OGC Testbed-16

Earth Observation Application Packages with Jupyter Notebooks

Publication Date: 2021-01-13

Approval Date: 2020-12-15

Submission Date: 2020-11-19

Reference number of this document: OGC 20-035

Reference URL for this document: http://www.opengis.net/doc/PER/t16-D027

Category: OGC Public Engineering Report

Editor: Christophe Noël

Title: OGC Testbed-16: Earth Observation Application Packages with Jupyter Notebooks

---

OGC Public Engineering Report

COPYRIGHT

Copyright © 2021 Open Geospatial Consortium. To obtain additional rights of use, visit http://www.opengeospatial.org/

WARNING

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Public Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER’S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR’s sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the

Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Table of Contents

Chapter 1. Subject

This OGC Testbed-16 Engineering Report (ER) describes all results and experiences from the “Earth Observation Application Packages with Jupyter Notebook” thread of OGC Testbed-16. The aim of this thread was to extend the Earth Observation Applications architecture developed in OGC Testbeds 13, 14, and 15 with support for shared and remotely executed Jupyter Notebooks [https://jupyter.org/]. The Notebooks make use of the Data Access and Processing API (DAPA) developed in the Testbed-16 Data Access and Processing API (DAPA) for Geospatial Data task and tested in joint Technology Integration Experiments.

Chapter 2. Executive Summary

2.1. Problem Statement

Previous OGC Testbeds developed an architecture for deploying and executing data processing applications close to the data products hosted by Earth Observation (EO) platforms. Testbed-16 participants and sponsored wished to complement this approach with applications based on Project Jupyter that enables developing Notebooks.

2.2. Use Cases

For the Testbed-16 (TB-16) demonstrations, the Testbed Call for Participation [https://portal.ogc.org/files/?artifact_id=91644] (CFP) considered three complementary scenarios that were refined as follows:

1. The first scenario explored the possible interactions of Jupyter Notebook with data and processing services through the Data Access and Processing API (DAPA) and various operations (OpenSearch query, GetMap, others).
2. The second scenario explored the interactive and batch mode execution on a hosted Jupyter.
3. The third scenario explored the conversion of Jupyter-based applications into a deployable ADES Application Package. Moreover, the scenario explored the chaining of ADES processes from a Jupyter Notebook.

2.3. Achievements

The TB-16 participants implemented two Jupyter Notebooks and also implemented two ADES endpoints, focusing on the following aspects:

- 52°North targeted the implementation of three Jupyter Notebook (D168) related to water masks in flooding situations, which combined form a single use case.
- Terradue developed a Jupyter Notebook related to volcanology thematic (Testbed 16 task D169) and explored the packaging of Notebooks based on a straightforward workflow from Jupyter Notebooks development to their deployment as a service in an Exploitation Platform supporting the ADES/EMS approach (Testbed 16 task D171).
- Geomatys provided an Application Deployment and Execution Service (ADES) endpoint (Testbed 16 task D170) supporting Jupyter Notebooks and demonstrated related workflows.

2.4. Findings and Recommendations

Section 7 of this ER discusses the challenges and main findings raised during TB-16. First the concepts and technologies that help understanding the Jupyter paradigm and the factors affecting proposed solutions are introduced.

Then, ADES support of non-interactive Notebooks is investigated as the major topic demonstrated during the TB-16 Jupyter task. Thereafter, the key ideas for provisioning interactive Notebooks are tackled. The focus is on highlighting the assets of a Jupyter-based development environment in EO platforms, especially when relying on developer-friendly libraries.

Finally, more challenges and lessons learned are elaborated, emphasizing in particular the great simplicity of code-based workflows implemented using Jupyter Notebooks.

2.5. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Contacts

Name	Organization	Role
Christophe Noël	Spacebel s.a.	Editor
Matthes Rieke	52° North GmbH	Contributor
Pedro Goncalves	Terradue Srl	Contributor
Fabrice Brito	Terradue Srl	Contributor
Guilhem Legal	Geomatys	Contributor

2.6. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 3. References

The following normative documents are referenced in this document.

- OGC: OGC 06-121r9, OGC® Web Services Common Standard, 2010 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2]
- OGC: OGC 14-065r2, OGC Web Processing Service 2.0.2 Interface Standard Corrigendum 2, 2018 [https://portal.opengeospatial.org/files/14-065r2]
- Commonwl.org: Common Workflow Language Specifications, v1.0.2 [https://www.commonwl.org/v1.0/]
- Telespazio: Master System Design Document - EOEPCA.SDD.001, Version 1.0, 2019 [https://eoepca.github.io/master-system-design/published/v1.0/]

Chapter 4. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply. In addition, the following terms and definitions apply.

- **Container**

  a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. (Docker [https://www.docker.com/resources/what-container]).

  ```
  NOTE: This is not a general definition of `container` as might be used in the OGC
  but instead is a definition within the context of this OGC ER.
  ```

- **OpenAPI Document**

  A document (or set of documents) that defines or describes an API. An OpenAPI definition uses and conforms to the OpenAPI Specification [OpenAPI [https://github.com/OAI/OpenAPI-Specification]]

- **OpenSearch**

  Draft specification for web search syndication, originating from Amazon's A9 project and given a corresponding interface binding by the OASIS Search Web Services working group.

- **Service interface**

  Shared boundary between an automated system or human being and another automated system or human being

- **Workflow**

  Automation of a process, in whole or part, during which electronic documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules (source ISO 12651-2:2014)

4.1. Abbreviated terms

- ADES Application Deployment and Execution Service
- AP Application Package
- CFP Call For Participation
- CWL Common Workflow Language
- DWG Domain Working Group
- EMS Execution Management Service
- EO Earth Observation
- ER Engineering Report
- ESA European Space Agency

- GUI Graphical User Interface
- JSON JavaScript Object Notation
- OAS3 OpenAPI 3 Specification
- OSDD OpenSearch Description Document
- OWS OGC Web Services
- REST REpresentational State Transfer
- SNAP SeNtinel Application Platform
- SWG Standards Working Group
- TIE Technology Integration Experiment
- UI User Interface
- URI Uniform Resource Identifier
- URL Uniform Resource Locator
- WFS Web Feature Service
- WPS Web Processing Service
- XP Exploitation

Chapter 5. Overview

Section 6 introduces the project context, background and the initial requirements for the OGC Testbed-16 Earth Observation Application Packages with Jupyter Notebooks activity.

Section 7 discusses the challenges and main findings raised during this TB-16 activity.

Section 8 presents the solutions and the results of the Technology Integration Experiments (TIEs).

Section 9 provides the conclusions and recommendations for future work.

Chapter 6. Context and Requirements

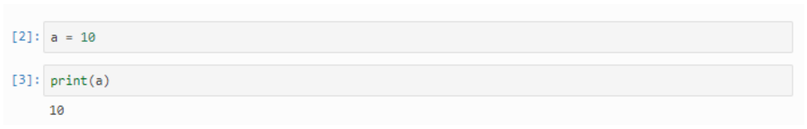
Previous OGC Testbeds defined and developed an Earth Observation Exploitation Platform architecture that enables deploying and executing processing applications close to the input data hosted on a Cloud infrastructure.

The architecture currently allows deploying and executing applications packaged as Docker containers. Testbed-16 intended to **complement this architecture with applications based on Project Jupyter** as detailed below.

6.1. Jupyter Notebooks

The goal of Project Jupyter "is to build open-source tools and create a community that facilitates scientific research, reproducible and open workflows, education, computational narratives, and data analytics". To that end, Jupyter enables developing Notebooks that consist of a web interface combining visual interactive components resulting from the execution of editable programming cells.

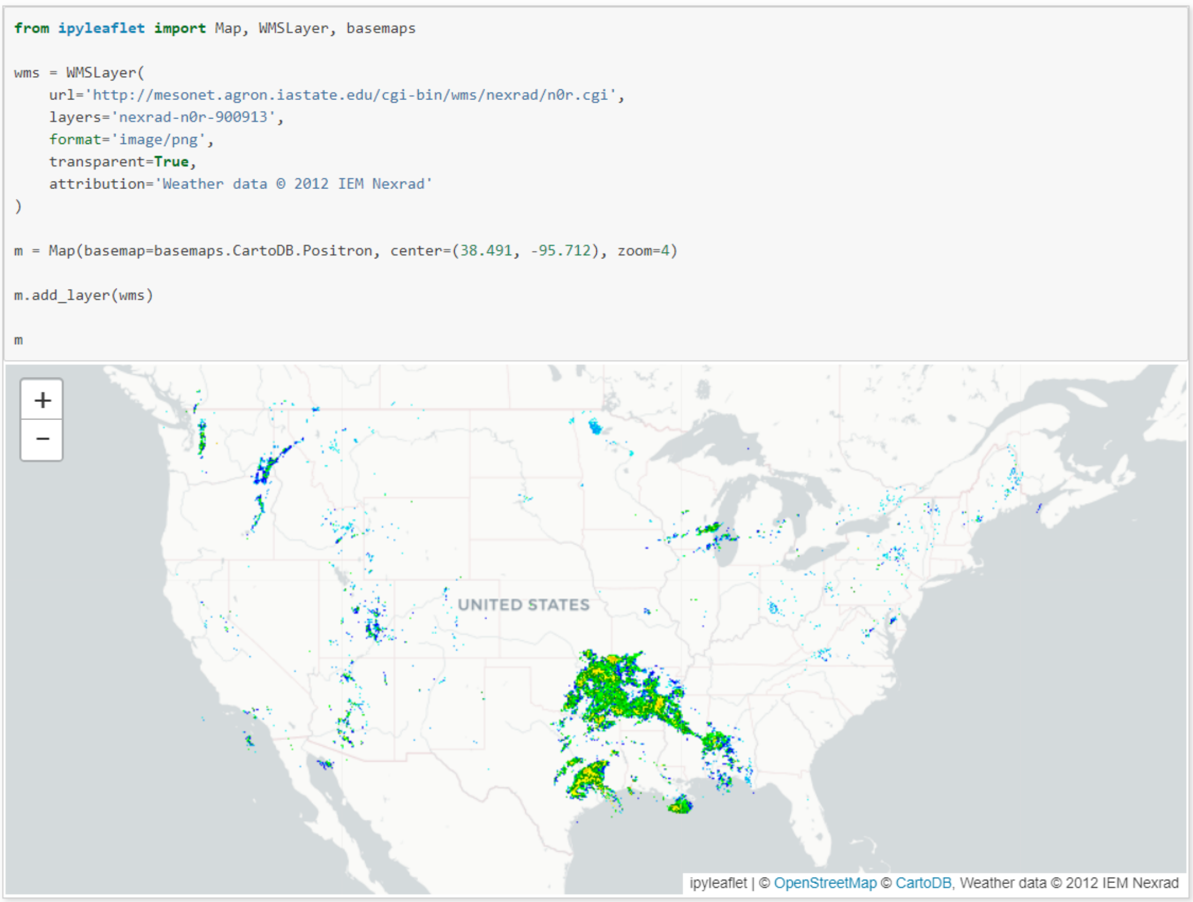
Jupyter features the ability to execute code changes on the fly from the browser, display the result of computation using visual representation, and/or capture the user input from graphical widgets. The figure below illustrates how code cells allow to enter and run code.



```
[2]: a = 10

[3]: print(a)
     10
```

Figure 1. Code cells used to enter and run code

The web-based interactive programming interface can be used to access Earth Observation data or to display results on a map as shown below.



```
from ipyleaflet import Map, WMSLayer, basemaps

wms = WMSLayer(
    url='http://mesonet.agron.iastate.edu/cgi-bin/wms/nexrad/n0r.cgi',
    layers='nexrad-n0r-900913',
    format='image/png',
    transparent=True,
    attribution='Weather data © 2012 IEM Nexrad'
)

m = Map(basemap=basemaps.CartoDB.Positron, center=(38.491, -95.712), zoom=4)

m.add_layer(wms)

m
```

+
−
UNITED STATES
ipyleaflet | © OpenStreetMap © CartoDB, Weather data © 2012 IEM Nexrad

Figure 2. WMS layers displayed from a Jupyter Notebook

The TB-16 Call For Participation (CFP) requested that participants explore two modes for executing Notebooks:

- Batch mode - Allowing the execution without user interface interaction.
- Interactive applications (Web interface) - Can be provided as either a classical Notebook enabling code editions or a rich interactive application hiding source code.

From the CFP: "The Jupyter Notebooks (D168, D169) to be developed will interact with data and processing capacities through the Data Access and Processing API [OGC 20-016]. The Notebooks will perform discovery, exploration, data requests and processing for both raster and vector data. Besides the standard interactive usage, the Jupyter Notebook should be used also as a Web application, hiding code and Notebook cell structure, as well as in batch mode".

6.2. Data Access and Processing API (DAPA)

For this TB-16 task, a new Data Access and Processing API (DAPA) was defined with the goal of developing an end-user optimized data access and processing API. The API was designed to provide simple access to processed samples of large datasets for subsequent analysis within an interactive Jupyter Notebook.

Existing data access web services often require complicated queries and are based on the characteristics of the data sources rather than the user perspective. A key design objective for the DAPA work is to provide a user centric API by making function-calls instead of accessing multiple generic Web services or local files.

The DAPA aims to provide the following capabilities:

- Specific data is bound to specific processing functions.
- Data access is based on the selection of fields requested for a given spatial coverage (point, area, or n-dimension cube) and for a given temporal sampling (instance or interval).
- Data access can be requested with a post-processing function, typically to aggregate the observations (by space and/or time).
- Multiple data encoding formats can be requested.

6.3. Earth Observation Exploitation Platform Architecture

The traditional approach for EO processing consisted of downloading available content to local infrastructure, enabling customers to perform local execution on the fetched data.

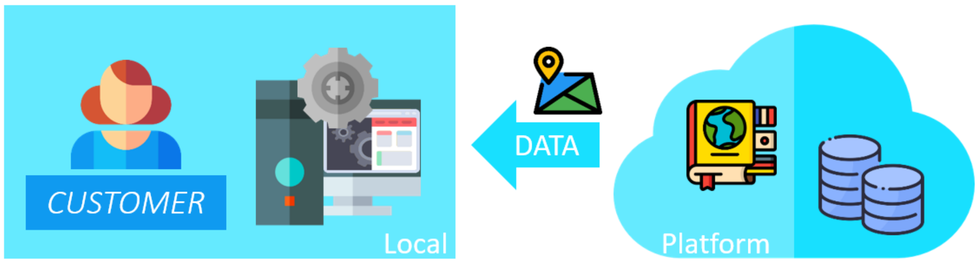


Figure 3. Traditional Approach for Data Processing

OGC Testbed-13, Testbed-14, Testbed-15 EO activities and the OGC Earth Observation Applications Pilot developed an architecture that enables the deployment and execution of applications close to the physical location of the source data. The goal of such architecture is to minimize data transfer between data repositories and application processes.

From a business perspective, the purpose of the proposed platform architecture is to reach additional customers beyond the usual platform Data Consumers. Indeed, Application Providers get offered new facilities to submit their developed applications in the Cloud infrastructure (with a potential reward). Alternatively the platform also provides user-friendly discovery and execution APIs for Application Consumers interested in the products generated by the hosted applications.

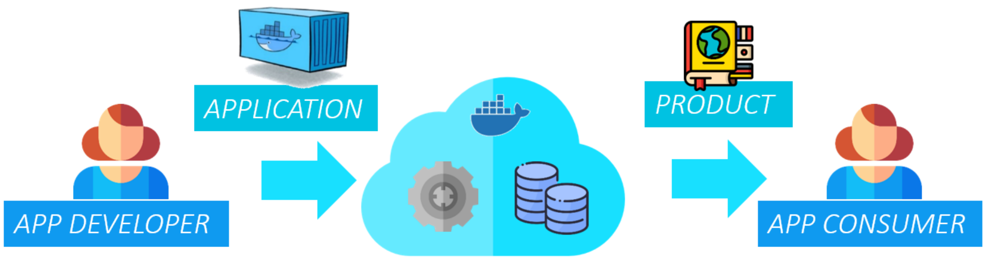


Figure 4. Exploitation Platform Architecture Customers

Two major Engineering Reports describe the architecture:

- OGC 18-049r1 [https://docs.ogc.org/per/18-049r1.html] - Testbed-14: Application Package Engineering Report

- OGC 18-050r1 [https://docs.ogc.org/per/18-050r1.html] - Testbed-14: ADES & EMS Results and Best Practices Engineering Report

Applications are packaged as Docker Images that are deployed as self-contained applications through the Application Deployment and Execution Service (ADES). OGC Web Processing Service (WPS) provides the standard interface to expose all ADES operations (including application deployment, discovery and execution).

Workflows are typically deployed as a CWL (Command Workflow Language) document through the Execution Management Service (EMS). Note that the BPMN alternative has also been explored in OGC 18-085 [https://docs.ogc.org/per/18-085.html].

The applications are described by an Application Descriptor (actually WPS Process Description) that provides all the information required to provide the relevant inputs and start an execution process. The descriptor might potentially be generated automatically when registering the application or can be explicitly provided.

6.4. Project Initial Requirements

The Testbed-16 CFP considered three complementary scenarios that might be modified during the Testbed, as long as the basic characteristics (i.e. data access, processing, and chaining) remain conserved:

1. The first scenario explores the possible interactions of Jupyter Notebooks with data and processing services through the Data Access and Processing API (DAPA) and potentially other OGC API endpoints.
2. The second scenario demonstrates the deployment and execution of Jupyter Notebooks in the EO Exploitation Platform environment. Use cases were prototyped for both an interactive mode and a batch mode execution.
3. The third scenario researched the execution of a workflow including steps with both containerized application and Jupyter Notebook executions. The CFP encouraged exploring other approaches than the CWL and BPMN approaches.

6.5. Project Refined Requirements

During the TB-16 Kick Off and the following weeks, Participants and the Initiative Manager refined the initiative architecture and settled upon specific use cases and interface models to be used as a baseline for prototype component interoperability. The following aspects were agreed:

- The scenario exploring interactions with OGC services must be explored without considering the ADES component.
- The scenario studying the ADES execution must be focused on the batch mode.
- The focus of the workflow aspects should be about the chaining of ADES applications from a (classical) Jupyter Notebook.
- ADES API: The Testbed-14 interface must be aligned to the latest draft version of OGC API - Processes.

The initial requirements were translated in a new set of three EO Application Packages to be demonstrated:

6.5.1. EOAP-1 Interactive Local Jupyter Notebook

The first scenario explored the handling of Jupyter Notebooks and the interaction with data through the Data Access and Processing API (DAPA). The scenario was refined as follows:

1. OpenSearch query (bbox, toi, keywords)
2. GetMap retrieval of map images
3. OpenSearch query (data domain for selected collections)
4. DAPA data request time-averaged map, area-averaged time series from variety of source data type.
5. DAPA data request with bounding polygon from different query.

6.5.2. EOAP-2 Hosted Jupyter Notebook (Alice)

The second scenario explored interactive and batch mode execution on hosted Jupyter. This scenario is described as follows:

1. Publish a Jupyter Notebook on an exploitation (XP) platform for Interactive use.
2. Publish a Jupyter Notebook on an XP platform for a batch mode execution.

6.5.3. EOAP-3 Packaged Jupyter Notebook (Bob)

The third scenario explored:

1. Convert the Jupyter Notebook (batch mode) into a deployable ADES Application Package.
2. Process chaining of ADES processes from a Jupyter Notebook (and possibly other Docker container applications)

Chapter 7. Findings and Lessons Learned

This section first introduces the concepts and technologies that help in understanding the Jupyter paradigm and the factors affecting proposed solutions used in this Testbed. Next, the use of a Notebook with ADES is discussed. This is the major topic demonstrated during this TB-16 activity. Then the key ideas for provisioning interactive Notebooks are presented with a focus on highlighting these assets for EO platforms. Finally, more challenges and lessons learned are elaborated.

7.1. Jupyter Concepts

Jupyter reshapes interactive computing by providing a web-based application able to capture a full computational workflow: developing, documenting, executing code and communicating the results.

Jupyter combines three components: A server, a Notebook document and a kernel. The Jupyter architecture and some specific aspects are detailed below.

7.1.1. Jupyter Notebook Architecture

The architecture of a Jupyter Notebook consists of a server rendering a Notebook document and an interacting programming language kernel:

- The **Notebook server** is responsible for loading the Notebook, rendering the Notebook web user interface and sending (through ZeroMQ messages) the cells of code to the specific execution environment (kernel) when the user runs the cells. The user interface components are updated according to the cell execution result.
- The **kernels** implement a Read-Eval-Print-Loop model for a dedicated programming language, which consists in the following loop: Prompt the user for some code, evaluate the code, then print the result. **IPython** is the reference Jupyter kernel, providing a powerful environment for interactive computing in Python.
- The **Notebook document** essentially holds the metadata, the markdown cells, and the code cells that contain the source code in the language of the associated kernel. The official Jupyter Notebook format is defined with a JSON schema (https://github.com/jupyter/nbformat/blob/master/nbformat/v4/nbformat.v4.schema.json).

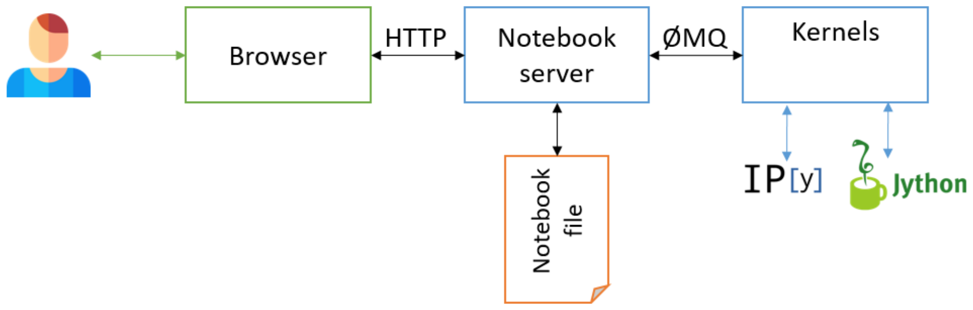


Figure 5. Jupyter Notebook Architecture

7.1.2. Jupyter Kernels and Environment

Jupyter **kernels** are the programming-language specific processes that run the Notebook cells code. IPython is the reference Jupyter kernel, providing a powerful environment for interactive computing in Python. A large set of kernels for other languages are available.

With administrator rights, kernels can be added to the Jupyter host using python or conda. **Conda** is an open-source cross-platform package and environment manager system for any language (e.g. Python, R, Java). The **conda** command easily creates, saves, loads and switches between environments.

The following commands create a conda environment, install a Python2 kernel in the environment, then register the conda environment as a kernel in Jupyter:

```
conda create -n py2 python=2 ipykernel
conda run -n py2 -- ipython kernel install
python -m ipykernel install --user --name py2 --display-name "Python (py2)"
```

A Notebook might depend on specific software (e.g. GDAL, Orfeo Toolboox, geopandas) used from the Jupyter kernel. Depending on the configured permissions, those **dependencies** can be installed either from the Jupyter terminal, or using system commands directly from the Notebook (enabling others users to repeat the installation commands). The environment is persisted in the user environment of the Jupyter host.

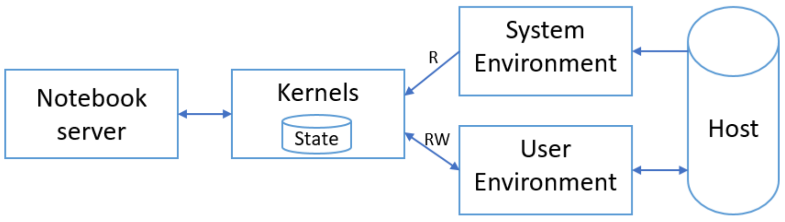


Figure 6. Kernel Access to Host Environment

The usual means for installing dependencies from a Notebook are through `pip` commands (python libraries) and `conda` (virtual environment).

Conda environments can be registered as a Jupyter kernel. The environment is specified in a configuration file (see example below) listing a set of channels (repositories hosting the packages - potentially custom software -) and dependencies.

Conda Environment Example

```yaml
name: my_environment_example
channels:
  - conda-ogc
dependencies:
  - otb
  - gdal
  - ipykernel
  - geopandas
```

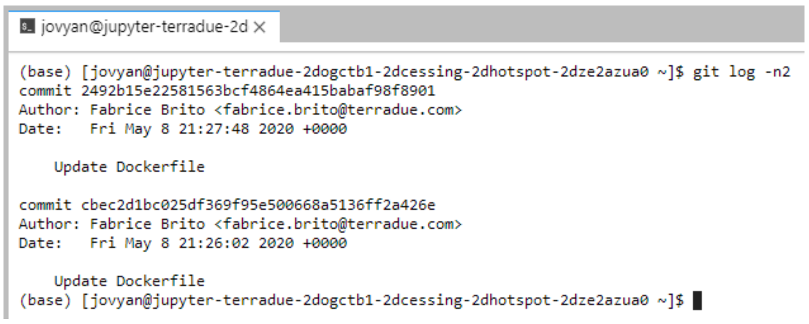
To create an environment from an environment.yml file, the following command should be used:

```
conda env create -f environment.yml
```

The kernel maintains the state of a Notebook's computations. When restarting the kernel, **the Notebook reset its states** and therefore looses all results of Notebook cells. However, the dependencies installed using system commands generally remains in the user environment depending on the specific implementation.

7.1.3. Notebooks Version Control

From any Jupyter session, it is possible to start a terminal and execute commands in the kernel. In particular, `git` commands can be used to **pull a repository and push changes**.



```
(base) [jovyan@jupyter-terradue-2dogctb1-2dcessing-2dhotspot-2dze2azua0 ~]$ git log -n2
commit 2492b15e22581563bcf4864ea415babaf98f8901
Author: Fabrice Brito <fabrice.brito@terradue.com>
Date:   Fri May 8 21:27:48 2020 +0000

    Update Dockerfile

commit cbec2d1bc025df369f95e500668a5136ff2a426e
Author: Fabrice Brito <fabrice.brito@terradue.com>
Date:   Fri May 8 21:26:02 2020 +0000

    Update Dockerfile
(base) [jovyan@jupyter-terradue-2dogctb1-2dcessing-2dhotspot-2dze2azua0 ~]$ █
```

Figure 7. JupyterLab Terminal

As illustrated below, from the terminal, the Notebook documents and the environment can be saved and retrieved from Git repositories (1). Then the environment can be provisioned using `pip` or `conda` (2).

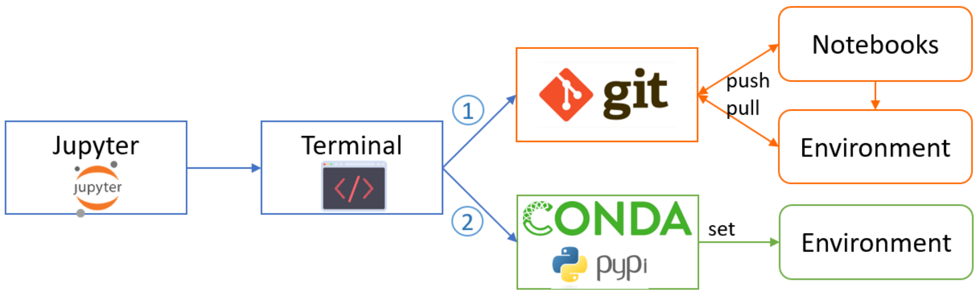


Figure 8. JupyterLab Configuration Management

In order to push back changes on the Notebook, the user can execute a git command using JupyterLab terminal, and potentially push the change to a user private repository. Therefore, as illustrated below, multiple users might collaborate on a shared Notebook repository.

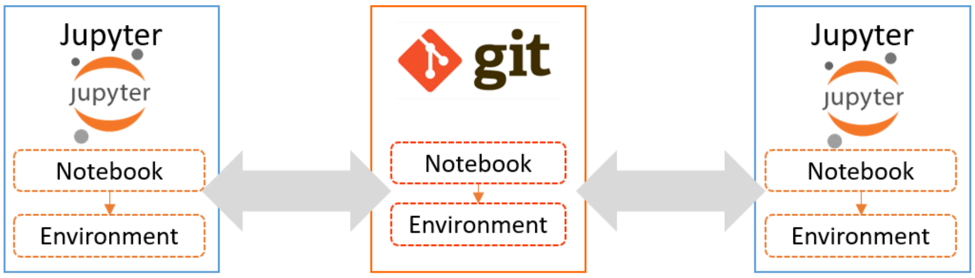


Figure 9. Notebook Collaboration using Git

7.1.4. Security Considerations

Security is a big challenge with sharing Notebooks. The potential for threat actors to attempt to exploit the Notebook for nefarious purposes is increasing with the growing popularity of Jupyter Notebook.

From the users' perspective, the Notebook server shall prevent untrusted code to be executed on user's behalf when the Notebook is opened. In particular, the security model of Jupyter is based on a trust attribute (a signature computed from a digest of the Notebook's contents plus a secret key) that prevents executing any untrusted code and sanitize the HTML source code. The Notebook's trust is updated when the user explicitly decides to execute a specific cell.

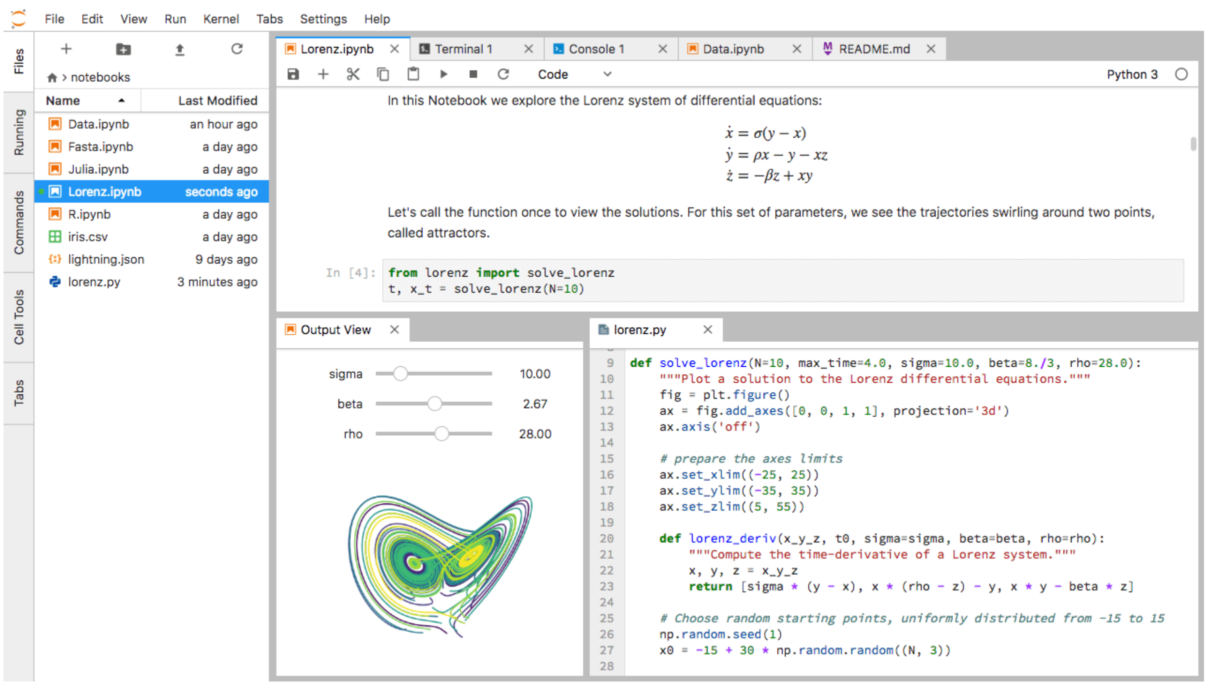
From the platform point of view, the security challenge implies offering the interactivity of a Notebook without allowing arbitrary code execution on the system by the end-user. In particular, restricting the access to the Notebook server is important. As the Notebook user might execute local system commands, the vulnerability of the system should be carefully reviewed - in particular regarding all the write permissions granted on the various offered resources.

7.2. Jupyter Notebook Technologies Review

A large set of technologies related to Jupyter Notebook have been explored and are detailed below. In particular, multiple solutions were considered for provisioning Jupyter Notebook environments.

7.2.1. Single-User Jupyter Notebook (JupyterLab)

In its simplest form, a Jupyter Notebook is simply executed by a kernel managed by a Jupyter server such as JupyterLab. **JupyterLab** provides the familiar building blocks of the classic Jupyter Notebook (Notebook, terminal, text editor, file browser, rich outputs, etc.) in a more flexible user interface and enabling plugin extensions.



Name	Last Modified
Data.ipynb	an hour ago
Fasta.ipynb	a day ago
Julia.ipynb	a day ago
Lorenz.ipynb	seconds ago
R.ipynb	a day ago
iris.csv	a day ago
lightning.json	9 days ago
lorenz.py	3 minutes ago

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

```
In [4]: from lorenz import solve_lorenz
        t, x_t = solve_lorenz(N=10)
```

```
sigma   10.00
beta     2.67
rho     28.00
```

```python
def solve_lorenz(N=10, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

```

Figure 10. JupyterLab

The JupyterLab interface consists of a main work area containing tabs of documents and activities, a collapsible left sidebar, and a menu bar. The left sidebar contains a file browser, the list of running kernels and terminals, the command palette, the Notebook cell tools inspector, and the tabs list.

TIP JupyterLab is relevant for local development of a Notebook but requires manual steps to share Notebooks on repositories and to provision the customized Notebook kernels.

7.2.2. Multi-User Jupyter Notebook (JupyterHub)

Sharing a Notebook server such as JupyterLab would make the concurrent users' commands collide and overwrite each other as the Notebook has **exactly one** interactive session connected to a kernel. In order to provision instances of a Jupyter Notebook and share the Notebook, a **hub serving multiple users** is required to spawn, manage, and proxy multiple instances of the single-user Jupyter Notebook server.

JupyterHub allows creating Jupyter single-user servers that guarantee the **isolation of the user Notebook servers**. The Notebook is copied to ensure the original deployed Notebook is preserved. JupyterHub includes JupyterLab and thus also supports the provisioning of the environment using Conda.

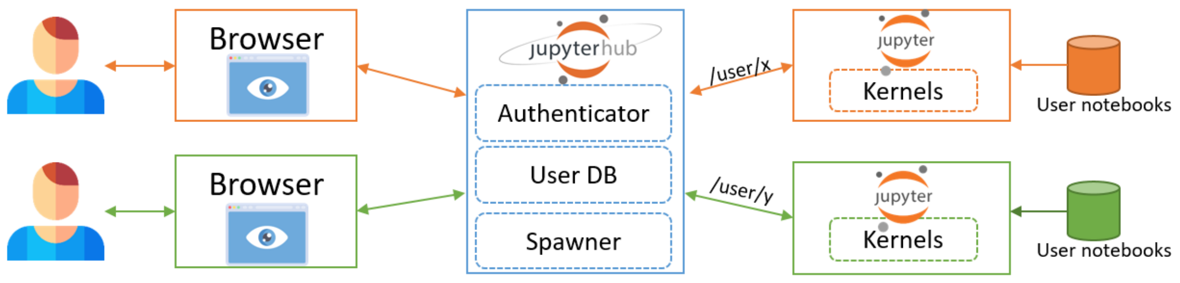


Figure 11. JupyterHub Architecture

Although nothing prevents a side component managing the sharing of Notebook documents, this support goes beyond the provided capabilities. Indeed, only extensions of JupyterHub provides Notebook sharing support.

TIP JupyterHub brings multi-users support and Jupyter isolation but it does not provide support for sharing Notebooks.

7.2.3. Containerized Jupyter Notebook (repo2Docker)

Full reproducibility requires the possibility of recreating the system that was originally used to generate the results. This can, to a large extent, be accomplished by using Conda to make a project environment with specific versions of the packages that are needed in the project. The limitations of Conda are reached when complex software installation is required (e.g. a specific configuration of a tool).

For the support of sophisticated systems, *repo2Docker* can be used to build a reproducible container that can be executed anywhere by a Docker engine. *repo2Docker* can build a computational environment for any repository that follows the Reproducible Execution Environment Specification (https://repo2Docker.readthedocs.io/en/latest/specification.html#specification).

As illustrated on the figure below, for building a Jupyter Notebook container, *repo2Docker* must be used from a repository that consists of the Notebook resources and a Dockerfile that builds the environment and install the Jupyter kernel.

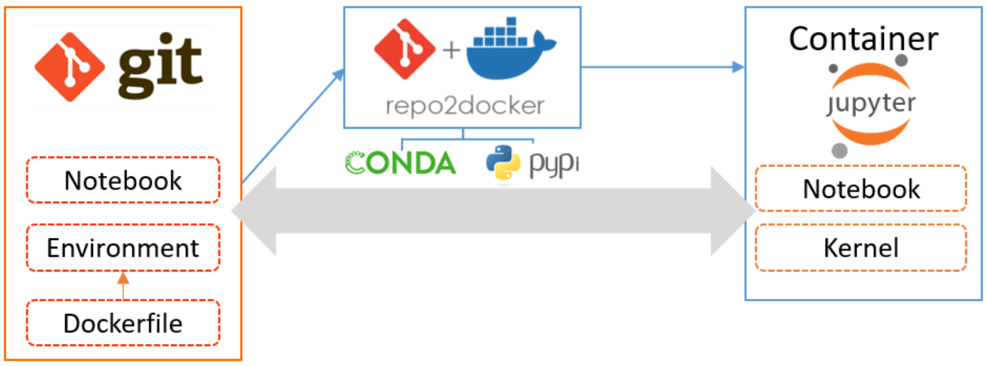


Figure 12. Repo2Docker Process

Assuming that the Jupyter kernel is installed in the container, the JupyterLab interface can be launched from within a user #session without additional configuration by simply appending /lab to the end of the URL like so:

```
http(s)://<server:port>/lab
```

Note that, the DockerSpawner (https://jupyterhub-Dockerspawner.readthedocs.io/en/latest/) from JupyterHub allows spawning Notebooks provisioned from a Docker image.

TIP

Supporting Docker provides a total flexibility for defining the Jupyter environment (e.g. if a custom GDAL configuration is needed to execute the Notebook). repo2Docker allows a user to reproduce the whole Jupyter environment in a container that can be deployed in any Docker engine.

7.2.4. Repository Based Jupyter Notebook (BinderHub)

BinderHub is an open-source tool that enables deploying a custom computing environment described on a Git repository and make it available (from an associated URL) by many remote users. A common use of Binder is the sharing of Jupyter Notebooks.

As illustrated on figure below, BinderHub combines the tools described above to generate the Notebook: **Repo2Docker** generates a Docker image from the Git repository (potentially setting a conda environment), and the JupyterLab container is provisioned online based on JupyterHub running on Kubernetes.

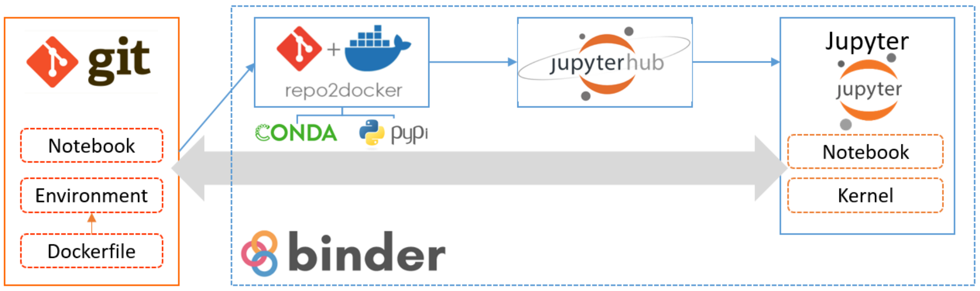
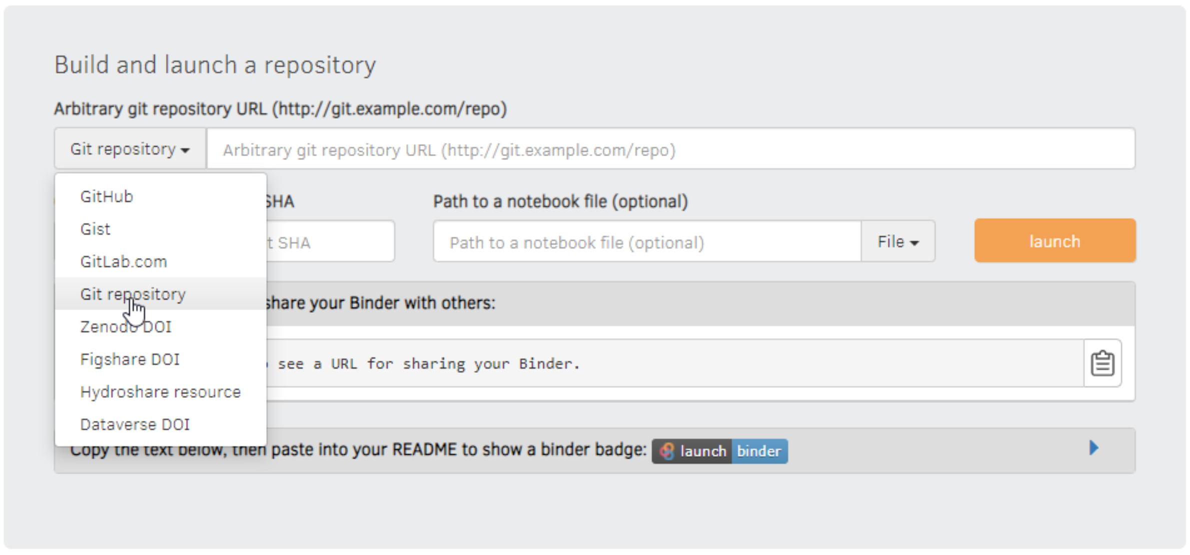


Figure 13. BinderHub Architecture

The Binder project also provides **mybinder.org**, a pre-existing and free to use BinderHub deployment used during this project. However, only a private BinderHub deployment might support accessing private Git repositories.

In order to share a Notebook developed locally with other developers, the Git repository can simply be linked with Binder for provisioning an online Notebook. On mybinder.org, the repository is turned to a Notebook by filling the form shown below.



Build and launch a repository

Arbitrary git repository URL (http://git.example.com/repo)

Git repository ▾ | Arbitrary git repository URL (http://git.example.com/repo)

GitHub
Gist
GitLab.com
Git repository
Zenodo DOI
Figshare DOI
Hydroshare resource
Dataverse DOI

SHA | t SHA

Path to a notebook file (optional) | Path to a notebook file (optional) | File ▾ | launch

share your Binder with others:

```
 see a URL for sharing your Binder.
```

Copy the text below, then paste into your README to show a binder badge: launch binder

Figure 14. Binder - Turn a Git repository into a Notebook

Binder is meant for interactive and ephemeral interactive coding. This means that the code is ideally suited for relatively short sessions that will automatically shut down after 10 minutes of inactivity (default configuration). The creation of a Binder session might take a long time as the following chain of events happens when a user clicks a Binder link:

1. BinderHub resolves the link to the repository.
2. Based on the Git repository hash, BinderHub determines if a Docker image already exists (if the Notebook uses a custom container).
3. If the image doesn’t exist, BinderHub uses repo2Docker to fetch the repository, build the Docker image and push it to the local Docker registry. BinderHub then communicates to JupyterHub the Docker image reference to provision a new server.
4. JupyterHub creates, if needed, a node for hosting the container.
5. The node pulls the Docker image, if not already present, and starts the Jupyter Notebook.

TIP Binder provisions a Notebook from a repository with its full environment but is essentially meant for provisioning ephemeral instances as it automatically shut down user sessions after 10 minutes of inactivity. Binder still does not enable collaborative work on a shared Notebook.

7.2.5. Hosted Jupyter Notebook

A set of Jupyter Notebook hosting solutions are available such as **Microsoft Azure Notebooks**, **Amazon EMR Notebook**, and **OnlineNotebook.net**. The online Jupyter hosting solutions enable Notebook collaboration and support considerable capability to extend the environment with additional libraries (e.g. pip, conda) but is, however, limited without the Dockerfile support. Moreover, free solutions are of course restricted by the availability of hardware resources.

In particular, **Google Colaboratory** (or "Colab") supports the import of libraries using pip and apt-get, the synchronization with Git repositories, and the access control management. Note also that Colab introduces new concepts and deviations from the original Jupyter interface.

Cocalc stands out from other hosting solutions with the support of real-time collaboration. However, Dockerfiles are also not supported.

7.2.6. Static Jupyter Notebook

nbviewer is a web application that lets you enter the URL of a Jupyter Notebook file, renders that Notebook as a static HTML web page, and gives you a stable link to that page which you can share with others. nbviewer also supports browsing collections of Notebooks (e.g., in a GitHub repository) and rendering Notebooks in other formats (e.g., slides, scripts).

In contrast to JupyterLab, *nbviewer* does not execute Notebooks. It only renders in a web page the inputs and outputs saved in a Notebook document.

TIP *nbviewer* renders statically (HTML, PDF, etc.) the saved inputs and outputs of a Notebook document.

7.3. Execution of Notebooks on ADES

The major objective of this TB-16 activity was about consolidating the OGC EO Cloud architecture with support of Jupyter-based applications. The capability to register and execute a Notebook on the Application Deployment and Execution Service (ADES) was therefore the key aspect that was demonstrated.

However, the real asset Jupyter Notebook would be to provide an interactive development environment on the EO platforms, offering all facilities for interacting with the offering data and processing services and developing application within the specific platform. Those considerations are addressed in the next section.

The successful deployment and execution of Jupyter Notebook on a target ADES depends on the following requirements:

- A command-line interface for running the container in batch mode (while Notebook are originally interactive applications).
- A Docker Application Package for deploying the container (in particular, a Docker Image and a Process Description or CWL).

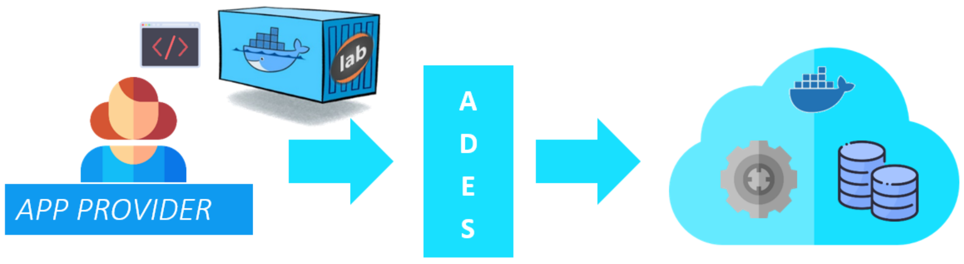


Figure 15. ADES Execution of a Notebook

7.3.1. Providing a Command-Line Interface

Papermill [https://papermill.readthedocs.io] was considered for enabling the execution of a Notebook in batch mode. An alternative solution based on nbconvert and Python dictionaries was also investigated.

7.3.1.1. Papermill Approach

Papermill is a tool providing a command-line interface for executing Notebooks in batch mode and returning outputs.

Papermill requires designating the parameters that should be submitted as command-line arguments. The parameterization relies on cell metadata and requires adding the *parameters* tag to the cell that contains the relevant Notebook variables:

```
"tags":["parameters"]
```

The command line interface offers a set of options [https://papermill.readthedocs.io/en/latest/usage-cli.html]: the Notebook path, the output path and the key-value pairs of arguments:

```
Usage: papermill [OPTIONS] NOTEBOOK_PATH OUTPUT_PATH -p <parameter_name>
<parameter_value>
```

As no code is required, *Papermill* thus provides a very straightforward means for exposing a Notebook with a command-line interface.

7.3.1.2. NbConvert Approach

With the aim of executing a Notebook on an ADES, an alternative method was explored, combining the parameterization of the Notebook with the description of the parameters which are **required for the ADES Application Package**.

The approach requires describing the parameters using Python dictionaries expanded with the information needed by the ADES component. In the example below, the identifier, title, and abstract of a parameter are provided:

```
aoi_wkt = dict([('identifier', 'aoi'),
                ('value', 'POLYGON((14.683 37.466,14.683 37.923,15.29 37.923,15.29
37.466,14.683 37.466))'),
                ('title', 'Area of interest in WKT or bbox'),
                ('abstract', 'Area of interest using a polygon in Well-Known-Text
format or bounding box'),
                ('maxOccurs', '1')])
```

The execution of the Notebook is done from a coded or generated application based on *nbconvert* which is used to perform the following actions:

1. Parse the Notebook.
2. Replace the parameter values with the command-line inputs.
3. Execute the Notebooks

In order to assist the nbconvert based solution, a utility command may be implemented to generate a command-line application based on nbconvert described above.

7.3.2. Building the Application Package

Notebooks can be deployed and executed by the ADES component using the **Docker Application Package** which is basically a Docker Image referenced in an Application Descriptor (e.g. typically based on CWL).

Independent of the command-line approaches described above, repo2Docker can be used for building the container embedding the Notebook application. Defining the container built from either a Conda environment or a Dockerfile offers enough flexibility for encapsulating any tool or library including *papermill* or the nbconvert application.

In particular, using the *papermill* approach, the following steps are required to prepare the Docker Application Package:

1. Define the arguments in the flagged cell;
2. Add the *papermill* dependency to the container (e.g. using command *python3 -m pip install papermill*);
3. Define the Application Descriptor with *papermill* as container base command.

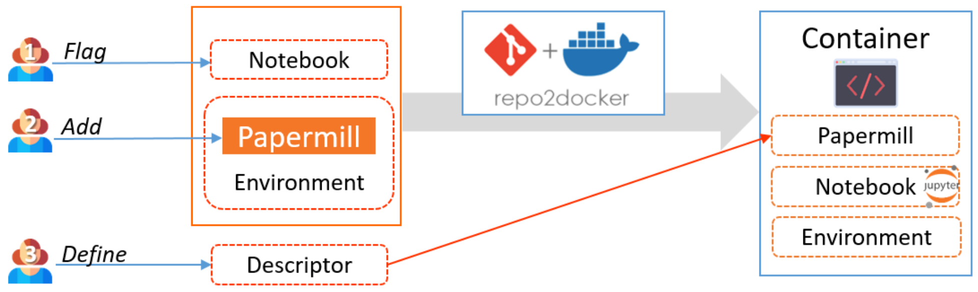


Figure 16. Building Package in Papermill Approach

On the other side, using the *nbconvert* approach, the following steps are required to prepare the Docker Application Package:

1. Describe the parameters using Python dictionaries;
2. Implement or generate (with a dedicated tool) the command-line application that uses nbconvert for parsing and running the Notebook and add the dependency to the container;
3. Add the *nbconvert* dependency to the container;
4. Define or generate the Application Descriptor with the nbconvert-based application as based command.

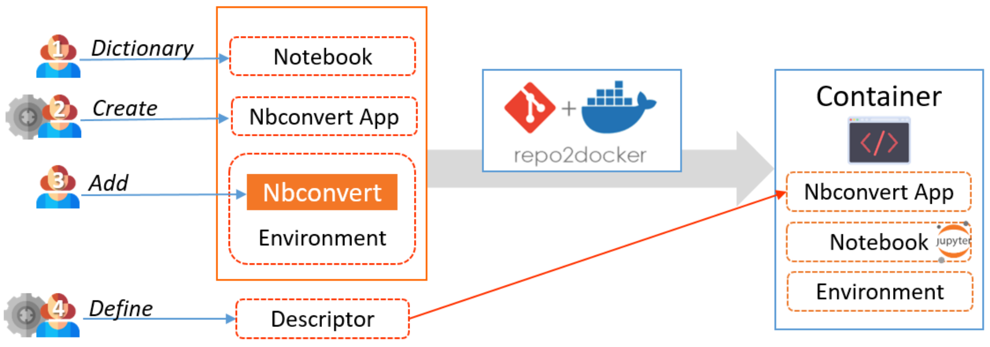


Figure 17. Building Package in NbConvert Approach

As mentioned above, a **dedicated tool** can be implemented to **generate** the command-line application and the Application Descriptor both from the Python dictionary. Note also that the possibility of **combining a dictionary** definition **with the *papermill*** approach was not investigated but would be feasible.

7.3.3. Interoperability Aspects

Since the ADES application package is based on containers, the **interoperability** for deploying the application is **ensured whatever command-line approach is used** (i.e. *papermill* or *nbconvert*).

However, in addition to the deployment packaging format, the execution concerns on the target platforms should be considered on different levels:

- Are all libraries and tools used by the Notebook available on the target platform? The build of the container should ensure that all software used on the development environment is encapsulated in the Docker image (using repo2Docker).
- Is the data required by the application available on the target platform? The data might be considered as a runtime parameter, which assumes that the inputs submitted by the client are referenced from the specific platform.
- Are the services APIs invoked by the Notebook offered on the remote platform? Indeed, the execution can only work if services queried by the Notebook are also offered on the second platform.

The coupling of the Notebook libraries with the platform offered services is illustrated on figure below.

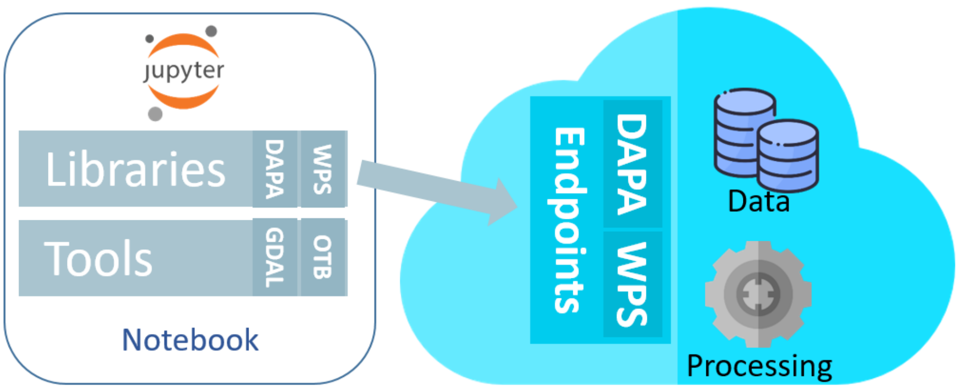


Figure 18. Notebook Interoperability

As proposed in the Master System Design Document (MSDD [https://eoepca.github.io/master-system-design/published/v1.0/]) of the EO Exploitation Platform Common Architecture, the Platform might declare its capabilities (e.g. support APIs, file formats, etc.) in a well-known 'bootstrap' URL. Additionally, the Application Descriptor may be enhanced for including all capabilities requirements, thus enabling the platform to detect conflicting deployments.

7.3.4. STAC Manifest

As in the OGC EO Apps Pilot project, support for a manifest with inputs and outputs based on the STAC specification was provided.

The **SpatioTemporal Asset Catalog** (STAC) specification provides a common language to describe a range of geospatial information and standardizes the way geospatial assets are exposed online and discovered. A SpatioTemporal Asset is a document representing information about the earth captured in a certain space and time including imagery (satellite, aerial, drone, etc.), derived data products and alternative geospatial captures. However, compared to a classic catalogue such as OpenSearch, STAC does not support specifying any search operation.

STAC is actually composed of three component specifications that together make up the core STAC specification but can be used independently in other contexts:

- The Item Specification defines a STAC Item, the core atomic unit representing a single spatiotemporal GeoJSON feature with links to related entities and assets.
- The Catalog Specification specifies a structure to link various STAC Items together to be crawled or browsed. The division of sub-catalogs is up to the implementor, but ease of browsing is a focus.
- The Collection Specification is collection of STAC Items that is made available by a data provider. This enables discovery at a higher level than individual items.

STAC defines only a minimal core and is designed for extensions. Extensions define the domain-specific fields that can be easily added to any STAC Item. This approach is reflected by the already available extensions such as: Data Cube, Electro-Optical, Project, SAR, Satellite, Tiled Assets, etc.

STAC is used as a solution to handle the discrepancy between platforms related to the format of EO products. For example, the Sentinel-2 SAFE format, illustrated on figure below, might be provided in a compressed archive on some platforms, with a different structure, or even a different format. Moreover, STAC can be used to provide additional information (spatial footprint, masks, bands, orbit direction, etc.) about both submitted inputs and generated outputs.

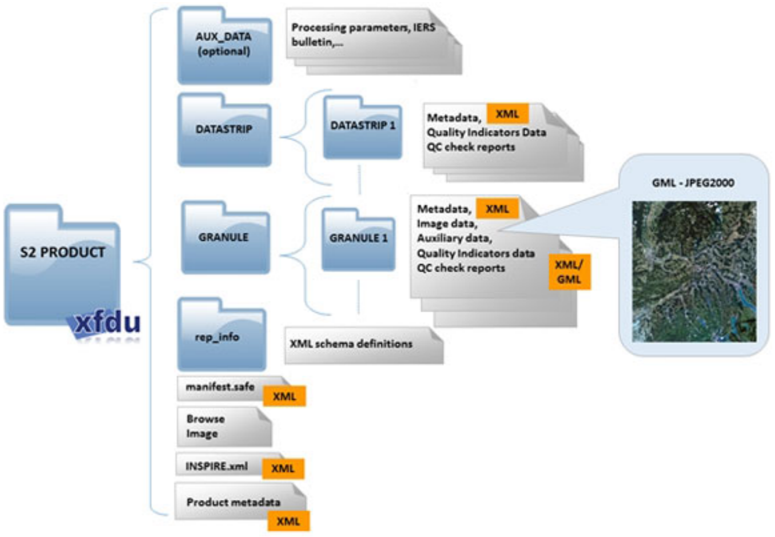


Figure 19. Sentinel-2 SAFE format

In summary, STAC facilitates the data stage-in and stage-out by conveying useful information between ADES and the Applications. The downside is the need to support the STAC profile in the application.

7.4. Interactive Jupyter Notebooks in EO Exploitation Platform

The previous section investigated the challenge of packaging Jupyter-based applications for deployment and execution on the ADES. Of course, the **real asset** of Notebooks relates to the **interactive** programming environment that eases the sharing and **collaboration** on directly editable applications.

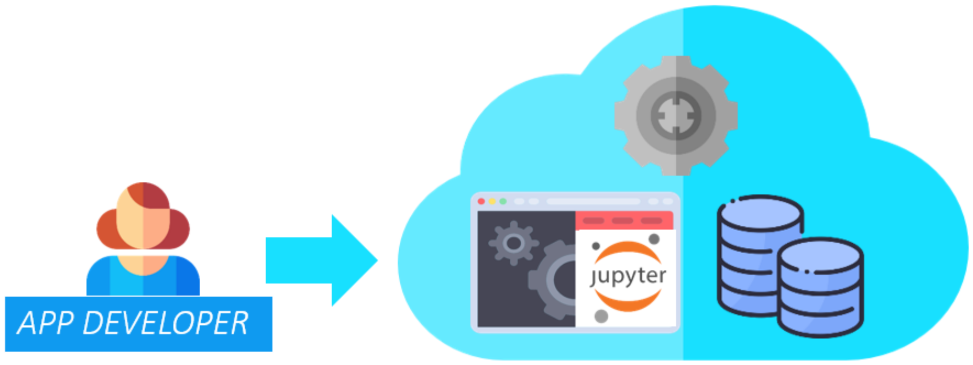


Figure 20. Exploitation Platform extended with Interactive Notebook support.

A great advantage of developing from within the Cloud platform is the possibility providing **user-friendly API libraries** that interact with the platform offered services. From the Notebook, developers can interactively implement the application and iteratively test the various steps including data discovery, data access and data processing.

Finally, the platform may provide support for the collaboration on shared Notebooks and even automate the build and packaging for an ADES deployment. In summary, Jupyter would provide an ideal environment development that plays a part in the already defined architecture, and also extends the architecture with new outstanding capabilities.

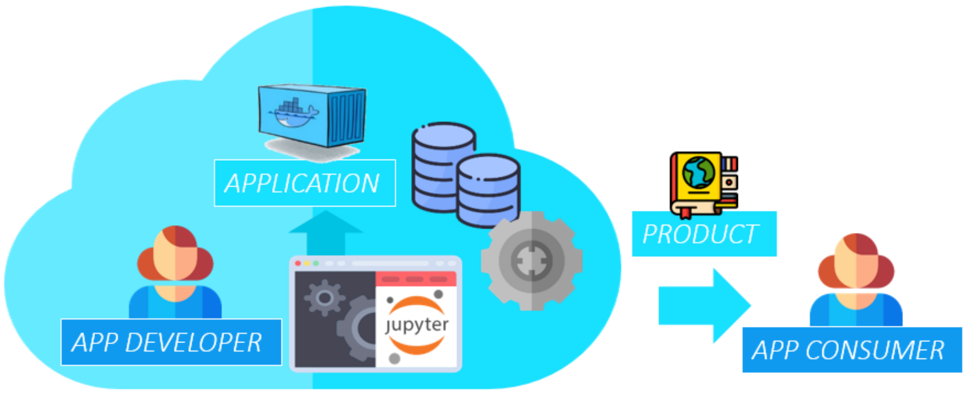


Figure 21. Jupyter Development Environment within the platform

This section provides recommendations for supporting interactive Jupyter Notebooks on an EO XP Platform, and aims to address the following questions:

- What are the benefits of developing Notebook in an EO XP Platform environment?
- How an interactive Jupyter Notebook server can be provisioned to the users of an EO XP Platform?

- How to ease sharing of a Notebook application between multiple users?
- How to facilitate collaboration on a Notebook?
- How to secure the Jupyter environment?

7.4.1. Examples of Interactive Application on EO Platforms

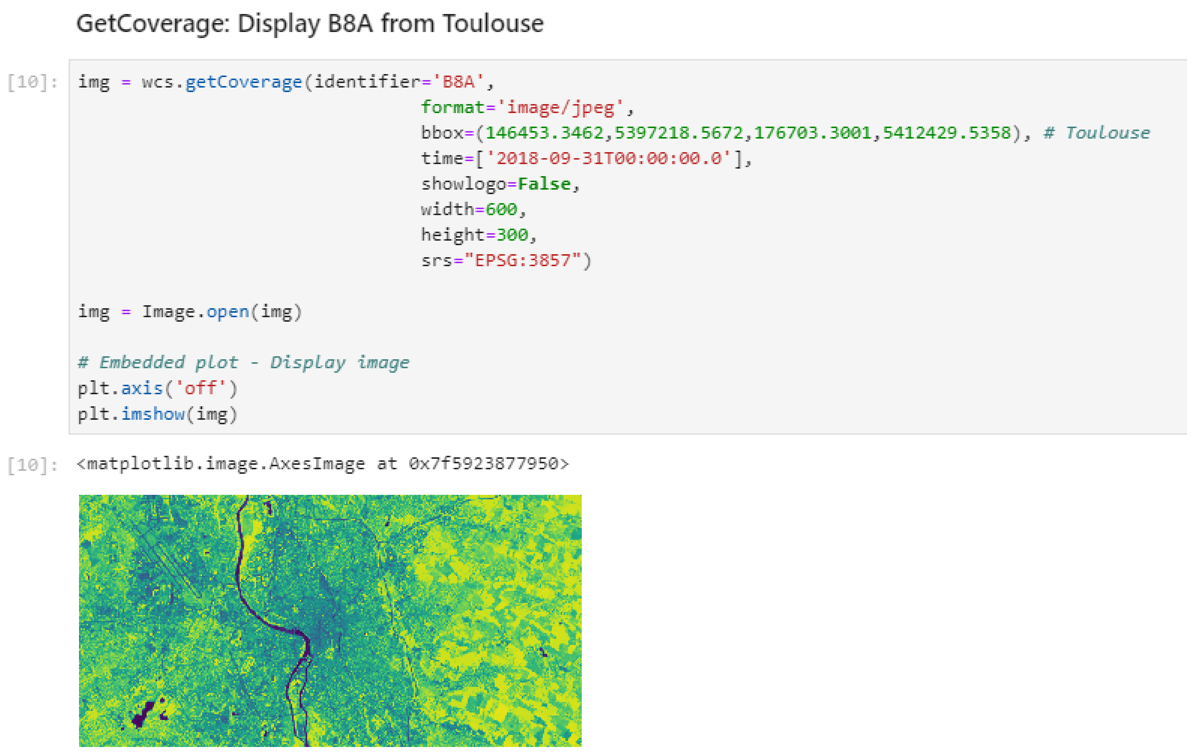
Several EO platforms already offer a Jupyter Notebook environment with built-in tools and API for operating the supported service. Some examples are Mundi (DIAS), CREODIAS, Open Data Cube, and Sentinel-Hub.

7.4.1.1. Mundi DIAS

The Mundi Jupyter Notebook has pre-installed tools enhanced to easily discover Mundi's services and enables an efficient access to data. Mundi offers the two kernel languages: Python 3 and R.

Mundi provides a Python API library (from mundilib import MundiCatalogue) that allows the user to query a large set of services (CSW, OpenSearch, WMS, WCS, WMTS) and tools (GDAL, matplotlib, etc.) without caring about the HTTP REST interface details.

The example below shows the retrieval of coverage crop of the band 8 from the Sentinel-2 L1C collection which is then transformed to an image using matplotlib:



GetCoverage: Display B8A from Toulouse

```
[10]: img = wcs.getCoverage(identifier='B8A',
                               format='image/jpeg',
                               bbox=(146453.3462,5397218.5672,176703.3001,5412429.5358), # Toulouse
                               time=['2018-09-31T00:00:00.0'],
                               showlogo=False,
                               width=600,
                               height=300,
                               srs="EPSG:3857")

      img = Image.open(img)

      # Embedded plot - Display image
      plt.axis('off')
      plt.imshow(img)

[10]: <matplotlib.image.AxesImage at 0x7f5923877950>
```

Figure 22. Notebook cell on Mundi DIAS

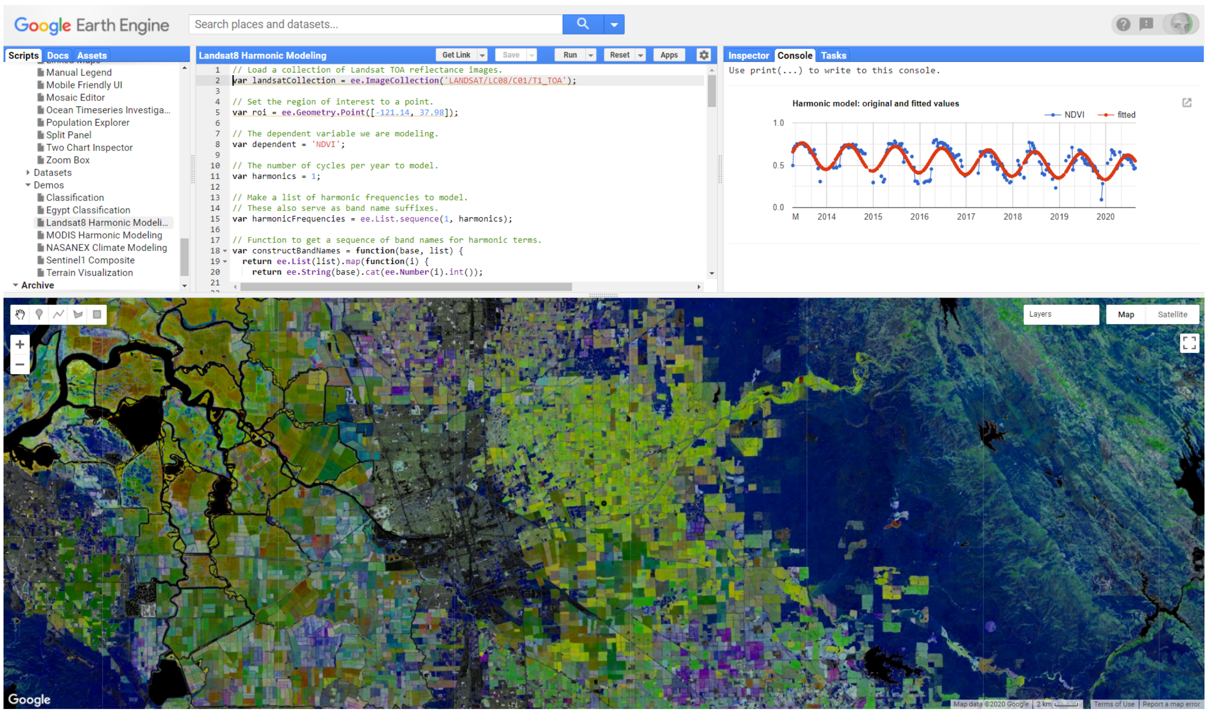
7.4.1.2. Google Earth Engine

Google Earth Engine, released in 2010, is not actually based on Jupyter Notebook and is restricted to geospatial data. However, the similar interactive code editor approach and the efficiency of the framework highlighted the asset of an interactive application environment offered in EP. Moreover, recently the Earth Engine API became available from Jupyter Notebooks developed with

Colaboratory.

Google Earth Engine is a cloud computing platform for processing satellite imagery and other geospatial and observation data. It provides access to a large database of satellite imagery and the computational power needed to analyze those images.

Beyond the large Catalogue and the outstanding performances, the success of Earth Engine is also related to the large set of access and analysis functions available using a very intuitive API. The abstraction of data and processing functions ease the development of applications as illustrated by the Landsat Harmonic Modeling consisting of about a hundred lines of code.



```
// Load a collection of Landsat TOA reflectance images.
var landsatCollection = ee.ImageCollection('LANDSAT/LC08/C01/T1_TOA');

// Set the region of interest to a point.
var roi = ee.Geometry.Point([-121.14, 37.98]);

// The dependent variable we are modeling.
var dependent = 'NDVI';

// The number of cycles per year to model.
var harmonics = 1;

// Make a list of harmonic frequencies to model.
// These also serve as band name suffixes.
var harmonicFrequencies = ee.List.sequence(1, harmonics);

// Function to get a sequence of band names for harmonic terms.
var constructBandNames = function(base, list) {
  return ee.List(list).map(function(i) {
    return ee.String(base).cat(ee.Number(i).int());
```

```
Use print(...) to write to this console.
```

Figure 23. Google Earth Engine Landsat Harmonic Modeling

7.4.2. Notebook as a Platform Frontend

Like the platforms mentioned above, a Jupyter Notebook environment can act as a platform frontend and be provisioned with a set of tools and libraries.

As shown on figure below, the environment provides some tools and a set of Python API libraries for interacting with the platform services such as the catalogue, data access and processing services.

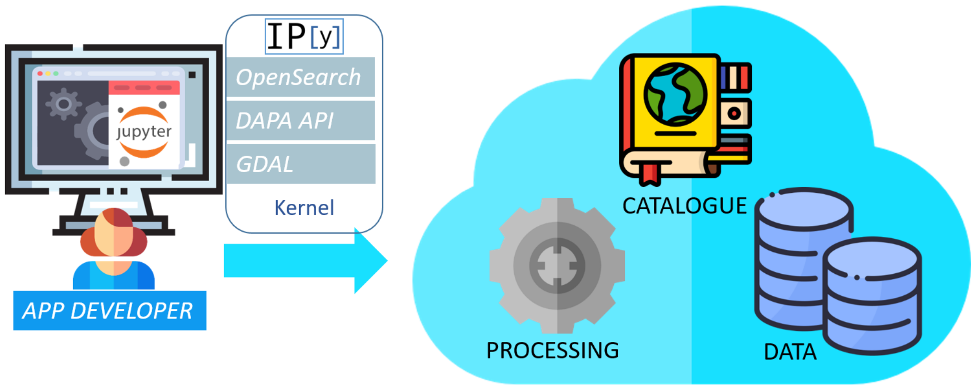
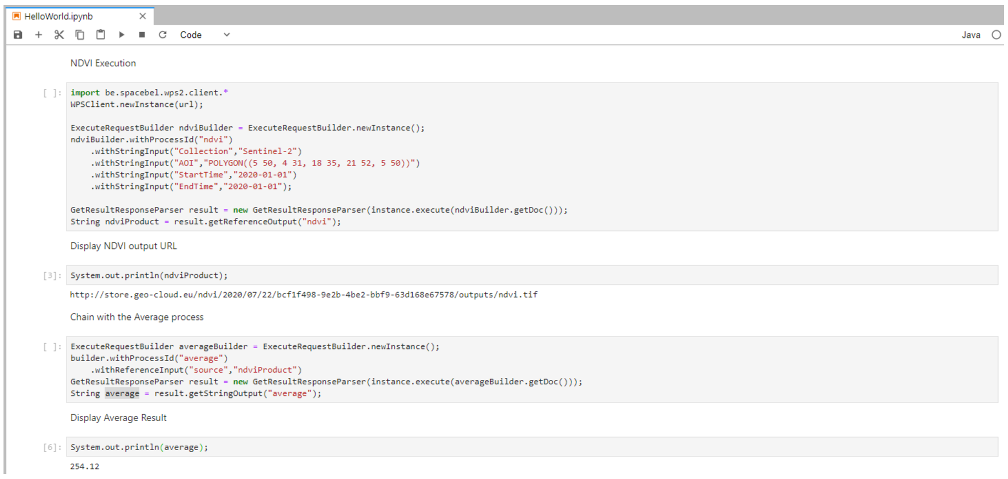


Figure 24. Exploitation Platform extended with Jupyter

Such a Notebook environment is a user-friendly place to easily develop applications close to the data. User-friendly Python libraries (or any other support kernel language) can be easily generated from the recent OGC APIs expressed in OpenAPI (such as the draft OGC Processes API or the draft OGC Data Access and Processing API). Indeed, the Swagger technologies enable automatically creating client libraries in a wide set of languages.

The libraries may facilitate a very intuitive composition of operations. In particular, a very simple data access call can be passed as a variable to a processing function, and thus **build in few lines a complex workflow**. The example below illustrates how a Jupyter Notebook (with a Java kernel) can perform the chaining sequence of a two-processes execution based on the client library implemented during OGC Testbed-14.



NDVI Execution

```
[ ]: import be.spacebel.wps2.client.*
     WPSClient.newInstance(url);

     ExecuteRequestBuilder ndviBuilder = ExecuteRequestBuilder.newInstance();
     ndviBuilder.withProcessId("ndvi")
         .withStringInput("Collection","Sentinel-2")
         .withStringInput("AOI","POLYGON((5 50, 4 31, 18 35, 21 52, 5 50))")
         .withStringInput("StartTime","2020-01-01")
         .withStringInput("EndTime","2020-01-01");

     GetResultResponseParser result = new GetResultResponseParser(instance.execute(ndviBuilder.getDoc()));
     String ndviProduct = result.getReferenceOutput("ndvi");
```

Display NDVI output URL

```
[3]: System.out.println(ndviProduct);

     http://store.geo-cloud.eu/ndvi/2020/07/22/bcf1f498-9e2b-4be2-bbf9-63d168e67578/outputs/ndvi.tif
```

Chain with the Average process

```
[ ]: ExecuteRequestBuilder averageBuilder = ExecuteRequestBuilder.newInstance();
     builder.withProcessId("average")
         .withReferenceInput("source","ndviProduct")
     GetResultResponseParser result = new GetResultResponseParser(instance.execute(averageBuilder.getDoc()));
     String average = result.getStringOutput("average");
```

Display Average Result

```
[6]: System.out.println(average);

     254.12
```

Figure 25. Jupyter Notebook with OGC API Processes interactions

As a possible solution, we recommend a careful attention to OpenEO, an open application programming interface (API) that connects clients in languages such as R, Python and JavaScript to Earth observation cloud back-ends in a simple and unified way. Many backend implementations are already provided (e.g. OGC WCPS).

7.4.3. Notebook as a Collaboration Tool

The following scenario reflects the collaboration activities on Notebooks that are considered for extending the EO XP Platform with Jupyter support:

1. Alice develops a Jupyter Notebook on EO XP Platform (optionally with built-in tools and APIs).

2. Alice **registers** the Jupyter Notebook on the platform.
3. Bob **discovers and accesses** Alice's Notebook.
4. Bob edits Notebook and **share updates** with Alice.
5. Alice exports the Notebook on a second platform.

7.4.3.1. Application Registration

Alice has developed a Jupyter Notebook in the environment managed by the EO XP Platform. In order to share the Notebook with other users, the Application Catalogue (as defined in OGC Testbed-15) collects the metadata resources describing the Jupyter Notebook, including the Notebook files.

An operation is implemented for extending the JupyterLab user interface with a control that allows registering the applications. The registration requires the following steps:

1. Push the Notebook files in a repository managed by the platform.
2. Request user for relevant information about the application (title, visibility, etc.).
3. Insert the metadata in the Application Catalogue.

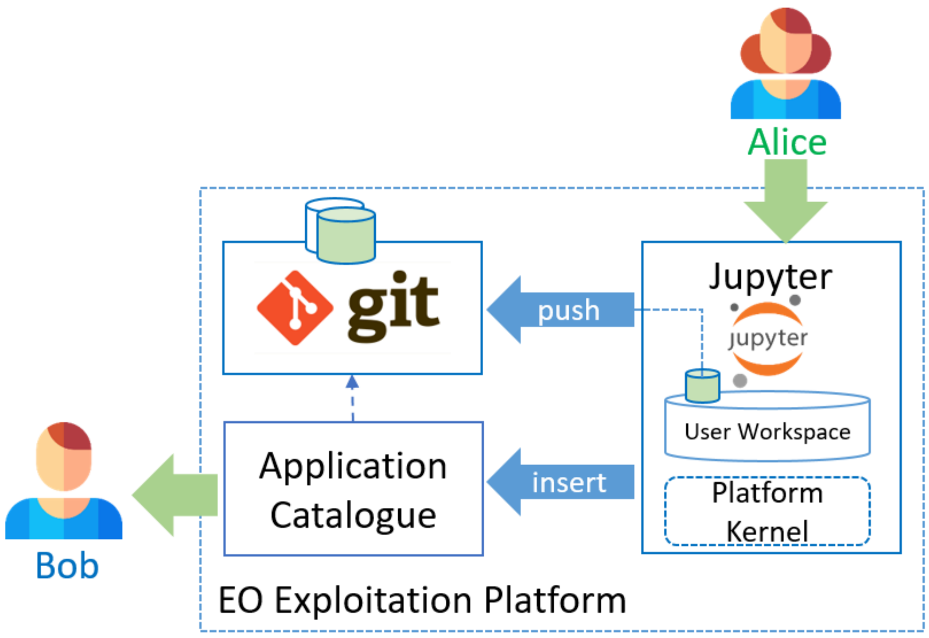


Figure 26. EO XP Platform Registering a Notebook

7.4.3.2. Access to Registered Notebooks

When Bob discovers the Jupyter Notebook from the Application Catalogue of the EO Exploitation Platform, an operation performs the following actions to provide access to the Notebook:

1. Provision a Jupyter Server (e.g. with Jupyter Hub) linked to the user workspace.
2. Copy the files from the Notebook repository to the user workspace.
3. Redirect to the JuperLab interface for displaying the selected Notebook.

In order to provision the shared Jupyter Notebook, the EO XP Platform may simply provision a

Jupyter instance:

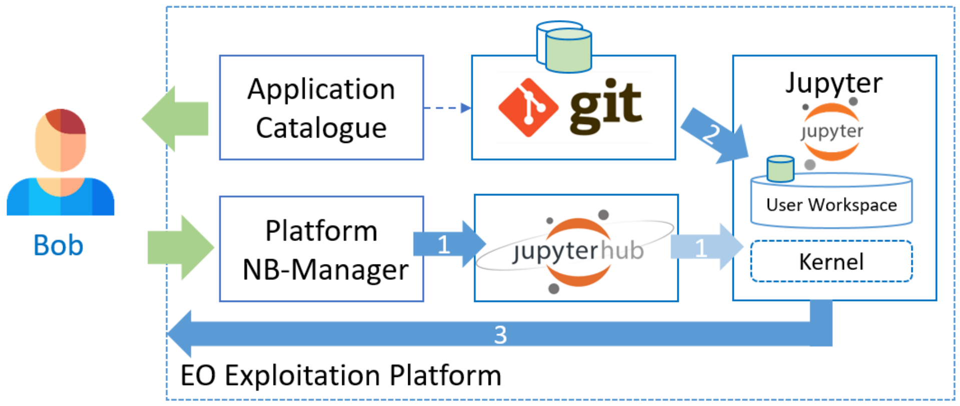


Figure 27. EO XP Platform provisioning a registered Notebook

7.4.4. Notebook as a Development Environment

As mentioned earlier, Jupyter Notebook would offer an ideal development environment for building applications that are executed on the ADES.

Developing the Jupyter-based application within the standard environment of the platform removes the need to install such environment. Moreover, the platforms can offer a set of predefined tools and include the relevant client libraries for accessing the services.

Finally, the platform might automate the creation of the Application Package by implementing the following sequence illustrated below:

1. The platform requests to the user the required information about the application (or read the Python dictionaries if this approach is implemented).
2. The container is built using repo2Docker. The container is based on the platform environment template (Dockerfile) and embeds the Notebook application with the command-line interface (i.e. papermill or nbconvert based command).
3. The application package is registered in the platform Application Catalogue (and optionally in the ADES registry).

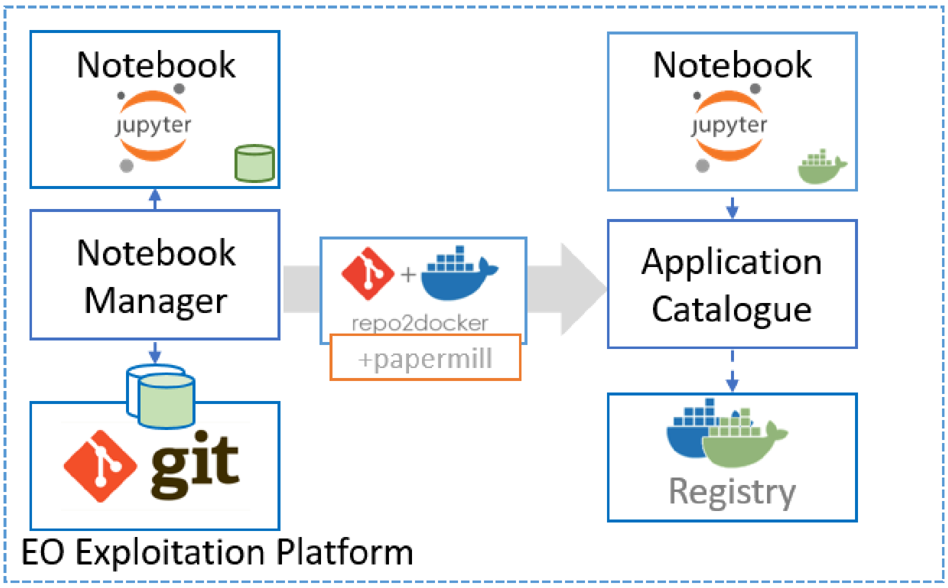


Figure 28. Export Implementation

7.4.4.1. Inter-platform Sharing

Application developers might be interested to export an interactive Jupyter Notebook to a second EO XP Platform, as illustrated on the figure below.

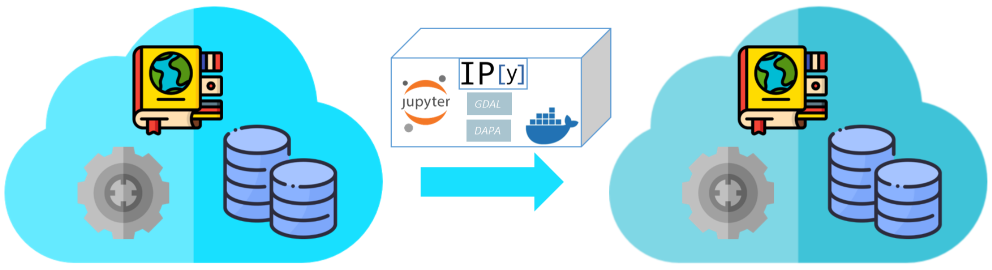


Figure 29. Inter-platform Sharing

Since the EO XP Platform potentially provides platform-specific tools, sharing the Notebook on a different platform requires packaging the Notebook in a container that contains those tools.

Therefore, the export operation shall first build the Docker container (e.g. typically using repo2Docker), then register the container on the remote platform (i.e. in the platform container registry) as illustrated on figure below. Optionally, the platform may register the Notebook in the Application Catalogue.

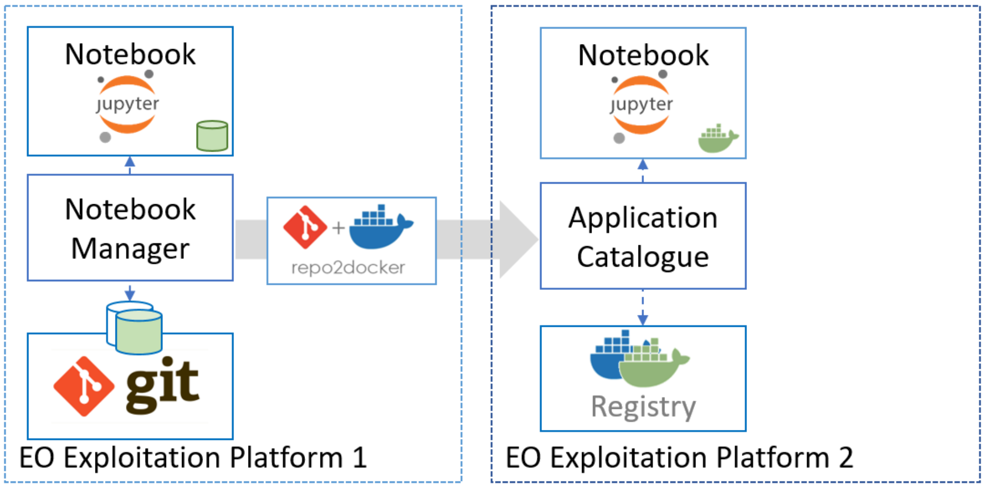


Figure 30. Export Implementation

As mentioned earlier, the Notebook reproducibility is ensured only if the standard services (e.g. DAPA, Processing API endpoint) queried by the Notebook are also offered on the target platform.

7.4.5. Implementation Aspects

From the many implementation approaches, no single solution appears to be obvious compared to others. However, BinderHub and JupyterHub provide valuable starting blocks for managing the Jupyter environment.

BinderHub enables defining containers but also supports alternatives. In particular, if only a Conda environment is defined in the repository, then the standard EO XP Platform container is provisioned. However, the Jupyter servers provisioned by BinderHub might be considered as stateless as the user loses any changes each time the session expires.

Unlike BinderHub which relies on repositories, JupyterHub based platforms (such as Mundi DIAS) handle Jupyter sessions that enable the user to manage files in a private user workspace. This aspect should be considered when designing a solution to develop and share Notebook on a EO platform.

7.5. Workflows and Processing Chains

A subset of previous OGC projects and Testbeds investigated solutions for building EO **processing chains**, i.e. a sequence of functions acting as a pipeline of processes. Expressing the flow as a Directed Acyclic Graph (DAG) seemed to capture most of the use cases for chaining geospatial applications.

Additionally, some projects explored execution of sophisticated **workflows** using technologies such as BPEL, BPMN and Oozie. Those approaches enable data consumers to integrate a multitude of functionality without the need to encapsulate the operations in a singular deployed application:

- Event triggering the execution (e.g. new product available in a collection).
- Data search and selection requests (e.g. a DAPA query).
- Processing patterns (e.g. conditions, parallel execution, map-reduce, etc.).

- Outputs registration (e.g. in platform Catalogue).
- Dissemination or notification for various protocols (e.g; FTP, SMS, Mail, etc.).

From an XP Platform perspective, the various workflows approaches can be grouped into the following categories:

1. **EMS**: As documented in OGC Testbed-14, the processing chain (a DAG) acts as a sequence of ADES execution requests that are managed by the EMS component which controls users' quotas, permissions, inter-platform data transfers, and selects at runtime the target federated platform for deploying and executing a specific processing step.
2. **Containerized** workflow: Based on the Docker Application Package, the workflow (and its potential engine if any) is packaged in a container acting as a single application executed by an ADES. The flexibility of containers enables the compatibility of any underlying technology.
3. **Code-based** workflow (specifically using Jupyter Notebook): Relying on a set of libraries (including the DAPA), the Notebook consists in a very simple sequence of steps (data access, data processing, etc.). Based on the underlying code language, developers are also able to implement very complex algorithms without limitations.

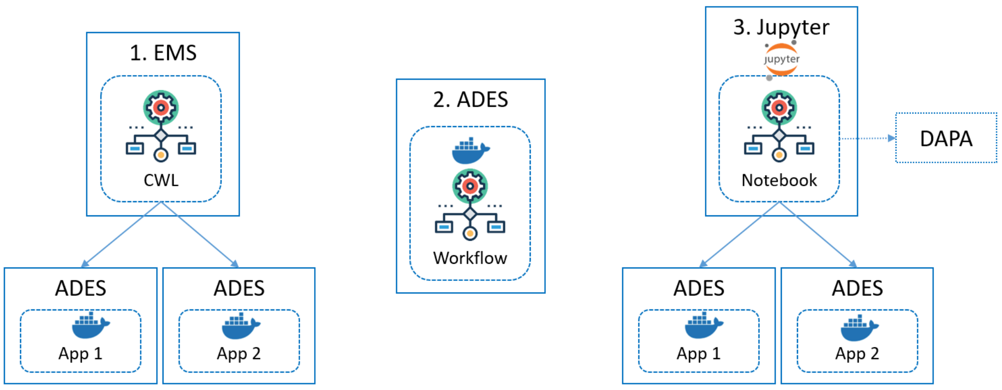


Figure 31. Workflows Approaches

During the projects, participants demonstrated the execution of containerized workflows based on the CWL language and also code-based workflows using Jupyter Notebook.

The approach based on Jupyter Notebook is very straightforward and flexible for building a sequence of data access and processing steps as illustrated below with pseudo-code. Moreover, Jupyter Notebook might execute directly CWL or container-based workflows, and extend those workflows with additional advanced steps (e.g. data access).

```
// Retrieve a set of inputs data from the Web Coverage Service
2DCoverage[] inputs = wcs.collection("Sentinel-
2.L1C").subset(Lat(10:11),Long(3,3.5),Time("2019-01-01:2020-01-31");
// Parallelize the inputs handling
for(2DCoverage i : inputs) {
    // Invoke a WPS execution and retrieves the single input
    WPSOutput ndviOutput =
wps.process("NDVI").inputs(i.url,i.bbox).getSingleOutput("NDVI");
    // Chain the output to a second function
    WPSOutput indiceeOutput =
wps.process("indicee").inputs(ndviOutput.url).getSingleOutput("indicee");
    // Copy the output on a remote FTP in a specific format
    ftpUtil.upload("ftp.ogc.com",indiceeOutput.url, indiceeOutput.filename,"zip");
}
```

Chapter 8. Solutions

8.1. 52°North (D168)

52°North targeted the implementation of three Jupyter Notebooks which when combined form a single use case. The split into sub-processes was motivated by the fact that certain steps can be parallelized (e.g. satellite scene pre-processing). This section describes the use cases, the implementation, the orchestration of the Notebooks as well as their publication on ADES instances, and finally making them available as OGC API Processes.

8.1.1. Use Cases

The mapping of water masks in flooding situations (e.g. for river gauges affected by severe weather events) is a valuable application of Sentinel-1 SAR data. Emergency response organizations are highly interested in the intensity and duration of floods so as to address flood-related damage events. In addition, the flooding history needs to be documented so that detailed information about the occurrence, frequency, and duration of flooding events for the affected areas is available. This information can be used, for example, to validate flood risk assessments and planning.

8.1.2. Implementation

The overall process is divided into three individual steps, all represented by a dedicated Notebook:

1. The discovery of relevant S1 scenes (based on time and area of interest). The scenes must all cover the same area of interest and cover different points in time of a severe weather event (before, during, after).
2. The binary classification of S1 scenes (no water vs. water).
3. An aggregation of all scenes into one GeoTIFF with the count of water occurrences by pixel.

8.1.2.1. #1 Data Discovery Notebook

The first Notebook is responsible for discovery and download of the relevant Sentinel-1 scenes. Two inputs are required:

- Area of interest (as WKT)
- Start and end date of interest (as ISO8601 date-time)

The Notebook creates a sole output:

- Array of Sentinel-1 product identifiers

The Notebook code is available at https://github.com/52North/testbed16-jupyter-Notebooks/blob/master/workflow_water_masks/nb1_request/nb1.ipynb

8.1.2.2. #2 Scene Classification Notebook

The second Notebook is responsible for doing the image classification using the Sentinel-1 backscatter values. The implementation is based on `snappy` which is the Python library of the ESA SNAP toolbox. `snappy` takes up the outputs of Notebook #1 and applies a pre-processing workflow. In particular, this process step only uses one entry of the output array. Thus, in an orchestrated workflow these Notebooks will be executed multiple times in parallel respectively for each entry of the Sentinel-1 product identifiers.

The individual steps of this Notebook for creating a water body classification are described in the following. The first step is the pre-processing of the data using "snappy". This includes the application of the Orbit files, radiometric calibration, subsetting by the area of interest as well as Speckle filtering, and terrain correction. An example of an intermediate result after the pre-processing is illustrated in Figure Figure 32 .

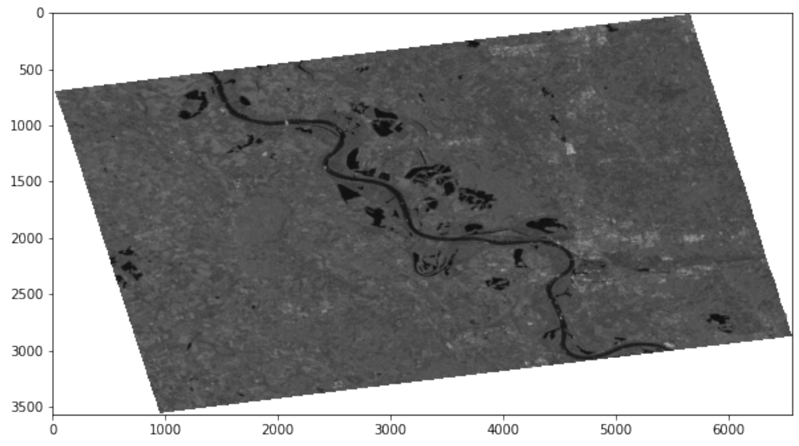


Figure 32. Pre-processing result

Finally, the water pixel classification is applied. Using "Multi-Otsu Thresholding", a threshold backscatter value is identified which is used to separate pixels between "water" and "no water". Figure Figure 33 illustrates the identification of a threshold.

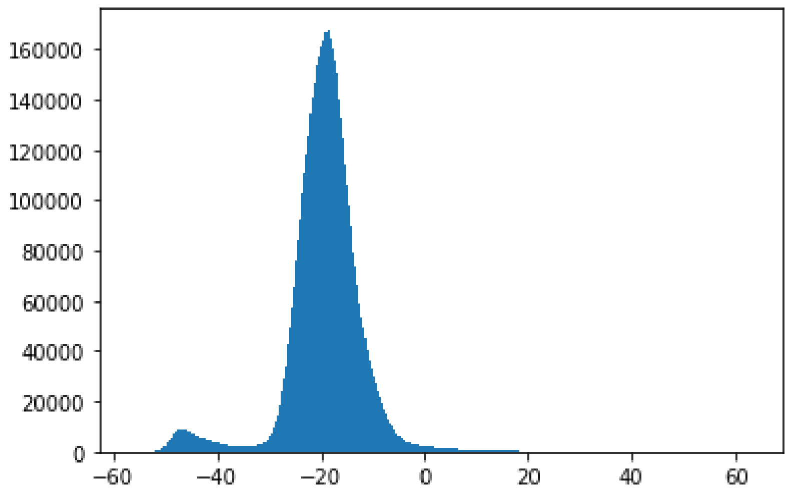


Figure 33. Threshold identification

The Notebook creates a sole output:

- A GeoTIFF representing water areas using boolean values (0 = no water, 1 = water)

An example is presented in Figure Figure 34.

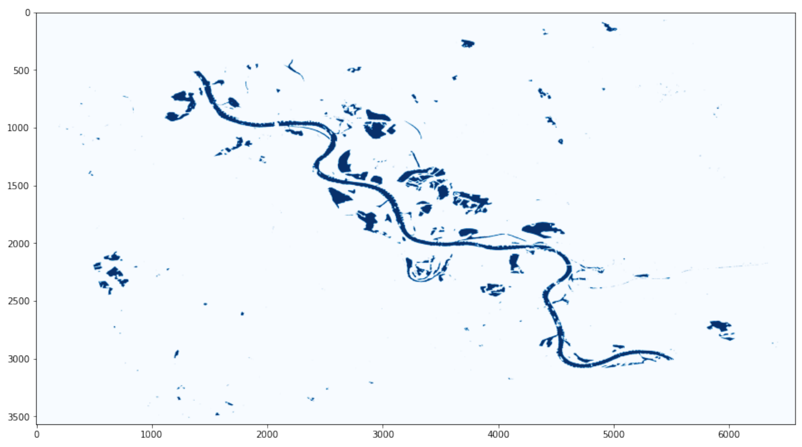


Figure 34. Water pixels after Threshold application

The Notebook code is available at https://github.com/52North/testbed16-jupyter-Notebooks/blob/master/workflow_water_masks/nb2_download_classify/nb2.ipynb

8.1.2.3. #3 Flood Mask Aggregation Notebook

The last Notebook defined is responsible for the combination of the outputs of the previously parallelized water pixel classification. Therefore, it uses the GeoTIFFs created by Notebook #2 as the input. The algorithm counts the amount of pixels that represent water and applies a normalization to `0..1` afterwards. As the scenes are all covering the same area and are distributed over time (covering the severe weather event), the total count of pixels provides insights into flooded areas. Figure Figure 35 illustrates an output.

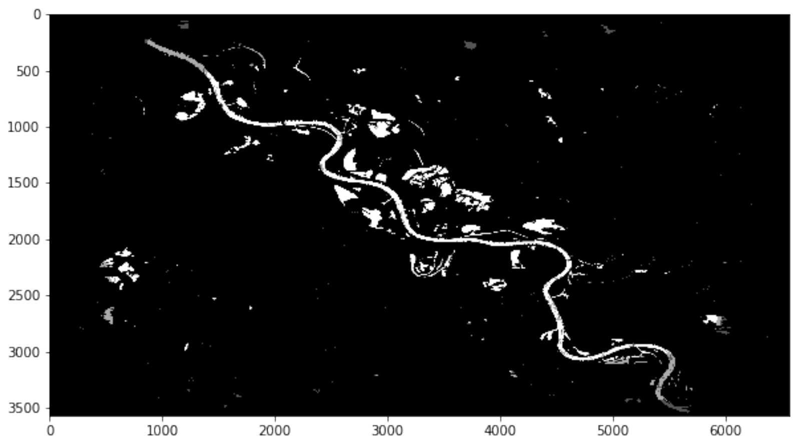


Figure 35. Water Mask using Pixel Aggregation

The overall result of this process is a raster image that enables the detection of flooded areas based on the different pixel values:

- `x >= 0.9` value pixels -→ persistent water body
- `0.1 < x < 0.9` value pixels -→ candidate for a flooded area

The Notebook creates a single output:

- Array of Sentinel-1 product identifiers

The Notebook code is available at https://github.com/52North/testbed16-jupyter-Notebooks/blob/master/workflow_water_masks/nb3_aggregate/nb3.ipynb

8.1.3. Orchestration

The chaining of the Notebooks was established using the following technologies:

- `Papermill` to externally parameterize Notebooks
- `Scrapbook` to extract outputs from an executed Notebooks
- `CWL` (i.e. cwl-runner) for the execution of the workflow, including input and output transfer

The CWL definition that was developed for the orchestration of the three Notebooks is:

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0
class: Workflow
requirements:
  SubworkflowFeatureRequirement: {}
  ScatterFeatureRequirement: {}
  ResourceRequirement:
    ramMin: 4000
    ramMax: 12000

inputs:
  areaOfInterest: string
  startDate: string
  endDate: string
  SCIHUB_UN: string
  SCIHUB_PW: string
outputs:
  floodmask_mosaic:
    type: File
    streamable: false
    outputSource: mosaic/floodmask_mosaic
steps:
  discover:
    in:
      areaOfInterest: areaOfInterest
      startDate: startDate
      endDate: endDate
      SCIHUB_UN: SCIHUB_UN
      SCIHUB_PW: SCIHUB_PW
    out: [nb1_out]
    run:
      class: CommandLineTool
      baseCommand: papermill
      hints:
        DockerRequirement:
          dockerPull: 52north/testbed16-wwm-discover:latest

      inputs:
        areaOfInterest:
          type: string
        startDate:
          type: string
        endDate:
          type: string
        SCIHUB_UN:
          type: string
        SCIHUB_PW:
          type: string
      arguments: ["/nb1.ipynb", "out.ipynb", "-p", "REQUEST_AREA",
$(inputs.areaOfInterest),
          "-p", "START_DATE", $(inputs.startDate), "-p", "END_DATE",
$(inputs.endDate),
          "-p", "SCIHUB_UN", $(inputs.SCIHUB_UN), "-p", "SCIHUB_PW",
$(inputs.SCIHUB_PW)]

      outputs:
        nb1_out:
          type: File
          outputBinding:
            glob: out.ipynb

  discover_parse:
    in:
      input_nb: discover/nb1_out
    out: [sentinelIds]
    run:
      class: CommandLineTool
      hints:
        DockerRequirement:
          dockerPull: 52north/testbed16-wwm-discover:latest
      baseCommand: ["python3", "parse.py"]
      requirements:
        InlineJavascriptRequirement: {}
        InitialWorkDirRequirement:
          listing:
            - entryname: parse.py
              entry: |
                import scrapbook as sb
                nb = sb.read_notebook("$(inputs.input_nb.path)")
                print(','.join(list(nb.scrap_dataframe["data"])[0]))

      stdout: csvItems

      inputs:
        input_nb:
          type: File

      outputs:
        sentinelIds:
          type:
            type: array
          items: string
        outputBinding:
          glob: csvItems
          loadContents: true
          outputEval: $(self[0].contents.replace('\n','').split(','))

  classify:
    run:
      class: CommandLineTool
      baseCommand: papermill

      hints:
        DockerRequirement:
          dockerPull: 52north/testbed16-wwm-classify:latest

      requirements:
        ResourceRequirement:
          ramMin: 4000
          ramMax: 12000

      arguments: ["/nb2.ipynb", "out.ipynb", "-p", "sentinel_ids",
$(inputs.sentinelSceneIds),
            "-p", "REQUEST_AREA", $(inputs.areaOfInterest),
          "-p", "SCIHUB_UN", $(inputs.SCIHUB_UN), "-p", "SCIHUB_PW",
$(inputs.SCIHUB_PW)]

      inputs:
        sentinelSceneIds:
          type: string
        areaOfInterest:
          type: string
        SCIHUB_UN:
          type: string
        SCIHUB_PW:
          type: string

      outputs:
        floodmask:
          outputBinding:
            glob: result.tif
          streamable: false
          type: File

    scatter: sentinelSceneIds
    in:
      sentinelSceneIds: discover_parse/sentinelIds
      areaOfInterest: areaOfInterest
      SCIHUB_UN: SCIHUB_UN
      SCIHUB_PW: SCIHUB_PW
    out: [floodmask]
  mosaic:
    run:
      class: CommandLineTool
      baseCommand: papermill

      requirements:
        InlineJavascriptRequirement: {}

      hints:
        DockerRequirement:
          dockerPull: 52north/testbed16-wwm-mosaic:latest

      arguments: ["/nb3.ipynb", "out.ipynb"]

      inputs:
        floodmasks_geotiff_as_array:
          type:
            type: array
            items: File
          inputBinding:
            prefix: -y
            valueFrom: |
              ${
                var yml = "floodmasks_geotiff: ["
                self.forEach(function(file){
                  yml = yml + file.path + ","
                })
                yml = yml + "]"
                return yml
              }
          streamable: false

      outputs:
        floodmask_mosaic:
          outputBinding:
            glob: mosaic.tif
          streamable: false
          type: File

    in:
      floodmasks_geotiff_as_array: classify/floodmask
    out: [floodmask_mosaic]
```

8.1.4. ADES Integration

The ADES implementations by the other participants support the execution of CWL definitions. The integration therefore is straightforward. In addition to the above presented CWL definition, each Notebook was accompanied with an individual CWL definition, making the Notebook individually executable.

An example CWL for Notebook #1 is:

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0
class: CommandLineTool
baseCommand: papermill

hints:
  DockerRequirement:
    dockerPull: 52north/testbed16-wwm-discover:latest

inputs:
  areaOfInterest:
    type: string
  startDate:
    type: string
  endDate:
    type: string
  SCIHUB_UN:
    type: string
  SCIHUB_PW:
    type: string
arguments: ["/nb1.ipynb", "out.ipynb", "-p", "REQUEST_AREA", $(inputs.areaOfInterest),
    "-p", "START_DATE", $(inputs.startDate), "-p", "END_DATE", $(inputs.endDate),
    "-p", "SCIHUB_UN", $(inputs.SCIHUB_UN), "-p", "SCIHUB_PW", $(inputs.SCIHUB_PW)]

outputs:
  nb1_out:
    outputBinding:
      glob: out.ipynb
    streamable: false
    type: File
```

Note that for demo purposes, required API credentials have been defined as inputs for the Notebook and CWL. In a production environment, the application platform APIs (e.g. S3 buckets for accessing data) shall be available without explicit authentication required.

8.2. TerraDue (D169, D171)

Terradue's Ellip Platform provides an application integration and processing environment with access to EO data to support Earth Sciences. The Ellip User Algorithm Hosting Service gives access to a dedicated application integration environment in the Cloud with exploitable software tools and libraries and access to distributed EO Data repositories to support the services adaptation and customisation.

This section reports the Testbed 16 Terradue activities on how Jupyter Notebooks can create Earth Observation Applications that are packaged, deployed, and executed as a service in Exploitation Platforms.

8.2.1. Usage Scenarios

The Testbed-16 scenarios selected by Terradue explored the packaging of Jupyter Notebooks following the recommendations of previous OGC Testbeds proposing. This was to provide a straightforward workflow from Jupyter Notebooks development to their deployment as a service in an Exploitation Platform supporting the ADES/EMS approach.

These scenarios demonstrate how a Jupyter Notebook can be used to discover, access and process (locally or remotely) Earth Observation products following three preliminary scenarios with specific steps and exploring the corresponding elements (e.g. Python Packages, OGC Services, Earth Observation data).

8.2.1.1. Scenario 1 - Process Earth Observation data in a Notebook

In this scenario, Alice is a data scientist in an Italian public institution working in the volcanology thematic field.

Alice has been tasked to contextualize using EO data and information the December 2018 Etna eruption event.

Alice knows that she may have to redo such a task for other eruption events occurring on Mt. Etna or for other active volcanoes

Therefore, Alice decides to produce a set of Jupyter Notebooks using the Python kernel for all data discovery, processing and visualization tasks.

To achieve her goal, Alice follows four major steps described below:

1. Volcano discovery through a Web Feature Service (WFS)
2. Volcano discovery with WFS dashboard
3. Earth Observation Data Discovery
4. Analysis of Volcano eruptive event with EO Band composition and Hotspot detection

8.2.1.1.1. Step 1 - Volcano discovery with WFS

Based on the assumption that her work must be reproducible for other eruption events for other active volcanoes, Alice first implements a Jupyter Notebook for volcano discovery.

The Department of Mineral Sciences from the National Museum Of Natural History of the Smithsonian Institution Global Volcanism Program (GVP) exposes OGC WFS services for the discovery of Holocene, Pleistocene and Holocene Volcanoes, for Eruptions dating from 1960 including Emissions data.

Within the Jupyter Notebook for volcano discovery, Alice consumes the GVP OGC WFS service for the discovery of the Etna volcano information. Since Etna is an Holocene volcano, she uses the Holocene Volcanoes WFS layer.

Using the OWSLib, a Python package for client programming with Open Geospatial Consortium (OGC) web service (hence OWS) interface standards and their related content models, Alice queries the WFS layer using the name queryable. The response format is JSON, a format quite easy to deserialize into a simple data frame containing the information about the Etna volcano.

While Alice is mostly interested in the Etna volcano geolocation, the information returned by the GVP WFS response contains a rich set of information such as the Etna volcano number, a unique identifier widely used by the volcanologists community.

8.2.1.1.2. Step 2 - Volcano discovery with WFS dashboard

Starting from the use-case 'Volcano discovery with WFS', Alice creates a Dashboard rendered by a Jupyter Notebook with interactions requiring a roundtrip to the kernel.

This dashboard displays the selected volcano information out of a list of volcano names.

8.2.1.1.3. Step 3 - Data Discovery

Now that Alice has the Etna geolocation as a Well-Known Text (WKT) expression, she continues with another Jupyter Notebook for the discovery of Copernicus Sentinel-2 data during the December 2018 eruption event. Alice hopes to discover Sentinel-2 cloud free products over Etna for producing RGB composites with several band compositions. She can uses the data to do a hotspot detection of the lava flows that occurred during the December 2018 event.

For the Sentinel-2 cloud free products discovery, she uses an OpenSearch catalogue and uses a Python package to interact with the Sentinel-2 collection OpenSearch endpoint.

Alice inspects the Sentinel-2 collection OpenSearch Description document to identify the search terms for querying:

1. Using the Etna geolocation WKT extended with a buffer of 0.2 degrees. This becomes the Etna Area of Interest (AOI).
2. The start/stop dates for discovering Sentinel-2 imagery captures during the 2018 eruption.
3. Cloud cover percentage less than 20%.
4. Sentinel-2 product type Level 2A (Bottom of atmosphere reflectance).

Alice queries the Sentinel-2 collection and gets the results as a GeoPandas data frame, a nice Python object to hold and manipulate tabular data with geographic information included (in this case it’s the Sentinel-2 footprints). The data frame columns include the Sentinel-2 metadata and the rows each Sentinel-2 acquisition.

Alice uses a Python module called `ipyleaflet` to plot the Etna AOI and the Sentinel-2 footprints. The map clearly shows that some Sentinel-2 acquisitions do intersect the Etna AOI but the area intersected is too small to provide information about the eruption.

Alice now wants to know what the Etna AOI coverage percentage is for each of the Sentinel-2 products discovered. For this analysis, Alice applies a function to each of the data frame rows creating a new column with the Etna AOI coverage percent. Alice decides to reject the Sentinel-2 acquisitions below a coverage percent threshold.

Now that Alice has a set of Sentinel-2 acquisitions over the Etna volcano and covering the December 2018 eruption, Alice wants to generate RGB composites with different band combinations and do the lava flows hotspot detection.

Alice saves the data frame with the Sentinel-2 discovered acquisitions as a geojson file.

8.2.1.1.4. Step 4 - Analysis of a Volcano eruptive event

Alice stages the Sentinel-2 discovered acquisitions as a Spatio-Temporal Asset Catalog (STAC) that provides her with a straightforward mechanism to access the EO acquisition data.

Alice uses GDAL to extract a geographic subset and rescale the list of EO bands required to produce the detection of lava flows hotspots - *nir08* and *swir22* - and an RGB composite for a visual identification of the lava flows with bands *swir22*, *nir08* and *red*.

Alice produces the hotspot detection and RGB composite using GDAL and numpy (Python programming language library to support large, multi-dimensional arrays and matrices) as GeoTIFF files.

Alices does an in-memory conversion of the GeoTIFFs to PNG and plots these as layers on an interactive map using the `ipyleaflet` Python module.

8.2.1.2. Scenario 2 - From a Notebook to a Processing Service

In this scenario Alice, Eric and Bob work in an Agency dealing with the detection and impact evaluation of wildfires in Australia that occurred in late 2019 into early 2020.

Alice was asked to evaluate the burned area using Sentinel-2 acquisitions with two different approaches. The first approach was based on the NBR and dNBR and the second based on a threshold method combining Normalized Difference Vegetation Index (NDVI) and Normalized Difference Water Index (NDWI).

Eric is interested in analyzing both approaches created by Alice and wants to execute them with Sentinel-2 acquisitions for validation purposes.

After Eric’s validation, Bob wants to expose the services developed by Alice as WPS services in a Platform.

To execute this scenario Alice, Eric and Bob follow four steps described below:

1. Data stage-in.
2. Process Data in a Notebook.

3. Script the Notebook execution.
4. Expose the Notebook as a Service on a Platform.

8.2.1.2.1. Step 1 - Data Discovery

Alice creates a Jupyter Notebook to discover and stage-in a pair (pre and post wildfire event) of Sentinel-2 acquisitions over the area of interest.

Alice writes a Notebook that does the data stage-in of a pair of Sentinel-2 acquisitions (pre and post wildfire event).

This Notebook generates two STAC local catalogs each with a single item for the EO product staged (pre and post wildfire event).

This Notebook writes the results as a self-contained STAC catalog with local assets described as bands with common names.

8.2.1.2.2. Step 2 - Data Processing in a Notebook

Vegetation Indexes

Alice proceeds by writing a Notebook to produce the NDVI, NDWI and NBR vegetation indexes.

This Notebook takes as input a Sentinel-2 STAC catalog containing a single item and accesses the *red*, *nir*, *swir16* and *swir22* assets to derive the three normalized differences:

- NDVI = (nir - red) / (nir + red)
- NDWI = (nir - swir16) / (nir + swir16)
- NBR = (nir - swir22) / (nir + swir22)

The Notebook writes the results as a self-contained STAC catalog with local assets described as bands with new common names *ndvi*, *ndwi* and *nbr*.

Burned area delineation

Alice continues her work to write a Notebook taking as input a STAC catalogs - pre and post event, each with an item having the *ndvi*, *ndwi* and *nbr* assets.

The Notebook performs the geographical subsetting and derives the burned area delineation products.

Alice uses the methods below:

- burned area intensity with: *dNBR = RBR_pre - RBR_post*
- a bitmask burned area delineation with a simple threshold based conditional expression: *(NDWI_post - NDWI_pre > 0.18 && NDVI_post - NDVI_pre > 0.19) ? 1 : 0*

This Notebook takes a pair of self-contained STAC catalogs with local assets described as bands with the common names *ndvi*, *ndwi* and *nbr* and produces both the delineation products.

This Notebook writes the results as a self-contained STAC catalog with local assets containing the two delineation products.

8.2.1.2.3. Step 3 - Script the Notebook execution

Eric is interested in Alice’s burned area Notebooks and he would like to run them against a few other pairs of Sentinel-2 acquisitions in Australia for validation purposes.

Eric wants to run a few other pairs of Sentinel-2 acquisitions in Australia for validation purposes using Alice’s Notebooks by parameterizing the Jupyter Notebook and driving the execution via scripting.

To invoke Alice’s Notebook, there must be a command line utility that:

- Replaces the parameters in the Notebook.
- Executes an instantiation of the Notebook with the values passed via the command line utility at runtime.

Once there is such a command line utility, invoking the utility in a shell script is quite straightforward.

Eric writes the command line utilities and scripts the execution of Alice’s Notebooks in other areas in Australia either using a shell script (e.g. bash) or a CWL workflow script.

8.2.1.2.4. Step 4 - Expose the Notebook as a Service on a Platform

Bob wants to expose Alice’s Notebooks as WPS services on an Exploitation Platform.

Bob creates a Docker image for each of the Notebooks and command line utilities and finally provides CWL scripts and associated default parameters as YAML files.

For each of the Docker images and associated CWL files, Bob creates an Application Package (CWL and Docker image published in a Docker registry) and deploys it on the Platform.

Bob invokes the deployed WPS services on the Exploitation Platform.

8.2.1.3. Scenario 3 - Chaining Notebooks

This scenario targets chaining applications as a combination of Jupyter Notebook based applications and Application Packages that link Docker Containers with arbitrary applications.

To complete this scenario, Bob follows two alternatives to chain the Notebooks.

The first method relies on using a Notebook to drive the application chaining in which the code cells define the inputs for each of the applications being chained, invokes the application execution and parses the results used as inputs for the next application in the chain.

The second method relies on using a CWL workflow to chain the applications and script its execution.

The inputs used in both methods are the applications implemented by Alice in Scenario 2, Step 2 and Eric's command line utilities and associated CWL scripts implemented in Scenario 2, Step 3.

8.2.1.3.1. Approach 1 - Application Chaining using a Notebook

Bob uses the assets developed by Alice and Eric to chain them using a Notebook. He creates a Notebook that prepares the CWL parameter files for each processing chain step and produces the burned area delineation products.

8.2.1.3.2. Approach 2 - Application Chaining using CWL

Bob uses the assets developed by Alice and Eric to chain them using a CWL workflow. Bob creates a single CWL script that invokes all processing steps described in the section above to script the application chaining execution.

This approach can also use the assets created by Bob in Scenario 2, Step 4 - the Docker images - instead of Eric's command line utilities as the CWL workflow script can reference a Docker image for each of the chained steps.

8.2.2. Notebook Execution Environments Requirements

The platform environment must provide developers with a well-defined set of operational processes and procedures to allow a single robust packaging of applications and allow their testing, validation and deployment in production in different execution scenarios. This section covers the different execution scenarios and environments for the use cases defined.

8.2.2.1. Write once, (deploy) and run everywhere

With the code developed by Alice, Eric and Bob want to be able to process it in several execution scenarios described below:

Use Case	Execution Scenario
Eric wants to reproduce and evolve the code.	Execute the Notebook on a JupyterLab instance.
Eric wants to demonstrate the code.	Deploy and Execute the Notebook on Binder.
Eric wants to execute the code in a console.	Script the execution of the Notebook via CLI.
Eric wants to batch process the code in a containerized environment.	Script the execution in a Docker container of the Notebook using CWL and cwltool.
Bob wants to perform on-demand or systematic processing on an Exploitation Platform.	Deploy the Application Package on an Exploitation Platform.

8.2.2.2. Processing scenarios global picture

Starting with a Notebook and its complementing set of files, the goal is to cover the use-cases above **without changing the code** on the Notebook nor **the contents of the git repository** to address the processing scenarios listed below:

	Execution Scenario	Environment
A	Local interactive execution on JupyterLab.	This execution may run in the Cloud or on a local machine that has Jupyter installed.
B	Interactive execution using a Docker container on Binder	This execution runs in the Cloud on the Binder resources.
C	Scripted execution in a Docker container.	This execution supports the interactive via JupyterLab and the scripting via JupyterLab terminal with a JupyterLab CLI. This execution may run in the Cloud or on a local machine.
D	Scripted execution using CWL using the Docker image as the processing environment.	This execution runs in the Cloud or on a local machine having docker installed.
E	Execution on an Exploitation Platform via the EMS/ADES after the deployment of an Application Package.	This execution runs in the Cloud.

The diagram below depicts the processing scenarios in an overall view:

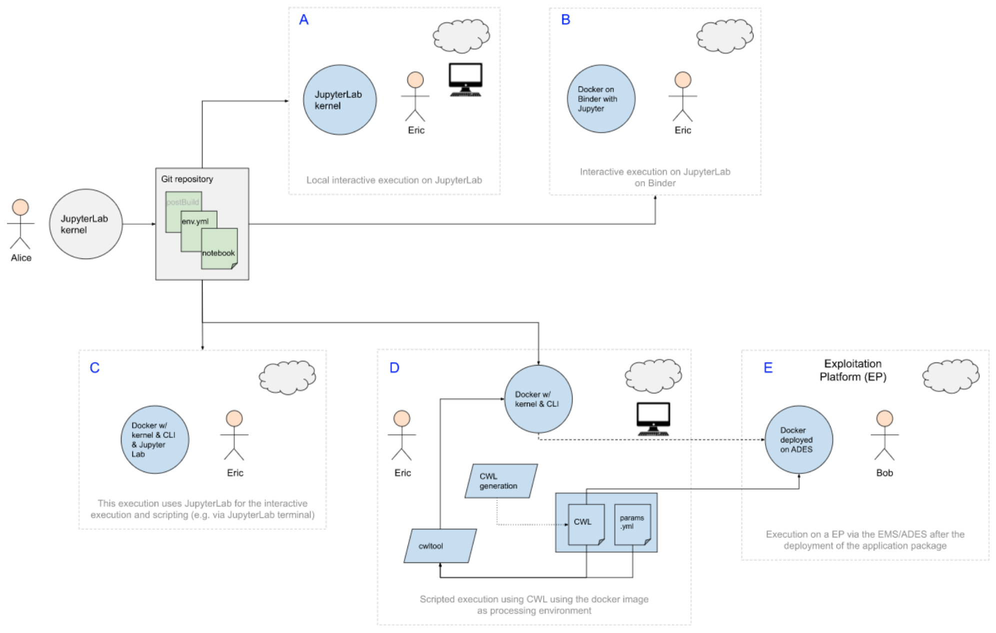

A
JupyterLab kernel
Eric
Local interactive execution on JupyterLab
B
Docker on Binder with Jupyter
Eric
Interactive execution on JupyterLab on Binder
Alice
JupyterLab kernel
Git repository
postBuild
env.yml
notebook
C
Docker w/ kernel & CLI & Jupyter Lab
Eric
This execution uses JupyterLab for the interactive execution and scripting (e.g. via JupyterLab terminal)
D
Docker w/ kernel & CLI
Eric
CWL generation
CWL
params .yml
cwltool
Scripted execution using CWL using the docker image as processing environment
E
Exploitation Platform (EP)
Docker deployed on ADES
Bob
Execution on a EP via the EMS/ADES after the deployment of the application package

Figure 36. Processing scenarios in different environments with the same code

8.2.3. Technical Solution

For the validation and deployment of the different execution scenarios, the starting elements are the assets created by Alice in Scenario 2, Step 2:

- The software repository with the NDVI, NDWI, RBR vegetation indexes Notebook.

- The software repository with the two delineation methods implemented in a Notebook.

The contents of the software repositories include the elements listed and described in the table below:

Element	Description
<file>.ipynb	This file is the Notebook.
<file(s)>.py	Optional Python helper file(s). These are useful to declutter the Notebook cells of functions and classes.
README.md	This file describes the software repository contents and may describe the installation steps, development guidelines and/or exploitation instructions.
environment.yml	This file describes the environment to run the Notebook. This is a YAML file containing a `Conda` environment definition. The file contains the Python modules used by the Notebook (and any helper files) and the `Conda` channels to get the Python modules.
.binder/Dockerfile	This is a Dockerfile enabling the execution on Binder or locally using Docker or Docker compose.
.binder/postBuild	Optional This file is a `bash` script used by the Dockerfile to provide additional build steps required to run the Notebook on Binder.

Element	Description
docker-compose.yml	Optional This file is a YAML Docker `Compose` file enabling the local execution with a docker-compose build/up two step process. `Compose` is a tool for defining and running multi-container Docker applications. With `Compose`, a developer uses a YAML file to configure the application's services. Then, with a single command, the consumer creates and starts all the services from the configuration.

8.2.3.1. Interactive Execution on JupyterLab

Eric reproduces Alice's Notebook experiment on his JupyterLab instance

The local interactive execution on JupyterLab described in this section is based on the assumption that Eric has access to a JupyterLab running instance.

Alice published the Python environment modules and channels in the `conda` **environment.yml** file available on the git repository.

This `conda` **environment.yml** is used to create:

- A Python environment for running the Notebook.
- A Jupyter kernel based on the environment created above.

Below is the environment.yml file used for the vegetation index Notebook:

```
name: env_vi
channels:
  - terradue
  - conda-forge
dependencies:
  - pystac=0.3.3
  - gdal>2.2.2
  - ipykernel
  - numpy
  - python=3.7
  - shapely
```

To reproduce Alice's Notebook experiment on his JupyterLab instance, Eric uses the JupyterLab terminal to clone Alice's repository with:

```
$ git clone https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-
index.git
$ cd vegetation-index
```

Eric reproduces the environment on a JupyterLab terminal with:

```
$ /opt/anaconda/bin/conda env create --file environment.yml
```

And creates the kernel with:

```
$ /opt/anaconda/envs/+${env_name}++/bin/python -m ipykernel install --name
kernel_++${env_name}+
```

Where *${env_name}* is the name of the environment defined in the environment.yml file. In the above example this is *env_vi*.

Now Eric can open Alice's Notebook, select the created kernel and run the cells one after the other.

The contents of the software repository ensured the reproducibility of Alice's work.

8.2.3.2. Interactive Execution in a Container on Binder

Eric reproduces Alice's Notebook experiment on Binder

Alice prepared a Dockerfile in the software repository to allow running the experiment on Binder.

She used her own Dockerfile instead of the base Docker image provided by Binder to have a Linux image (e.g. Centos distribution). To do so, Alice followed the instructions provided in the Binder documentation.

Alice uses a base image that includes JupyterLab. Her Dockerfile adds the elements needed to make sure the Notebook she wrote works as expected on Binder.

Note that some of the Dockerfile steps defined by Alice are the same as performed by Eric in the local interactive execution on JupyterLab described in the previous section.

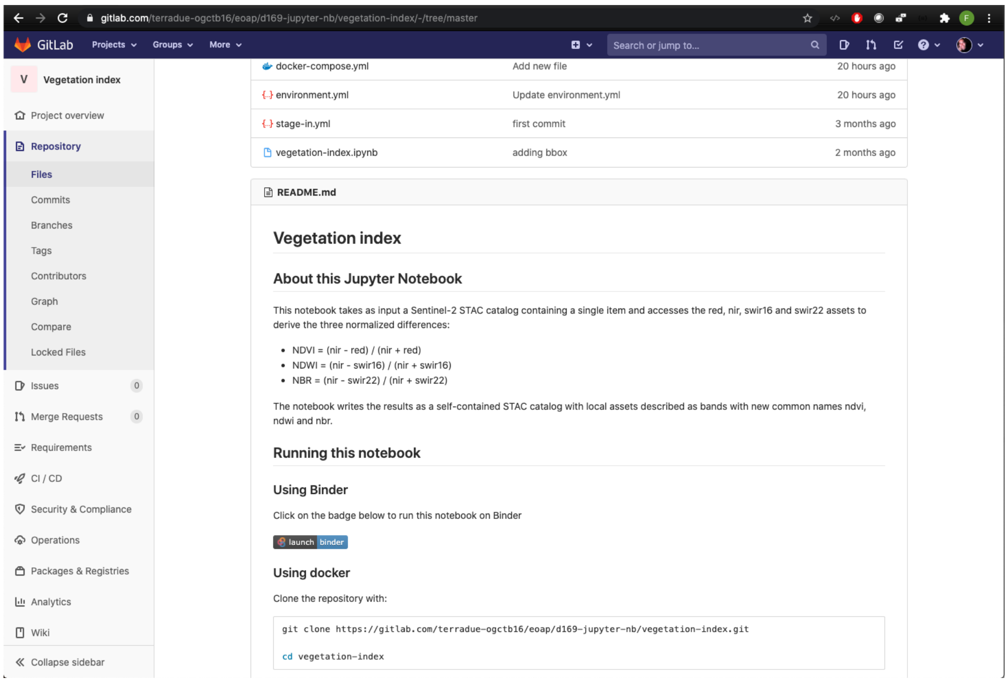
The Dockerfile is shown below:

```
FROM terradue/l1-binder:latest
MAINTAINER Terradue S.r.l
COPY . ${HOME}
USER root
RUN /opt/anaconda/bin/conda env create --file ${HOME}/environment.yml
ENV PREFIX /opt/anaconda/envs/env_vi
RUN ${PREFIX}/bin/python -m ipykernel install --name kernel
RUN chown -R ${NB_UID} ${HOME}
RUN mkdir -p /workspace/data && chown -R ${NB_UID} /workspace
USER ${NB_USER}
RUN test -f ${HOME}/postBuild && chmod 755 ${HOME}/postBuild && ${HOME}/postBuild ||
true
WORKDIR ${HOME}
```

Alice adds the Binder badge on the README.md file providing a direct URL to the Binder service for reproducing her Notebook experiment.

Below an example of such an URL:

https://mybinder.org/v2/gl/terradue-ogctb16%2Feoap%2Fd169-jupyter-nb%2Fvegetation-index/master?urlpath=lab



docker-compose.yml	Add new file	20 hours ago
environment.yml	Update environment.yml	20 hours ago
stage-in.yml	first commit	3 months ago
vegetation-index.ipynb	adding bbox	2 months ago

README.md

Vegetation index

About this Jupyter Notebook

This notebook takes as input a Sentinel-2 STAC catalog containing a single item and accesses the red, nir, swir16 and swir22 assets to derive the three normalized differences:

- NDVI = (nir - red) / (nir + red)
- NDWI = (nir - swir16) / (nir + swir16)
- NBR = (nir - swir22) / (nir + swir22)

The notebook writes the results as a self-contained STAC catalog with local assets described as bands with new common names ndvi, ndwi and nbr.

Running this notebook

Using Binder

Click on the badge below to run this notebook on Binder

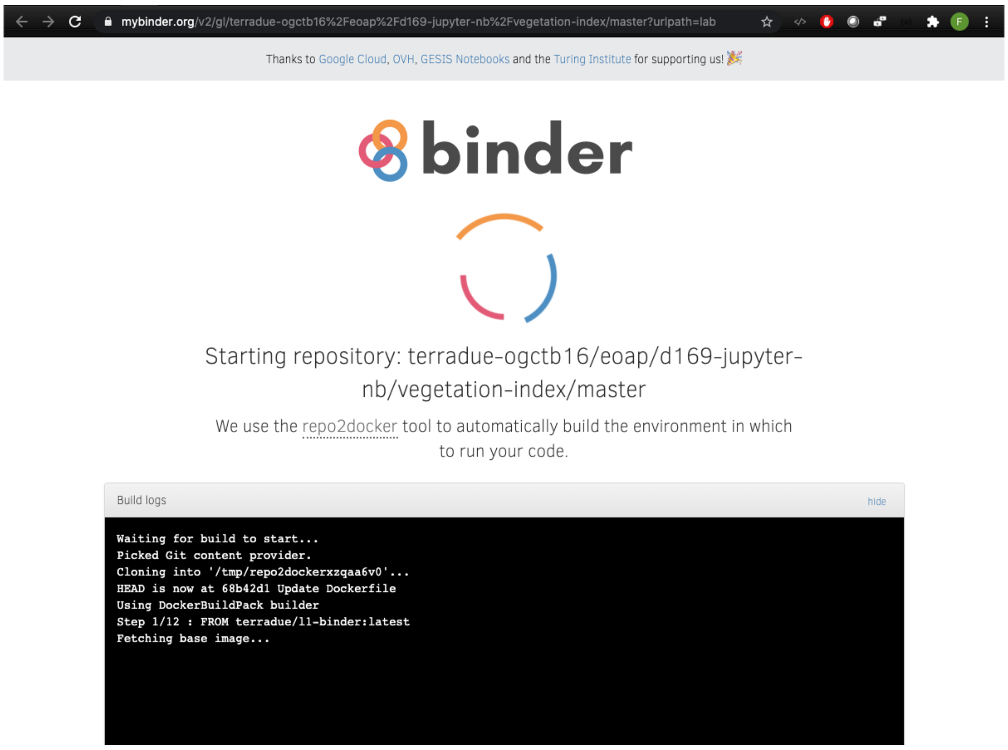
Using docker

Clone the repository with:

```
git clone https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-index.git

cd vegetation-index
```

Figure 37. Vegetation index application repository in GitLab ready to exploited in Binder

Eric simply clicks on the Binder badge on the README and waits for Binder to run the container. Eric selects the installed kernel and re-runs the Notebook cells one after the other.


mybinder.org/v2/gl/terradue-ogctb16%2Feoap%2Fd169-jupyter-nb%2Fvegetation-index/master?urlpath=lab

Thanks to Google Cloud, OVH, GESIS Notebooks and the Turing Institute for supporting us!

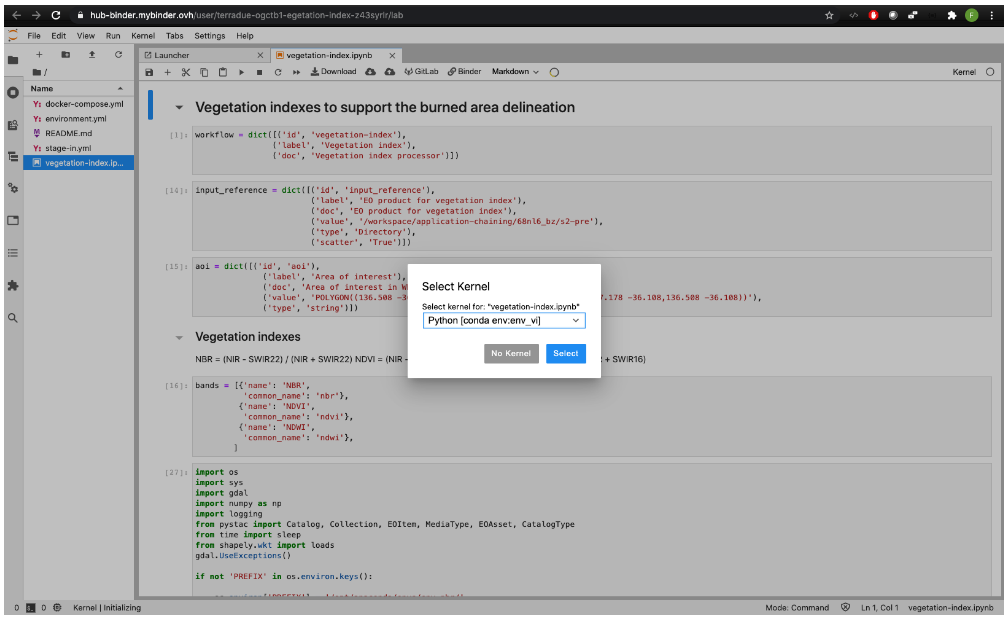
Starting repository: terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-index/master

We use the repo2docker tool to automatically build the environment in which to run your code.

Build logs

```
Waiting for build to start...
Picked Git content provider.
Cloning into '/tmp/repo2dockerxzqaa6v0'...
HEAD is now at 68b42d1 Update Dockerfile
Using DockerBuildPack builder
Step 1/12 : FROM terradue/l1-binder:latest
Fetching base image...
```

Figure 38. Binder Web Page starting the deployment process from the Vegetation index application repository


hub-binder.mybinder.ovh/user/terradue-ogctb1-egetation-index-z43syrlr/lab

Vegetation indexes to support the burned area delineation

```
[1]: workflow = dict([('id', 'vegetation-index'),
                      ('label', 'Vegetation index'),
                      ('doc', 'Vegetation index processor')])

[14]: input_reference = dict([('id', 'input_reference'),
                              ('label', 'EO product for vegetation index'),
                              ('doc', 'EO product for vegetation index'),
                              ('value', '/workspace/application-chaining/68nl6_bz/s2-pre'),
                              ('type', 'Directory'),
                              ('scatter', 'True')])

[15]: aoi = dict([('id', 'aoi'),
                  ('label', 'Area of interest'),
                  ('doc', 'Area of interest in W
                  ('value', 'POLYGON((136.508 -3                 7.178 -36.108,136.508 -36.108))'),
                  ('type', 'string')])
```

Vegetation indexes

NBR = (NIR - SWIR22) / (NIR + SWIR22) NDVI = (NIR - … + SWIR16)

```
[16]: bands = [{'name': 'NBR',
                'common_name': 'nbr'},
               {'name': 'NDVI',
                'common_name': 'ndvi'},
               {'name': 'NDWI',
                'common_name': 'ndwi'},
              ]

[27]: import os
      import sys
      import gdal
      import numpy as np
      import logging
      from pystac import Catalog, Collection, EOItem, MediaType, EOAsset, CatalogType
      from time import sleep
      from shapely.wkt import loads
      gdal.UseExceptions()

      if not 'PREFIX' in os.environ.keys():
```

Select Kernel

Select kernel for: "vegetation-index.ipynb"

Python [conda env:env_vi]

Figure 39. Notebook with Vegetation index source code and available kernels

8.2.3.3. Interactive Execution in a Docker Container

***Eric reproduces Alice's Notebook experiment using JupyterLab in a Docker container*.**

Alice included a file named docker-compose.yml in her software repository. In conjunction with the *docker-compose* command line utility, this file provides the elements to build and run the Docker image Alice created for Binder.

Below an example of a docker-compose.yml file:

```
version: '3'

services:
  jupyterlab:
    build:
      context: .
      dockerfile: .binder/Dockerfile
    ports:
    - "9005:8888"
    entrypoint: sh -c 'jupyter lab --ip=0.0.0.0 --port=8888 --no-browser --Notebook
-dir /home/jovyan --allow-root --NotebookApp.token=""'
```

The difference of this execution scenario is that the execution is done locally but without the need of having a JupyterLab instance running as for the scenario described in section dealing with the interactive execution on JupyterLab.

To reproduce Alice’s Notebook experiment using a Docker image, Eric uses his local terminal to clone Alice’s repository with:

```
$ git clone https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-
index.git

$ cd vegetation-index
```

Then Eric builds the elements required to run Alice’s Notebook using JupyterLab in a Docker image:

```
$ docker-compose build
```

This step may take a few minutes as during the build process the base Docker is downloaded and a new Docker image is created.

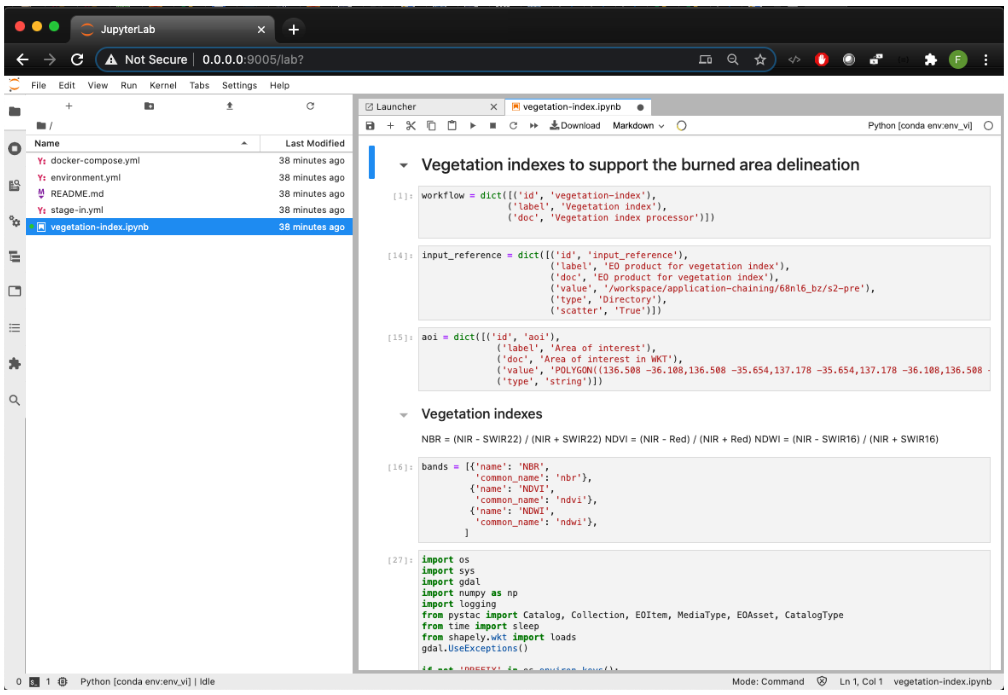
Now Eric runs the built elements with:

```
$ docker-compose up
```

The output shows a JupyterLab process starting:

```
Creating network "vegetation-index_default" with the default driver
Creating vegetation-index_jupyterlab_1 ... done
Attaching to vegetation-index_jupyterlab_1
jupyterlab_1  | [I 10:21:58.951 LabApp] [nb_conda_kernels] enabled, 3 kernels found
jupyterlab_1  | [I 10:21:58.962 LabApp] Writing Notebook server cookie secret to
/home/jovyan/.local/share/jupyter/runtime/Notebook_cookie_secret
jupyterlab_1  | [W 10:21:59.260 LabApp] All authentication is disabled.  Anyone who
can connect to this server will be able to run code.
jupyterlab_1  | [I 10:21:59.651 LabApp] JupyterLab extension loaded from
/opt/anaconda/envs/Notebook/lib/python3.7/site-packages/jupyterlab
jupyterlab_1  | [I 10:21:59.651 LabApp] JupyterLab application directory is
/opt/anaconda/envs/Notebook/share/jupyter/lab
jupyterlab_1  | [I 10:21:59.656 LabApp] Registered QuickOpenHandler extension at URL
path /api/quickopen to serve results of scanning local path /home/jovyan
jupyterlab_1  | [I 10:21:59.765 LabApp] [nb_conda] enabled
jupyterlab_1  | [I 10:21:59.769 LabApp] nteract extension loaded from
/opt/anaconda/envs/Notebook/lib/python3.7/site-packages/nteract_on_jupyter
jupyterlab_1  | [I 10:21:59.847 LabApp] Serving Notebooks from local directory:
/home/jovyan
jupyterlab_1  | [I 10:21:59.847 LabApp] The Jupyter Notebook is running at:
jupyterlab_1  | [I 10:21:59.847 LabApp] http://bd01fa317168:8888/
jupyterlab_1  | [I 10:21:59.847 LabApp] Use Control-C to stop this server and shut
down all kernels (twice to skip confirmation).
```

Eric opens his browser at one the addresses 127.0.0.1:9005 or 0.0.0.0:9005. The browser displays a JupyterLab instance with the Notebook ready to run interactively.



Name	Last Modified
docker-compose.yml	38 minutes ago
environment.yml	38 minutes ago
README.md	38 minutes ago
stage-in.yml	38 minutes ago
vegetation-index.ipynb	38 minutes ago

Vegetation indexes to support the burned area delineation

```
[1]: workflow = dict([('id', 'vegetation-index'),
                      ('label', 'Vegetation index'),
                      ('doc', 'Vegetation index processor')])
```

```
[14]: input_reference = dict([('id', 'input_reference'),
                              ('label', 'EO product for vegetation index'),
                              ('doc', 'EO product for vegetation index'),
                              ('value', '/workspace/application-chaining/68nl6_bz/s2-pre'),
                              ('type', 'Directory'),
                              ('scatter', 'True')])
```

```
[15]: aoi = dict([('id', 'aoi'),
                  ('label', 'Area of interest'),
                  ('doc', 'Area of interest in WKT'),
                  ('value', 'POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508 -
                  ('type', 'string')])
```

Vegetation indexes

NBR = (NIR - SWIR22) / (NIR + SWIR22) NDVI = (NIR - Red) / (NIR + Red) NDWI = (NIR - SWIR16) / (NIR + SWIR16)

```
[16]: bands = [{'name': 'NBR',
                'common_name': 'nbr'},
               {'name': 'NDVI',
                'common_name': 'ndvi'},
               {'name': 'NDWI',
                'common_name': 'ndwi'},
              ]
```

```
[27]: import os
      import sys
      import gdal
      import numpy as np
      import logging
      from pystac import Catalog, Collection, EOItem, MediaType, EOAsset, CatalogType
      from time import sleep
      from shapely.wkt import loads
      gdal.UseExceptions()
```

Figure 40. Interactive Execution in a Docker Container

This execution scenario is technically the same as the one described in the section dealing with the execution in a Container using Binder but this time it runs locally with potentially more CPU and RAM than Binder provides and with no culling after a period of inactivity (Binder's standard behavior).

8.2.3.4. Scripted Execution

Eric scripts the Notebook execution using command line utilities and optionally CWL.

Notebooks were designed to be executed as a step-by-step execution of cells containing code. Instead, this scenario is about the scripted execution of a Notebook.

Therefore, the scripted execution of a Notebook requires a mechanism to identify the cells containing the variables to be defined at runtime.

As such Notebook is used as a template. During the scripted execution, the template is instantiated, the variable values updated and finally the instantiated Notebook cells are executed in a sequential and non-attended process.

The scripted execution of a Notebook requires the creation of a command-line utility (CLI) that:

- Exposes the Notebook variables as parameters.
- Instantiates the Notebook (used as a template) updating the variables with the values passed via the CLI.
- Executes the cells one after another.
- Saves the resulting Notebook.

The Terradue implementation of such a feature followed the *nbconvert* approach by using its Python API to:

- Parse the Notebook looking for variables to parametrize.
- Replace the variable values at runtime.
- Execute the instantiated Notebook.
- Save the Notebook.

The first two steps listed above require the definition of a convention to define parameters in a Notebook. Terradue chose to use Python dictionaries to define the parameters in a Notebook. This choice has the benefit of allowing the defining of additional information to describe the parameters. CLI utilities provide a help context so providing a description for the parameter seems a good practice. Also providing a unique identifier is useful as well as a default value.

A parameter could thus be defined as:

```
ndvi_threshold =  dict([('id', 'ndvi_threshold'),
                        ('description', 'The NDVI threshold, it default value is 0,18
'),
                        ('value', '0.18')])
```

As discussed in section 7, the CWL specification has proven its utility to provide a simple way to invoke a command-line utility and to describe a simple workflow as the Application Package. While the scripted execution of a Notebook does not require using CWL, the execution scenario on an Exploitation Platform does. As such, the Python dictionary could include other key/value pairs to allow generating the CWL script that invokes the CLI derived from the Notebook.

The resulting parameter definition thus becomes:

```
ndvi_threshold =  dict([('id', 'ndvi_threshold'),
                        ('label', 'NDVI threshold'),
                        ('doc', 'The NDVI threshold, its default value is 0.18'),
                        ('value', '0.18'),
                        ('type', 'string')])
```

To define a parameter the provides options, this definition becomes:

```
false_color_1 = dict([('id', 'false_color_1'),
                      ('label', 'False Color 1 (S5, S3, S2)'),
                      ('doc', 'False Color RGB composite using bands S5, S3 and S2'),
                      ('value', 'Yes'),
                       ('symbols', ['Yes', 'No']),
                       ('type', 'enum')])
```

At this stage, Alice knows how to define parameters to support the parametrization and scripted execution. Nevertheless, there is an element missing for the creation of a CWL script with a workflow that invokes the CLI. The missing element defines the workflow metadata: id, doc and label in the CWL terminology. Such information can also be defined with a Python dictionary:

```
workflow = dict([('label', 'Normalized burn ratio'),
                 ('doc', 'Normalized burn ratio for burned area intensity assessment'
),
                 ('id', 'nbr')])
```

At this stage, Alice's Notebook has all the elements to support the generation of a CLI utility and associated CWL script.

The paradigm "Write once, Execute anywhere" introduced in the previous section provides a strong restriction related to the implementation options where the entry point must be a software repository with a Notebook and its associated files (at least a `conda` environment definition file).

During the testbed, Terradue implemented a tool called *repo2cli* that converts a git software repository with a Notebook and its associated files to a CLI utility and CWL script display.

The steps the *repo2cli* CLI utility performs are:

1. Git clone a repository that contains at least a Notebook and a conda environment file.
2. Create the environment.
3. Publish the Jupyter kernel associated with that environment.

4. Instantiates a Python project template enabling:
    a. The creation of the CLI utility that invokes the Notebook.
    b. The creation of the CLI utility flag to display the CWL workflow script.
5. Installs the instantiated Python project.
6. Runs the postBuild if included in the repository.
7. Adds the CLI utility that invokes the Notebook to the environment PATH variable.

Once run against Alice’s git repository URL, the CLI allowing the scripted execution of the Notebook is created and made available in the environment path.

Eric uses the CLI *repo2cli* to create the CLI utility that will allow him to script Alice’s Notebook execution:

```
$ repo2cli -r https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-
index.git
```

Eric has now:

- A new conda environment named env_vi.
- A new Jupyter kernel based on the environment above.
- A CLI to script the Notebook execution and generate the CWL workflow script.

```
(base) [eric@sb-10-160-0-21 workspace]$ /opt/anaconda/envs/env_vi/bin/vegetation-index
--help
usage: vegetation-index [-h] [--kernel KERNEL] [--output NB_TARGET] [--docker DOCKER]
[--cwl] [--params] [--input_reference INPUT_REFERENCE] [--aoi AOI]

Vegetation index Vegetation index processor

optional arguments:
  -h, --help            Show this help message and exit
  --kernel KERNEL       Kernel for Notebook execution
  --output NB_TARGET    Output Notebook
  --docker DOCKER       Sets the docker image for the CWL DockerRequirement
  --cwl                 Prints the CWL script and exits
  --params              Prints the default parameters and exits
  --input_reference INPUT_REFERENCE
                        EO product for vegetation index
  --aoi AOI             Area of interest in WKT
(base) [eric@sb-10-160-0-21 workspace]$
```

Eric stages a Sentinel-2 Level 2A acquisition as local STAC catalog to a folder (e.g. /workspace/data/s2_pre) and invokes the vegetation-index command line utility:

```
(base) [eric@sb-10-160-0-21 workspace]$ /opt/anaconda/envs/env_vi/bin/vegetation-index
\
     --input_reference /workspace/data/s2_pre \
     --aoi 'POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178
-36.108,136.508 -36.108))'
```

Once the execution is completed, Eric will have as results:

- A local STAC catalog with a STAC item and its assets NBR, NDVI and NDWI GeoTIFF files.
- Alice's vegetation-index.ipynb Notebook instantiated and executed with Eric's parameters passed via the command line utility.

8.2.3.4.1. Approach 1 - Scripted execution using CWL

Eric invokes the vegetation-index command line utility using CWL.

The CWL file is retrieved with:

```
$ docker run --rm -it vegetation-index:0.1 vegetation-index --cwl
```

The capacity of the vegetation-index command line utility to print a CWL workflow script allowing its invocation is done by the *repo2cli* tool.

The content Alice added to the Notebook with the Python dictionaries declaring the Notebook parameters and workflow metadata provide the information to do such automated CWL script generation.

The CWL script, printed in the stdout is saved in an ASCII file named vi.cwl.

Eric invokes the CWL workflow script with:

```
$ cwltool vi.cwl#vegetation-index params.yml
```

Where *vegetation-index* is the CWL workflow identifier (this tells the *cwltool* what entry point to use) and params.yml a YAML file containing the values of the parameters for the Notebook execution, eg:

```yaml
input_reference: {class: Directory, path: /workspace/data/s2_pre}
aoi: 'POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508
-36.108))'
```

The *cwltool* execution launches the vegetation-index command line utility and generate the results listed to `stdout` as a JSON structure providing the local paths to the files and folders generated by the execution:

- A local STAC catalog with a STAC item and its assets NBR, NDVI and NDWI GeoTIFF files.

- Alice’s vegetation-index.ipynb Notebook instantiated and executed with Eric’s parameters passed via the command line utility.

8.2.3.4.2. Approach 2 - Scripted execution using CWL and Docker

Eric creates a new Docker file with the content:

```
FROM terradue/c7-repo2cli:latest

RUN repo2cli -r https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-
index.git

ENV PREFIX /opt/anaconda/envs/env_vi

ENV PATH /opt/anaconda/envs/env_vi/bin:$PATH
```

To simplify the Dockerfile, the base Docker image used contains the *repo2cli* utility.

Eric builds the Docker image with:

```
$ docker build . -t vegetation-index:0.1
```

The CWL file is retrieved with:

```
$ docker run --rm -it vegetation-index:0.1 vegetation-index --cwl --docker
'vegetation-index:0.1'
```

The CWL script, printed in the stdout is saved in an ASCII file named docker-vi.cwl.

Eric invokes the CWL workflow script with:

```
$ cwltool docker-vi.cwl#vegetation-index params.yml
```

Where *vegetation-index* is the CWL workflow identifier (this tells the *cwltool* what entry point to use) and params.yml a YAML file containing the values of the parameters for the Notebook execution, eg:

```
input_reference: {class: Directory, path: /workspace/data/s2_pre}
aoi: 'POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508
-36.108))'
```

The *cwltool* execution launches the Docker execution of the Notebook and the results are listed to `stdout` as a JSON structure providing the local paths to the files and folders generated by the execution.

This scripted execution approach leaves Eric's computer free of any application files as these are all self-contained in a Docker image.

8.2.3.5. Execution on a Platform via ADES

Bob deploys an Application Package and consumes the WPS service on an Exploitation Platform.

Bob creates an Application Package with the assets Eric created in the "*Scripted execution using CWL and Docker execution*" scenario.

Bob builds and pushes the Docker image to a docker registry (e.g. DockerHub). In example below is part of the Terradue DockerHub organization.

```
$ docker build . -t terradue/vegetation-index:0.1
$ docker login
$ docker push terradue/vegetation-index:0.1
```

Bob regenerates the CWL script to update the docker reference with:

```
$ docker run --rm -it vegetation-index:0.1 vegetation-index --cwl --docker
'terradue/vegetation-index:0.1'
```

Bob then:

- Posts the Application Package in a Resource Manager.
- Deploys the Application Package on a WPS endpoint exposed by an Exploitation Platform.
- Invokes the deployed processing service using WPS requests.

8.2.3.6. Chaining Notebooks

Alice implemented two Notebooks. The first Notebook takes a Sentinel-2 Level 2A local STAC catalog and produces the NBR, NDVI and NDWI vegetation indexes. The second takes a pair of pre and post wildfire event local STAC catalogs with the NBR, NDVI and NDWI assets to generate the burned area bitmask and burned area intensity.

This section describes two methods to chain these two applications. Both methods use the command-line utilities generated by the *repo2cli* tool.

The steps the *repo2cli* CLI tool performs are:

- Git clone a repository that contains at least a Notebook and a conda environment file.
- Create the environment.
- Publish the Jupyter kernel associated with that environment.
- Instantiates a Python project template enabling:
  - The creation of the CLI utility that invokes the Notebook;

    - The creation of the CLI utility flag to display the CWL workflow script.
- Installs the instantiated Python project.
- Runs the postBuild if included in the repository
- Adds the CLI utility that invokes the Notebook to the environment P.ATH variable.

Bob runs *repo2cli* on both Alice's repositories with:

```
$ repo2cli -r https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-
index.git
```

Which adds the vegetation-index CLI:

```
(base) [bob@sb-10-160-0-21 workspace]$ /opt/anaconda/envs/env_vi/bin/vegetation-index
--help
usage: vegetation-index [-h] [--kernel KERNEL] [--output NB_TARGET] [--docker DOCKER]
[--cwl] [--params] [--input_reference INPUT_REFERENCE] [--aoi AOI]

Vegetation index Vegetation index processor

optional arguments:
  -h, --help            show this help message and exit
  --kernel KERNEL       kernel for Notebook execution
  --output NB_TARGET    output Notebook
  --docker DOCKER       Sets the docker image for the CWL DockerRequirement
  --cwl                 Prints the CWL script and exits
  --params              Prints the default parameters and exits
  --input_reference INPUT_REFERENCE
                        EO product for vegetation index
  --aoi AOI             Area of interest in WKT
(base) [eric@sb-10-160-0-21 workspace]$
```

and

```
$ repo2cli -r https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/burned-area.git
```

Which adds the burned-area CLI:

```
(base) [bob@sb-10-160-0-21 workspace]$ /opt/anaconda/envs/env_burned_area/bin/burned-
area --help
usage: burned-area [-h] [--kernel KERNEL] [--output NB_TARGET] [--docker DOCKER] [--
cwl] [--params] [--pre_event PRE_EVENT] [--post_event POST_EVENT] [--ndvi_threshold
NDVI_THRESHOLD] [--ndwi_threshold NDWI_THRESHOLD]

Burned area delineation using two techniques:

optional arguments:
  -h, --help            show this help message and exit
  --kernel KERNEL       kernel for Notebook execution
  --output NB_TARGET    output Notebook
  --docker DOCKER       Sets the docker image for the CWL DockerRequirement
  --cwl                 Prints the CWL script and exits
  --params              Prints the default parameters and exits
  --pre_event PRE_EVENT
                        Pre-event product for burned area delineation
  --post_event POST_EVENT
                        Post-event product for burned area delineation
  --ndvi_threshold NDVI_THRESHOLD
                        NDVI difference threshold
  --ndwi_threshold NDWI_THRESHOLD
                        NDWI difference threshold
```

Bob’s machine now has all the assets needed to chain the two applications.

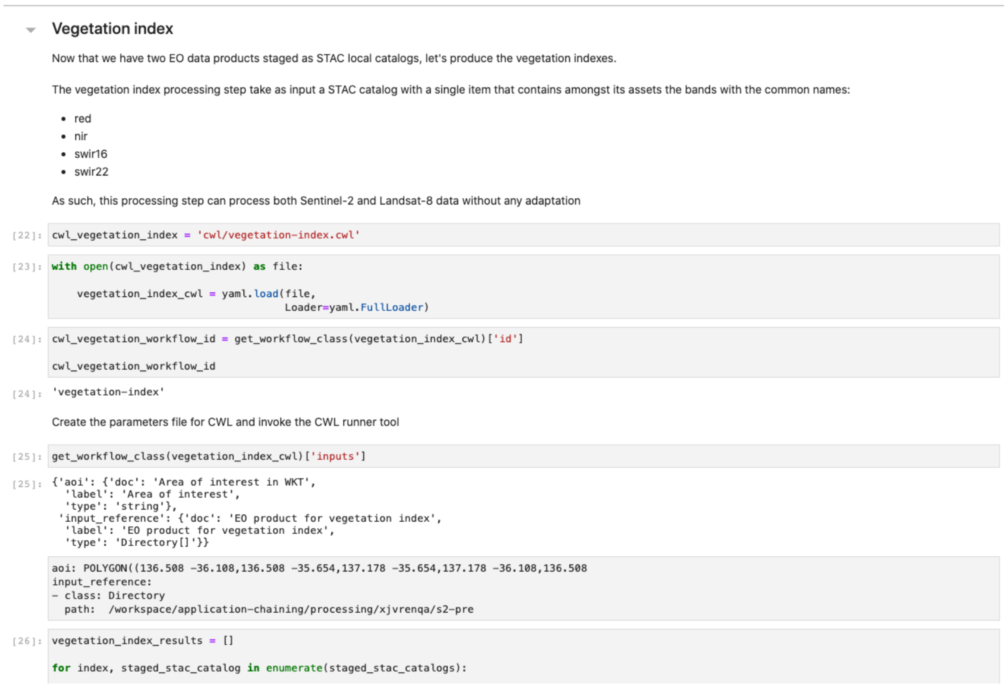
8.2.3.6.1. Approach 1 - Application Chaining using a Notebook

Bob uses a Notebook to chain the two applications. One option to do so would be to do a system call to each of the vegetation-index and burned-area CLIs but this option was soon discarded as it would require knowing their interfaces and clumsy manipulation of inputs and outputs.

Instead, Bob invokes the two command line tools above with the --cwl option to generate the CWL scripts vegetation-index.cwl and burned-area.cwl.

This approach allows Bob’s chaining Notebook to do a simple parsing of the CWL scripts to discover the CWL Workflow class and provide the right parameters as a YAML file.

Then Bob’s Notebook does a system call to the *cwltool* and simply parses the JSON provided as the *cwltool* execution stdout.



Vegetation index

Now that we have two EO data products staged as STAC local catalogs, let's produce the vegetation indexes.

The vegetation index processing step take as input a STAC catalog with a single item that contains amongst its assets the bands with the common names:

- red
- nir
- swir16
- swir22

As such, this processing step can process both Sentinel-2 and Landsat-8 data without any adaptation

```
[22]: cwl_vegetation_index = 'cwl/vegetation-index.cwl'
```

```
[23]: with open(cwl_vegetation_index) as file:

          vegetation_index_cwl = yaml.load(file,
                                           Loader=yaml.FullLoader)
```

```
[24]: cwl_vegetation_workflow_id = get_workflow_class(vegetation_index_cwl)['id']

      cwl_vegetation_workflow_id
```

```
[24]: 'vegetation-index'
```

Create the parameters file for CWL and invoke the CWL runner tool

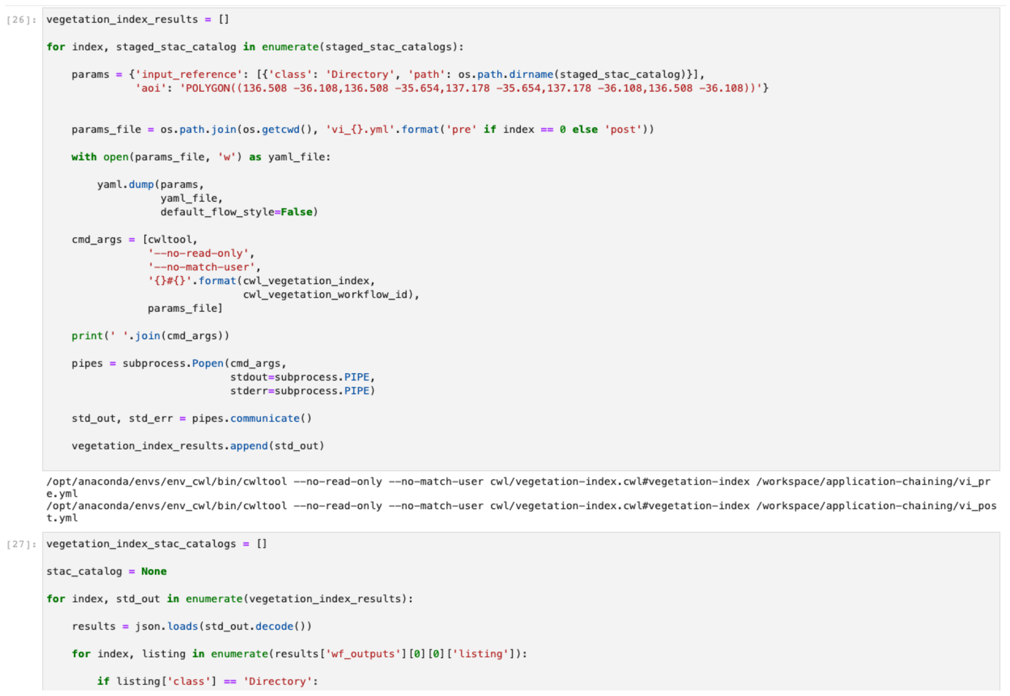
```
[25]: get_workflow_class(vegetation_index_cwl)['inputs']
```

```
[25]: {'aoi': {'doc': 'Area of interest in WKT',
        'label': 'Area of interest',
        'type': 'string'},
       'input_reference': {'doc': 'EO product for vegetation index',
        'label': 'EO product for vegetation index',
        'type': 'Directory[]'}}
```

```
aoi: POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508
input_reference:
- class: Directory
  path:  /workspace/application-chaining/processing/xjvrenqa/s2-pre
```

```
[26]: vegetation_index_results = []

      for index, staged_stac_catalog in enumerate(staged_stac_catalogs):
```

Figure 41. Chaining Notebook - vegetation index 1

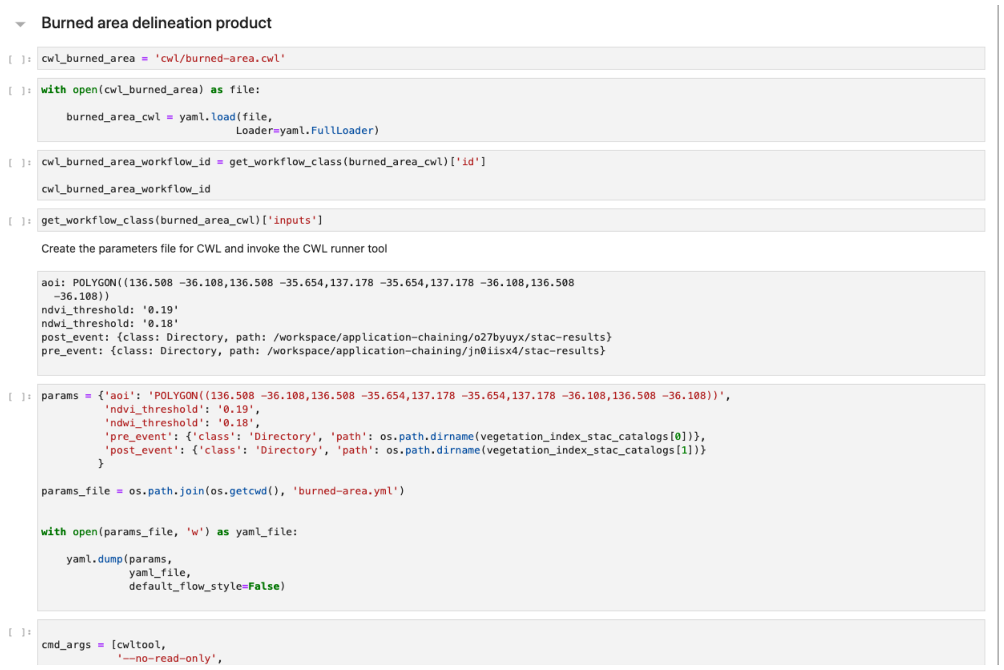


```
[26]: vegetation_index_results = []

      for index, staged_stac_catalog in enumerate(staged_stac_catalogs):

          params = {'input_reference': [{'class': 'Directory', 'path': os.path.dirname(staged_stac_catalog)}],
                    'aoi': 'POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508 -36.108))'}


          params_file = os.path.join(os.getcwd(), 'vi_{}.yml'.format('pre' if index == 0 else 'post'))

          with open(params_file, 'w') as yaml_file:

              yaml.dump(params,
                        yaml_file,
                        default_flow_style=False)

          cmd_args = [cwltool,
                      '--no-read-only',
                      '--no-match-user',
                      '{}#{}'.format(cwl_vegetation_index,
                                     cwl_vegetation_workflow_id),
                      params_file]

          print(' '.join(cmd_args))

          pipes = subprocess.Popen(cmd_args,
                                   stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE)

          std_out, std_err = pipes.communicate()

          vegetation_index_results.append(std_out)
```

```
/opt/anaconda/envs/env_cwl/bin/cwltool --no-read-only --no-match-user cwl/vegetation-index.cwl#vegetation-index /workspace/application-chaining/vi_pr
e.yml
/opt/anaconda/envs/env_cwl/bin/cwltool --no-read-only --no-match-user cwl/vegetation-index.cwl#vegetation-index /workspace/application-chaining/vi_pos
t.yml
```

```
[27]: vegetation_index_stac_catalogs = []

      stac_catalog = None

      for index, std_out in enumerate(vegetation_index_results):

          results = json.loads(std_out.decode())

          for index, listing in enumerate(results['wf_outputs'][0][0]['listing']):

              if listing['class'] == 'Directory':
```

Figure 42. Chaining Notebook - vegetation index 2



Burned area delineation product

```
[ ]: cwl_burned_area = 'cwl/burned-area.cwl'
```

```
[ ]: with open(cwl_burned_area) as file:

         burned_area_cwl = yaml.load(file,
                                     Loader=yaml.FullLoader)
```

```
[ ]: cwl_burned_area_workflow_id = get_workflow_class(burned_area_cwl)['id']

     cwl_burned_area_workflow_id
```

```
[ ]: get_workflow_class(burned_area_cwl)['inputs']
```

Create the parameters file for CWL and invoke the CWL runner tool

```
aoi: POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508
  -36.108))
ndvi_threshold: '0.19'
ndwi_threshold: '0.18'
post_event: {class: Directory, path: /workspace/application-chaining/o27byuyx/stac-results}
pre_event: {class: Directory, path: /workspace/application-chaining/jn0iisx4/stac-results}
```

```
[ ]: params = {'aoi': 'POLYGON((136.508 -36.108,136.508 -35.654,137.178 -35.654,137.178 -36.108,136.508 -36.108))',
               'ndvi_threshold': '0.19',
               'ndwi_threshold': '0.18',
               'pre_event': {'class': 'Directory', 'path': os.path.dirname(vegetation_index_stac_catalogs[0])},
               'post_event': {'class': 'Directory', 'path': os.path.dirname(vegetation_index_stac_catalogs[1])}
              }

     params_file = os.path.join(os.getcwd(), 'burned-area.yml')


     with open(params_file, 'w') as yaml_file:

         yaml.dump(params,
                   yaml_file,
                   default_flow_style=False)
```

```
[ ]:
     cmd_args = [cwltool,
                 '--no-read-only',
```

Figure 43. Chaining Notebook - burned area delineation

8.2.3.6.2. Approach 2 - Application Chaining using CWL

Bob complements the application chaining activities with an approach that uses CWL as the only asset to chain Alice's Notebooks.

To do so, Bob creates and publishes two Docker images, one for each of Alice's software repositories.

Bob creates the same new Docker file as Eric did in the "Scripted execution using CWL and Docker" step:

```
FROM terradue/c7-repo2cli:latest

RUN repo2cli -r https://gitlab.com/terradue-ogctb16/eoap/d169-jupyter-nb/vegetation-
index.git

ENV PREFIX /opt/anaconda/envs/env_vi

ENV PATH /opt/anaconda/envs/env_vi/bin:$PATH
```

Bob builds the Docker image with:

```
$ docker build . -t terradue/vegetation-index:0.1
$ docker login
$ docker push terradue/vegetation-index:0.1
```

The CWL file is retrieved with:

```
$ docker run --rm -it vegetation-index:0.1 vegetation-index --cwl --docker
'terradue/vegetation-index:0.1'
```

The CWL script, printed to `stdout` is saved in an ASCII file named docker-vi.cwl.

Bob repeats the steps for the burned area software repository.

At this stage, Bob has:

- A docker-vi.cwl CWL workflow script.
- A docker-burned-area.cwl CWL workflow script.
- Two docker images, terradue/vegetation-index:0.1 and terradue/burned-area:0.1 published on DockerHub.

Bob now writes a CWL chaining script that uses the `SubworkflowFeatureRequirement` and the `ScatterFeatureRequirement` CWL requirements.

The `SubworkflowFeatureRequirement` allows Bob to reference the docker-vi.cwl and docker-burned-area.cwl in his chaining CWL workflow and thus avoiding copy/pasting the content of the CWL scripts to chain.

The `ScatterFeatureRequirement` is used to process in parallel the vegetation index step as the vegetation index processing step takes a single Sentinel-2 Level 2A acquisition at the time so there are two executions: One for the pre wildfire event acquisition and one for the post event acquisition.

Bob runs the CWL chaining script using the *cwltool* which pulls the Docker images and trigger their execution and generate the inputs and outputs automatically.

8.2.3.6.3. Final considerations on the chaining methods

Both application chaining approaches described above are valid and may address different scopes and/or targets.

The one using a Notebook may be more user friendly for CWL neophytes and is useful for supporting the development of the underlying application Notebooks and their chaining as it eases the development/test cycles.

The second method creates a more complex CWL script and relies on docker images and containers to run. This approach is better fit when the deployment of an application is the main goal. This approach has two interesting benefits behind it:

- The requirements to follow such an approach is to have docker and *cwltool*.
- The chaining CWL is an Application Package that can be deployed on an Exploitation platform as described in this section.

8.3. Geomatys (D170)

For Testbed 16, Geomatys built various Jupyter Notebooks to research a simple method for scientists to easily package their Jupyter Notebooks. This method will determine how to make a Jupyter Notebook batch executable (mandatory for ADES execution) and package all the components necessary to its deployment as a service in an Exploitation Platform supporting the ADES/EMS approach.

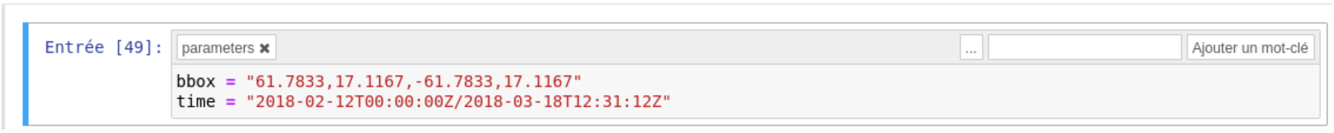
A first Notebook demonstrates how a Notebook can interact with a DAPA service. Below are described the different steps to use, package and deploy this Notebook into an ADES/EMS plateform. Three other Notebooks where developed in order to demonstrate other ways of chaining processes not using CWL/BPMN as done in previous testbeds. The TB-16 approach was creating a workflow directly into a Jupyter Notebook.

- NDVIMultiSensor : produce NDVI GeoTIFF from various input data (Sentinel2, Deimos, PROBA-V)
- NDVIStacker : produce NDVI GeoTIFF merged from multiple GeoTIFF.
- NDVIWorkflow : a workflow chaining the two previous Notebooks.

8.3.1. DAPA service Notebook use case

Alice create a Notebook that requests a DAPA service (https://t16.ldproxy.net/ghcnd/), serving station-based measurements from land-based stations worldwide. This Notebooks takes Area of Interest (AoI) and Time of Interest (ToI) input parameters and return a JSON file containing the matching stations.

The first cell of the Notebook contains the variables AoI and ToI to fill. The cell has a metadata tag "parameters" that enables `papermill` batch execution.



```
bbox = "61.7833,17.1167,-61.7833,17.1167"
time = "2018-02-12T00:00:00Z/2018-03-18T12:31:12Z"
```

The next cellules performs a search query on the DAPA service and builds the JSON result file with the matching stations.

```
{
    "stations": [{
            "name": "ST JOHNS COOLIDGE FLD",
            "geometry": "POINT (-61.7833 17.1167)",
            "elevation": 10.1,
            "observations":
"http://t16.ldproxy.net/ghcnd/collections/observation/items?locationCode=ACW00011604"
        }, {
            "name": "ST JOHNS",
            "geometry": "POINT (-61.7833 17.1333)",
            "elevation": 19.2,
            "observations":
"http://t16.ldproxy.net/ghcnd/collections/observation/items?locationCode=ACW00011647"
        }, {
            "name": "QAIROON HAIRITI",
            "geometry": "POINT (54.083 17.25)",
            "elevation": 881.0,
            "observations":
"http://t16.ldproxy.net/ghcnd/collections/observation/items?locationCode=MUM00041315"
        }]
}
```

8.3.2. Application package creation use case

Alice must produce the following items :

- The Jupyter Notebook (e.g. dapa-stations.ipynb)
- The `Conda` environment file (e.g. environment.yml)
- A CWL file (e.g. dapa-stations.cwl)
- A WPS process description used for ADES deployment

8.3.3. Notebook and environment creation use case

Alice builds her Notebook and installs all the dependencies needed using `Conda`.

Once the Notebook is ready to be shared, she exports all the installed dependencies by using:

```
conda env export --no-builds -f environment.yml
```

8.3.4. WPS Process description / CWL creation

In order to execute the Jupyter Notebook in batch mode, a generic method using `papermill`, CWL and a bash script that need to be incorporated into the Docker image was developed.

The following generic bash script run.sh transforms the input parameters into a `papermill` command line :

```
arguments="$@";
command="papermill ";
parameters="";
index=0;
multi=0;
for arg in "$@"; do
    if [ $index -gt 1 ];
    then
        case ${arg} in
        -S*) propName=$(echo $arg | cut -c 3-)
             parameters=${parameters}${propName}':'
             multi=0;;
        -M*) propName=$(echo $arg | cut  -c 3-)
             parameters=${parameters}${propName}':\n'
             multi=1;;
         *)  if [ $multi -eq 1 ];
             then
                parameters=${parameters}' - '${arg}'\n'
             else
                parameters=${parameters}' '${arg}'\n'
             fi
    esac
    else
        command="${command} ${arg}"
    fi
    index=$((index+1))
done

$command -f input.yml
```

This script transforms the inputs sent to the script from a command line, into a YAML file used by `papermill` to override the input parameters in the Jupyter Notebook and then call `papermill` to execute the Notebook in batch mode.

The supported parameters are limited to literal inputs : Numeric, character string, files and URL. The single parameters are prefixed with "-S" and the multiple parameters by "-M"

The CWL file calls this script within the Docker image and must map the parameters using the same prefix:

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0
class: CommandLineTool

hints:
  DockerRequirement:
    dockerPull: images.geomatys.com/tb16/dapa-stations:latest
arguments: ["run.sh", "dapa-stations.ipynb","out.ipynb"]
inputs:
  bbox:
    type: string
    inputBinding:
      position: 1
      prefix: -Sbbox
  time:
    type: string
    inputBinding:
      position: 2
      prefix: -Stime


outputs:
  output:
    outputBinding:
      glob: "./result.json"
    type: File
```

Finally the WPS process description specifies the inputs and outputs, with their types and cardinality. The process description also specifies the CWL location and docker image. The ADES needs this information in order to pull and execute the Jupyter Notebook.

```
    "processDescription": {
        "process": {
            "id": "dapa-stations",
            "title": "DAPA stations",
            "owsContext": {
                "offering": {
                    "code": "http://www.opengis.net/eoc/applicationContext/cwl",
                    "content": {
                        "href": "https://raw.githubusercontent.com/Alice/dapa-
stations/dapa-stations.cwl"
                    }
                }
            },
            "abstract": "A juyter Notebook searhing stations on a DAPA service",
            "keywords": ["DAPA"],
            "inputs": [
                {
                    "id": "bbox",
                    "title": "Area Of Interest",
                    "minOccurs": "1",
                    "maxOccurs": "1",
                    "input": {
                        "literalDataDomains": [{
                                "dataType": {
                                    "name": "String",
                                    "reference": "http://www.w3.org/TR/xmlschema-
2/#String"
                                }
                            }]
                    }
                },{
                    "id": "time",
                    "title": "Time Of Interest",
                    "minOccurs": "1",
                    "maxOccurs": "1",
                    "input": {
                        "literalDataDomains": [{
                                "dataType": {
                                    "name": "String",
                                    "reference": "http://www.w3.org/TR/xmlschema-
2/#String"
                                }
                            }]
                    }
                }
            ],
            "outputs": [{
                    "id": "output",
                    "title": "station output file",
                    "output": {
                        "formats": [{
                                "mimeType": "application/json",
                                "default": true
                            }]
                    }
                }]
        },
        "processVersion": "1.0.0",
        "jobControlOptions": [
            "async-execute"
        ],
        "outputTransmission": [
            "reference"
        ]
    },
    "immediateDeployment": true,
    "executionUnit": [{
            "href": "images.geomatys.com/tb16/dapa-stations:latest"
        }],
    "deploymentProfileName": "http://www.opengis.net/profiles/eoc/workflow"
}
```

NB: This json file will be later referred to as `dapa-station-wps-description.json`.

8.3.5. Packaging

Alice then hosts those files on a GitHub repository with the following structure:

```
dapa-stations
|-- binder
      |-- environment.yml
|-- dapa-stations.ipynb
|-- run.sh
|-- dapa-stations.cwl
```

NB: The CWL does not need to be in this repository. The CWL can be hosted anywhere.

Alice needs now to generate a Docker container and host it on a Docker repository.

To do this she can use `jupyter-repo2docker`. This tool creates a Docker image from a GitHub repository.

```
jupyter-repo2docker --image-name "images.geomatys.com/tb16/dapa-stations:latest"
https://github.com/Alice/dapa-stations/
```

```
docker push images.geomatys.com/tb16/dapa-stations:latest
```

8.3.6. Execution

Eric wants to retrieve Alice's Notebook for modification and local execution.

```
# Eric clone the gitlab repository
git clone https://github.com/Alice/dapa-stations/ .
```

8.3.6.1. Local Execution using jupyter-Notebook (interactive mode)

Eric locally executes the process using `jupyter-Notebook`:

```
    # reproduces the environment
    conda env create --file environment.yml

    # start jupyter-Notebook
    jupyter-Notebook

    # extract Jupyter Notebook URL in output
    example:
http://127.0.0.1:8888/?token=7bc6f1e07e131a994f24498c4677413274e94c77784657d6

    # open the url on a browser and select "dapa-stations.ipynb" file

    # execute the Notebook
```

The above process produces a result.json file.

8.3.6.2. Local Execution using bash script (batch mode)

Eric executes the Notebook using the `bash` script provided by Alice:

```
    # reproduces the environment
    conda env create --file environment.yml

    # execute the Jupyter Notebook in batch mode with the bash script
    ./run.sh dapa-stations.ipynb out.ipynb -Sbbox 61.7833,17.1167,-61.7833,17.1167
-Stime 2018-02-12T00:00:00Z/2018-03-18T12:31:12Z
```

The above process produces a result.json file.

8.3.6.3. Local Execution using papermill (batch mode)

Eric executes the Notebook with `papermill`:

```
    # reproduces the environment
    conda env create --file environment.yml

    # execute the Jupyter Notebook in batch mode with papermill
    papermill NDVIMultiSensor.ipynb out.ipynb -y "
    bbox: 61.7833,17.1167,-61.7833,17.1167,
    time: 2018-02-12T00:00:00Z/2018-03-18T12:31:12Z
    "
```

The above process produces a result.json file.

8.3.6.4. Local execution using docker (interactive mode)

Eric does not want to modify Alice’s Notebook, so he can use the Docker image directly

```
    # Work directory is current directory
    export WORKDIR=`pwd`

    # start interactive Jupyter Notebook
    docker run -v ${WORKDIR}/result.json:/result.json -p 8888:8888
images.geomatys.com/tb16/dapa-stations:latest

    # extract Jupyter Notebook URL in docker output
    example:
http://127.0.0.1:8888/?token=7bc6f1e07e131a994f24498c4677413274e94c77784657d6

    # open the url on a browser and select "dapa-stations.ipynb" file

    # Execute the cells
```

The result (e.g. result.json) is available within ${WORKDIR}.

8.3.6.5. Local execution using docker (Batch mode)

Eric executes the process in batch mode using Docker:

```
    # Work directory is current directory
    export WORKDIR=`pwd`

    # execute the Notebook by calling the script run.sh
    docker run -v -v ${WORKDIR}/result.json:/result.json
images.geomatys.com/tb16/dapa-stations:latest ./run.sh  dapa-stations out.ipynb
-Sbbox 61.7833,17.1167,-61.7833,17.1167 -Stime 2018-02-12T00:00:00Z/2018-03-
18T12:31:12Z
```

The result (e.g. result.json) is available within ${WORKDIR}.

8.3.6.6. Local Execution using cwl-runner (batch mode)

Eric executes the process using `cwl-runner`:

Eric writes parameters in a JSON file that references the input files to process:

```
{
    "bbox": "62.1,16.2,-62.1,17.2",
    "time": "2018-02-12T00:00:00Z/2018-03-18T12:31:12Z"
}
```

Eric executes the `cwl`:

```
    # Work directory is current directory
    export WORKDIR=`pwd`

    cwl-runner --no-read-only --preserve-entire-environment
https://raw.githubusercontent.com/Alice/dapa-stations/dapa-stations.cwl params.json
```

The result (e.g. result.json) is available within ${WORKDIR}.

8.3.7. Deployment and execution on ADES

Bob want to deploy the `dapa-stations` application package on an XP platform.

8.3.7.1. WPS Deployment

Bob deploys the application package on a ADES WPS service with the following request (see WPS process description above):

```
    # Set ADES WPS URL
    export ADES_WPS_URL=https://tb16.geomatys.com/WS/wps/default

    # Set the path to the deploy json file
    export DEPLOY_PROCESS_JSON=dapa-station-wps-description.json

    # Deploy process
    curl -X POST \
    -i "${ADES_WPS_URL}/processes" \
    -H "Authorization: Bearer Th34cc3ssTok3nFrom4lice" \
    -H "Content-Type: application/json" \
    -d "@${DEPLOY_PROCESS_JSON}"
```

The response should be :

```
    HTTP/1.1 100

    {
    "id": "OK",
    "processSummary": {
        "id": "dapa-stations",
        "title": "DAPA stations",
        "keywords": [],
        "metadata": [],
        "additionalParameters": [],
        "version": "1.0.0",
        "jobControlOptions": ["async-execute"],
        "outputTransmission": ["reference"],
        "processDescriptionURL":
"https://tb16.geomatys.com/WS/wps/default/processes/dapa-stations",
        "abstract": ""
    }
    }
```

8.3.7.2. WPS Execution

Bob executes the application package on a ADES WPS service with the following request:

```
{
    "inputs": [
        {
            "id": "bbox",
            "input": {
                "dataType": {
                    "name": "string"
                },
                "value": "62.1,16.2,-62.1,17.2"
            }
        },{
            "id": "time",
            "input": {
                "dataType": {
                    "name": "string"
                },
                "value": "2018-02-12T00:00:00Z/2018-03-18T12:31:12Z"
            }
        }
    ],
    "outputs": [
        {
            "id": "output",
            "transmissionMode": "reference"
        }
    ],
    "mode": "async",
    "response": "document"
}
```

```
    # Set ADES WPS URL
    export ADES_WPS_URL=https://tb16.geomatys.com/WS/wps/default/processes

    # Set the path to the deploy json file
    export EXECUTE_PROCESS_JSON=execute.json

    # Execute process
    curl -X POST \
    -i "${ADES_WPS_URL}/processes/dapa-stations/jobs" \
    -H "Authorization: Bearer Th34cc3ssTok3nFrom4lice" \
    -H "Content-Type: application/json" \
    -d "@${EXECUTE_PROCESS_JSON}"
```

The response should be:

```
    HTTP/1.1 201 Crée
    Server: Apache-Coyote/1.1
    Access-Control-Allow-Origin: *
    Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
    Access-Control-Allow-Headers: Origin, access_token, X-Requested-With, Content-
Type, Accept
    Access-Control-Allow-Credentials: true
    Location: https://tb16.geomatys.com/WS/wps/default/processes/dapa-
stations/jobs/00d2a55c-3193-471e-bbbe-937a75527ce6
    Cache-Control: no-cache, no-store, max-age=0, must-revalidate
    Pragma: no-cache
    Expires: 0
    X-XSS-Protection: 1; mode=block
    X-Content-Type-Options: nosniff
    Content-Length: 0
    Date: Tue, 05 May 2020 10:42:34 GMT
```

The **Location** property indicates the URL to access the job status:

```
# Bob get the status of the job
curl -X GET "${ADES_WPS_URL}/processes/dapa-stations/jobs/0ee6840c-61d1-4fca-b231-
82cd01249f1d"
```

The response should be :

```
{"status":"succeeded","message":"Process completed.","jobId":"00d2a55c-3193-471e-bbbe-
937a75527ce6"}
```

Once the status is success (i.e. {"status":"succeeded"}), Bob can get the result of the job:

```
curl -X GET "${ADES_WPS_URL}/processes/dapa-stations/jobs/0ee6840c-61d1-4fca-b231-
82cd01249f1d/result"
```

The response should be:

```
    {
    "outputs": [{
            "id": "output",
            "href": "https://tb16.geomatys.com/WS/wps/default/products/00d2a55c-3193-
471e-bbbe-937a75527ce6-results/result.json"
        }]
    }
```

8.3.8. Workflow using Jupyter Notebook

During previous OGC Testbeds, process workflows where orchestrated and executed through various workflow languages such as CWL/BPMN. In TB-16 Geomatys experimented with the orchestration of a workflow directly inside a Jupyter Notebook.

We built two Notebooks:

- The first (NDVIMultiSensor) takes sentinel-2/Deimos/PROBA-V data and performs a NDVI computation and then returns GeoTIFF data files.
- The second (NDVIStacker) takes multiple GeoTIFF files, stacks them together, and returns a single GeoTIFF file.

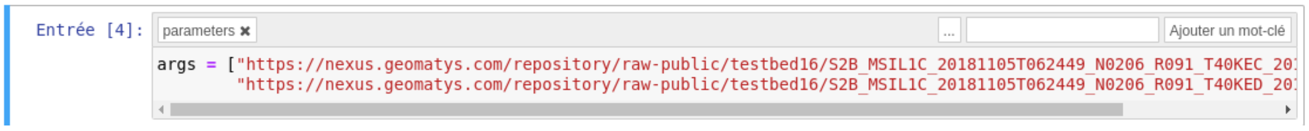
This two Jupyter Notebooks are packaged with the method described above and deployed to an EMS/ADES platform. Geomatys then built a third Notebook taking sentinel-2/Deimos/PROBA-V data as input, called the first NDVIMultiSensor process through the ADES WPS API, retrieved the results, and then called the second NDVIStacker process with the previous results. The workflow chaining the two process is achieved within the Notebook.

To perform WPS call, Geomatys wrote a single simple Python cell that sent an execute request, looped on the getStatus method, and finally called the getResult method.

This workflow Notebook can be packaged and deployed as any other Juptyer Notebook, making it available in batch mode for anyone.

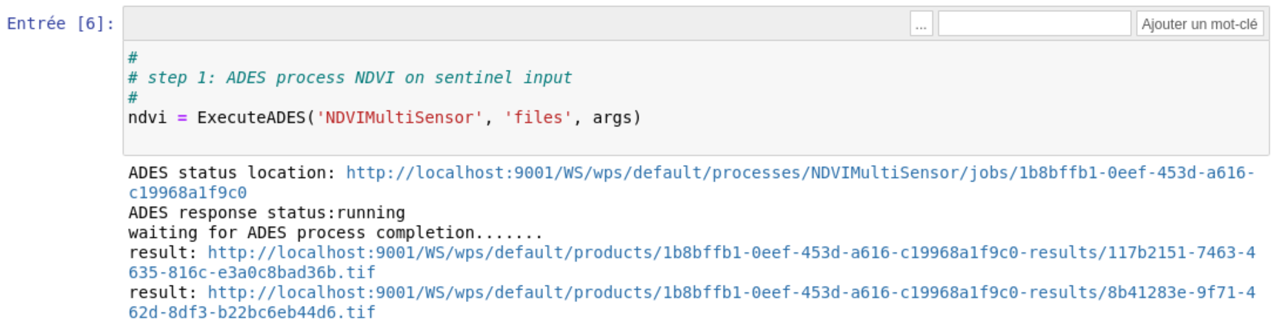
8.3.8.1. Notebook execution

The first cell contains input parameters (taggued as parameters for papermill in order to be overriden in batch mode).



```
args = ["https://nexus.geomatys.com/repository/raw-public/testbed16/S2B_MSIL1C_20181105T062449_N0206_R091_T40KEC_20
        "https://nexus.geomatys.com/repository/raw-public/testbed16/S2B_MSIL1C_20181105T062449_N0206_R091_T40KED_20
```

Then we called the process NDVIMultiSensor on ades with the specified parameters:



```
#
# step 1: ADES process NDVI on sentinel input
#
ndvi = ExecuteADES('NDVIMultiSensor', 'files', args)
```

```
ADES status location: http://localhost:9001/WS/wps/default/processes/NDVIMultiSensor/jobs/1b8bffb1-0eef-453d-a616-
c19968a1f9c0
ADES response status:running
waiting for ADES process completion.......
result: http://localhost:9001/WS/wps/default/products/1b8bffb1-0eef-453d-a616-c19968a1f9c0-results/117b2151-7463-4
635-816c-e3a0c8bad36b.tif
result: http://localhost:9001/WS/wps/default/products/1b8bffb1-0eef-453d-a616-c19968a1f9c0-results/8b41283e-9f71-4
62d-8df3-b22bc6eb44d6.tif
```

The ADES returns references to step 1 results (two NDVI geotiff computed from the sentinel2 source):

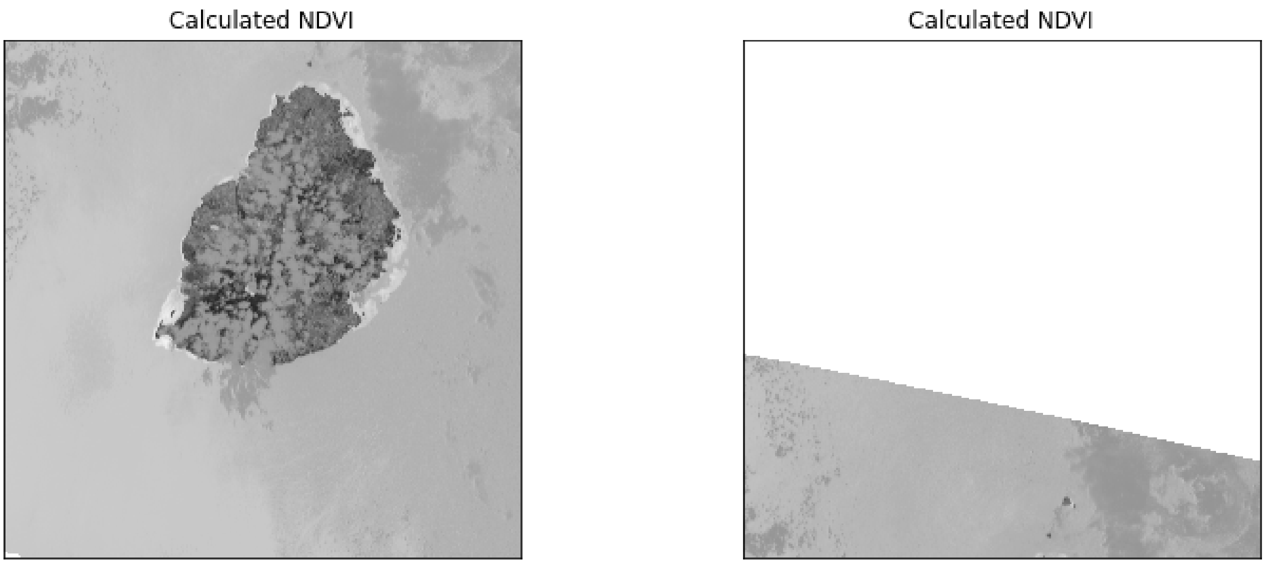
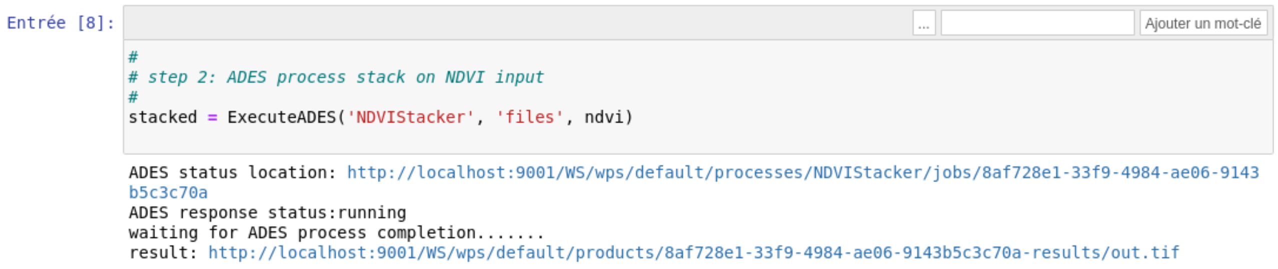


Figure 44. Two NDVI geotiff computed from the sentinel2 source

The second process NDVIStacker is called with the results obtained from the previous process:



```
Entrée [8]:
#
# step 2: ADES process stack on NDVI input
#
stacked = ExecuteADES('NDVIStacker', 'files', ndvi)
```

```
ADES status location: http://localhost:9001/WS/wps/default/processes/NDVIStacker/jobs/8af728e1-33f9-4984-ae06-9143
b5c3c70a
ADES response status:running
waiting for ADES process completion.......
result: http://localhost:9001/WS/wps/default/products/8af728e1-33f9-4984-ae06-9143b5c3c70a-results/out.tif
```

The ADES return a reference to a new geotiff combination of the two previoulsy computed.

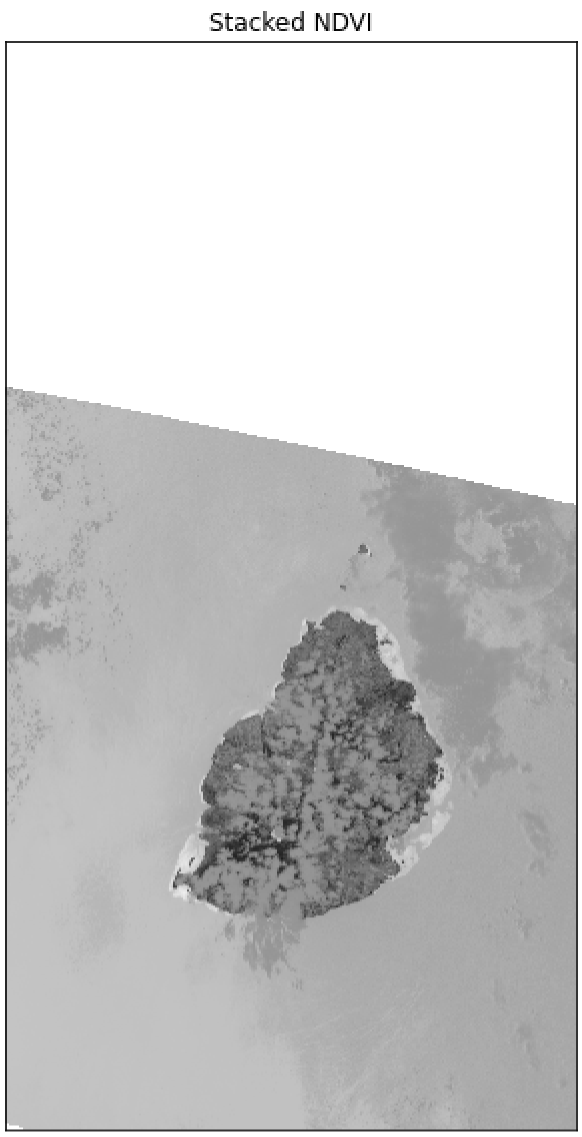


Figure 45. A new geotiff combination of the two previoulsy computed

The final step is to download and save the GeoTIFF file in order to make it available to the user (and return it in batch mode):

Entrée [11]:

... Ajouter un mot-clé

```
#
# prepare final result
#
import os
from pathlib import Path
import uuid

path = str(Path.home()) + "/outputs/"

if not os.path.exists(path):
    os.makedirs(path)

dlRes = []
for p in stacked:
    dlpath = path + str(uuid.uuid4())
    url = p
    ext = ''
    if '.' in url: ext = '.' + url.rsplit('.', 1)[1]
    r = requests.get(url, allow_redirects=True)
    open(dlpath + ext, 'wb').write(r.content)
    p = dlpath + ext
    print(p)
    dlRes.append(p)
```

```
/home/guilhem/outputs/f775c2a8-913a-406f-9f8f-c1ce9de30013.tif
```

8.4. Technology Integration Experiments

The deployment and execution of the Jupyter Notebook (D168, D169) on the ADES endpoint (D170, D171) has been tested to demonstrate the interoperability of the components as the Technical Integration Experiments (TIEs) listed below:

Identifier	Notebook	ADES Endpoint
TIE-1	52°North	Geomatys
TIE-2	TerraDue	TerraDue

As exposed in the earlier sections, the Jupyter Notebooks are containerized with all required dependencies. Therefore, the interoperability only depends on the availability of the services and data consumed by the Notebook. In a real case scenario, the user discovers the data provided by the platform and submit some products available to the executed application.

Other components combinations have not been executed as the Jupyter Notebook and ADES endpoint from Terradue relies on the STAC approach described earlier in this document.

Chapter 9. Conclusions

9.1. Outcomes

Previous OGC Testbeds developed an architecture for deploying and executing processing applications close to the data products hosted by Earth Observation (EO) platforms. The aim of the Testbed-16 EO Thread was to complement that approach with applications based on Project Jupyter that enables developing Notebooks. The support of non-interactive notebooks was the key aspect demonstrated during the project.

As intended, the Docker Application Package defined for the ADES can embed any kind of software including Jupyter Notebooks. Therefore, the challenge was providing a command-line interface to the notebook. Both an approach based on `papermill` (not requiring any code implementation) and an approach on `nbconvert` (enabling a greater flexibility) were demonstrated. The implemented scenarios also showed that describing the Notebook interface in a self-contained Python dictionary enables automated generation of the Application Descriptor.

Additionally, participants demonstrated that various technologies introduced in previous Testbeds (CWL, STAC, etc.) could be managed easily from within the Notebook. A method for building a container from a `git` repository, relying on repo2Docker, was also introduced and might be reused for any kind of application that needs to be Containerized.

Finally, new approaches for building processing chains were prototyped:

- The chaining of processing steps (including notebook-based application) based on CWL.
- The interactive development of code-based workflows using Jupyter Notebook which appeared to be a very straightforward and flexible.

9.2. Future Work

Provisioning an interactive notebook environment to EO platforms users is a key advantage of Jupyter which potentially offers numerous benefits:

- Means for operating all platform services through **user-friendly API libraries** (including a possible integration of OpenEO);
- Integrated interactive environment for developing applications deployed on the platform;
- Collaboration on shared notebooks;
- Automated ADES packaging of applications.

Only a mockup of the possible solution design was discussed in this report. The Testbed particpants recommend going further by extending the EO XP Platform architecture with the outstanding features of interactive Jupyter Notebook.

Appendix A: Revision History

Table 1. Revision History

Date	Editor	Release	Primary clauses modified	Descriptions
November 19, 2020	C. Noël	1.0	all	version released to EO EX Platform DWG

Appendix B: Bibliography