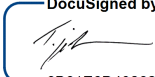


Customer	: ESA	Document Ref	: EOEPKA-TN-012
Contract No	: 40000125714/18/I-LG	Issue Date	: 28/02/2024
WP No	: WP12000	Issue	: 1.0

**Title** : **EO Exploitation Platform Common Architecture – GTIF Use Case for EO Exploitation Platform Architecture Project**

**Abstract** : Green Transition Information Factory use case Task 1: potential integration of openEO and EOEPKA architectures

**Author** : 

DocuSigned by:  
  
6B61E6D4026241A...

Thomas Jellicoe  
Software Developer

**Approval** : 

DocuSigned by:  
  
AC7DD627110842B...

Simon Farman  
Project Manager

**Distribution** :  
  
**Hard Copy File:**  
**Filename:** EOEPKA-TN-012\_1.0.doc

*This document, produced by Telespazio UK Ltd under ESA's EOEPKA project, is published under the OSI approved Apache 2.0 Licence."*

**Telespazio VEGA UK Ltd**  
**350 Capability Green, Luton, Bedfordshire LU1 3LU, United Kingdom**  
**Tel: +44 (0)1582 399 000 Fax: +44 (0)1582 728 686**  
**<https://telespazio.co.uk/>**

**TABLE OF CONTENTS**

<b>1. INTRODUCTION .....</b>	<b>4</b>
1.1 Purpose and Scope .....	4
1.2 Structure of the Document.....	4
1.3 Referenced Documents .....	4
<b>2. OVERVIEW .....</b>	<b>5</b>
<b>3. WORK PERFORMED .....</b>	<b>6</b>
3.1 Project Outcomes .....	6
3.2 EOEPCA and ADES .....	7
3.2.1 Python and CWL .....	7
3.2.2 Convert Script.....	8
3.2.3 Python and EOEPCA .....	8
3.3 Open Earth Observation (openEO) .....	8
3.3.1 Google Earth Engine .....	9
3.3.2 Local Processing .....	9
3.3.3 openEO Platform Cloud Processing .....	10
3.3.4 openEO User-Defined Functionality .....	11
3.3.4.1 openEO User-Defined Processes .....	11
3.3.4.2 openEO User-Defined Functions .....	11
3.4 Integration of EOEPCA and openEO .....	12
3.4.1 openEO within EOEPCA .....	13
3.4.1.1 openEO CWL Implementation - Development .....	14
3.4.1.2 openEO CWL Implementation – Further Work .....	14
3.4.2 EOEPCA within openEO .....	15
3.4.2.1 Using UDFs and UDPs.....	15
3.4.2.2 openEO Specification .....	16
3.4.2.2.1 Running CWL Scripts Locally .....	16
3.4.2.2.2 Getting data from openEO into EOEPCA.....	17
3.4.2.2.3 openEO Specification Updates .....	17
3.4.2.3 openEO Implementation.....	19
3.4.2.4 Putting it all together.....	20
3.5 Future Work .....	21
<b>4. PROJECT CONCLUSIONS .....</b>	<b>23</b>
<b>5. OTHER INFORMATION.....</b>	<b>24</b>
5.1 Git Repositories .....	24
<b>6. GLOSSARY.....</b>	<b>25</b>

**AMENDMENT POLICY**

This document shall be amended by releasing a new edition of the document in its entirety.  
The Amendment Record Sheet below records the history and issue status of this document.

**AMENDMENT RECORD SHEET**

ISSUE	DATE	DCI No	REASON
1.0	28/02/2024	N/A	Issue for submission

## **1. INTRODUCTION**

### **1.1 Purpose and Scope**

This document provides an overview of the work done to complete Task 1 of the GTIF Use Case work under contract 4000125714/18/I-LG. Focusing on the possible integration of the openEO and EOEPCA platforms.

### **1.2 Structure of the Document**

This document follows a similar template to previous Technical Note style documents and covers the full narrative workflow for this work package, from initial prototyping to final conclusions.

### **1.3 Referenced Documents**

The following is a list of documents with a direct bearing on the content of this report. Where referenced in the text, these are identified as Rd., where 'n' is the number in the list below:

RD.1 NoR Sponsorship Request – Network of Resources, 22/11/2023

RD.2 GTIF Use Case Statement of Work, 4000125714/18/I-LG

## **2. OVERVIEW**

This report summarises an investigation into potential integration of openEO and EOEPCA architectures.

This project focuses on two Earth Observation services, EOEPCA and openEO, that allow for data access and processing of EO resources. While the final outcome is to define some method of integration between the two services, the route of the project serves as a good introduction to both specifications taking you through the engineer's full method of development. This includes early experimentation and prototyping to gain a better understanding of each service, and defining updates to the specification to provide new functionality.

The analysis carried out under this project seeks to answer the question of whether there is potential for interaction between EOEPCA and openEO for Earth Observation data access and processing. Through carrying out this analysis we will start to understand where the building blocks for these two services differ and how we might be able to improve one through integration with the other. Such integration could help to avoid time spent re-implementing processes that are already defined in one of the respective application package formats – either as an openEO process or an OGC application package for execution within EOEPCA.

While both EOEPCA and openEO can be deployed locally on your own machine, for example by using Minikube or [client-side processing](#) respectively, openEO is better understood by gaining access to one of the provided back-ends to experiment with some of the built-in processes. To access these processes, it was suggested we gain access to the openEO Platform which includes the Vito back-end, among others. This service provides access to cloud processing environments for execution of process graphs as well as offering a web editor UI for defining them visually. A free 30-day trial of openEO Platform is available but to allow for future project expansion, we extended this access via a NOR request registered with ID **3b22qR**.

We will make frequent reference throughout this document to the terms 'OGC Application Package', which is a package containing CWL script(s), defining algorithms that can be executed on a given input. In EOEPCA this execution is done via the ADES. Within openEO, process definitions are instead provided as 'openEO Process Graph(s)', defined in JSON, containing nodes that represent individual operations which combine in sequence to define an algorithm which is then executed on the data.

### 3. WORK PERFORMED

The work done under this project has been completed in stages, to allow for greater levels of experimentation and understanding of underlying architectures. This also allows for a gradual increase in complexity to develop prototype applications that serve as a proof of concept for future integration between EOEPCA and openEO. The following subsections define each of the stages developed during this project with the later sections then focusing on potential future development in this area to extend the EOEPCA-openEO integration.

We have broken down this project into two main areas for consideration:

1. openEO processing could be used as steps within an EOEPCA workflow, with each step passing data to an openEO back-end for processing and the output being returned to the EOEPCA workflow. This allows processes to be built in openEO using any of the offered clients (Python, R or JavaScript) or the raw JSON definitions, which can then be executed just as any other EOEPCA application, provided a CWL script is defined alongside the containerised openEO code.
2. EOEPCA processing could be used as steps (processing nodes) within an openEO process graph. Each node can be used to execute an EOEPCA application package, defined as a CWL script, being run either locally or via the ADES, with the output returned to the openEO process for further processing.

Suggestion 2 allows complex EOEPCA processing to be executed within an openEO environment, potentially avoiding the need to reimplement complex algorithms in an openEO back-end. This can also help to divide up processing among different back-ends with EOEPCA processing being done externally via the ADES. For computationally expensive algorithms this could be extremely useful, particularly with certain openEO back-ends, if they offer limited processing capability.

#### 3.1 Project Outcomes

In this section we introduce the most significant outcomes – namely, the integration of each processing workflow approach as steps within the other – resulting in the possibility for hybrid workflows that combine both approaches.

These are described in detail in sections 3.4.1 and 3.4.2, each defining the work and providing example executables that demonstrate both methods of integration between EOEPCA and openEO. A summary of each is provided below.

##### Calling openEO processes from an OGC application package

In the first instance, we have successfully been able to define an openEO process graph and executed it just as an OGC application package. We achieved this using the openEO Python Client, chosen as it is the most familiar option for the developer, containerising this script using Docker and then generating a CWL script to invoke this image when required. This script then uses openEO functionality to access data via the Google Earth Engine back-end at a specified location and time, computes the NDVI or NDWI on this dataset, saves it as a TIF file and then exports the data in a STAC-catalogue format. Note, the STAC output data is only a draft definition here to ensure integration with the EOEPCA stage-out process. Further work will be required to ensure the STAC data is accurate.

The code created to demonstrate this integration can be cloned from a GitHub repository (available here: [https://github.com/tjellcoie-tpzuk/openeo\\_GEE/tree/main](https://github.com/tjellcoie-tpzuk/openeo_GEE/tree/main)). Both the CWL script defining the application package itself, **get-eo-data-wrflw.cwl**, and a sample http file, **openeo-export-app.http**, are provided for user experimentation.

##### Executing OGC application packages within openEO process graphs

The second outcome from this project is more complex as it suggests an update to the openEO specification to allow OGC application packages (CWL scripts) to be executed directly within openEO process graphs. The work completed in this section demonstrates, and provides an implementation of, a method to execute OGC application packages within openEO process graphs. These EOEPCA processes can be incorporated, no changes required, into openEO process graphs, with only a CWL script pointing to a containerised application being required.

The update to the openEO specification is provided as three new processes, defined in JSON files, just as with all openEO built-in processes. These three processes define: a function to prepare input data for execution via the ADES, a function to execute a CWL script on this data remotely, via the ADES, and a final function to execute a CWL script locally within the openEO back-end. These process definitions are available in a GitHub repository (found here <https://github.com/tjellicoe-tpzuk/openeo-processes-cwl/tree/main>, see ***cwl\_preparation***, ***run\_cwl\_ades*** and ***run\_cwl\_local***.

In addition to the suggested specification update, we have also provided example implementations for each of these processes. Ordinarily, the implementation is provided independently, by the developers that are writing their own implementation of the openEO spec, rather than alongside the specification itself. However, as a proof of concept, and to allow experimentation, we have provided our own suggested implementation of these new processes. The implementation is written in Python and can be found in the same GitHub repository as above. Defining the implementation also means we can provide some example process graphs as well as a Jupyter Notebook through which these graphs can be executed, with outputs ready to be exported for analysis. The Jupyter Notebooks are available in the GitHub repository here: <https://github.com/tjellicoe-tpzuk/openeo-processes-cwl/tree/main/examples>.

The other sections of this Technical Note help to document the full project workflow that enabled us to generate the outputs discussed above, from initial prototyping in EOEPCA using CWL, to executing scripts via the ADES and experimenting with the openEO Python Client. These sections provide an insight into the developer's workflow and could be used as inspiration for a future engineer to undertake a similar piece of work to build their knowledge and experience of both EOEPCA and openEO services.

## 3.2 EOEPCA and ADES

The Earth Observation Exploitation Platform Common Architecture (EOEPCA) aims to facilitate adoption of a freely available common architecture that supports a paradigm shift from “bring the data to the user” (i.e. user downloads data locally) to “bring the user to the data” (i.e. move user exploitation to hosted environments with collocated computing and storage). This leads to a platform-based ecosystem that provides infrastructure, data, computing and software as a service. A key component of EOEPCA is the Application, Deployment and Execution (ADES) which provides a platform-hosted execution engine through which users can initiate parameterised processing jobs using applications made available within the platform. As [the OGC Best Practice](#) specifies, application packages to be deployed via the ADES must be in the form of CWL files and this leads us to finding some way to integrate openEO scripts, most likely written in Python, within CWL files.

### 3.2.1 Python and CWL

As part of the EOEPCA Deployment Guide an example is provided to demonstrate the deployment of a CWL script via the ADES to convert a given image file by a scale percentage. Looking in the CWL script provided, available at <https://raw.githubusercontent.com/EOEPCA/convert/main/convert-url-app.cwl>, this relies on a shell script to reformat the file and save it locally as a STAC catalog item. This example demonstrates that CWL can easily be used to run containerised shell scripts and that the ADES can be used to execute the CWL. However, as openEO only provides clients in R, JavaScript and Python it would be a good step to understand how we can reimplement such functionality in one of these languages instead of just using the shell script. Therefore, as a quick proof of concept, and as an opportunity to build knowledge on the definition of CWL scripts, it was decided to produce a python script and

container image that can be deployed via the ADES to output a similar result as when using the example script. This will then help us understand the potential to write an openEO process within Python and potentially execute such a process graph within the EOEPCA application package.

### 3.2.2 Convert Script

To demonstrate that EOEPCA can work with Python scripts, we have developed the scripts available at [https://github.com/tjellicoe-tpzuk/cwl\\_python\\_convert](https://github.com/tjellicoe-tpzuk/cwl_python_convert). The Python code here has been containerised using Docker, and accompanying CWL scripts are also defined that run these Docker containers. The CWL script here can be run easily with the inputs as defined in the input<filetype>.yml files and will generate an output file with the image converted as specified by the *func* input variable.

### 3.2.3 Python and EOEPCA

The subsequent step to demonstrate Python integration with EOEPCA is to execute these scripts via the ADES on an EOEPCA deployment. This requires some minor tweaking to the CWL scripts, to create workflows, rather than command line tools, and the docker images need to be made available via DockerHub but otherwise the process is almost identical, provided an EOEPCA processing environment is running locally on the system. We can achieve this by using Minikube and the [EOEPCA helm charts](#) to build Kubernetes clusters locally. For more information on getting EOEPCA running, please see the deployment guide, available at <https://github.com/EOEPCA/deployment-guide>.

To see an example of this and to try running the deployment yourself, the updated CWL scripts and some example http requests can be found in the repository at [https://github.com/tjellicoe-tpzuk/ades\\_python\\_convert](https://github.com/tjellicoe-tpzuk/ades_python_convert).

This has therefore shown that Python scripts can be executed via the ADES just as with the previous shell script examples, leading to potential for openEO code to be executed in the same way.

## 3.3 Open Earth Observation (openEO)

Open Earth Observation (openEO) provides a specification for EO data processing through the construction of directed acyclic graphs built up of different process and sub-process nodes. Any EO data is then stored and processed using datacubes. The specification also defines several built-in processes that could be provided within an openEO implementation, including inputs, outputs and a description of the process itself.

Interaction with an openEO application can currently be achieved by writing code in Python, R or JavaScript, or by using the online visualisation workspace provided by [openEO Platform](#), for which you can access a 30 day free trial. Note, any interaction relies on some back-end being configured for use, this could be provided by Vito, EODC or Google Earth Engine for cloud computing, or could be provided locally through an individual implementation such as the Dask implementation discussed later in this document.

The final output of any openEO code is a JSON file defining a process graph, which itself can be made up of multiple processes and process graphs. See the example JSON definition below which loads a dataset and saves the output as a NETCDF. Note, when coding an openEO implementation, a process registry object must be created which associates process ids (seen below) with executable Python code.

```
{
  "process_graph": {
    "load_collection": {
      "process_id": "load_collection",
      "description": "Loading the data",
```



```

    "arguments": {
      "id": "Sentinel-2",
      "spatial_extent": {
        "west": 16.1,
        "east": 16.6,
        "north": 48.6,
        "south": 47.2
      },
      "temporal_extent": ["2018-01-01", "2018-02-01"],
      "bands": ["B02", "B04", "B08"]
    },
  },
  "save": {
    "process_id": "save_result",
    "arguments": {
      "data": {
        "from_node": "load_collection"
      },
      "format": "_ "
    },
    "result": true
  }
}

```

**Code 1: Example of JSON output for an openEO process graph to load some data and save it out to a file.**

This example only uses two standard processes which are defined in the openEO specification, but 241 built-in processes are provided within the openEO specification, for example *ndvi* and *atmospheric\_correction*. As stated earlier, it is important to remember that openEO serves as a specification for an EO implementation but does not define the implementation itself, this is handled within the individual back-ends or local applications.

### 3.3.1 Google Earth Engine

It was initially chosen to focus on the openEO back-end provided by Google Earth Engine, as this has an easy to navigate [webpage](#) identifying all available datasets and there is clear documentation to help with getting started. There is also no need to create an account to run processes on the GEE cloud platform, as they provide several test accounts which can be used. We created some sample scripts to extract data from the GEE server, apply some process on this data and then export the result as a TIF file. The code written to demonstrate this functionality is available in a GitHub repository found at [https://github.com/tjellicoe-tpzuk/openeo\\_GEE](https://github.com/tjellicoe-tpzuk/openeo_GEE). Consider running the *openeo-func* package with the variable *testing* set to True. Note how the script builds up the process graph from several PGNodes making use of the *reduce\_dimension* function to apply the processes across the bands. The additional CWL scripts in this repo are discussed later in this document.

While this serves as a good demonstration of openEO's functionality, in future it makes more sense to run a local openEO implementation (such as the Dask implementation), as this allows more flexibility when testing different processes. This also enables us to look at how the implementation has been written to understand any shortcomings or areas for potential improvement.

### 3.3.2 Local Processing

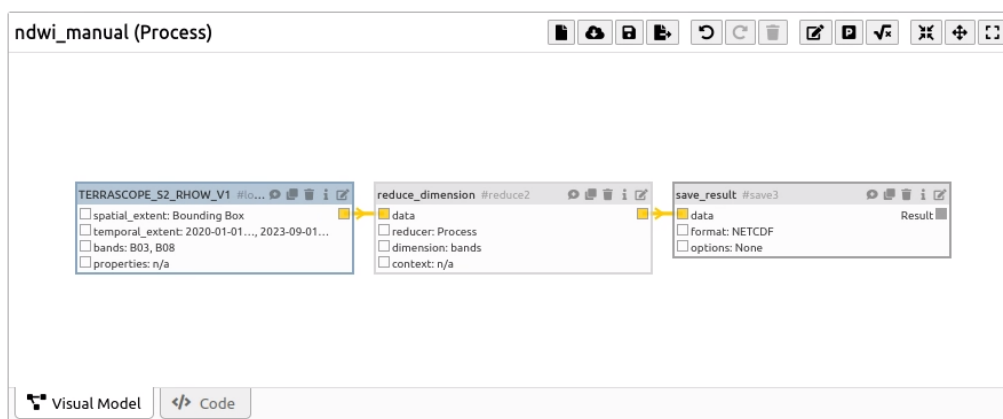
For a local processing environment it was suggested to look at the [Dask implementation](#) which we can build from source on our local machines and more easily analyse the code. As a brief introduction to

using this implementation, which also relies on the `openeo_pg_parser_networkx` package to execute openEO process graphs, we have implemented a [simple script](#), based on the [online guidance](#). This script loads data from a STAC item, computes the Normalized Difference Vegetation Index (NDVI) values for this data and saves the output as both a TIF and NETCDF file which we can then examine through use of software such as QGIS Desktop viewer. You can also see the outputs from each job printed to the terminal. As we can see, the Dask application implements the datacube data structures as Xarray DataArrays, which are quite nice to work with and allow easy outputting as NETCDF files.

### 3.3.3 openEO Platform Cloud Processing

The openEO Platform provides a new cloud processing and analytics environment built on top of openEO. It allows interfacing with multiple back-end providers and presents data around which processes can be run on which back-ends, automatically selecting the most suitable provider for a given processing graph. The Platform aims to bring openEO to production and offers data access and data processing services to the EO community. Software developers can work in their own programming IDE to integrate services into larger or dedicated applications. A key aspect of the Platform service is the openEO Editor which provides the ability to instantly convert any code to the JSON or a visual representation of the processing graph, see examples below for a process which loads a dataset and computes the Normalised Difference Water Index (NDWI) values for the data:

First as a visualised process graph



**Figure 1: Visualisation of a process graph in openEO Platform**

For simplicity the JSON and Python script examples are available in a GitHub repository at [https://github.com/tjellicoe-tpzuk/openeo\\_localProcessing/tree/main/openEO\\_exports](https://github.com/tjellicoe-tpzuk/openeo_localProcessing/tree/main/openEO_exports)

The above code also serves as a good introduction to the `reduce()` and `apply()` process types, although these processes are not explicitly used above, instead relying on `reduce_dimension()`, which are often required when carrying out computations on datacubes. A lot of processes work on single dimension or pixel values and so we need to tell openEO how to apply these to the larger multi-dimensional datacubes.

This ability to define and view processes in multiple formats improves our understanding of how openEO is interpreting the code being executed and helps us to see where we might be able to provide further improvements or new processes. It also provides three different ways to interact with openEO and consider how we might wish to integrate EOEPCA processing within the application.

### 3.3.4 openEO User-Defined Functionality

Now that we have had a brief introduction to openEO processing we can consider two of the more complex areas of functionality available within openEO that allow definition of more intricate procedures. The hope is that combining the following methods together could allow us to write new processes within a current back-end provider that can export data to a remote server and run CWL scripts on that data via the ADES. This would avoid having to make any changes to the openEO specification itself.

#### openEO User-Defined Processes

A user-defined process (UDP) provides the ability to define a set of processes that you may wish to repeat in future with a different dataset and potentially different context parameters. This can then be saved with a new process name by the user and recalled later as a step in a process graph. While similarly named, UDPs provided different functionality to user-defined functions (UDF) which are discussed in the next section.

In the context of this project, UDPs might be used to create steps that run specific processes via EOEPKA, with each of these UDPs containing some processes that first prepare the data for ADES execution and then run a CWL with this data as input. A UDP will have a parameter definition for the input data, which in our case is likely to either be a datacube itself or a link to a URL where this data can be found, and any additional context arguments for CWL input.

Below, we combine this UDP functionality with UDFs to allow Python code to be run on openEO Platform with user-defined inputs.

#### openEO User-Defined Functions

A user-defined function (UDF) provides the ability to run raw code as an openEO process. This functionality is provided exclusively via the Vito back-end and currently only two runtimes are provided, both being Python. Thus, for this project we will focus on executing Python code within an openEO process graph. The uploaded code must follow a set format as defined in the [documentation](#) and can take as inputs both the data to be processed and any additional context variables, given as a dictionary.

As an initial demonstration of a UDF and to get comfortable with the implementation of such, we will write a test function that calculates the Normalised Difference Water Index (NDWI), as this is not a built-in function provided by openEO, although this could be achieved by using the built-in *normalized\_difference* process. NDWI is calculated as in the formula below:

$$ndwi = \frac{G - NIR}{G + NIR}$$

where G is the green spectrum channel and NIR is the near-infrared channel.

Setting up the UDF can be done either using the Python client by defining the function as a string using the **openeo.UDF** declaration (Figure 1), by copying the code directly into the openEO web interface (Figure 2), by uploading the script to the openEO server and providing the file path or by providing a URI to the script.

```

udf = openeo.UDF("""
from openeo.udf import XarrayDataCube

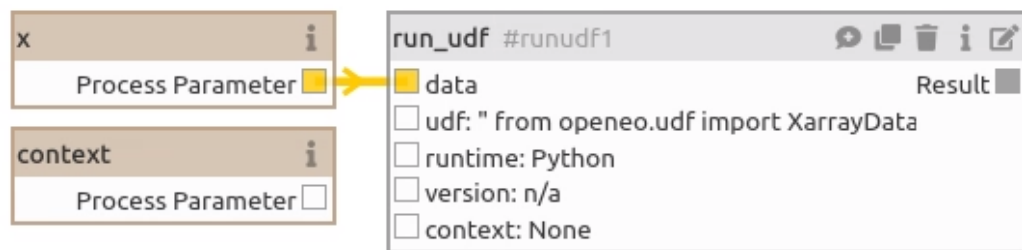
def apply_datacube(cube: XarrayDataCube, context: dict) -> XarrayDataCube:

    array_G = cube.sel(bands="B3")
    array_NIR = cube.sel(bands="B8")
    ndwi_cube = (array_G - array_NIR) / (array_G + array_NIR)
    return cube
""")

```

**Figure 2: Example of UDF defined using the openEO Python client**

In the openEO Platform web interface, the same code is inserted within a **run\_udf** process graph node.



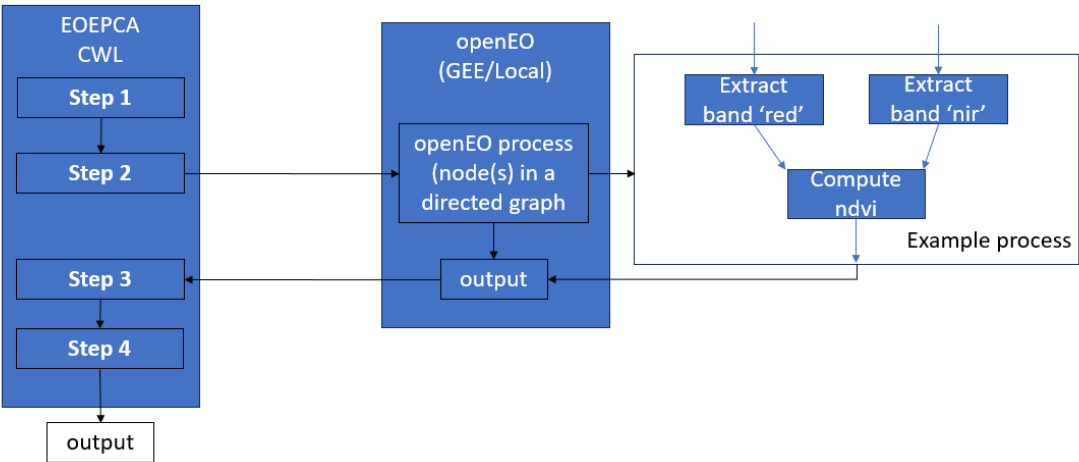
**Figure 3: Example of UDF defined within the openEO web interface**

Now we have the basis for running Python scripts within the openEO environment, we can start to consider expanding this functionality to work with more complex Python scripts. Potentially allowing us to create processes that run CWL scripts on an EOEPCA deployment via the ADES.

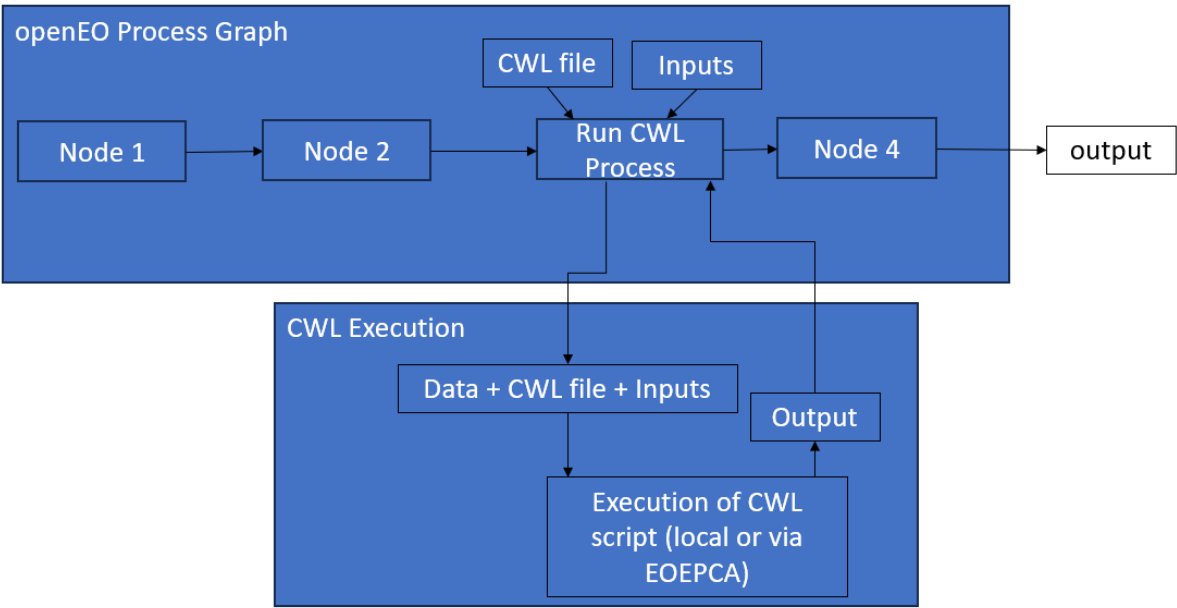
### 3.4 Integration of EOEPCA and openEO

There are two methods of integration we wish to consider under this project:

1. Running openEO scripts within an EOEPCA environment, allowing us to execute applications, via CWL, that make use of openEO processing. For example, having a step in the CWL script call out to an openEO processing graph to perform some a calculation on the input data, returning the output for further processing within EOEPCA. Consider the below diagram to demonstrate this integration:



2. Running OGC application packages, via execution of CWL scripts, as a process within an openEO process graph. For example, creating a process that can execute CWL scripts on a given input(s), returning the output for further processing within openEO. Consider the below diagram to demonstrate this integration:



3.4.1 openEO within EOEPKA

A major component of EOEPKA is the ADES package which allows applications to be executed in a platform-hosted (cloud or local) environment. To ensure conformance to the OGC Best Practices this application must be in the form of CWL script. Therefore, it is vital we can run openEO processes through CWL scripts and it is here that this section of the document will focus on. Once we can run openEO processes, whether it be a single process or a graph – openEO treats these the same – from a CWL script, we can then look to pass this CWL script to an EOEPKA deployment.

This section of the document discusses method 1 from the above introduction (see section 3.4)

### 3.4.1.1 openEO CWL Implementation - Development

To demonstrate the potential for EOEPKA to run openEO processes as steps, we will take the Python script defined in section 3.3.1, using the GEE back-end, containerise this script and create a CWL file to accompany this, allowing it to be executed as a command line tool. This CWL script can then be passed to an EOEPKA deployment via the ADES to execute openEO processing and output the results as specified in the Python script. Here, the Python code outputs a TIF file for the EO data along with an example STAC catalog definition, which currently serves as an example only and does not fully define the data contained within the TIF file.

The code developed for this prototype is available at [https://github.com/tjellicoe-tpzuk/openeo\\_GEE](https://github.com/tjellicoe-tpzuk/openeo_GEE) with the ADES deployment being handled via the *openeo-export-app.http* file. We suggest opening this file in VS Code with the REST Client extension installed to send the http requests and view the responses. Note, this requires a deployed EOEPKA instance and the outputs will be available via the Minio web client.

It is important to note that when running CWL scripts with version v1.2 or newer, you must set the `networkAccess` item, under the `NetworkAccess` requirement, to `true`. As has been done in the *get-eo-data.cwl* script.

When this CWL application is executed via the ADES, the python script creates a process graph in openEO to compute either the NDVI or NDWI, depending on the inputs. This process graph can be made as complex as desired, provided the inputs and outputs align with the CWL definition. This Python script can also make use of other back-end providers or can run a local openEO package, allowing more freedom in the data types being handled.

Note, the [CWL script](#) produced under this stage of the project serves as a good template if future openEO processing wishes to be initiated via the ADES. This can also be further integrated into more complex CWL scripts to include other processing steps, whether using openEO or some other processing functionality.

The above OGC application package demonstrates the possibility to run entire openEO process graphs within an EOEPKA deployment and shows that openEO processing can be executed regardless of platform, in this case using the ADES. If instead we wish to integrate openEO as a single step in a multi-step OGC application package, we can also achieve this. The script developed in the Git repository at [https://github.com/tjellicoe-tpzuk/openeo\\_eoepka\\_processing](https://github.com/tjellicoe-tpzuk/openeo_eoepka_processing) demonstrates the ability to use openEO processing as a single step in an EOEPKA process. Here we pass in a STAC item, which openEO then loads, computes the NDVI on and returns the output in a STAC-like format. This output can then be further processed by additional steps in our OGC application package.

### 3.4.1.2 openEO CWL Implementation – Further Work

A logical next step would then be to develop the integration between EOEPKA processing and openEO further and include multiple steps in the application package (CWL script) to call both openEO processes and prior EOEPKA functions, e.g. [snuggs.cwl](#) in the same execution. However, while we have had an initial attempt at this this, see the [openeo-download-ndvi.cwl](#) script, there is an additional complexity to ensure intermediate data is of the correct format for either process. We require openEO to generate outputs as STAC Catalog items for further EOEPKA processing. As we have had issues generating such data, any outputs are unable to be staged-in to subsequent EOEPKA processes. This therefore has been noted as a future piece of work for this project. It has been noted that the API

provided by the openEO Platform specification includes a [results](#) endpoint which may enable us to generate STAC items for results in future.

While openEO can import STAC items as datacubes, it is more complex to output STAC items, as openEO processing relies mostly on datacubes. Also note the `load_stac` function, available on openEO Platform, is currently experimental and, from some initial experimentation, does not provide the necessary output for continued processing.

Some consideration also needs to be given to the usability of such functionality, as it is probably more likely that the CWL scripts currently executed via EOEPKA will be more useful as part of an openEO process, rather than the other way round, as openEO is a newer specification. However, it is important to consider all methods of potential integration here, as openEO is likely to undergo significant development in the future and if we can harness any improvement within EOEPKA quickly, it is likely any development can be shared.

### **3.4.2 EOEPKA within openEO**

openEO process graphs are built up of one or more processes, with each taking some input, performing a calculation, and generating an output. To integrate openEO processes with EOEPKA processing, we will attempt to provide the ability to execute OGC application packages (CWL scripts) within an openEO implementation. This highlights a key functionality gap between the two specifications; that openEO processes can only be defined directly within a provided client or via a UDF, written in Python. EOEPKA's containerisation and CWL functionality means any processes are language independent, provided they can be containerised and run from a CWL script. The key limitation therefore being that OGC application packages are not currently supported within the openEO specification.

We will look at two methods to achieve this functionality, the first being to simply run CWL scripts locally using a CWL runner within a process and the second being to execute the CWL scripts using EOEPKA via the ADES. Using the ADES may be more suitable for a user when they need more extensive processing power or wish to run a process remotely and then feed the output back into openEO once complete.

This section of the document discusses method 1 from the above introduction (see section 3.4).

As openEO itself is just the specification for an application package, the outcome for this section of the project will be to suggest an extension to this specification to include additional processing that allows CWL scripts to be executed on given inputs. Therefore, in an attempt to provide this specification update we are suggesting the upcoming definitions are added to the openEO spec. Note, the method of execution for these CWL files will form part of the implementation, rather than part of the specification. As with EOEPKA, the specification here should conform to the OGC best practice standards, including inputs and outputs.

#### **3.4.2.1 Using UDFs and UDPs**

The least complex way to introduce new processes within openEO is through use of User-defined Functions and Processes, as discussed above. This means we can conform to the openEO specification while potentially providing completely new functionality within our process graphs. As a quick proof of concept here, we produced a user-defined function that takes input data as a data-cube and saves it to an external data storage service, here GitHub, and then does the reverse of this process to import the data back into openEO. This step will be vital in allowing data to pass between openEO and EOEPKA instances, as EOEPKA relies on cloud-hosted data access.

I have written a UDP (available here: [https://github.com/tjellicoe-tpzuk/openeo-processes-cwl/blob/main/examples/data/save\\_to\\_git.json](https://github.com/tjellicoe-tpzuk/openeo-processes-cwl/blob/main/examples/data/save_to_git.json)) that makes use of these UDFs to demonstrate this functionality, however we are restricted by what we are able to run on openEO platform (via the Vito back-end), due to the available Python runtimes, *Python* and *Python Jep*, having limited package



imports. So, for the minute this serves as a proof of concept only and cannot be executed within openEO Platform.

This also helps to highlight a similar issue when trying to execute CWL scripts within a Python UDF as the packages required to achieve this, whether done locally via a CWL Runner package or externally via http calls, using the *requests* package, will fail as these packages are not installed in either runtime.

The UDP first introduces a UDF that saves a datacube into a git repository as a NETCDF file and returns the file location for global access. Another UDF then reads the file located at a given URL (assumed to be NETCDF) and loads it back into the openEO environment as a datacube. The suggestion is that between these two UDFs another process can be defined which deploys a process via the ADES to further process the data, and returns the location of the output, currently a Min.io bucket, ready to be read back into openEO.

The main outstanding question here is whether we can define new runtimes, with different installed packages on which a UDF can be run. This does look to be an area of development for openEO (see [here](#)), but it is currently unclear how to make use of such functionality. To avoid this blocker entirely we can instead seek to define our own processes at the openEO specification level, rather than the implementation, as discussed below.

### 3.4.2.2 openEO Specification

openEO provides around 240 processes (at [version 2.0.0-rc.1](#)), which can then be implemented within a given back-end by the individual development team. We call the current set of processes the 'built-in' processes. The current openEO specification defines each of these 'built-in' processes as a JSON file which sits alongside the code implementation itself. The GitHub repository at <https://github.com/eodcgmbh/openeo-processes/tree/master> contains some example JSON definitions for the native openEO processes and openEO's website then presents these processes in an [easy to understand UI](#). Our aim here is to add new processes to the 'built-in' provision that allow CWL scripts to be executed as part of an openEO process graph.

Before we can update the openEO specification, we need to experiment with the best way to incorporate this functionality into openEO process graph execution and determine the optimal method for handling OGC application packages.

Note that all code here is inspired by the work of the [Dask openEO Implementation](#) which implements a number of openEO processes in Python.

#### 3.4.2.2.1 Running CWL Scripts Locally

The aim here is to allow OGC application packages, CWL scripts, to be executed on a given input as a step within an openEO process graph. The first, and perhaps most logical, method of development here is therefore simply to execute CWL scripts locally within a Python environment. The *cwltool* Python package provides a CWL Runner that can execute CWL directly within a Python script, something that EOEPCA uses the ADES to achieve.

To demonstrate this functionality, we have implemented a process called [run cwl local](#), see section 3.4.2.5 below, that makes use of the *cwltool* package in the Python implementation. This process saves the input data to a local file, executes the CWL script on this data and then returns the output from this execution in the most suitable format. Note, CWL scripts often refer to container images in which any processing should be run, in our examples we are working exclusively with Docker containers and execution using Docker but the *cwltool* package also supports other methods such as *Singularity* and *Apptainer*, as defined in the tool [reference documentation](#).

This process has no reliance on a running EOEPCA instance and allows all processing to be done within the openEO process graph parser, without any calls to external processing. It should be noted



that in future implementations the actual method of execution for the provided CWL script will be left up to the developer and could of course instead rely on an EOEPCA instance or make use of other packages. But the output data must always be imported back into the local environment if further processing is required.

The simplicity and efficiency of this process looks to offer the most functional improvement to the openEO specification, as it enables us to implement new processes that can run CWL natively without having to make external calls to other services. Also note that just as with EOEPCA, the CWL script can point to any containerised process and is therefore language-agnostic. This allows any programming language to be used when defining steps within a process graph.

#### 3.4.2.2.2 Getting data from openEO into EOEPCA

For a more complex solution that makes direct use of EOEPCA processing we can instead directly integrate EOEPCA processing within openEO process graphs by sending http requests via the ADES for CWL execution. This removes any reliance on a CWL Runner within the openEO implementation and allows processing to be done externally via a running EOEPCA instance.

To demonstrate that we can indeed make use of EOEPCA here, the next step is to demonstrate interaction between openEO and EOEPCA via the ADES. This is an extension to the above work, as we can simply update our implementation for the `run_cwl_local` process to make calls to the ADES api via http requests. The final code is available in the [run\\_cwl\\_ades](#) process, see section 3.4.2.5 below. So rather than running the CWL script locally using a CWL Runner, we can pass data to the ADES for execution and then load the result back into local memory. However, an additional step will also be required to ensure the data is in a location suitable for cloud execution prior to engaging with the ADES.

As an EOEPCA instance is always run in the cloud, this means we need to get any processed data out of openEO local storage into a cloud-accessible environment where the data can be exposed to the ADES for processing via EOEPCA. Note that EOEPCA interfaces with STAC items (as defined by the OGC) and therefore we need to ensure that any data coming out of openEO is of the correct format.

To ensure the data is indeed in the correct format, the process implementation needs to include steps to take the current datacube and produce a STAC catalogue item that can be read by EOEPCA. This data can then be saved in a cloud location, in our prototype we use GitHub just as with our UDF experimentation, that is then accessible by the ADES. Once this has been done, the processing can be executed by EOEPCA with a URI pointing to this newly saved data.

The use of GitHub here is suboptimal, and it would be more efficient to use some shared network storage, such as S3. We can avoid this step altogether by allowing CWL execution natively within the openEO spec, as discussed in section 3.4.2.3, rather than relying on an external ADES instance. However, as this project only serves as a proof of concept, we will continue with the use of GitHub but note that future implementations should follow alternative methods.

Once EOEPCA processing is complete, the service needs to import the data back into an openEO environment ready for further processing. This can be done via the `boto3` Python package which allows direct interaction with S3 storage, including downloading of files into local environments. Our implementation identifies the NETCDF file within the STAC item produced by EOEPCA and loads this back in as a `DataArray`.

#### 3.4.2.2.3 openEO Specification Updates

Now that we have functioning implementations for new openEO processes that work independently as Python scripts, the next step is to integrate these into an openEO implementation. To introduce new processes to openEO, we first need to create a JSON definition for these processes and then link these to our implemented functions.

Note, the below definition is based on the openEO process spec for *run\_udf* (see the openEO Processes website, [openEO processes \(2.0.0-rc.1\)](https://openeo.org/en/latest/processes/), for full definitions), as this also allows the execution of a text file, although *run\_udf* is currently limited to Python executables.

As a prototype, we have generated definitions for three new processes in JSON format, just as with the built-in openEO processes. These definitions can be found in the repository at <https://github.com/tjellicoe-tpzuk/openeo-processes-cwl/tree/main>, as additions to the previous process definition directory. We have provided new definitions for three processes found below. The formatting here is designed to mimic the definitions found on the openEO processes website, linked above.

### [cwl\\_preparation.json](#)

Takes an input and generates a file that can be passed to a CWL script as a local-path or URL (user-specified by an input parameter). The path to this file is returned as a string. In future, this will be combined with the *run\_cwl\_ades* process below. This process is only required to precede *run\_cwl\_ades*.

Below definition is extracted from the JSON definition.

- **Summary:** Convert input data to a file location that can be used as an input to a *run\_cwl\_ades* process
- **Description:** This process takes input data, often a raster data cube but any data type can be supported, saves it to a remote server location as a file and returns the location of this file for future recall.
- **Parameters:**
  - **data:** The data to be passed to the preparation process.
  - **context:** A dictionary of additional inputs required within this process. In our implementation this should include:
    - **save\_location:** Parameter to define whether the prepared file should be saved locally on the server or uploaded to a remote server location.
    - **git\_token:** Git-generated token to be used with Git API to gain access to specified repository and allow file upload.
    - **git\_repository:** Location of Git repository to upload data to in the format '<user-name>/<repo-name>'. A new directory will be created in this repository called 'data'.
- **Returns:** The location of the generated file, either as a url or local file path.

### [run\\_cwl\\_ades.json](#)

Runs a CWL script via the ADES on the given input file along with any additional inputs. A running EOEPCA instance is required before running this process and the domain of such an instance is required as an input. The input data must be in the form of URL path to the data that is to be processed e.g. a STAC catalog definition or NETCDF file. This data will be passed as the *input\_data* parameter in the CWL script. Once processed the output will be returned in the most suitable format, either URL or loaded data itself. The output is likely to be in the form of a STAC catalogue and if so this process will need to be followed by a *load\_stac* process. Note, the [load\\_stac process](#) is experimental doesn't currently support STAC catalog items.

Below definition is extracted from the JSON definition.

- **Summary:** Run a CWL on a file uploaded to the server.
- **Description:** Runs a given CWL script on a file that has been uploaded to the server, as well as allowing additional context data to be provided as required by the CWL definition.
- **Parameters:**
  - **cwl\_url:** Absolute URL to the CWL file

- **cwl\_inputs:** A dictionary of additional parameter inputs to be passed into the CWL file when run. Note, the inputs here must align with those names in the input CWL file.
- **data:** The data to input to the CWL script, provided as a URL. This data will be passed to the CWL as the *input\_data* input and so the file type must match that defined in the CWL script here.
- **context:** A dictionary of additional inputs required within this process. In our implementation this should include:
  - **Domain:** The domain address of a running EOEPCA instance to be used when deploying the application package via the ADES.
- **Returns:** The data processed by the CWL tool. The returned value can be of any data type and is exactly what the CWL tool returns.

### [run\\_cwl\\_local.json](#)

Runs a CWL script on the given input file along with any additional inputs. The input data is first saved as a local file, with type defined in the implementation, e.g. a STAC catalog definition or NETCDF file. The location of this file will then be passed as the *input\_data* parameter in the CWL script. Once processed the output will be returned in the most suitable format, either URL or loaded data itself.

Below definition is extracted from the JSON definition.

- **Summary:** Run a CWL script on the input data.
- **Description:** Runs a given CWL script on the input data, as well as allowing additional context data to be provided as required by the CWL definition. This process will run the #main workflow within the CWL tool. Desired output for next process must be called 'results'.
- **Parameters:**
  - **cwl\_location:** An absolute URL pointing to a CWL script to be applied to the input data. This CWL script needs to comply with the OGC Best Practice definition and needs to contain exactly one input with id 'input\_data'.
  - **cwl\_inputs:** A dictionary of additional parameter inputs to be passed into the CWL file when run. Note, the inputs here must align with those names in the input CWL file.
  - **data:** The data to be passed to the CWL script as 'input\_data' in the CWL file.
  - **context:** A dictionary of additional inputs required within this process. In our implementation this can be ignored.
- **Returns:** The data processed by the CWL tool. The returned value can be of any data type and is exactly what the CWL tool returns.

While these JSON definitions are sufficient to describe any new processes to be added to the openEO specification, this stops short of any implementation, as this should be kept independent from the specification. To demonstrate these new processes and allow for further experimentation we need to provide an implementation for each.

Therefore, alongside each of these JSON definitions we need to provide Python code to tell the openEO parser what to compute for each node. The mapping from each process\_id, as in the JSON definitions, to each Python callable is defined in a "process registry". This mapping is then used when executing a process graph to call the correct function at each step.

Note, openEO often passes around datacubes between process steps and so there will need to be a stage-in process added to convert this data into some format that a CWL file can read and process. In our examples this is handled within the process itself by first saving the data to a file and then passing this to the CWL runner as a path.

### 3.4.2.3 openEO Implementation

Now that we have provided JSON definitions and Python implementations for our new openEO processes this is sufficient to define an updated openEO implementation, building on the Dask

implementation. The [Dask implementation](#) is a Python implementation of several openEO processes which allows for local execution of process graphs without relying on any separate back-ends. In order to parse the process graphs correctly, this implementation also requires the [openeo-pg-parser-networkx](#) package. Both of these packages are developed by the EODC.

As discussed above, in addition to the previous set of processes, we have included three additional ones; the first two allow for ADES execution of CWL scripts while the third executes CWL scripts locally. Eventually, these functions can be combined into a single process that will allow OGC application packages to be executed within an openEO process graph. Future implementations can allow for multiple CWL execution methods in a single process, and the user can identify the optimal execution method on the fly, whether local or via the ADES.

All the implementation code discussed here is available in the Git repository at <https://github.com/tjellicoe-tpzuk/openeo-processes-cwl>. The best way to try out the new processes is by running the Jupyter Notebook, [minibackend\\_demo\\_cwl.ipynb](#), and experimenting with the different example JSON definitions provided, the two in bold are the **best** demonstrators for the new processes:

- `cwl-prepare-data.json` – a test process graph that runs the `cwl_preparation` process on the output and saves the created URL to a text file.
- **`cwl-example-ades.json`** – converts the input data to a file format and saves it to a public GitHub repo, available at <https://github.com/tjellicoe-tpzuk/openeo-read-write>. Then executes a provided CWL script on this uploaded data via the ADES and returns a datacube of the result. The example CWL script here applies a scale factor to each pixel in the dataset.
- **`cwl-example-local.json`** – executes a CWL script on the input data locally, using `cwltool`, and returns a datacube of the result. The example CWL script here applies a scale factor to each pixel in the dataset.
- `cwl-example-stac.json` – a test process graph to check the functionality of the `load_stac` process provided by the Dask implementation. This process graph loads a STAC item and saves it to a file as a NETCDF.
- `openeo-example-basic.json` – a test process graph that loads data into memory as a datacube and saves this datacube as a NETCDF.

### 3.4.2.4 Putting it all together

Now that these new functions are fully defined and implemented, we can build a process graph making use of each and execute it to see the results. The above Jupyter Notebook allows for further experimentation here using the provided JSON definitions for process graphs. However, we can also use these process graphs directly in a Python environment just as in previous examples using local processing. An example script is provided in the [openeo-cwl-local.py](#) script which creates two basic process graphs: one to load an example dataset, apply a CWL script on this data and save the output to a NETCDF; and another to execute the CWL script remotely via the ADES and again save the output locally. The CWL script used as an example here is based on the convert example discussed in section 3.2.2 and scales the input data pixels by a scale factor provided as input. The below code snippet demonstrates how the process graph is built up and executed.

```
input_cube = load_collection()

## Construct graph with local CWL execution

cwl_location = '../convert-netcdf-basic/convert-nc-app.cwl'
cwl_inputs = {
    "fn": "scale",
    "size": "0.1"
}

## Execute CWL script on given data
output_cube = run_cwl_local(data=input_cube, cwl_location=cwl_location, cwl_inputs=cwl_inputs)
save_result(data=output_cube, file_name="local_output.nc")
```

**Figure 4: Code snippet of the Python client being used to build a process graph using our new `run_cwl_local` process to execute a CWL script.**

Work has also been done to create another process implementation for `run_cwl_local`, to allow CWL execution within openEO just as if executing via a command line interface. In this new process, titled [run\\_cwl\\_local\\_generic](#), any inputs to the CWL script are allowed, just as if executed via the command line, and must be defined in the input dictionary variable, `cwl_inputs`. Whereas before the data input was handled separately from other inputs and needs to be converted to a CWL compliant format, e.g. local NETCDF file, first. This new process is currently experimental but serves as a proof of concept for generic CWL execution within openEO process graphs. This script can be tested using the [Jupyter Notebook](#), just as before, and is currently being tested running the EOEPKA [snuggs.cwl](#) example. Note that running this CWL script locally can take significant time and compute power.

### 3.5 Future Work

There are three key areas of further development that could be done next under this project:

- Improve our understanding around STAC objects and generating them from datacubes. This would allow for more seamless integration between EOEPKA and openEO, particularly when incorporating openEO processing into an OGC application package. If we can unify these data formats, it would enable us to create more complex workflows that make multiple calls to openEO processes as well as other EOEPKA processing. We have identified a STAC extension, available at <https://github.com/stac-extensions/datacube>, that enables datacubes to be integrated into STAC catalog items which offers a good start point for investigation into this area. STAC items are also vital to meet the OGC Best Practice guidelines. We can also consider methods to get STAC items out of openEO using the API endpoint discussed briefly in section 3.4.1.2.
- Finalise our openEO specification updates including combining the new openEO processes into a single `run_cwl` process that is fully generic and leaves any execution decisions up to the final implementation, i.e. whether to be done locally or using a hosted platform. The user can then combine in the steps contained in the `cwl_preparation` process if they wish to use an online execution environment, such as the ADES. We can also provide examples within the JSON definition and need to ensure thorough exception handling. Once finalised we can then raise a pull request for these updates.

- As discussed in the previous section, a new process, with an improved level of abstraction has been defined in the `run_cwl_local_generic` process, however this is currently only experimental. We would like to carry out additional experimentation and testing here, if it is decided that such a process is to be supported in the openEO specification. This process starts to combine the definitions of the `run_cwl_local` and `run_cwl_adcs` processes as it allows CWL execution just as if done via the command line interface, and is defined independently from the execution method, whether done locally or via the ADES, and expects input data in a CWL ready format, e.g. file path or URL.
- As any CWL script needs to be run on a saved file, rather than the in-memory binary data itself, we must first save the data to be parsed in a file format. There are many options available for EO data, particularly those types that integrate well with Xarray. For this project most of the work was done using NETCDF file formatting, as this type works well with openEO, being available with the built-in `save_result` process and being integrated within Xarray via the `to_netcdf()` function. However, other more advanced formats are available such as Zarr type files which allow multiple compression options and support reading and writing from different back-end datastores. Xarray also has built-in functions for converting data to Zarr format. Further work could be done to analyse the best file type to use here, to improve efficiency and compression, as each time a CWL script is executed we need some local file to pass as input.

## **4. PROJECT CONCLUSIONS**

This project has successfully discussed and demonstrated how EOEPCA and openEO can be integrated, in both directions, with positive results. We have provided several Git repositories, mentioned throughout this document, that enable the reader to carry out their own experimentation and see this work in action. We appreciate that this is only the first step in terms of developing fully integrated systems, however the ability to run CWL scripts within an openEO environment as part of a process graph is a very positive outcome and one that should be formalised to ensure it becomes part of future openEO implementations, should developers wish to support EOEPCA-like functionality through execution of CWL scripts. The ability to execute OGC application packages in openEO is likely to offer lots of new functionality for openEO users as they can start to consider the ways in which EOEPCA processing can be incorporated into their processing.

While openEO and EOEPCA both provide a specification for the access and exploitation of, and execution of processes on, Earth Observation data, they go about offering this functionality in rather different ways. One relying on CWL scripts to process this data through steps in a document and the other building up more visual, graph-type structures through coding directly in other programming languages. Having this ability to create complex process graphs in a language such as Python or JavaScript via the openEO clients might be more appealing to certain developers. However, this previously meant they were limited to using the processes that openEO define in their specification with all other processes relying on UDPs and UDFs. The ability to execute CWL scripts effectively allows user processes to be created in any programming language. It also opens the possibility for CWL scripts to be borrowed from other services, such as EOEPCA, removing some of the complexity of writing your own UDFs. Offering these users the ability to execute CWL scripts, means that they can avoid having to re-implement complex processes that have been employed elsewhere and can make use of prior implementation, provided there exists an associated OGC application package.

With both services helping many users to realise and achieve their Earth Observation objectives and by bringing data and processing to a wide variety of users it will be vital that any development in either service can be shared. Whether this be by executing openEO processes from an OGC application package executed via the ADES, running remote processes within EOEPCA directly from openEO or allowing complex OGC application packages to be combined into openEO process graphs. If we can encourage users to understand and appreciate the use cases of both services this can potentially lead to increased uptake and faster development in both.

## **5. OTHER INFORMATION**

### **5.1 Git Repositories**

Development under this project:

- [Python CWL Convert](#)
- [Python ADES Integration](#)
- [ADES Deployment of openEO Process Graphs](#)
- [openEO Process Update for CWL Execution](#)
- [openEO OGC Application Package](#)

External Git Repositories, not developed under this project:

- [EOEPCA Deployment Guide](#)
- [openEO Process Implementation](#)
- [openEO Processes Dask](#)



## **6. GLOSSARY**

The following acronyms and abbreviations are used in this report.

EOEPCA	Earth Observation Exploitation Platform Common Architecture
NoR	Network of Resources
openEO	Open Earth Observation
CWL	Common Workflow Language
TIF	Tagged Image File format
NETCDF	Network Common Data Form file format
UDP	User Defined Process (see section 3.3.4.1)
UDF	User Defined Function (see section 3.3.4.2)