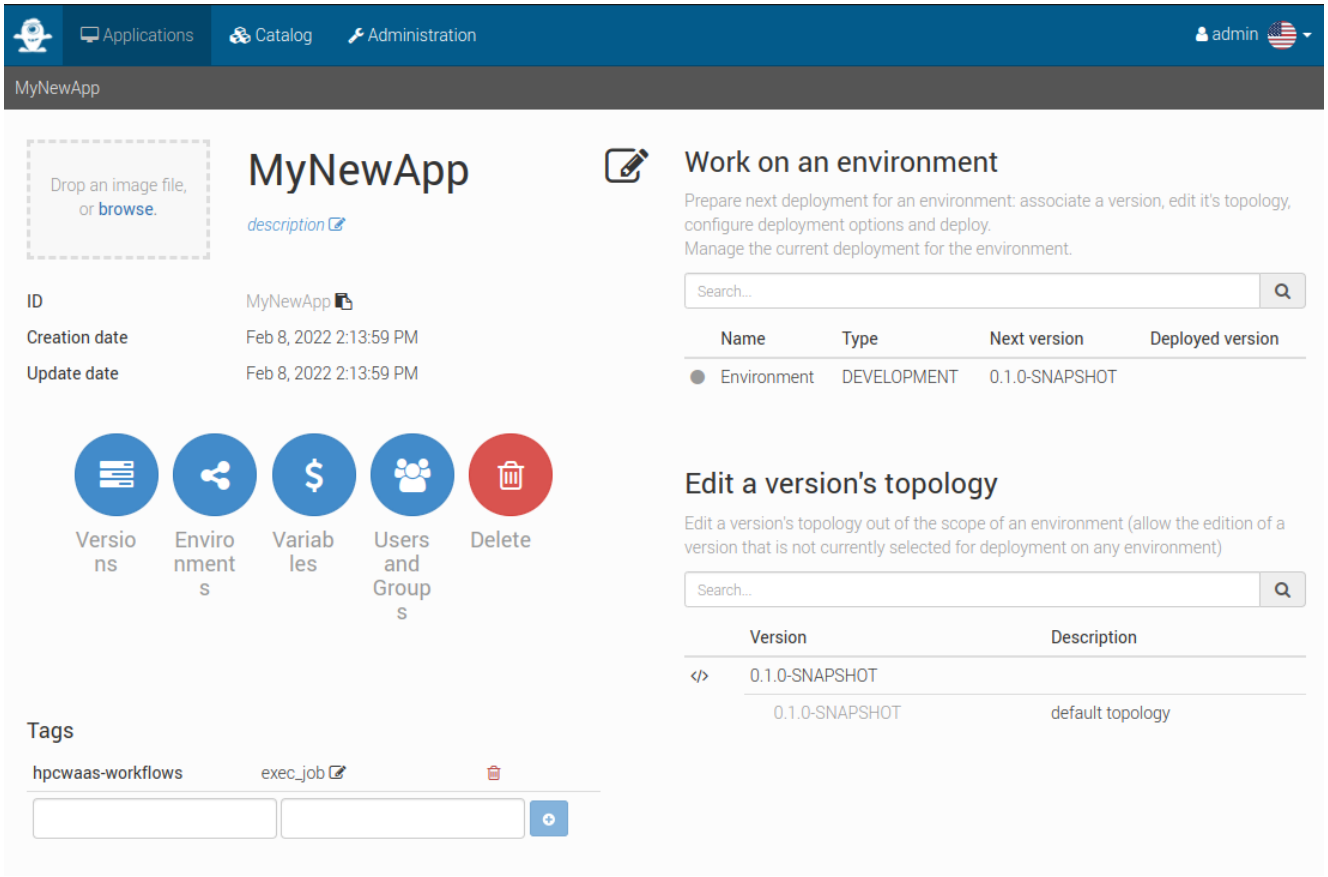


Figure 12: Alien4Cloud add tags to an application

4.1.4.1 Image Creation Service TOSCA component

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

This components interacts with the Image Creation Service RESTful API to trigger and monitor the creation of container images for specific hardware architectures.

Code 6 is a simplified (for the sake of clarity) version of the TOSCA type definition of the Image Creation Service that shows the configurable properties that can be set for this component.

Code 6: Extract of the TOSCA definition for Image Creation Service

```
data_types:
  imagecreation.ansible.datatypes.Machine:
    derived_from: tosca.datatypes.Root
    properties:
      platform:
        type: string
        required: true
      architecture:
        type: string
        required: true
      container_engine:
        type: string
        required: true

node_types:
  imagecreation.ansible.nodes.ImageCreation:
    derived_from: org.alien4cloud.nodes.Job
    properties:
      service_url:
        type: string
        required: true
      insecure_tls:
        type: boolean
        required: false
        default: false
      username:
        type: string
        required: true
      password:
        type: string
        required: true
      machine:
        type: imagecreation.ansible.datatypes.Machine
        required: true
      workflow:
        type: string
        required: true
      step_id:
        type: string
        required: true
      force:
        type: boolean
        required: false
        default: false
      debug:
```

(continues on next page)

(continued from previous page)

```
        type: boolean
        description: Do not redact sensible information on logs
        default: false
      run_in_standard_mode:
        type: boolean
        required: false
        default: true
```

- The `imagecreation.ansible.datatypes.Machine` data type allows to define the build-specific properties for the container image to be created
  - `platform` is the expected operating system for instance: `linux/amd64`
  - `architecture` is the expected processor architecture for instance `sandybridge`
  - `container_engine` is the expected container execution engine typically `docker` or `singularity`
- `workflow` property is the name of the workflow within the workflow-registry github repository
- `step_id` property is the name of the sub step of the given workflow in the workflow registry
- `service_url`, `insecure_tls`, `username` and `password` are properties used to connect to the Image Creation Service
- `force` property allows to force the re-creation of the image even if an existing image with the same configuration already exists
- `debug` will print additional information in Alien4Cloud's logs, some sensible information like passwords could be reveled in these logs, this should be used for debug purpose only
- `run_in_standard_mode` this property controls in which TOSCA workflows this component interacts with the Image Creation Service by setting this property to `true` this components will be run in the `standard` mode meaning at the application deployment time. This is an advanced feature and the default value should fit most of the needs.

4.1.4.2 Data Logistics Service TOSCA components

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

These components interact with the Airflow RESTful API to trigger and monitor the execution of Airflow pipelines.

These components leverage TOSCA inheritance to both allow to run generic Airflow pipelines and to make it easier to create TOSCA components with properties specific to a given pipeline. `dls.ansible.nodes.DLSDAGRun` is the parent of all others DLS TOSCA components. It allows to run any DLS pipeline with an arbitrary configuration. Other DLS TOSCA components extend it by adding or overriding some properties.

Code 7 is a simplified version of the TOSCA type definition of the Data Logistics Service that shows the configurable properties that can be set for these components. We removed components that are note used in the Pillar I use case.

Code 7: Extract of the TOSCA definition for DLS

```
dls.ansible.nodes.DLSDAGRun:
  derived_from: org.alien4cloud.nodes.Job
  properties:
    dls_api_url:
      type: string
      required: false
    dls_api_username:
      type: string
      required: true
    dls_api_password:
      type: string
      required: true
    dag_id:
      type: string
```

(continues on next page)

(continued from previous page)

```
      required: true
    extra_conf:
      type: map
      required: false
      entry_schema:
        description: map of key/value to pass to the dag as inputs
        type: string
    debug:
      type: boolean
      description: Do not redact sensible information on logs
      default: false
    user_id:
      type: string
      description: User id to use for authentication may be replaced with workflow input
      required: false
      default: ""
    vault_id:
      type: string
      description: User id to use for authentication may be replaced with workflow input
      required: false
      default: ""
    run_in_standard_mode:
      type: boolean
      required: false
      default: false
  requirements:
    - environment:
        capability: eflows4hpc.env.capabilities.ExecutionEnvironment
        relationship: tosca.relationships.DependsOn
        occurrences: [ 0, UNBOUNDED ]

dls.ansible.nodes.HTTP2SSH:
  derived_from: dls.ansible.nodes.DLSDAGRun
  properties:
    dag_id:
      type: string
      required: true
      default: plainhttp2ssh
    url:
      type: string
      description: URL of the file to transfer
      required: false
    force:
      type: boolean
      description: Force transfer of data even if target file already exists
      required: false
      default: true
    target_host:
      type: string
      description: the remote host
      required: false
    target_path:
      type: string
      description: path of the file on the remote host
      required: false
    input_name_for_url:
```

(continues on next page)

(continued from previous page)

```
      type: string
      description: >
        Name of the workflow input to use to retrieve the URL.
        If an input with this name exists for the workflow, it overrides the url property.
      required: true
      default : "url"
    input_name_for_target_path:
      type: string
      description: >
        Name of the workflow input to use to retrieve the target path.
        If an input with this name exists for the workflow, it overrides the target_path␣
↪property.
      required: true
      default : "target_path"

dls.ansible.nodes.DLSDAGStageInData:
  derived_from: dls.ansible.nodes.DLSDAGRun
    properties:
      oid:
        type: string
        description: Transferred Object ID
        required: false
      target_host:
        type: string
        description: the remote host
        required: false
      target_path:
        type: string
        description: path of the file on the remote host
        required: false
      input_name_for_oid:
        type: string
        description:
        required: true
        default : "oid"
      input_name_for_target_path:
        type: string
        description:
        required: true
        default : "target_path"

dls.ansible.nodes.DLSDAGStageOutData:
  derived_from: dls.ansible.nodes.DLSDAGRun
  properties:
    mid:
      type: string
      description: Uploaded Metadata ID
      required: false
    target_host:
      type: string
      description: the remote host
      required: false
    source_path:
      type: string
      description: path of the file on the remote host
      required: false
```

(continues on next page)

(continued from previous page)

```
    register:
      type: boolean
      description: Should the record created in b2share be registered with data cat
      required: false
      default: false
    input_name_for_mid:
      type: string
      required: true
      default: mid
    input_name_for_source_path:
      type: string
      required: true
      default: source_path
    input_name_for_register:
      type: string
      required: true
      default: register

dls.ansible.nodes.DLSDAGImageTransfer:
  derived_from: dls.ansible.nodes.DLSDAGRun
  properties:
    image_id:
      type: string
      description: The image id to transfer
      required: false
    target_host:
      type: string
      description: the remote host
      required: false
    target_path:
      type: string
      description: path of the file on the remote host
      required: false
    run_in_standard_mode:
      type: boolean
      required: false
      default: true
```

- `dls.ansible.nodes.DLSDAGRun` is the parent TOSCA component with the following properties:
  - `dls_api_url`, `dls_api_username` and `dls_api_password` are used to connect to the Airflow REST API.
    - `dls_api_url` could be overridden by the `dls_api_url` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together
    - `dls_api_username` and `dls_api_password` can be provided as plain text for testing purpose but the recommended way to provide it is to use the `get_secret` TOSCA function as shown in Code 21
  - `dag_id` is the unique identifier of the DLS pipeline to run
  - `extra_conf` is a map of key/value properties to be used as input parameters for the DLS pipeline
  - `debug` will print additional information in Alien4Cloud's logs, some sensible information like passwords could be reveled in these logs, this should be used for debug purpose only
  - `user_id` and `vault_id` are credentials to be used connect to the HPC cluster for data transfer
  - `run_in_standard_mode` this property controls in which TOSCA workflows this component interacts with the DLS by setting this property to `true` this components will be run in the `standard` mode meaning at the application deployment time. This is an advanced feature and the default value should fit most of the needs and it is overridden in derived TOSCA components if needed.
- `dls.ansible.nodes.HTTP2SSH` is a TOSCA component that allows to trigger a pipeline that will download a file and copy it to a cluster through SSH

  - `dag_id` overrides the pipeline identifier to `plainhttp2ssh`
  - `url` is the URL of the file to be downloaded
  - `force` forces transfer of data even if target file already exists
  - `target_host` the remote host to copy file on. This could be overridden by the `cluster_login_host` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together.
  - `input_name_for_url` is the name of the workflow input to use to retrieve the URL. If an input with this name exists for the workflow, it overrides the url property. The default value is `url`.
  - `input_name_for_target_path` is the name of the workflow input to use to retrieve the target path. If an input with this name exists for the workflow, it overrides the target_path property. The default value is `target_path`.
- `dls.ansible.nodes.DLSDAGStageInData` interacts with the DLS pipeline that download data from the data catalogu and copy it to the HPC cluster through SSH
  - `oid` is the Obejct ID of the file in the data catalogue
  - `target_host` the remote host to copy data to. This could be overridden by the `cluster_login_host` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together.
  - `target_path` is the path of a directory to store the file on the remote host
  - `input_name_for_oid` is the name of the workflow input to use to retrieve the OID. If an input with this name exists for the workflow, it overrides the oid property. The default value is `oid`.
  - `input_name_for_target_path` is the name of the workflow input to use to retrieve the target path. If an input with this name exists for the workflow, it overrides the target_path property. The default value is `target_path`.
- `dls.ansible.nodes.DLSDAGStageOutData` interacts with the DLS pipeline that copy data from the HPC cluster through SSH and upload it to the data catalogue
  - `mid` is the Metadata ID of the file in the data catalogue
  - `target_host` the remote host to copy data from. This could be overridden by the `cluster_login_host` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together.
  - `source_path` is the path of the file on the remote host
  - `register` controls if the record created in b2share should be registered within the data catalogue
  - `input_name_for_mid` is the name of the workflow input to use to retrieve the MID. If an input with this name exists for the workflow, it overrides the mid property. The default value is `mid`.
  - `input_name_for_source_path` is the name of the workflow input to use to retrieve the source path. If an input with this name exists for the workflow, it overrides the source_path property. The default value is `source_path`.
  - `input_name_for_register` is the name of the workflow input to use to retrieve the register flag. If an input with this name exists for the workflow, it overrides the register property. The default value is `register`.
- `dls.ansible.nodes.DLSDAGImageTransfer`:
  - `image_id` is the identifier of the container image to transfer from the Image Creation Service. If this component is linked to an Image Creation Service component then this id is automatically retrieved from the image creation execution.
  - `target_host` the remote host to copy the container image to. This could be overridden by the `cluster_login_host` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together.
  - `target_path` is the path of the container image on the remote host
  - `run_in_standard_mode` container image creation is typically designed to be run at application deployment time so this property is overridden to run at this stage.

4.1.4.3 PyCOMPSs TOSCA component

The source code of this component is available in the pycompss-yorc-plugin github repository in the eFlows4HPC organization.

This component is different from the above ones as it does not have an implementation in pure TOSCA. Instead the implementation is done by a plugin directly shipped with the Yorc orchestrator. This allows to handle more complex use-cases like interacting with workflows inputs.

That said a TOSCA component should still be defined to configure how the plugin will run the PyCOMPSs job.

Code 8 is a simplified version of the TOSCA type definition of the PyCOMPSs execution that shows the configurable properties that can be set for this component.

Code 8: Extract of the TOSCA definition for PyCOMPSs

```
data_types:
  org.eflows4hpc.pycompss.plugin.types.ContainerOptions:
    derived_from: tosca.datatypes.Root
    properties:
      container_image:
        type: string
        required: false
        default: ""
      container_compss_path:
        type: string
        required: false
        default: ""
      container_opts:
        type: string
        required: false
        default: ""

  org.eflows4hpc.pycompss.plugin.types.COMPSsApplication:
    derived_from: tosca.datatypes.Root
    properties:
      command:
        type: string
        required: true
      arguments:
        type: list
        required: false
        entry_schema:
          description: list of arguments
          type: string
      container_opts:
        type: org.eflows4hpc.pycompss.plugin.types.ContainerOptions


  org.eflows4hpc.pycompss.plugin.types.SubmissionParams:
    derived_from: tosca.datatypes.Root
    properties:
      compss_modules:
        type: list
        required: false
        entry_schema:
          description: list of arguments
          type: string
        default: ["compss/3.0", "singularity"]
```

(continues on next page)

(continued from previous page)

```
      num_nodes:
        type: integer
        required: false
        default: 1
      qos:
        type: string
        required: false
        default: debug
      python_interpreter:
        type: string
        required: false
        default: ""
      extra_compss_opts:
        type: string
        required: false
        default: ""

  org.eflows4hpc.pycompss.plugin.types.Environment:
    derived_from: tosca.datatypes.Root
    properties:
      endpoint:
        type: string
        description: The endpoint of the pycomps server
        required: false
      user_name:
        type: string
        description: user used to connect to the cluster may be overridden by a workflow input
        required: false

node_types:
  org.eflows4hpc.pycompss.plugin.nodes.PyCOMPSJob:
    derived_from: org.alien4cloud.nodes.Job
    metadata:
      icon: COMPSs-logo.png
    properties:
      environment:
        type: org.eflows4hpc.pycompss.plugin.types.Environment
        required: false

      submission_params:
        type: org.eflows4hpc.pycompss.plugin.types.SubmissionParams
        required: false

      application:
        type: org.eflows4hpc.pycompss.plugin.types.COMPSsApplication
        required: false

      keep_environment:
        type: boolean
        default: false
        required: false
        description: keep pycompss environment for troubleshooting
    requirements:
      - img_transfer:
          capability: tosca.capabilities.Node
          relationship: tosca.relationships.DependsOn
```

(continues on next page)

(continued from previous page)

```
        occurrences: [ 0, UNBOUNDED ]
    - environment:
        capability: eflows4hpc.env.capabilities.ExecutionEnvironment
        relationship: tosca.relationships.DependsOn
        occurrences: [ 0, UNBOUNDED ]
```

- The `org.eflows4hpc.pycompss.plugin.types.ContainerOptions` data type allows to define container specific options for the PyCOMPSs job
  - `container_image` is the path the container image to use to run the execution. If connected to a `dls.ansible.nodes.DLSDAGImageTransfer` component the path of the transferred image is automatically detected.
  - `container_compss_path` is the path where compss is installed in the container image
  - `container_opts` are the options to pass to the container engine
- The `org.eflows4hpc.pycompss.plugin.types.COMPSsApplication` data type allows to define how a PyCOMPSs application is run
  - `command` is the actual command to run
  - `arguments` is a list of arguments
  - `container_opts` is `org.eflows4hpc.pycompss.plugin.types.ContainerOptions` data type described above
- The `org.eflows4hpc.pycompss.plugin.types.SubmissionParams` data type defines PyCOMPSs parameters related to job submission
  - `compss_modules` is the list of modules to load for the job. This could be overridden by the `pycompss_modules` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together.
  - `num_nodes` is the number of nodes a job should run on
  - `qos` is the quality of Service to pass to the queue system
  - `python_interpreter` Python interpreter to use (python/python3)
  - `extra_compss_opts` is an arbitrary list of extra options to pass to PyCOMPSs
- The `org.eflows4hpc.pycompss.plugin.types.Environment` data type define properties related to the cluster where the job should be run
  - `endpoint` the remote host to run jobs on. This could be overridden by the `cluster_login_host` attribute of a `eflows4hpc.env.nodes.AbstractEnvironment` if components are linked together.
  - `user_name` user used to connect to the cluster may be overridden by a workflow input
- The `org.eflows4hpc.pycompss.plugin.nodes.PyCOMPSJob` TOSCA component
  - `environment` is `org.eflows4hpc.pycompss.plugin.types.Environment` data type described above
  - `submission_params` is `org.eflows4hpc.pycompss.plugin.types.SubmissionParams` data type described above
  - `application` is `org.eflows4hpc.pycompss.plugin.types.COMPSsApplication` data type described above
  - `keep_environment` is a flag to keep pycompss execution data for troubleshooting

4.1.4.4 Environment TOSCA component

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

This components holds properties of an HPC cluster. It is an abstract TOSCA component, meaning that it's values does not need to be known when designing the application and can be matched to a concrete type just before the deployment. This is a powerful tool combined with Alien4Cloud's services that allows to define concrete types for abstract components.

Code 9 is a simplified version of the TOSCA type definition of the Environment that shows attributes of this component.

Code 9: Extract of the TOSCA definition for Environment

```
eflows4hpc.env.nodes.AbstractEnvironment:
  derived_from: tosca.nodes.Root
  abstract: true
  attributes:
    cluster_login_host:
      type: string
    pycompss_modules:
      type: string
    dls_api_url:
      type: string
```

- `cluster_login_host` the host (generally a login node) of the HPC cluster to connects to
- `pycompss_modules` a coma-separated list of PyCOMPSs modules installed on this cluster and that should be loaded by PyCOMPSs
- `dls_api_url` the URL of the Data Logistics Service API

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 `ghcr.io/eflows4hpc/hpcwaas-api:main-cli` 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 are running instances of this API on both Juelich and BSC clouds, ask to the team (eflows4hpc@bsc.es) for an access to the API.

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. 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 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.

For a more detailed usage please refer to *Step-by-step Example*.

Chapter 5

Step-by-step Example

This chapter provides a step-by-step guide about how developers and final users can implement, deploy and execute a workflow using the eFlows4HPC Methodologies. To illustrate it, we will use one of the workflows implemented during the first period of the project.

5.1 Pillar I: Reduced Order Model workflow

Figure 13 shows an overview of what we want to achive with the Pillar I workflow. This workflow aims at creating a Reduced-Order Model (ROM) from the training data generated by Full Order Model (FOM) simulations. The HPC FOM simulations performed with the *Kratos Multiphysics* software are combined with distributed machine learning algorithms implemented with the *dislib*. The input for these simulations are available in HTTP repository, and the generated Reduce Order Model must be uploaded to an B2SHARE repository in order to be available for final users. All required software will be deployed as containers in the HPC sites and all the required data movement and executions will be automatically orchestrated by the eFlows4HPC components.

The following sections describe the different steps to implement, deploy and execute the Reduce Order Model workflow using the eFlows4HPC methodologies:

- *Step 1*: Implement the computational workflow integrating different types of computations using the eFlows4HPC programming interfaces.
- *Step 2*: Enable the automatic creation of container images by including the workflow software requirements in the workflow description.
- *Step 3*: Implement the data logistic pipelines to manage workflow data movements between the parallel file system of HPC clusters and external data repositories.
- *Step 4*: Integrate the different workflow parts in TOSCA application to enable the automation of the deployment and execution processes.
- *Step 5*: Deploy the workflow to an HPC clusters using Alien4Cloud and make it accessible to users
- *Step 6*: Configure the credentials and Execute the workflow with the HPCWaaS execution API

5.1.1 Implementation of the Reduced Order Model Computation

PyCOMPSs is a task-based programming model which allows developers to define parallel workflows as simple sequential python scripts. To implement a PyCOMPSs application, developers have 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 hundreds of milliseconds) that can potentially run concurrently with other parts of the application. Those methods have to be annotated with the *@task* decorator and indicate the directionality of the parameters. Based on the task definitions, the runtime is able to detect dependencies between task invocations and infer the inherent parallelism of a Python script.

PyCOMPSs has been extended in eFlows4HPC, with two new decorators (*@software* and *@dt*) to facilitate the integration of different kinds of computations in a PyCOMPSs workflow and facilitating their reuse in other workflows. This section will show how this methodology is applied in the case of the Reduced Order Model

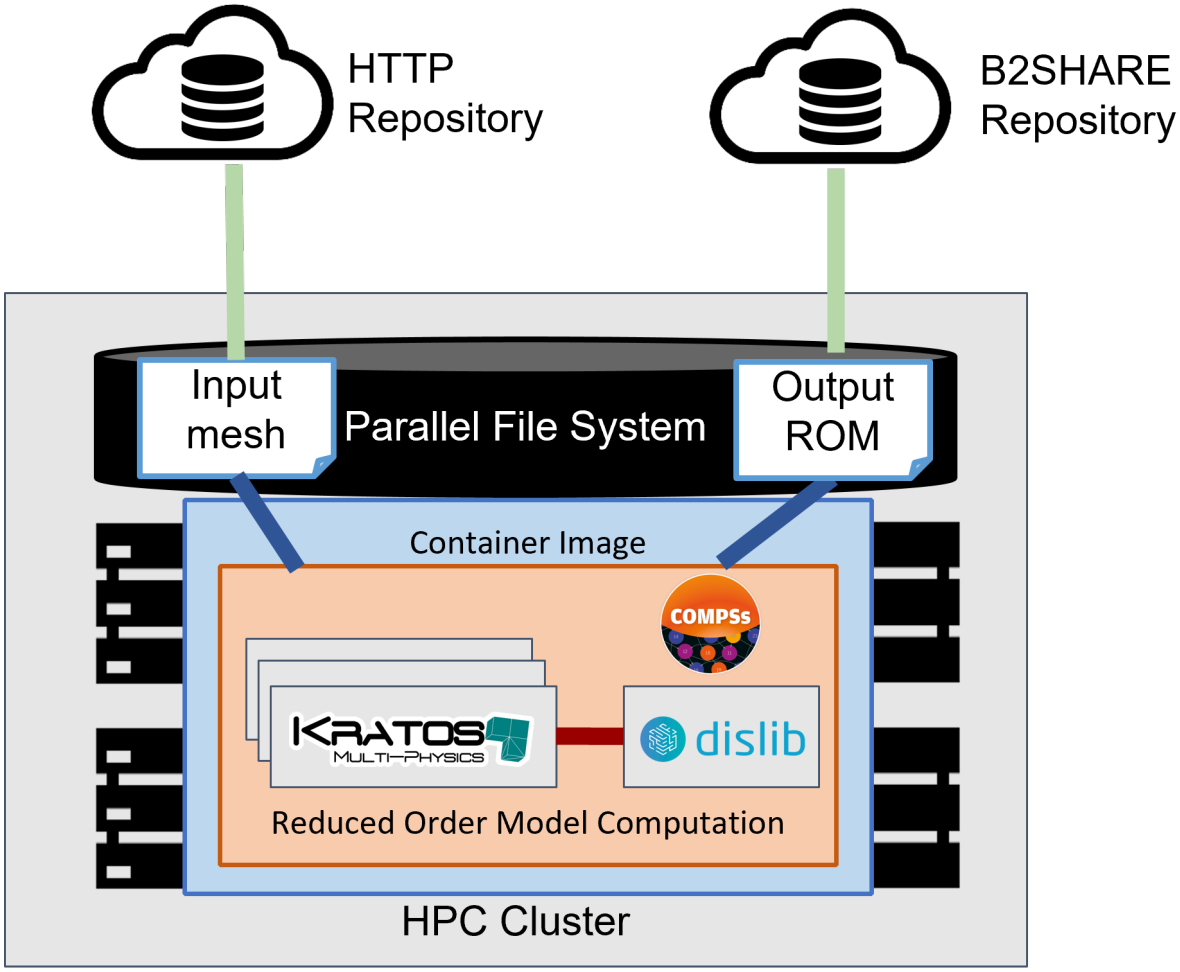


Figure 13: Overview of the Pillar I workflow.

(ROM) workflow. More details about how to use these decorators in other cases are available at the *Programming Interfaces* section.

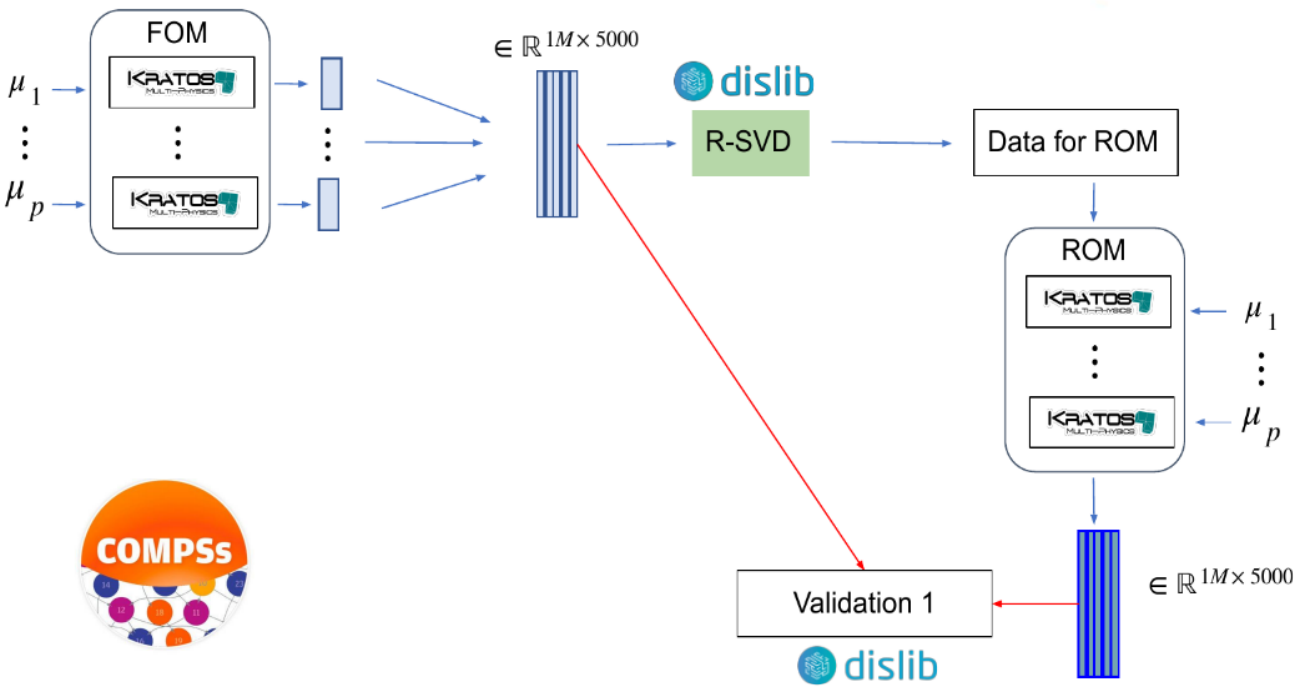


Figure 14: Reduced Order Model Computation Overview.

Figure 14 shows an overview of the first version of the Reduced Order Model workflow implemented in Pillar I. The workflow starts with a set of Full Order Model simulations of the system in which we want to create the ROM. These simulations are executed using the Python API of the *Kratos Multiphysics* software, and the results of these simulations are used as a training dataset for the ROM computation. To calculate the ROM, we have implemented the Randomized SVD algorithm using the *dislib* which implements distributed ML algorithms on top of PyCOMPSs. The workflow finishes with another set of Kratos computations which repeats the simulations using the obtained ROM and the results are compared with the FOM results.

Code 10 shows a code snippet about how the workflow has been implemented with PyCOMPSs. You can observe the main code of the workflow is a simple Python script where after parsing the arguments and loading the model and parameters, the different FOM simulations (*execute_FOM_instance*) are invoked with different configuration values. The results of these simulations (*sim_results* variable) are passed to the randomized SVD computation (*rSVD*) which produces the ROM (*rom* variable). This ROM is used as input for the ROM simulations (*execute_ROM_instance*) and their results are compared by invoking the *compare_ROM_vs_FOM* functions.

Code 10: Code snippet of the PyCOMPSs code for the ROM computation.

```
if __name__ == '__main__':

    model_file, sim_cfgs, desired_rank, output_rom_file = parse_arguments()
    model, parameters = load_model_parameters(model_file)

    # Full Order Model (FOM) simulation for each simulation parameter.
    sim_results=[]
    for cfg in sim_cfgs:
        sim_results.append(execute_FOM_instance(model,parameters,[cfg]))

    # Computes the "fixed rank" randomized SVD in parallel using the dislib library
```

(continues on next page)

(continued from previous page)

```
    rom = rSVD(sim_results, desired_rank)

    # Reduced Order Model simulations for the same simulation parameters used for the FOM
    rom_results=[]
    for cfg in sim_cfgs:
        sim_results.append(execute_ROM_instance(model,parameters,[cfg],rom))

    compare_ROM_vs_FOM(rom_results, sim_results)
```

Code 11 and Code 12 shows a code snippet about how the FOM simulation has been implemented with PyCOMPSs. On top of the function which includes the Kratos Multiphysics calls we have included *@software* decorator to indicate the function is a Kratos FOM invocation described by the fom.json file stored in the Software Catalog. This description indicates that it will be executed as a task, consuming the number of cores indicated by *$KRATOS_CUS* environment variable.

Code 11: Code snippet of the PyCOMPSs definition of FOM simulation.

```
@software(config_file = SW_CATALOG+"/kratos/fom.json")
def execute_FOM_instance(model,parameters, sample):
    import KratosMultiphysics
    from kratos_simulations import GetTrainingData
    current_model = KratosMultiphysics.Model()
    model.Load("ModelSerialization",current_model)
    del(model)
    current_parameters = KratosMultiphysics.Parameters()
    parameters.Load("ParametersSerialization",current_parameters)
    del(parameters)
    # get sample
    simulation = GetTrainingData(current_model,current_parameters,sample)
    simulation.Run()
    return simulation.GetSnapshotsMatrix()
```

Code 12: Definition of FOM simulation(fom.json).

```
{
    "execution" : {
            "type":"task"
    },
    "constraints" : {
            "computing_units": "$KRATOS_CUS"
    },
    "parameters" : {
            "model" : "IN",
            "parameters" : "IN",
            "sample" : "IN",
            "returns" :1
    }
}
```

In the case of the randomized SVD, the code snippet and configuration file can be found in Code 13 and Code 14. On top of the function which includes the dislib calls, we have included the *@dt* and *@software* decorators. On one side, the *@software* decorator indicates the function is a dislib code whose execution is described in the dislib.json file stored in the Software Catalog. This description indicates that it will be treated as a PyCOMPS workflow. On the other side, the *@dt* decorator indicates the transformation required to the *blocks* to the ds-array used by dislib as implemented in the *load_blocks_rechunk* function.

Code 13: Code snippet of the PyCOMPSs definition of the Randomized SVD.

```
@dt("blocks", load_blocks_rechunk, shape=expected_shape, block_size=simulation_block_size,
     new_block_size=desired_block_size, is_workflow=True)
@software(config_file = SW_CATALOG + "/py-dislib/dislib.json")
def rSVD(blocks, desired_rank=30):
    from dislib_randomized_svd import rsvd
    u,s = rsvd(blocks, desired_rank, A_row_chunk_size, A_column_chunk_size)
    return u
```

Code 14: Definition of dislib algorithm.

```
{
    "execution" : {
            "type" : "workflow"
    }
}
```

Following the same procedure as above, we have defined the ROM simulations as depicted in Code 15 and the ROM/FOM comparison as depicted in Code 16. The ROM simulations have a lot of similarities to the FOM simulations. In this case, a data transformation has been added to serialize the *rom* object to the ROM file required by Kratos Multiphysics. In the case of the ROM/FOM comparison, as it is implemented as a dislib algorithm, two *load_blocks_rechunk* data transformations have been included to convert the ROM and FOM results to dislib's ds-arrays.

Code 15: Code snippet of the PyCOMPSs definition of the ROM simulation.

```
@dt(target="rom", function=ROM_file_generation, type=OBJECT_TO_FILE, destination=rom_file)
@software(config_file = SW_CATALOG + "/kratos/rom.json")
def execute_ROM_instance(model,parameters,sample,rom):
    import KratosMultiphysics
    from kratos_simulations import RunROM_SavingData
    load_ROM(rom)
    current_model = KratosMultiphysics.Model()
    model.Load("ModelSerialization",current_model)
    current_parameters = KratosMultiphysics.Parameters()
    parameters.Load("ParametersSerialization",current_parameters)
    # get sample
    simulation = RunROM_SavingData(current_model,current_parameters,sample)
    simulation.Run()
    return simulation.GetSnapshotsMatrix()
```

Code 16: Code snippet of the PyCOMPSs definition of the ROM/FOM comparison.

```
@dt("SnapshotsMatrixROM", load_blocks_rechunk, shape=expected_shape, block_size=simulation_
↪block_size,
     new_block_size=desired_block_size, is_workflow=True)
@dt("SnapshotsMatrixFOM", load_blocks_rechunk, shape=expected_shape, block_size=simulation_
↪block_size,
     new_block_size=desired_block_size, is_workflow=True)
@software(config_file = SW_CATALOG + "/py-dislib/dislib.json")
def compare_ROM_vs_FOM(SnapshotsMatrixROM, SnapshotsMatrixFOM):
    import dislib as ds
    import numpy as np
```

(continues on next page)

(continued from previous page)

```
    #using the Frobenious norm of the snapshots of the solution
    original_norm= np.linalg.norm((SnapshotsMatrixFOM.norm().collect()))
    intermediate = ds.data.matsubtract(SnapshotsMatrixROM, SnapshotsMatrixFOM) #(available on␣
↪latest release)
    intermediate = np.linalg.norm((intermediate.norm().collect()))
    final = intermediate/original_norm
    np.save('relative_error_rom.npy', final)
```

All this code has been stored in the Workflow Reposirory as indicated in *this section*.

5.1.2 Enabling HPC Ready Container Image Creation

In the previous section, we have seen how to implement the computational workflow with PyCOMPSs. To enable the deployment of this workflow with containers, we have to indicate what software is required for the execution of this service. Previously, we have seen that the Reduced Order Model computation requires Kratos Multiphysics for the FOM and ROM simulations, dislib for rSVD and results comparison and COMPSs as the runtime system to manage PyCOMPSs workflows. These requirements have to be defined as a simplified Spack environment (*spack.yml*) inside the workflow folder stored in the Workflow Registry as shown in Code 17. Developers just need to indicate the required software package name and optionally specify the version and variants for this software. For instance, this workflow requires *Kratos* version *9.1.4* with the *app* variant which indicates the Kratos applications to include in the compilation. No other Spack environment information must be included in this description, the rest of the options of the spack environment will be included during the image creation process depending on the description of the target supercomputer.

Code 17: Workflow Software requirements as simplified spack environment.

```
spack:
        specs:
                - compss
                - py-dislib
                - kratos@9.1.4 apps=LinearSolversApplication,FluidDynamicsApplication,
↪StructuralMechanicsApplication,ConvectionDiffusionApplication,RomApplication
```

The description of the software packages indicated in *spack.yml* must be included in the *Software Catalog* or supported by Spack. This description must follow the Spack package description format. It is a Python class which defines the packaging type (Autotools, CMake, Python modules, etc.), the location from where to download the sources, available versions, software dependencies, and other options depending on the packaging type. For instance, Code 18 shows how this description has been implemented for COMPSs. It uses the basic packaging type (extends *Package*) and the installation procedure is described by implementing the *install* method. More examples can be found in the Software Catalog repository.

Code 18: Spack package description for COMPSs.

```
class Compss(Package):
    """COMP Superscalar programming model and runtime."""

    url      = "https://compss.bsc.es/repo/sc/stable/COMPSs_2.10.tar.gz"
    version('2.10', sha256='0795ca7674f1bdd0faeac950fa329377596494f64223650fe65a096807d58a60',
↪ preferred=True)
    ...

    # dependencies.
    depends_on('python')
    depends_on('openjdk')
    depends_on('boost')
```

(continues on next page)

(continued from previous page)

```
    depends_on('libxml2')
    ...

    def install(self, spec, prefix):
        install_script = Executable('./install')
        install_script('-A', '--only-python-3', prefix.compss)

    def setup_run_environment(self, env):
        env.set('COMPSS_HOME', self.prefix.compss)
        env.prepend_path('PATH', self.prefix.compss + '/Runtime/scripts/user')
```

Once the workflow software requirements and the software package description have been included in the Workflow Registry and Software Catalog respectively, we can start to create the container images for this workflow. With this goal, developers have to request it to the Container Image Creation service using the command line interface (CLI). Code 19 and Code 20 shows the command and JSON file of the request for creating the ROM workflow container image for the MareNostrum4. This supercomputer has a *skylake_avx512* architecture and supports Singularity containers. In this section, we have shown how a developer can use the Container Image Creation to build the workflow containers, however the Container Image Creation can be also part of the workflow deployment and it can be executed via the TOSCA descriptions. More details about this option can be found in *Image Creation Service TOSCA component*.

Code 19: Container Image Creation CLI command for ROM image creation request for MN4.

```
$ image_creation> ./cic_cli user pass https://bscgrid20.bsc.es build rom_for_MN4.json
Response:
{"id":"f1f4699b-9048-4ecc-aff3-1c689b855adc"}
```

Code 20: ROM image creation request for MN4 supercomputer.

```
{
     "machine": {
          "platform": "linux/amd64",
          "architecture": "skylake_avx512",
          "container_engine": "singularity"
     },
     "workflow":"rom_pillar_I",
     "step_id" :"reduce_order_model"
}
```

More details about the Container Image Creation service can be found in *this link*.

5.1.3 Implementing a Data Logistics Pipeline

Data movements in the eFlows4HPC Workflow-as-a-Service are orchestrated by the Data Logistics Service and defined as Airflow Pipelines. The pipelines are formally Direct Acyclic Graphs (DAGs) and are defined programmatically using Python.

Each DAG definition consists of a set of tasks and additional metadata. The metadata can be used, for example, to 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 custom operators.

The following is a brief introduction to Data Logistics Pipelines using the eFlows4HPC Pillar 1 workflow as an example. The complete source code of the workflow pipeline can be found in repository. The workflow is built on the principle of Extract Transform Load (ETL) and uses the Airflow taskflow API to define a DAG.

5.1.3.1 DAG Definition: HTTP-based transfer

The stage-in of the data in Pillar I is fairly straightforward. The source of the data is a B2DROP repository that provides HTTP access. The destination is an HPC system accessed via SSH.

```
@dag(default_args=default_args, schedule_interval=None, start_date=days_ago(2), tags=['wp4',
↪'http', 'ssh'])
def plainhttp2ssh():

    @task
    def stream_upload(connection_id, **kwargs):
        params = kwargs['params']
        force = params.get('force', True)
        target = params.get('target', '/tmp/')
        url = params.get('url', '')
        if not url:
            print('Provide valid url')
            return -1

        print(f"Putting {url} --> {target}")
        ssh_hook = get_connection(conn_id=connection_id, **kwargs)

        with ssh_hook.get_conn() as ssh_client:
            return http2ssh(url=url, ssh_client=ssh_client, remote_name=target, force=force)

    setup_task = PythonOperator(python_callable=setup, task_id='setup_connection')
    a_id = setup_task.output['return_value']
    cleanup_task = PythonOperator(python_callable=remove, op_kwargs={'conn_id': a_id}, task_
↪id='cleanup')

    setup_task >> stream_upload(connection_id=a_id) >> cleanup_task

dag = plainhttp2ssh()
```

The DAG is defined as a Python annotated function `plainhttp2ssh`. The submethods (only one in this case) are annotated with `@task` are Operators, finally the dependencies between tasks are defined with help of `>>` operator.

5.1.3.2 Data Movement Tasks

The workflow includes the following data movements:

- download from B2DROP repository,
- upload to the target system using SCP/SFTP.

The `http2ssh` method streams the data directly to the target location, without an intermediate storage on the DLS server. Although, the transfer pipeline is implemented with B2DROP in mind, any data source that provides HTTP-based access can be used here. The `force` parameter passed to the pipeline defines what to do if the requested data are already exist at the target location (overwrite or not). This is useful if the workflow needs to be run multiple times.

5.1.3.3 Connection setup

The credentials required to access storages are passed to the DAG through external component `vault`. Based on their contents a temporary Airflow connection is created, used by Data Movement Tasks and then removed. The connection management is taken care of by `setup` and `remove` tasks. The data movement method is provided with `connection_id` that is dynamically created for the particular data transfer.

5.1.3.4 DAG Definition: Singularity image upload

After the successful stage-in of the data, a computation step follows. The computations in Pillar I workflow are performed using Singularity containers. This requires a Singularity image to be present on the target machine. The HPC nodes usually don't have Internet access, thus the image needs to be uploaded to the right place before the computation happens. The following code shows how such a transfer can be performed.

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

    @task
    def stream_upload(connection_id, **kwargs):
        params = kwargs['params']
        force = params.get('force', True)
        target = params.get('target', '/tmp/')
        image_id = params.get('image_id', 'wordcount_skylake.sif')
        target = os.path.join(target, image_id)
        url = f"https://bscgrid20.bsc.es/image_creation/images/download/{image_id}"

        ssh_hook = get_connection(conn_id=connection_id, **kwargs)

        with ssh_hook.get_conn() as ssh_client:
            return http2ssh(url=url, ssh_client=ssh_client, remote_name=target, force=force)

    setup_task = PythonOperator(python_callable=setup, task_id='setup_connection')
    a_id = setup_task.output['return_value']
    cleanup_task = PythonOperator(python_callable=remove, op_kwargs={'conn_id': a_id}, task_
↪id='cleanup')

    setup_task >> stream_upload(connection_id=a_id) >> cleanup_task


dag = transfer_image()
```

This pipeline is almost identical to the previous one as the images are downloaded from the eFlows4HPC image service which provides HTTP-based access and uploaded to the target location using SSH. The only difference is the use of the `image_id` parameter instead of the full `url` as in the previous example.

5.1.3.5 Final 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 more information.

If you intend to use eFlows4HPC resources accessed via SSH, reuse `setup_task` and `cleanup_task`.

The data movements are part of the overall workflow and are executed via the TOSCA descriptions (see *Data Logistics Service TOSCA components* for more details). For testing purposes, however, you can start the pipelines directly either via the Airflow UI or via API calls.

```
curl -X POST -u airflowuser:airflowpass \
        -H "Content-Type: application/json" \
        --data '{"conf": {"image_id": "wordcount_skylake.sif", "target": "/tmp/", "host":
↪"sshhost", "login": "sshlogin", "vault_id": "youruserid"}}' \
        https://datalogistics.eflows4hpc.eu/api/v1/image_transfer/dagRuns
```

If you don't have credentials registered in vault (or are using local standalone Airflow) you can provide ssh credentials in the API call:

```
curl -X POST -u airflowuser:airflowpass \
       -H "Content-Type: application/json" \
       --data '{"conf": {"image_id": "wordcount_skylake.sif", "target": "/tmp/", "host":
↪"sshhost", "login": "sshlogin", "key": "sshkey"}}' \
       http://localhost:5001/api/v1/image_transfer/dagRuns
```

5.1.4 Integration in TOSCA

eFlows4HPC uses TOSCA to describe the high-level execution lifecycle of a workflow, enabling the orchestration of tasks with diverse nature. For the Pillar I use case, TOSCA is used to coordinate the creation of a container image, its transfer to a target cluster, stage-in of input data, PyCOMPSs computation, and stage-out the computation result to a data catalog.

An exhaustive list of TOSCA components developed in the context of the eFlows4HPC project and their configurable properties can be found in section *eFlows4HPC TOSCA Components*.

Section *ROM Pillar I topology template* describes how these components are assembled together in a TOSCA topology template to implement the ROM Pillar I use case. More specifically you can refer to Code 21 to see how properties of the TOSCA components are used in this particular context.

5.1.4.1 ROM Pillar I topology template

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

This topology template composes the different components described above into a TOSCA application that allows to implement the ROM Pillar I workflow.

The ROM Pillar I workflow is composed of two phases. First at deployment time the Image Creation Service is invoked to generate a container image containing the required softwares, this image is then transferred to the target HPC cluster using the Data Logistic Service (the `DLSDAGImageTransfer` TOSCA component). Once deployed the execution workflow can be invoked as many time as required. This execution workflow consists in transferring input data from an HTTP server to the HPC cluster thanks to the DLS (the `HTTP2SSH` TOSCA component), then run a PyCOMPSs job on those data (the `PyCOMPSJob` TOSCA component) and finally upload computation results to an EUDAT repository using the DLS (the `DLSDAGStageOutData` TOSCA component).

Code 21 shows how are defined the components and how they are connected together in order to run in sequence. Figure 15 shows the same topology in a graphical way.

Code 21: Extract of the TOSCA topology template for ROM Pillar I workflow

```
topology_template:
  inputs:
    debug:
```

(continues on next page)

(continued from previous page)

```
      type: boolean
      required: true
      default: false
      description: "Do not redact sensible information on logs"
    user_id:
      type: string
      required: false
      default: ""
      description: "User id to use for authentication may be replaced with workflow input"
    vault_id:
      type: string
      required: false
      default: ""
      description: "User id to use for authentication may be replaced with workflow input"
    container_image_transfer_directory:
      type: string
      required: false
      description: "path of the image on the remote host"
    mid:
      type: string
      required: true
      description: "Uploaded Metadata ID"
    register_result_in_datacat:
      type: boolean
      required: false
      default: false
      description: "Should the record created in b2share be registered with data cat"
  node_templates:
    StageOutData:
      type: dls.ansible.nodes.DLSDAGStageOutData
      properties:
        mid: { get_input: mid }
        register: { get_input: register_result_in_datacat }
        input_name_for_mid: mid
        input_name_for_source_path: "result_data_path"
        input_name_for_register: register
        dls_api_username: { get_secret: [/secret/data/services_secrets/dls, data=username] }
        dls_api_password: { get_secret: [/secret/data/services_secrets/dls, data=password] }
        dag_id: "upload_example"
        debug: { get_input: debug }
        run_in_standard_mode: false
      requirements:
        - dependsOnAbstractEnvironmentExec_env:
            type_requirement: environment
            node: AbstractEnvironment
            capability: eflows4hpc.env.capabilities.ExecutionEnvironment
            relationship: tosca.relationships.DependsOn
        - dependsOnPyCompsJob2Feature:
            type_requirement: dependency
            node: PyCOMPSJob
            capability: tosca.capabilities.Node
            relationship: tosca.relationships.DependsOn
    ImageCreation:
      type: imagecreation.ansible.nodes.ImageCreation
      properties:
        service_url: "https://bscgrid20.bsc.es/image_creation"
```

(continues on next page)

(continued from previous page)

```
      insecure_tls: true
      username: { get_secret: [/secret/data/services_secrets/image_creation, data=user] }
      password: { get_secret: [/secret/data/services_secrets/image_creation, data=password]␣
↪}
      machine:
        container_engine: singularity
        platform: "linux/amd64"
        architecture: sandybridge
      workflow: "rom_pillar_I"
      step_id: "reduce_order_model"
      force: false
      debug: { get_input: debug }
      run_in_standard_mode: true
  DLSDAGImageTransfer:
    type: dls.ansible.nodes.DLSDAGImageTransfer
    properties:
      target_path: { get_input: container_image_transfer_directory }
      run_in_standard_mode: true
      dls_api_username: { get_secret: [/secret/data/services_secrets/dls, data=username] }
      dls_api_password: { get_secret: [/secret/data/services_secrets/dls, data=password] }
      dag_id: "transfer_image"
      debug: { get_input: debug }
      user_id: { get_input: user_id }
      vault_id: { get_input: vault_id }
    requirements:
      - dependsOnImageCreationFeature:
          type_requirement: dependency
          node: ImageCreation
          capability: tosca.capabilities.Node
          relationship: tosca.relationships.DependsOn
      - dependsOnAbstractEnvironmentExec_env:
          type_requirement: environment
          node: AbstractEnvironment
          capability: eflows4hpc.env.capabilities.ExecutionEnvironment
          relationship: tosca.relationships.DependsOn
  AbstractEnvironment:
    type: eflows4hpc.env.nodes.AbstractEnvironment
  PyCOMPSJob:
    type: org.eflows4hpc.pycompss.plugin.nodes.PyCOMPSJob
    properties:
      submission_params:
        qos: debug
        python_interpreter: python3
        num_nodes: 2
        extra_compss_opts: "--cpus_per_task --env_script=/reduce_order_model/env.sh"
      application:
        container_opts:
          container_opts: "-e"
          container_compss_path: "/opt/view/compss"
        arguments:
          - "$(dirname ${staged_in_file_path})"
          - "/reduce_order_model/ProjectParameters_tmpl.json"
          - "${result_data_path}/RomParameters.json"
        command: "/reduce_order_model/src/UpdatedWorkflow.py"
      keep_environment: true
    requirements:
```

(continues on next page)

(continued from previous page)

```
      - dependsOnDlsdagImageTransferFeature:
          type_requirement: img_transfer
          node: DLSDAGImageTransfer
          capability: tosca.capabilities.Node
          relationship: tosca.relationships.DependsOn
      - dependsOnAbstractEnvironmentExec_env:
          type_requirement: environment
          node: AbstractEnvironment
          capability: eflows4hpc.env.capabilities.ExecutionEnvironment
          relationship: tosca.relationships.DependsOn
      - dependsOnHttp2SshFeature:
          type_requirement: dependency
          node: HTTP2SSH
          capability: tosca.capabilities.Node
          relationship: tosca.relationships.DependsOn
  HTTP2SSH:
    type: dls.ansible.nodes.HTTP2SSH
    properties:
      dag_id: plainhttp2ssh
      url: "https://b2drop.bsc.es/index.php/s/fQ85ZLDztG2t5j3/download/GidExampleSwaped.mdpa
↪"
      force: true
      input_name_for_url: url
      input_name_for_target_path: "staged_in_file_path"
      dls_api_username: { get_secret: [/secret/data/services_secrets/dls, data=username] }
      dls_api_password: { get_secret: [/secret/data/services_secrets/dls, data=password] }
      debug: { get_input: debug }
      user_id: ""
      vault_id: ""
      run_in_standard_mode: false
    requirements:
      - dependsOnAbstractEnvironmentExec_env:
          type_requirement: environment
          node: AbstractEnvironment
          capability: eflows4hpc.env.capabilities.ExecutionEnvironment
          relationship: tosca.relationships.DependsOn
workflows:
  exec_job:
    inputs:
      user_id:
        type: string
        required: true
      vault_id:
        type: string
        required: true
      result_data_path:
        type: string
        required: true
      staged_in_file_path:
        type: string
        required: true
      num_nodes:
        type: integer
        required: false
        default: 1
    steps:
```

(continues on next page)

(continued from previous page)

```
        StageOutData_executing:
          target: StageOutData
          activities:
            - set_state: executing
          on_success:
            - StageOutData_run
        HTTP2SSH_submitted:
          target: HTTP2SSH
          activities:
            - set_state: submitted
          on_success:
            - HTTP2SSH_executing
        PyCOMPSJob_submitting:
          target: PyCOMPSJob
          activities:
            - set_state: submitting
          on_success:
            - PyCOMPSJob_submit
        PyCOMPSJob_submit:
          target: PyCOMPSJob
          operation_host: ORCHESTRATOR
          activities:
            - call_operation: tosca.interfaces.node.lifecycle.Runnable.submit
          on_success:
            - PyCOMPSJob_submitted
        StageOutData_submitted:
          target: StageOutData
          activities:
            - set_state: submitted
          on_success:
            - StageOutData_executing
        StageOutData_submitting:
          target: StageOutData
          activities:
            - set_state: submitting
          on_success:
            - StageOutData_submit
        StageOutData_run:
          target: StageOutData
          operation_host: ORCHESTRATOR
          activities:
            - call_operation: tosca.interfaces.node.lifecycle.Runnable.run
          on_success:
            - StageOutData_executed
        HTTP2SSH_executing:
          target: HTTP2SSH
          activities:
            - set_state: executing
          on_success:
            - HTTP2SSH_run
        PyCOMPSJob_submitted:
          target: PyCOMPSJob
          activities:
            - set_state: submitted
          on_success:
            - PyCOMPSJob_executing
```

(continues on next page)

(continued from previous page)

```
    HTTP2SSH_submitting:
      target: HTTP2SSH
      activities:
        - set_state: submitting
      on_success:
        - HTTP2SSH_submit
    StageOutData_submit:
      target: StageOutData
      operation_host: ORCHESTRATOR
      activities:
        - call_operation: tosca.interfaces.node.lifecycle.Runnable.submit
      on_success:
        - StageOutData_submitted
    HTTP2SSH_run:
      target: HTTP2SSH
      operation_host: ORCHESTRATOR
      activities:
        - call_operation: tosca.interfaces.node.lifecycle.Runnable.run
      on_success:
        - HTTP2SSH_executed
    HTTP2SSH_executed:
      target: HTTP2SSH
      activities:
        - set_state: executed
      on_success:
        - PyCOMPSJob_submitting
    StageOutData_executed:
      target: StageOutData
      activities:
        - set_state: executed
    PyCOMPSJob_executing:
      target: PyCOMPSJob
      activities:
        - set_state: executing
      on_success:
        - PyCOMPSJob_run
    HTTP2SSH_submit:
      target: HTTP2SSH
      operation_host: ORCHESTRATOR
      activities:
        - call_operation: tosca.interfaces.node.lifecycle.Runnable.submit
      on_success:
        - HTTP2SSH_submitted
    PyCOMPSJob_executed:
      target: PyCOMPSJob
      activities:
        - set_state: executed
      on_success:
        - StageOutData_submitting
    PyCOMPSJob_run:
      target: PyCOMPSJob
      operation_host: ORCHESTRATOR
      activities:
        - call_operation: tosca.interfaces.node.lifecycle.Runnable.run
      on_success:
        - PyCOMPSJob_executed
```

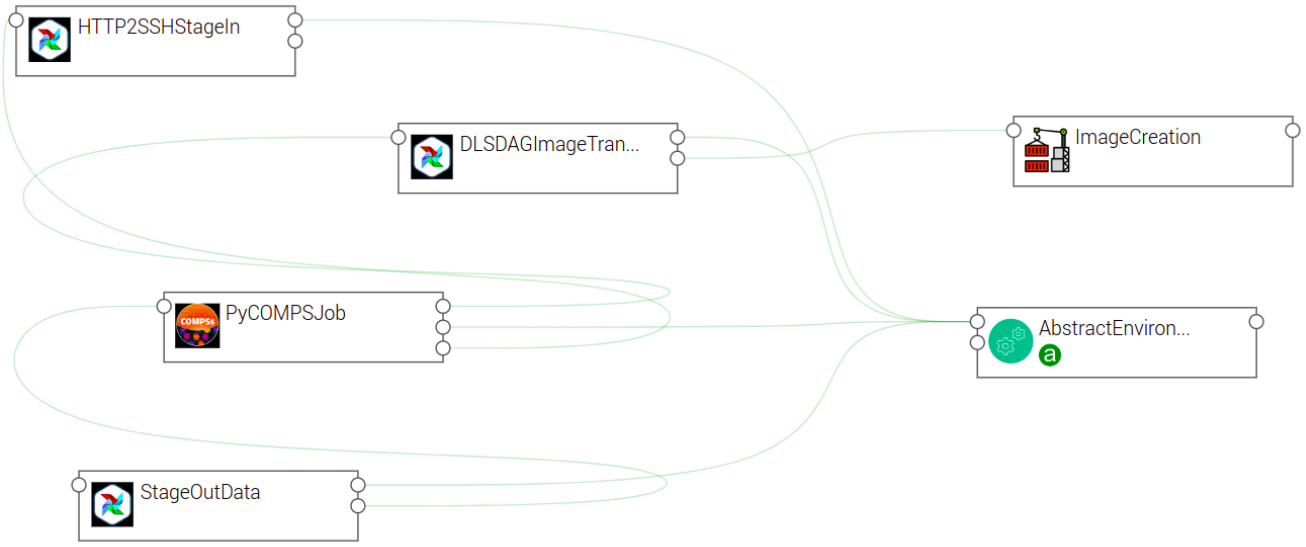


Figure 15: Alien4Cloud ROM Pillar I topology

5.1.5 Workflow Deployment

In this section, we will provide guidance on how to deploy workflows as a "Workflow Developer" in eFlows4HPC. A "Workflow Developer" is responsible for defining and deploying workflows that can be triggered by "end-users" through the HPCWaaS API.

The main interaction for a workflow developer is with Alien4Cloud, where TOSCA applications and associated TOSCA workflows are defined. In this guide, we will utilize the TOSCA Topology Template feature in Alien4Cloud to create an application from a pre-existing template and configure it to meet our specific needs prior to deployment. Upon deployment, we will cover how to test workflows, as well as expose the application to the HPCWaaS API for end-user access.

5.1.5.1 Create an application from a Topology Template

Figure 16 shows how to create an application from a topology template. Log in to Alien4Cloud and navigate to the `Applications` tab. Click on the `New Application` button, provide a unique name for the application, and then switch to the `Topology Template` tab under the `Initialize topology from` section. Select the desired topology template and click the `Create` button.

5.1.5.2 Configure the application before deployment

To prepare for deployment, navigate to the `Environment` section under `Work on an Environment` (see Figure 17). Then click on `Prepare next deployment` (see Figure 18) and select the proposed location (see Figure 19).

Under the `Topology` tab, you can examine the TOSCA topology created from the template and make any necessary modifications. In most cases, this step is not required as the topology is designed to be configurable through defined `inputs`. Access the `Inputs` tab and fill all necessary inputs (see Figure 20). Note that the information bubble provides additional input description.

It is important to understand that these inputs are fixed properties that will be selected prior to deployment and cannot be altered for a specific workflow execution. To do this workflows can be defined with their own specific inputs.

Next, under the `Matching` tab (see Figure 21), you should match abstract TOSCA components to their concrete implementations. This enables the definition of reusable topologies and facilitates their adaptation to target specific HPC clusters, such as the login node address or the PyCOMPSs modules to be loaded for job execution. To do

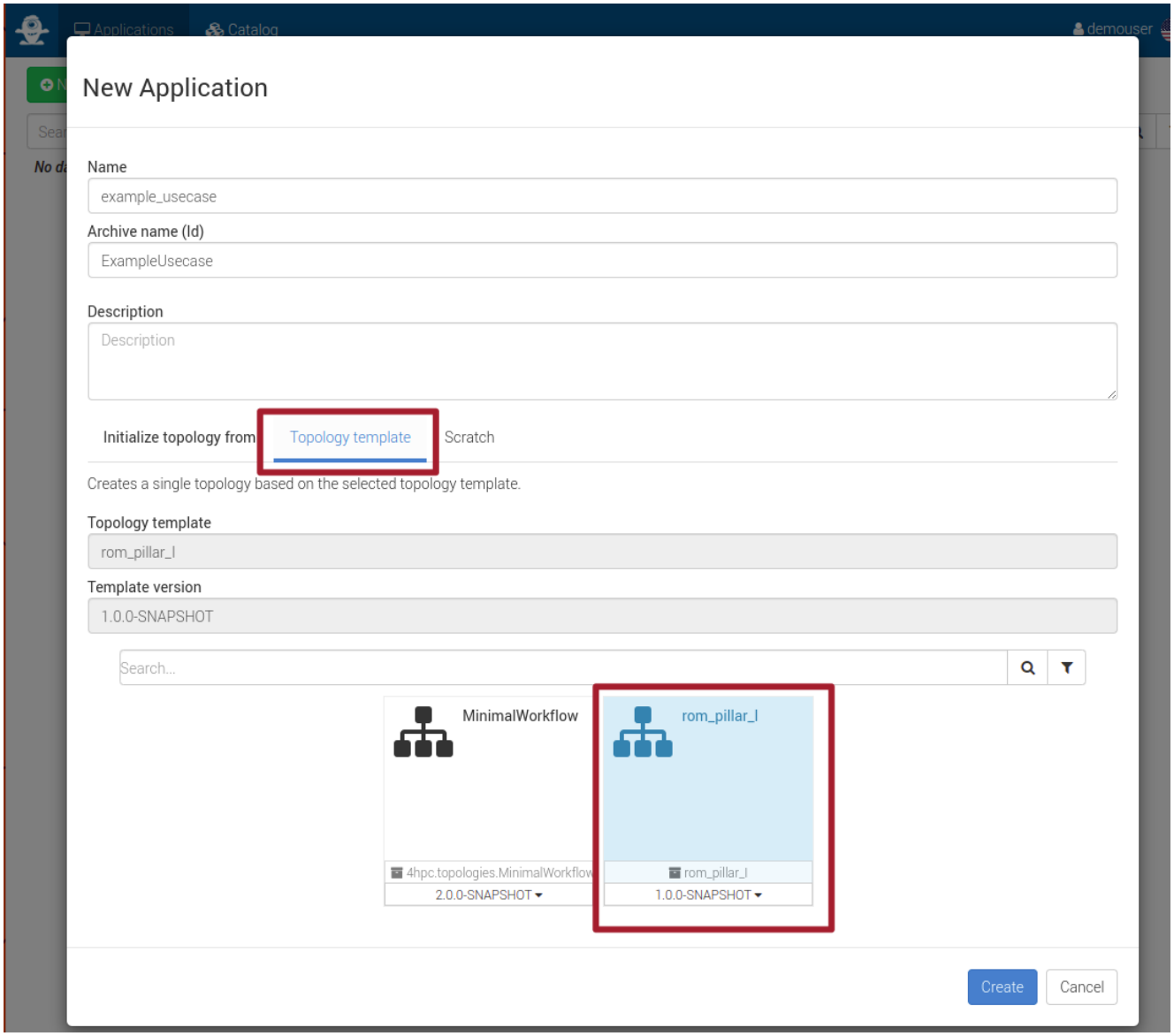


Figure 16: Create an application from a Topology Template

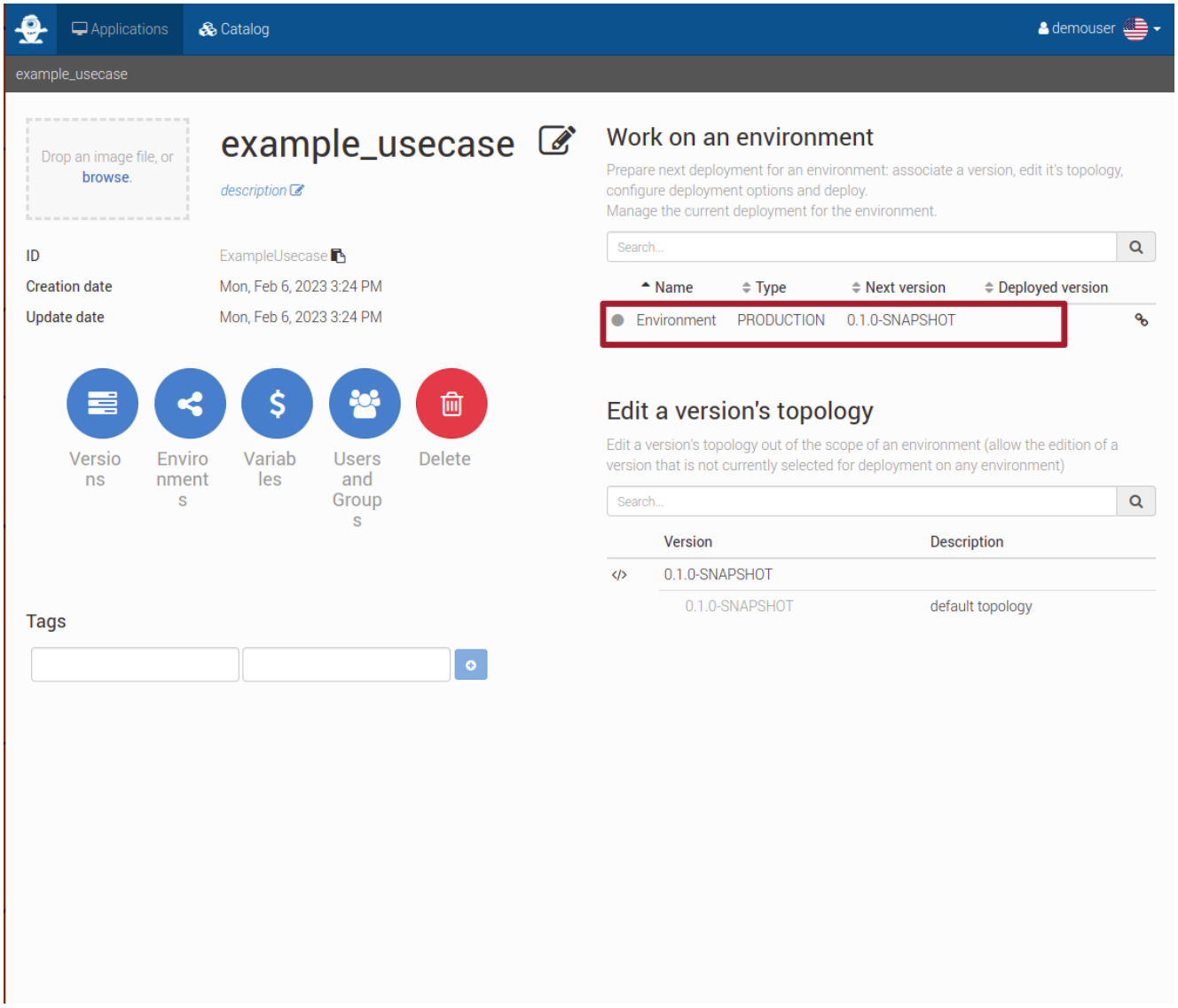


Figure 17: Select the environment

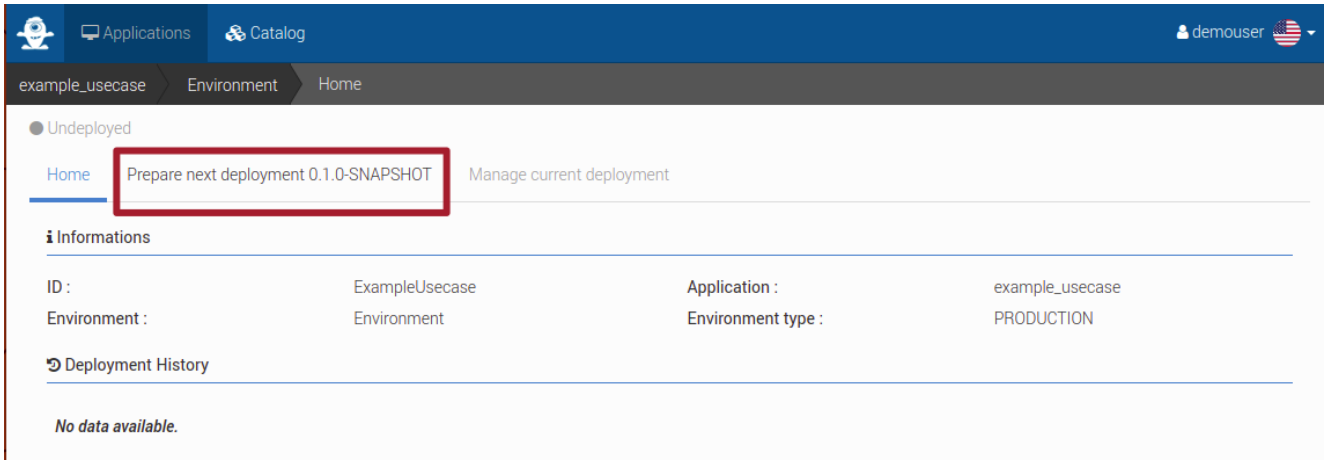


Figure 18: Prepare next deployment

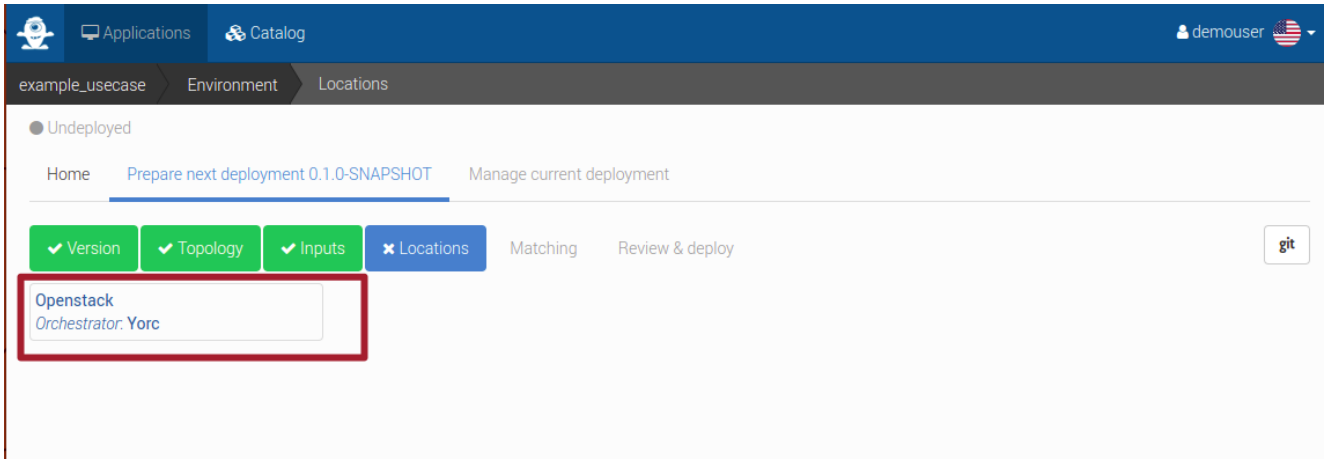


Figure 19: Select location

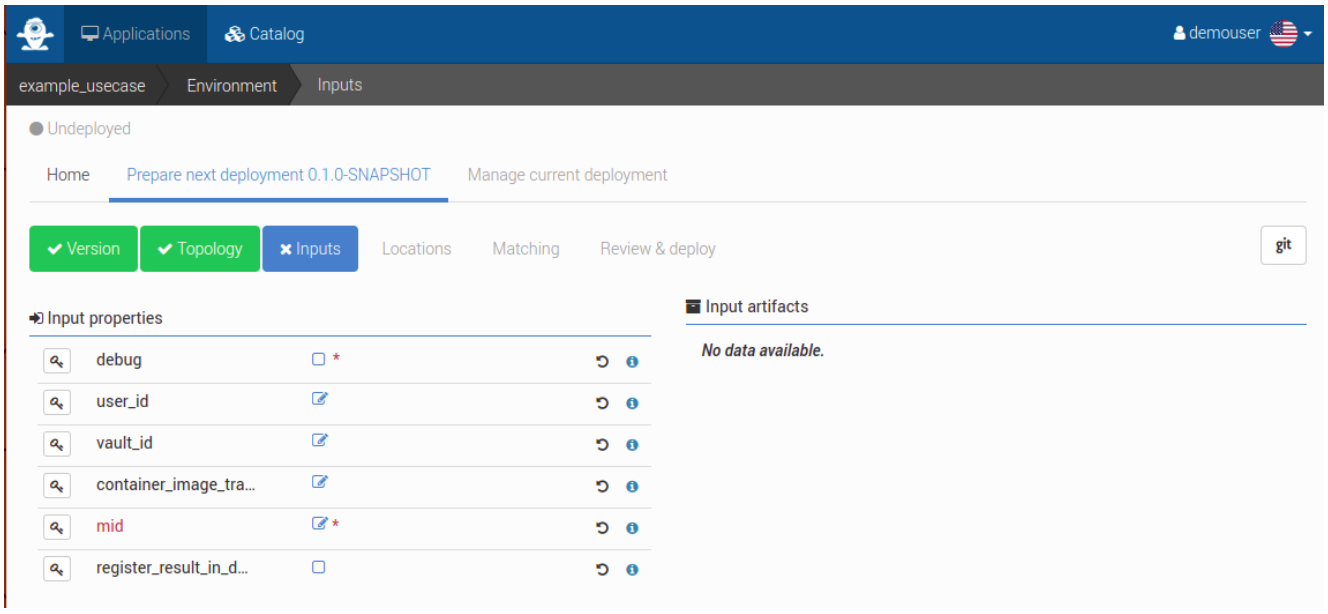


Figure 20: Fill deployment inputs

this, click on the `Nodes matching` tab and expand the `AbstractEnvironment` node. Finally, select the desired concrete implementation for the execution environment.

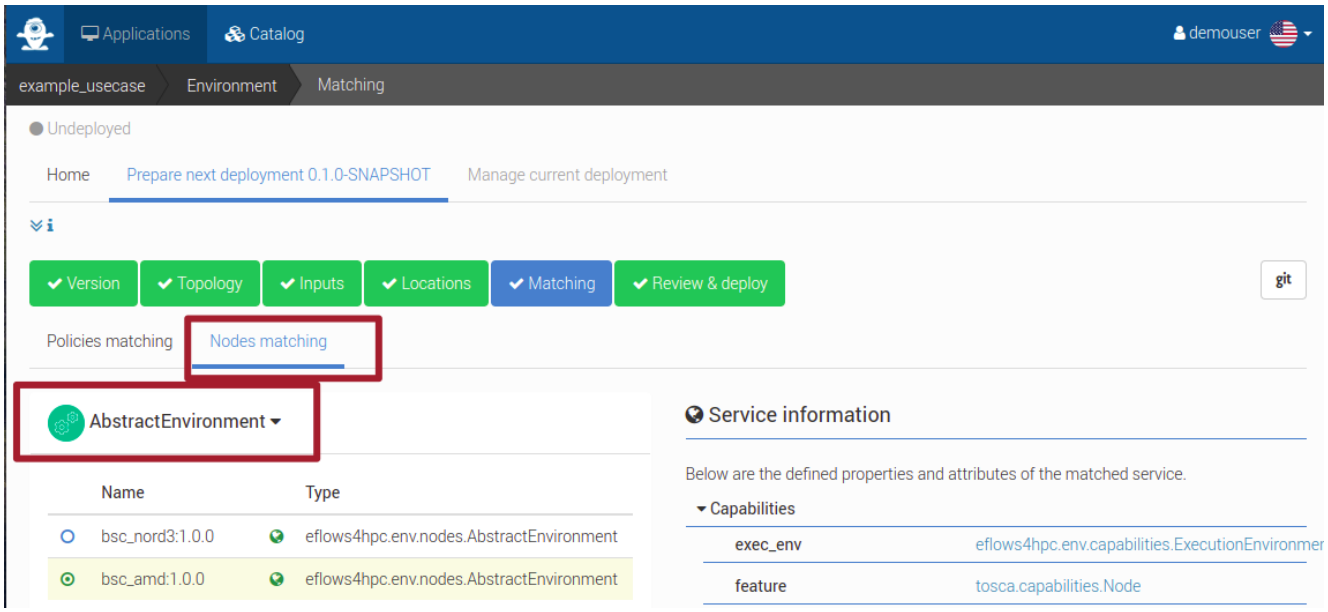


Figure 21: Match abstract components to concrete implementations

5.1.5.3 Deploy an application

To finalize the deployment process, navigate to the `Review and deploy` tab (see Figure 22). Carefully review the configurations made in previous sections and, if satisfactory, click the `Deploy` button.

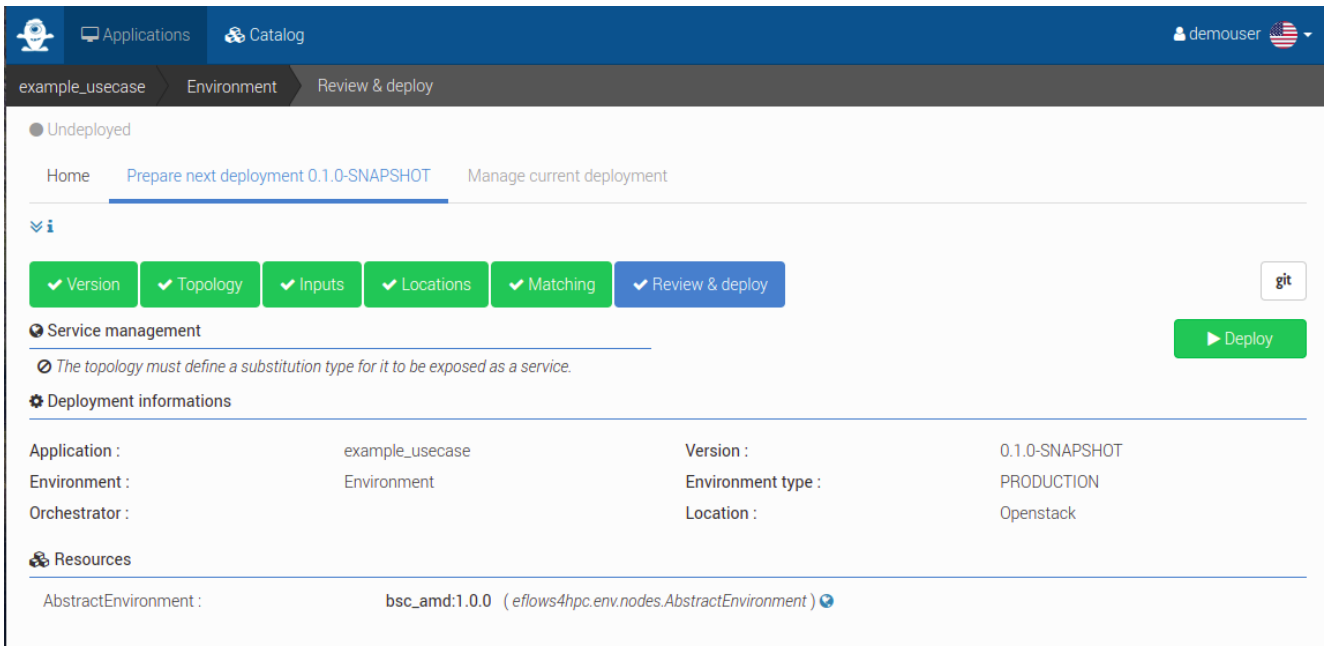


Figure 22: Deploy application

You will be automatically redirected to the `Manage current deployment` tab (see Figure 23). Here, you can monitor the progress of the deployment. For more comprehensive insights, you may access the `Workflow` (see Figure 24) or `Logs` (see Figure 25) tabs.

Wait for the deployment workflow to complete before moving to the next section.

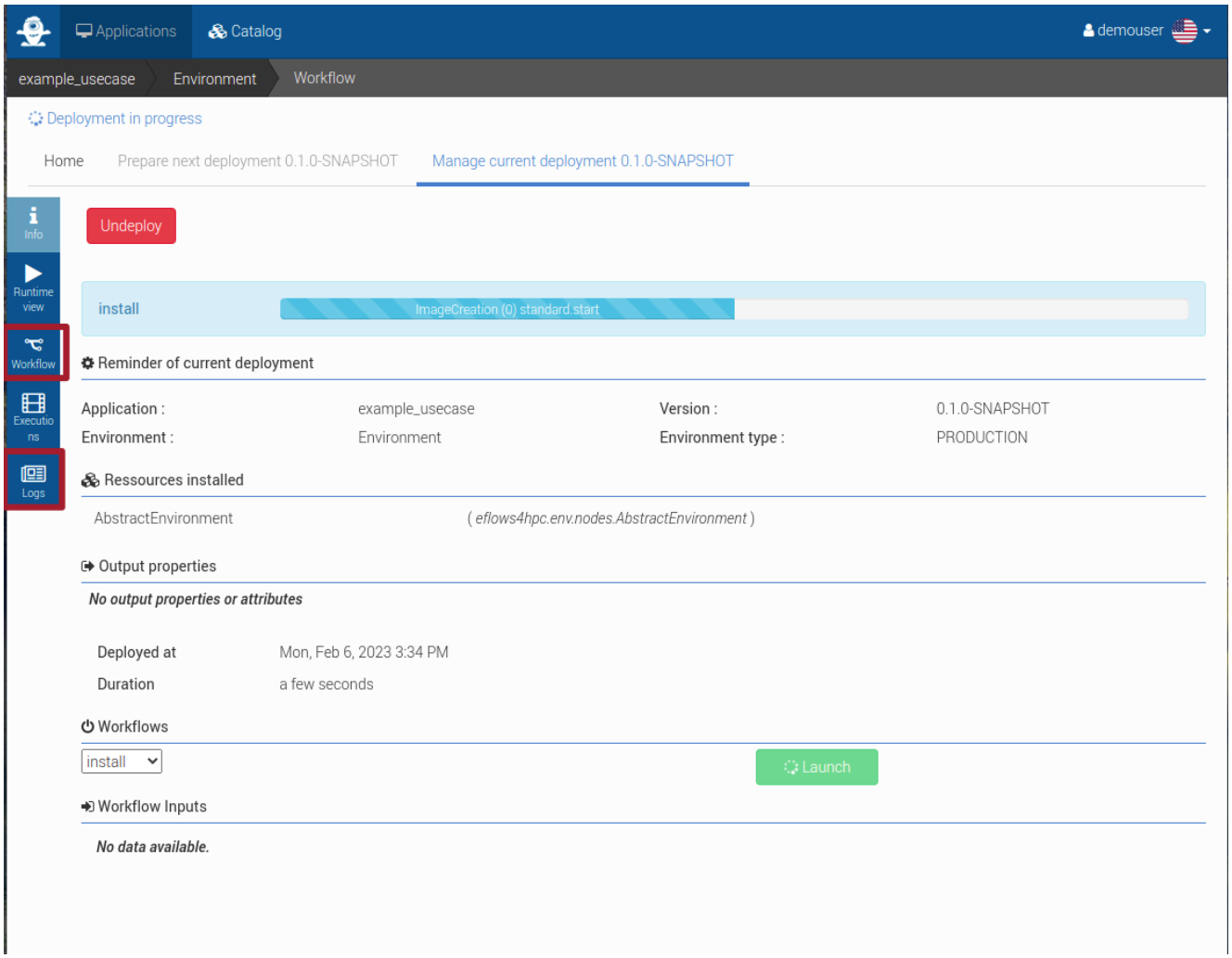


Figure 23: Deployment of an application

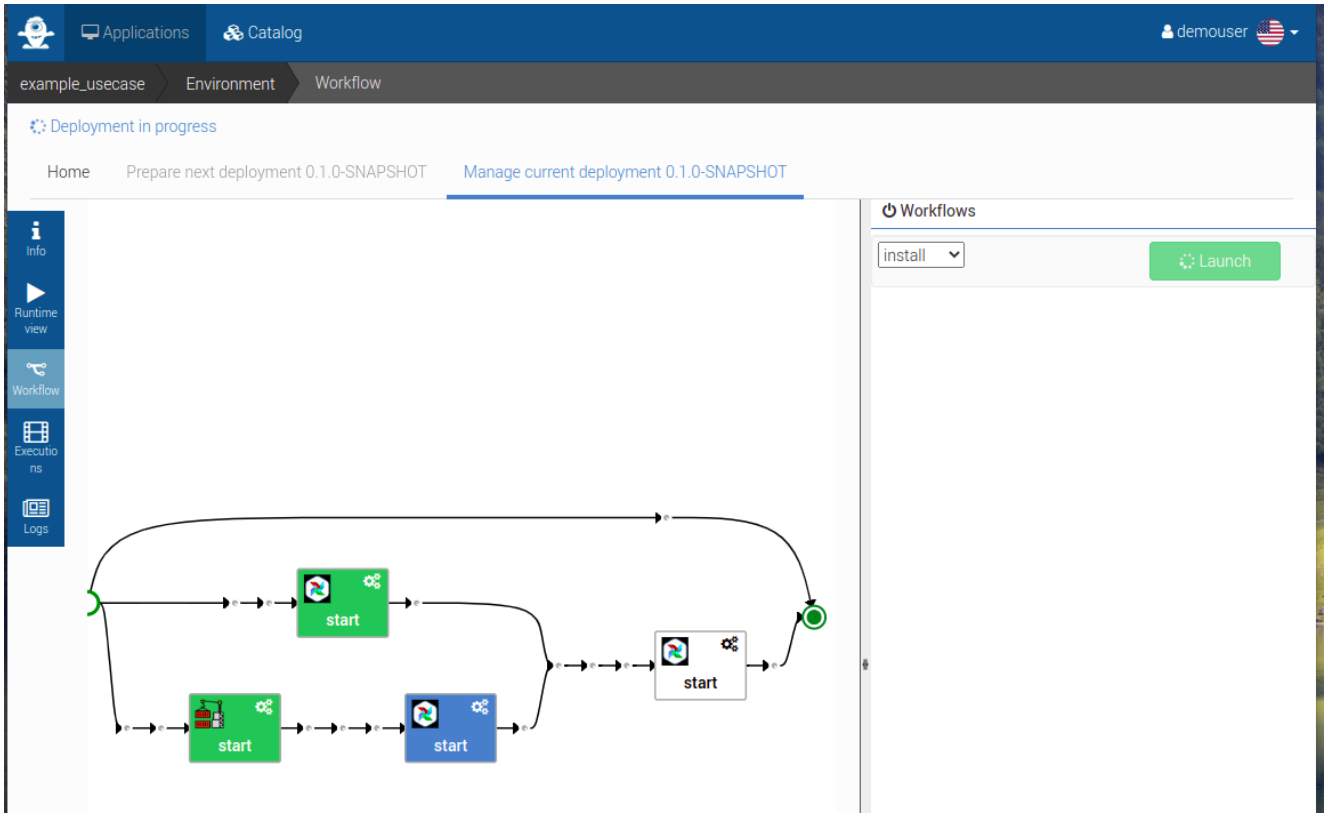


Figure 24: Workflow view of a deployment of an application

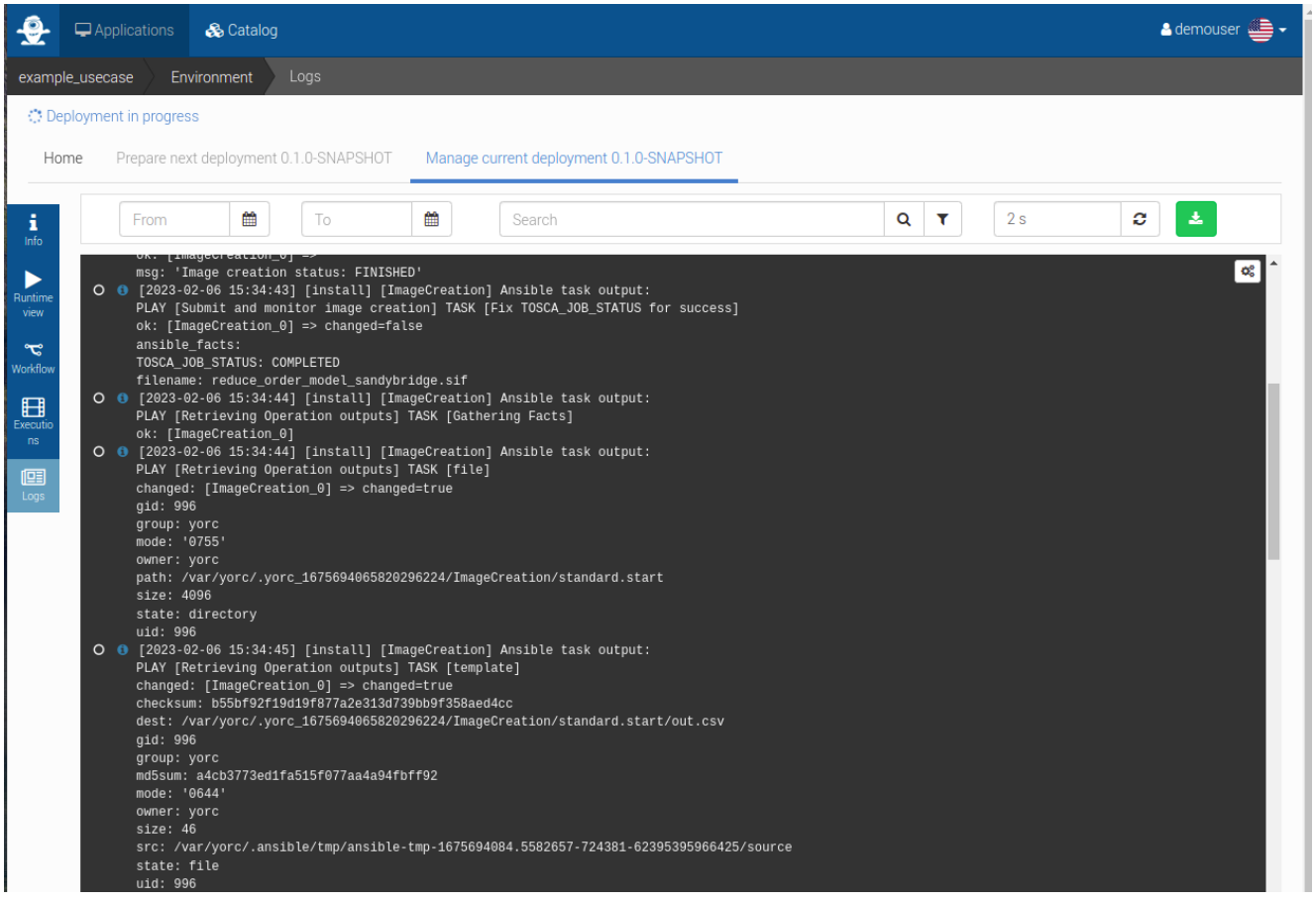


Figure 25: Logs view of a deployment of an application

5.1.5.4 Test a workflow directly from Alien4Cloud

As a workflow developer, it is advisable to perform testing of the workflow prior to making it available to end-users. This can be achieved directly within Alien4Cloud without the need for additional tools. To initiate the testing process, navigate to the `Workflow` tab and select the desired execution workflow from the dropdown menu (see Figure 26). Next, provide the necessary inputs for the workflow and initiate the launch by clicking on the `Launch` button.

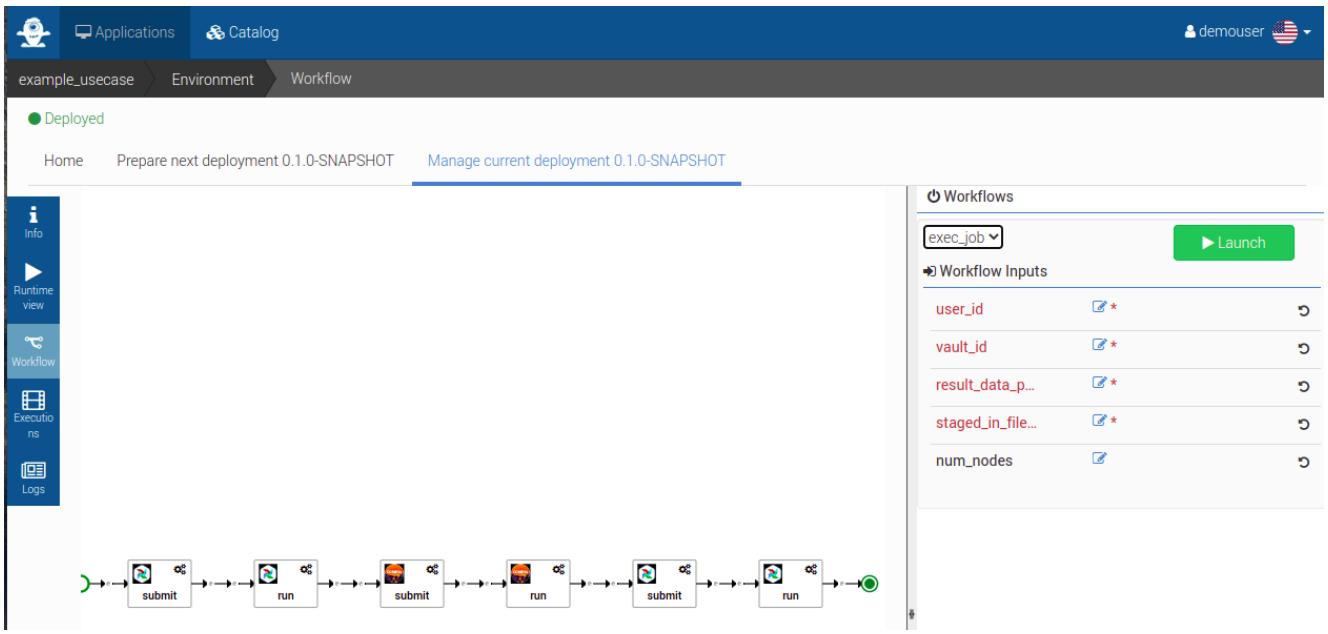


Figure 26: Triggering a workflow for testing purpose

The execution of this workflow can be monitored in a similar manner as previously described in the previous section, by accessing the Manage current deployment tab and monitoring its progress through the `Workflow` or `Logs` tabs.

5.1.5.5 Expose a workflow to the HPCWaaS API

To expose your application to the HPCWaaS API, navigate to the main page of your application by clicking on its name in the top left corner (see Figure 27).

Figure 27: Back to application's main page

Utilize the `Tags` section to configure the interaction between your application and the HPCWaaS API (see Figure 28). The following tags are recognized by the HPCWaaS API:

- `hpcwaas-workflows` represents a list of comma-separated workflows names from your application that should be made available to the API.
- `hpcwaas-authorized-users` refers to a list of comma-separated users who are authorized to utilize this workflow. If this tag is not specified, all authenticated users will have access to the workflow.

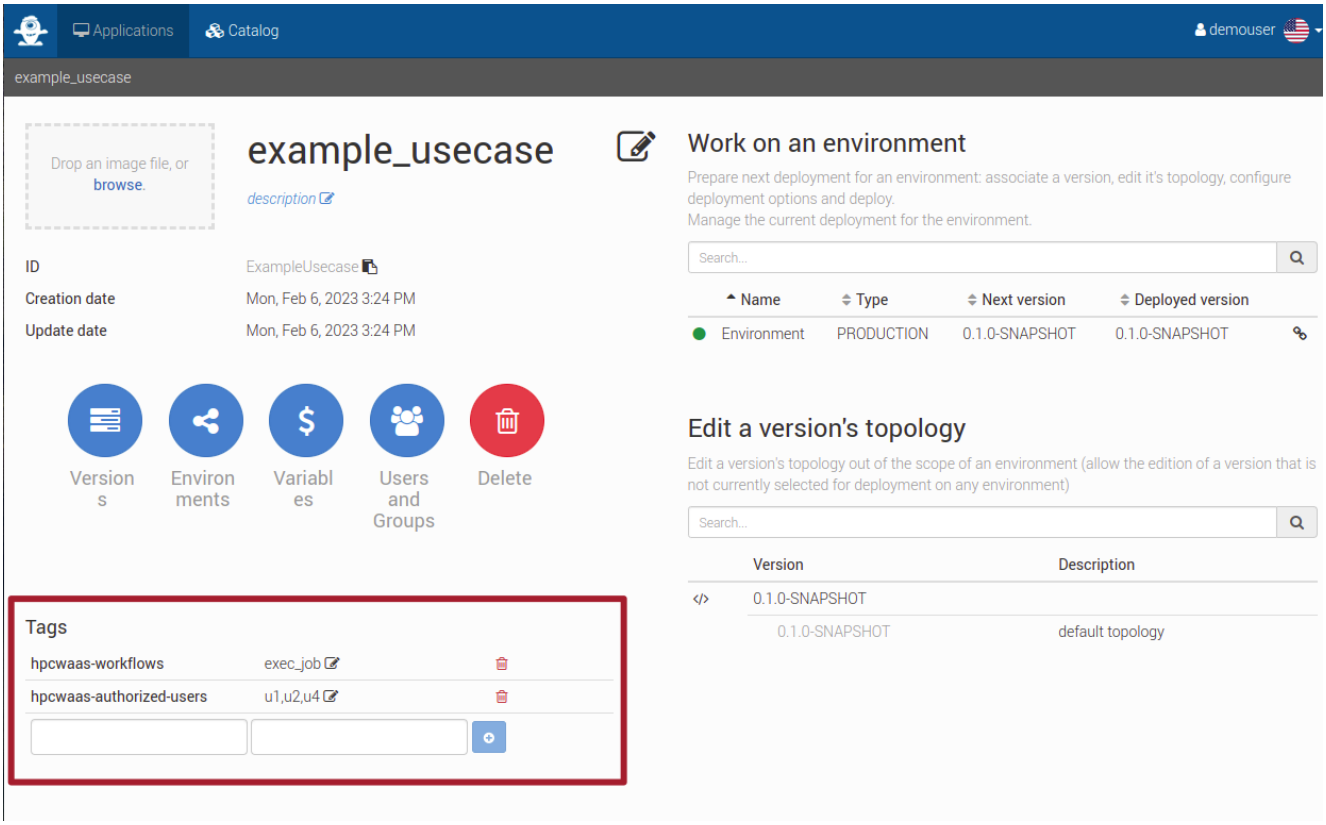


Figure 28: Configure application tags

5.1.6 Credentials setup and Workflow Execution

As a end-user in eFlows4HPC, the process of executing workflows involves defining inputs and triggering the workflow execution via the HPCWaaS API. This section provides guidance for end-users to perform these tasks effectively.

5.1.6.1 Download required tools

Visit the HPCWaaS API release page on GitHub to download a binary version of the `waas` Command Line Interface (CLI) that is compatible with your computer.

Alternatively, a Docker image (ghcr.io/eflows4hpc/hpcwaas-api:main-cli) containing the CLI can also be obtained. To utilize the Docker image, the following command can be executed in the terminal:

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

This is equivalent to executing:

```
./waas help
```

5.1.6.2 Setup your credentials

To enable secure data transfer and execution of workflows, it is necessary to generate a pair of private and public SSH keys using the CLI. The system generates the key pair and securely stores it in a Vault, with the private key being kept confidential and not accessible.

The public key and key ID are returned upon successful key pair generation and should be carefully recorded as they cannot be retrieved later

```
$ ./waas --api_url <api_url> -u <user>:<password> ssh_keys key-gen
INFO: Below is your newly generated SSH public key.
INFO: Take note of it as you will not see it again.
INFO: You are responsible for adding it to the authorized_keys file on the systems you want␣
↪to run your workflows.

INFO: SSH key ID: 31...3f
INFO: SSH Public key: ssh-rsa AAA...mH
```

To grant access to the designated HPC clusters, the SSH Public Key generated during the key pair generation process must be copied to the `authorized_keys` file located in the `.ssh` directory of the user's home directory (`${HOME}/.ssh/authorized_keys`).

5.1.6.3 List available workflows

```
./waas --api_url <api_url> -u <user>:<password> workflows list
```

The above command lists the workflows accessible to you. Take note of the workflow ID of the desired workflow for the next step.

5.1.6.4 Trigger a workflow execution

To initiate the execution of a workflow, you must first determine the inputs required for the workflow from the Workflow developer. Then, execute the following command to trigger the workflow execution:

```
./waas --api_url <api_url> -u <user>:<password> workflows trigger -f \
    -i input1Name=input1Value -i input2Name=input2Value \
    <workflow_id>
```

5.1.6.5 Monitor a workflow execution

In order to monitor a workflow execution, one can use the `-f` flag on the `trigger` command. This flag enables the continuous retrieval of the execution status from the HPCWaaS API.

Alternatively, the execution status can be obtained using the `execution status` command along with the Execution ID, which is returned by the `trigger` command. The syntax for this command is as follows:

```
./waas --api_url <api_url> -u <user>:<password> executions status <Execution_ID>
```

It is to be noted that the `execution status` command also has its own `-f` flag, which can be used for continuously monitoring the execution status.

5.1.6.6 Cancel a workflow execution

You may cancel a workflow execution that is currently in progress by utilizing the `executions cancel` command.

```
./waas --api_url <api_url> -u <user>:<password> executions cancel <Execution_ID>
```