OGC Earth Observations Applications Pilot

Terradue Engineering Report

Publication Date: 2020-10-22

Approval Date: 2020-09-23

Submission Date: 2020-09-03

Reference number of this document: OGC 20-042

Reference URL for this document: http://www.opengis.net/doc/PER/EOAppsPilot-Terradue

Category: OGC Public Engineering Report

Editor: Pedro Gonçalves

Title: OGC Earth Observations Applications Pilot: Terradue Engineering Report

OGC Public Engineering Report

Table of Contents

Chapter 1. Subject

This OGC Engineering Report (ER) documents the findings and experiences resulting from Terradue Activities on the OGC Earth Observation Applications Pilot. More specifically, this ER provides a way forward for the implementation of the "applications to the data" paradigm in the context of Earth Observation (EO) satellite data processing and Cloud-based platforms to facilitate and standardize the access to Earth observation data and information.

Chapter 2. Executive Summary

The availability of a growing volume of environmental data from space represents a unique opportunity for science, general R&D, and applications (apps), but it also poses a major challenge to achieve its full potential in terms of data exploitation. Firstly, because the emergence of large volumes of data (Petabytes era) raises new issues in terms of discovery, access, exploitation, and visualization of “Big Data”, with profound implications on how users do “data-intensive” Earth Science. Secondly, because the inherent growing diversity and complexity of data and users, whereby different communities – having different needs, skills, methods, languages and protocols – need to cooperate to make sense of a wealth of data of different nature (e.g. EO, in-situ, model), structure and format.

Responding to these technological and community challenges requires the development of new ways of working, capitalizing on Information and Communication Technology (ICT) developments to facilitate the exploitation, analysis, sharing, mining and visualization of massive EO data sets and high-level products within Europe and beyond. Evolution in information technology and the consequent shifts in user behavior and expectations provide new opportunities to provide more significant support to EO data exploitation.

Earth Observation Platforms provide customers with an environment allowing them to focus on their core business and outsource other aspects to a platform that supplies services to a large number of customers with similar needs. The success of platforms in the business world is based on their ability to minimize cost and time to market for their customers, thereby also reducing the risk of exploring unproven business cases. In recent years, Platforms for the Exploitation of Earth Observation data have been developed by public and private companies in order to foster the usage of EO data and expand the market of Earth Observation-derived information. The domain is composed of platform providers, service providers who use the platform to deliver a service to their users, and data providers. The availability of free and open data (e.g. Copernicus Sentinel), together with the availability of affordable computing resources, creates an opportunity for the wide adoption and use of Earth Observation data in a growing number of fields in our society.

OGC activities in Testbed-13, Testbed-14, and Testbed-15 initiated the development of an architecture to allow the ad-hoc deployment and execution of applications close to the physical location of the source data with the goal to minimize data transfer between data repositories and application processes.

The activity described in this Engineering Report responds to the invitation for Earth observation platform operators to implement the OGC Earth Observation Applications Pilot architecture as it has been defined in those previous OGC Innovation Program (IP) initiatives. The goal of the pilot is to evaluate the maturity of those specifications in a real-world environment with several Earth Observation applications brought by several application developers that work with Earth observation satellites. These developers brought different views and requirements in terms of data discovery, data loading, data processing, and result delivery to which the platform readiness is challenged and evolutions are proposed to the architecture.

This Engineering Report initiates by introducing the Earth Observation (EO) Platform architecture and documents the encoding and interfaces needed for defining, deployment and execution of EO Applications brought by the different application developers. The ER concludes with a summary of the main challenges found during the pilot activities and provides further recommendations to advance the architecture, integration and implementation strategies taking in consideration the viewpoints of both EO platforms and EO application developers.

2.1. Document contributor contact points

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

Contacts

Name	Organization	Role
Pedro Gonçalves	Terradue	Editor
Fabrice Brito	Terradue	Contributor
Emmanuel Mathot	Terradue	Contributor

2.2. 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
- OGC: OGC 14-065r2, OGC Web Processing Service 2.02
- OGC: OGC 13-026r8, OGC OpenSearch Extension for Earth Observation 1.0, 2016
- OGC: OGC 13-032r8, OGC OpenSearch Geo and Time Extensions 1.0.0, 2014
- OGC: OGC 12-084r2, OWS Context Atom Encoding Standard, 1.0.0, 2014
- OGC: OGC 14-055r2, OGC OWS Context GeoJSON Encoding Standard, 1.0., 2017
- OGC: OGC 17-069r3, OGC API - Features - Part 1: Core, 1.0, 2019

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.

- **Application**

  A self-contained set of operations to be performed, typically to achieve a desired data manipulation. The Application must be implemented (codified) for deployment and execution on the platform.

- **Application Deployment and Execution Service (ADES)**

  WPS-T (REST/JSON) service that incorporates the Docker execution engine, and is responsible for the execution of the processing service (as a WPS request) within the 'target' Exploitation Platform.

- **Application Descriptor**

  A file that provides the metadata part of the Application Package. Provides all the metadata required to accommodate the processor within the WPS service and make it available for execution.

- **Application Package**

  A platform independent and self-contained representation of a software item, providing executable, metadata and dependencies such that it can be deployed to and executed within an Exploitation Platform. Comprises the Application Descriptor and the Application Artefact.

- **Compute Platform**

  The Platform on which execution occurs (this may differ from the Host or Home platform where federated processing is happening).

- **Docker**

  Docker is a set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

- **Execution Management Service (EMS)**

  The EMS is responsible for the orchestration of workflows, including the possibility of steps running on other (remote) platforms, and the on-demand deployment of processors to local/remote ADES as required.

- **Exploitation Platform**

  An on-line collection of products, services and tools for exploitation of EO data.

- **Spatiotemporal Asset**

  Any file that represents information about the earth captured in a certain space and time.

- **GeoTIFF**

  A public domain metadata standard which allows georeferencing information to be embedded within a TIFF file. The potential additional information includes map projection, coordinate systems, ellipsoids, datums, and everything else necessary to establish the exact spatial reference for the file.

- **JPEG2000**

  An image compression standard and coding system

- **Processing**

  A set of predefined applications that interact to achieve a result. For the exploitation platform, comprises on-line processing to derive data products from input data, conducted by a hosted processing service execution.

- **Processing Result**

  The Products produced as output of a Processing Service execution.

- **Processing Service**

  A non-interactive data processing that has a well-defined set of input data types, input parameterization, producing Processing Results with a well-defined output data type.

- **Products**

  EO data (commercial and non-commercial) and Value-added products and made available through the EP. It is assumed that the Hosting Environment for the EP makes available an existing supply of EO Data.

- **Transactional Web Processing Service (WPS-T)**

  Transactional extension to WPS that allows ad hoc deployment / undeployment of user-provided application.

- **Web Processing Services (WPS)**

  OGC standard that defines how a client can request the execution of a process, and how the output from the process is handled

4.1. Abbreviated terms

- ADES Application Deployment and Execution Service
- API Application Programming Interface
- CWL Common Workflow Language
- DIAS Copernicus Data and Information Access Services
- DLR Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)
- EMS Execution Management Service
- EO Earth Observation
- EP Exploitation Platform

- ER Engineering Report
- ESA European Space Agency
- GDAL Geospatial Data Abstraction Library
- IW Interferometric Wide
- JSON JavaScript Object Notation
- OGC Open Geospatial Consortium
- OS Operating System
- OWS OGC Web Services
- REST Representational State Transfer
- SAFE Standard Archive Format for Europe
- SatCen European Union Satellite Centre
- SLC Single Look Complex
- SNAP Sentinel Application Platform toolbox
- STAC SpatioTemporal Asset Catalog
- TBD To Be Determined
- TIE Technology Integration/Interoperability Experiments
- TIFF Tagged Image File Format
- URL Uniform Resource Locator
- WPS Web Processing Service
- WPS-T Transactional Web Processing Service
- XML Extensible Markup Language
- YAML YAML Ain’t Markup Language

Chapter 5. Overview

Section 6 introduces the Earth Observation (EO) Platform architecture targeting the deployment and execution of EO Applications in distributed Cloud Platforms. The section provides an overview of EO applications design patterns, package and data interfaces. The section presents the approach followed by previous OGC Testbeds (13, 14 and 15) and the evolutions implemented to respond to specific challenges addressed during the pilot activities.

Section 7 presents the full cycle of definition, deployment and execution of Earth Observation Applications on an Exploitation Platform. This section presents the application packaging using the Common Workflow Language (CWL) [1] and data stage-in/out strategies for the deployment and publication of the application through a Web Service endpoint, OGC Web Processing Service (WPS).

Section 8 presents the EO Applications brought by the different application developers and documents all the steps for their deployment and execution.

Section 9 provides a summary of the main findings and provides further recommendations to advance the architecture, integration and implementation strategies taking in consideration the viewpoints of both EO platforms and EO application developers.

Please note that this ER refers to WPS and OGC API – Processes interchangeably. This is because the OGC API – Processes draft specification emerged from the draft specification of the REST binding of the WPS standard.

Chapter 6. Earth Observation Platform

This section explains the concepts from the Platform viewpoint.

6.1. Overview

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 easily exploitable software tools and libraries and access to distributed EO Data repositories to support the services adaptation and customization. The service provides developers with well-defined operational processes and procedures in order to allow for robust packaging of applications, test and validation activities, and application deployment in production. The adaptation and customization of the services is focused on supporting the developer in defining the parallelization strategy, the data requirements needs, tools and libraries necessary to execute the applications and the overall production strategy.

Ultimately, the User Algorithms are exposed through a Web Service endpoint, Web Processing Service (OGC WPS), that allows end-user portals and B2B client applications to pass processing parameters, trigger a data processing or systematic requests and establish the data pipeline to retrieve the information produced.

This section shows how Ellip was adapted and the evolutions implemented to respond to specific challenges addressed during the pilot activities.

6.2. Architecture

Previous OGC Testbeds (13, 14 and 15) initiated the design of an architecture to allow the packaging, deployment and execution of Earth Observation Applications in distributed Cloud Platforms.

These testbed activities built on OGC standards (e.g. WPS, OWS-Context) to describe data processing applications or workflows as Application Packages that can be deployed on and executed within diverse Cloud Platforms.

The WPS service allows end-user portals and B2B client applications to pass processing parameters, trigger on-demand or systematic data processing requests and establish the data pipeline to retrieve the information produced.

The Application Package includes information about the execution unit to be executed or specific workflow script that can be invoked on the processor directly together with the mappings for parameterization or tailoring.

The testbeds defined an architecture where the Application Package is used with an Execution Management Service (EMS) and Application Deployment and Execution Service (ADES) as seen in the figure below. The EMS provides the workflow orchestration service (WPS-T) to deploy and invoke processing services workflows on multiple deployment and execution services. The ADES provides the execution engine service of the application that was previously deployed as a WPS service by the EMS.

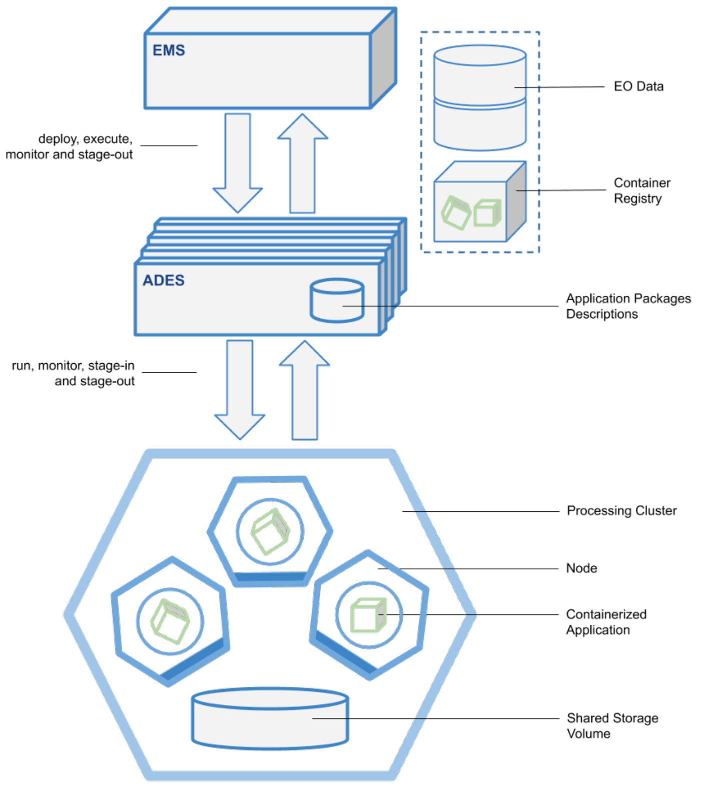

EMS
EO Data
deploy, execute, monitor and stage-out
Container Registry
ADES
Application Packages Descriptions
run, monitor, stage-in and stage-out
Processing Cluster
Node
Containerized Application
Shared Storage Volume

Figure 1. Platform High-level Architecture

The Execution Management Service (EMS) provides the orchestration service to deploy and invoke services of an application package on an ADES of the selected platform. It is responsible for managing the on-demand deployment and execution of the workflow building blocks on the “local” or “remote” platform.

The main responsibilities of the ADES are:

- Validate and accept an application deployment request from the EMS
- Validate and accept an execution request from the EMS
- Submit the process execution to the processing cluster
- Monitor the process execution
- Retrieve the processing results

In order to accomplish the execution and monitor steps above, the internal sub-steps also need to be responsible for the operations of:

- Data Stage-in for the process inputs
- Data Stage-out for the process outputs

6.3. Application Integration Scenarios

An Earth Observation Platform provides processing functions, tools and applications invoked individually or utilized in workflows. Users are able to integrate their own processing services into the platform and make them available for exploitation by other users also individually or in their own workflows. To support their integration activities, users are provided with an environment where they can integrate, build, test & debug and deploy.

The platform gives access to a dedicated application integration environment with exploitable software tools and libraries and access to distributed EO Data repositories to support the services adaptation and customization. It allows the design and integration of services as scalable data processing chains, leveraging selected data programming models.

The environment considers two scenarios of EO application integration:

- Importing: The application is directly packaged as a black-box. It relies upon the stage-in/out of data to the applications existing data access expectations by the Processing Framework.
- Adapting: The application is adapted to use the data access interfaces offered for data input and output.

The main objective of the current pilot project is to cover the importing scenario.

6.4. Application Design Pattern

6.4.1. Data-driven application with a fan-in application pattern

The data driven application fan-in patterns refers to the execution of a data processing function that aggregates several input products. The platform application accesses a list of input products, retrieves and proceeds with the stage-in of the products making them available to the application execution block.

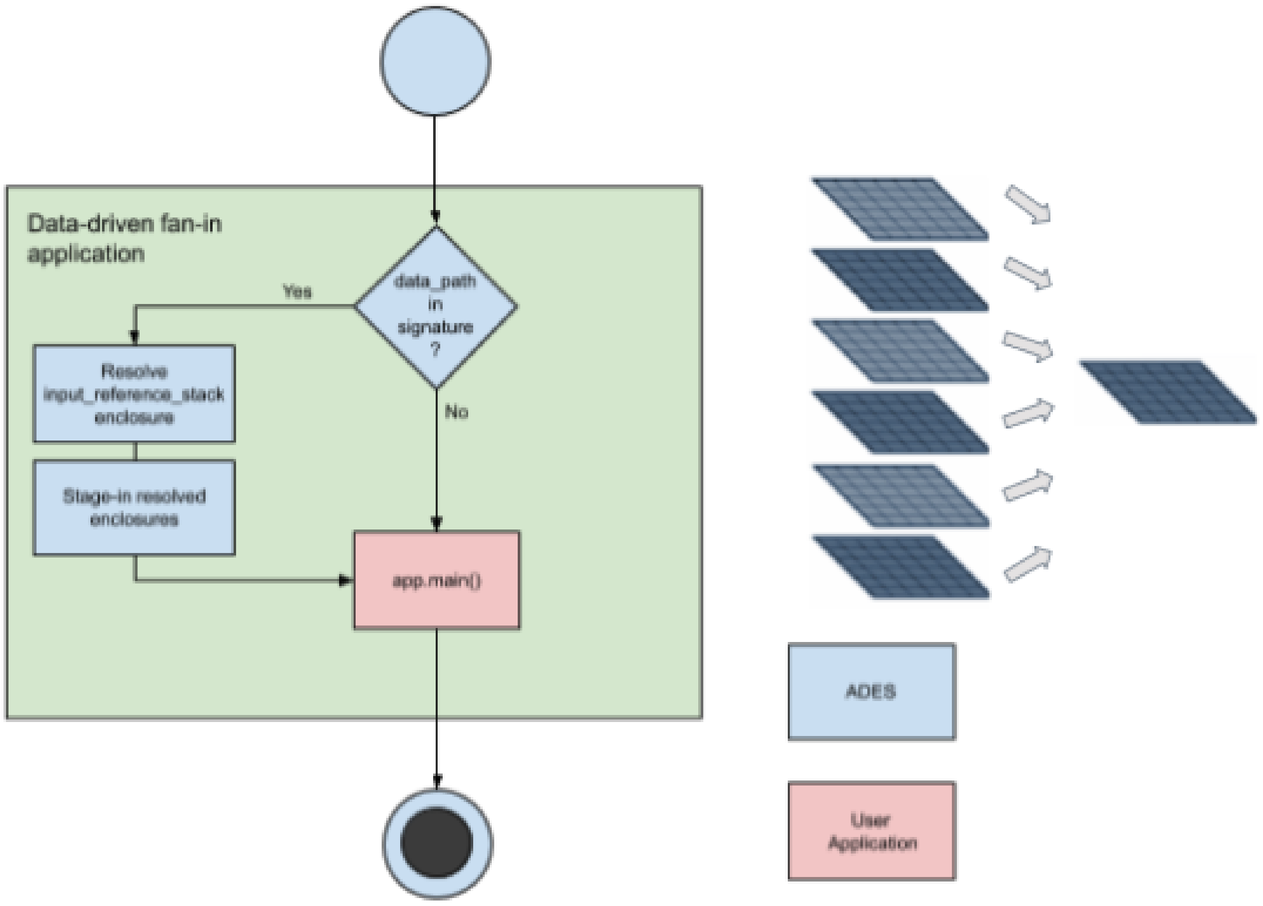

Data-driven fan-in application
data_path in signature ?
Yes
No
Resolve input_reference_stack enclosure
Stage-in resolved enclosures
app.main()
ADES
User Application

Figure 2. Data-driven application with fan-in input references where an application aggregates n-input EO products

6.4.2. Data-driven application with a fan-out application pattern

The data driven application fan-out patterns refers to the execution of a data processing function that processes concurrently several products generating independent output for each input. The platform application loops from a list of input products, retrieves and proceeds with the stage-in of the individual products making them available to the application execution block. The platform can apply different strategies to parallelize the execution of each individual product.

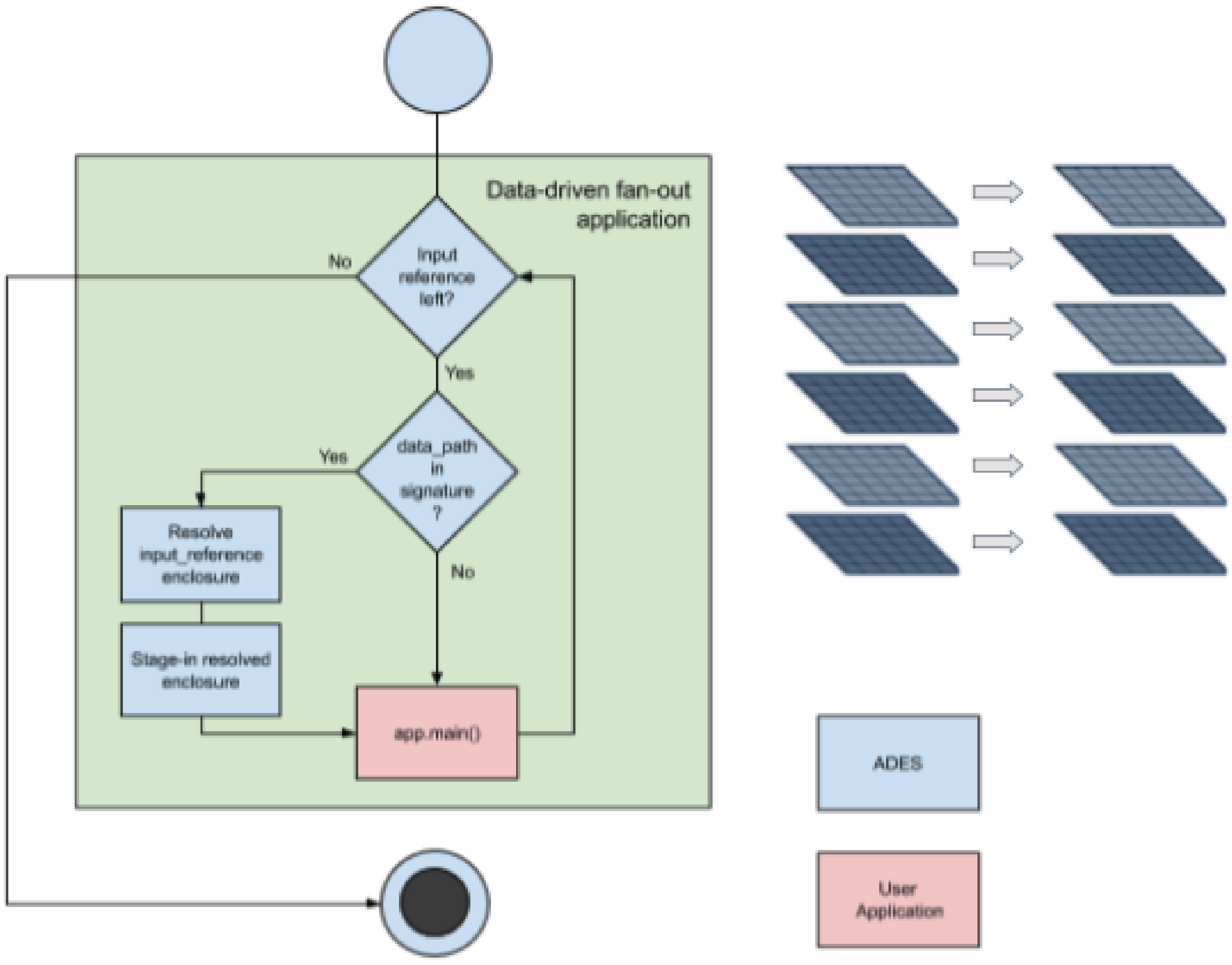

Data-driven fan-out application
Input reference left?
No
Yes
data_path in signature ?
Yes
No
Resolve input_reference enclosure
Stage-in resolved enclosure
app.main()
ADES
User Application

Figure 3. Data-driven application with fan-out input references where an application needs n-input EO products.

6.4.3. Parameter-driven application pattern

Parameter-driven data flows permit cyclic, systematic retrieval of selected groups of input products between selected the parameter intervals (e.g. start and end dates). In this scenario, the parameter interval acts as a step function, determining how the next batch of products is to be selected.

When considering temporal parameters, this pattern creates a "moving window" of processing mostly suitable for daily or weekly composites, or any task where input products must be grouped for processing by time interval.

The platform can apply different strategies to parallelize the execution of each individual product.

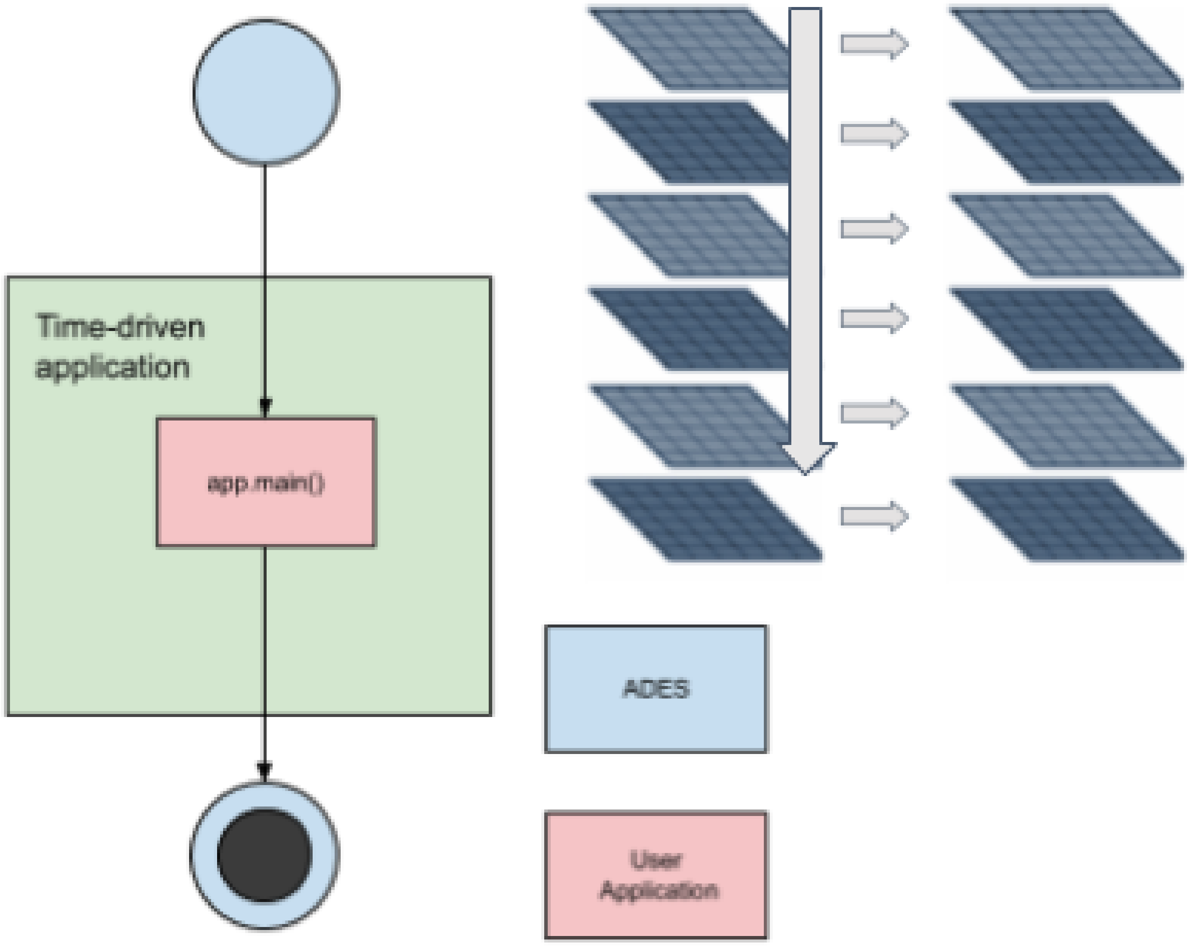


Figure 4. Parameter-driven application pattern where an application is systematically executed with the same parameters over a temporal range with different input EO products

6.5. Application Package

The Application Package targets algorithms (developed by a third-party developer) that are to be deployed in a Cloud Platform. The Application Package will contain all the information necessary to allow applications to be specified and deployed in federated resources providers. The Application Package takes into consideration past efforts to integrate OGC services with well-established formats and protocols that are natively ready for interoperability within Cloud solutions.

The main objective of the Application Package description file is to define a data processing application or workflow providing the information about the parameters, software item, executable, metadata and dependencies such that it can be deployed to and executed within an Exploitation Platform. This file must require a simple encoding so that application developers are able to create it, ensure that the application is fully portable among all supporting platforms and supports automatic deployment in a Machine-To-Machine (M2M) scenario. The Application Package information model must also allow the deployment of the application as a WPS service.

The execution block (i.e. Application Artefact) describes the ‘software’ component that represents the execution unit in a specific container (e.g. Docker) to be executed or specific workflow script that can be invoked on the processor directly. Based on the context information provided with the processor, the execution block maps how the container can be parameterized or tailored.

Docker is currently the selected container format and enables applications to be quickly assembled from components and eliminates the friction between development and production environments. As a result, applications can ship faster and run the same process into large multi-tenant data centers in the Cloud. Docker images work as an isolated process in the host operating system, which shares a kernel with other containers. Thereby, still having the benefits of virtualization, it is more portable and more effective by using less disk space. The application packaging allows the developers to remove their focus from the infrastructure details.

Previous work in OGC Testbed-13 (OGC 17-023) defined two alternative encodings for the Application Package. One as an OWS Context Document (OGC 12-084r2) with an XML encoding and an alternative where the information was included in the WPS Process Description in the `ows:metadata` element.

Subsequent OGC Testbed-14 (OGC 18-049r1) updated this model with an OWS Context JSON encoding (OGC 14-055r2) with the addition of a (embedded) Common Workflow Language (CWL) file (in JSON or YAML) that could be used in the simplest case to describe a single application and how to invoke the command-line within the container.

The CWL is a specification for describing analysis workflows and tools in a way that makes them portable and scalable across a variety of software and hardware environments, from workstations to cluster, cloud, and high-performance computing (HPC) environments. While CWL is not heavily used in the EO domain, it is designed to meet the needs of data-intensive science, such as Bioinformatics, Medical Imaging, Astronomy, Physics, and Chemistry.

The CWL contains two main specifications. The *Command Line Tool Description Specification* [https://www.commonwl.org/v1.1/CommandLineTool.html] that specifies the document schema and execution semantics for wrapping and executing command line tools and the *Workflow Description Specification* [https://www.commonwl.org/v1.1/Workflow.html] that specifies the document schema and execution semantics for composing workflows from components such as command line tools and other workflows.

To summarize, OGC Testbed-14 proposed the use of CWL initially for the command line invocation, the activities during this pilot showed a simple way of pointing to a Docker container from inside the CWL file included in the Application Packaging. We concluded that the CWL file has the potential to reference the application containers (e.g. Dockers) and also allows the definitions of the application parameters, input/output interface and the overall process offering parameters.

6.6. Data Flow Management

Previous OGC Testbeds (13, 14 and 15) followed an approach where the ADES is responsible for providing the data stage-in of product input references specified on the WPS request and stage-out the results of the application.

Data stage-in is the process to locally retrieve the inputs for the processing. Processing inputs are provided as EO Catalogue references and the ADES is responsible to translate those references into inputs available for the local processing.

Data stage-out is the process to upload the outputs of the processing onto external system(s), and make them available for later usage. ADES retrieves the processing outputs and automatically stores them onto an external persistent storage. Additionally, ADES should publish the metadata of the outputs onto a Catalogue and provide their references as an output.

In Testbed-13 the ADES makes the files available in a specific location defined by an environment variable pointing to the working directory. In Testbed-14, the input files location was provided directly in the CWL file, mentioned in the previous section, by defining one or more input files or folder directories.

However, EO product files come in different formats (e.g. GeoTIFF, HDF5, SAFE) and might have sub-items (e.g. metadata, bands, masks) that can be encoded in the same file or follow a given folder structure.

For example, SENTINEL-2 products are made available to users in SENTINEL-SAFE format, including image data in JPEG2000 format, quality indicators (e.g. defective pixels mask), auxiliary data and metadata. The SAFE format wraps a folder containing image data in a binary data format and product metadata in XML. A SENTINEL-2 product refers to a directory folder that contains a collection of information that can include several files like seen in Figure 5.

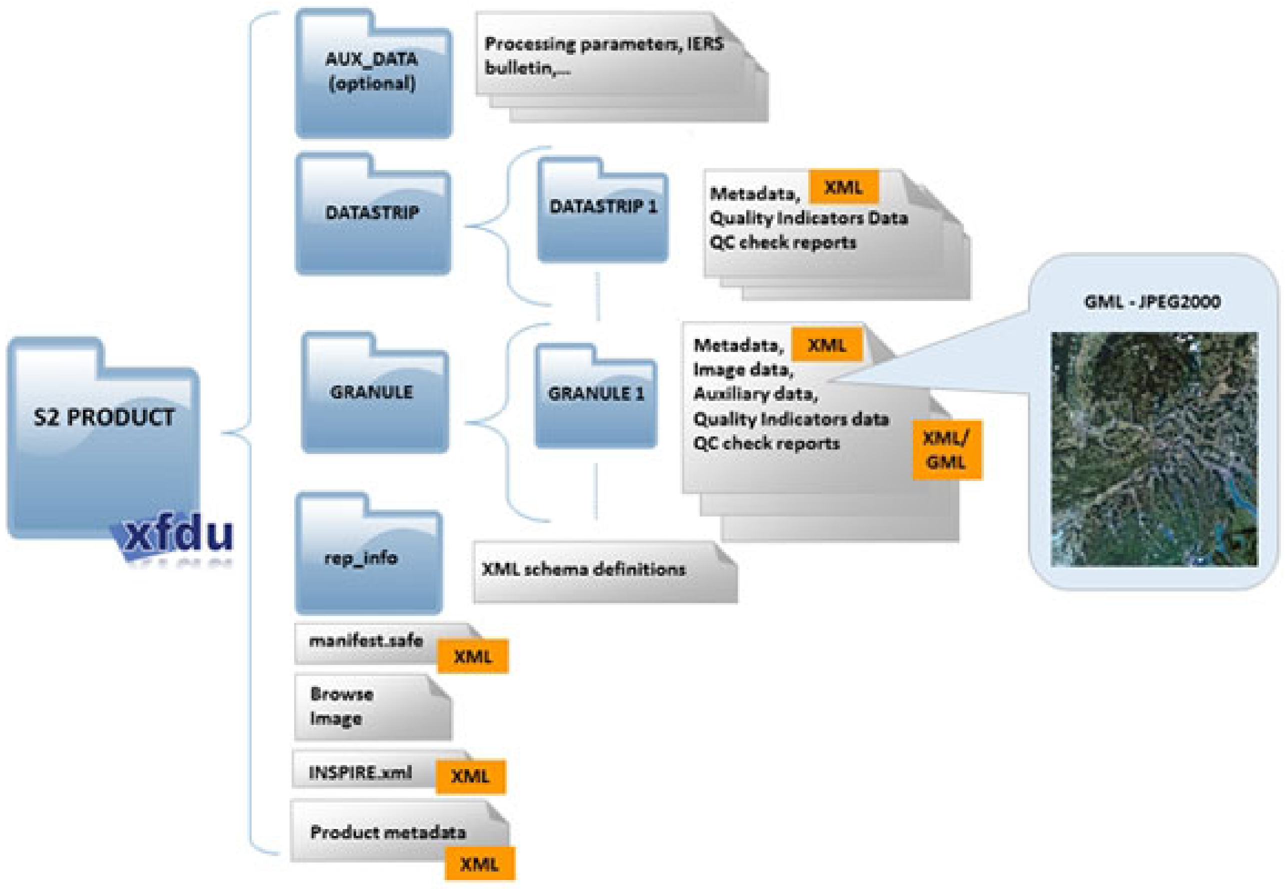


Figure 5. SENTINEL-2 product physical format

Different platforms will catalog and stage the products differently. Some will reproduce the exact folder structure and return the folder root or the main XML manifest file. Other platforms will compress the folder structure and return the compressed file. Other platforms will even convert the file and return a GeoTIFF aggregating all the information.

In general, the approach followed in the previous testbeds assumed that the applications were responsible for navigating the input folder directory and programmatically reacting to how the file was staged-in by the platform. This implies that the application developer needs to consider all possible cases when developing their read routines.

Conversely, the outputs of the application are fully managed by the developer that must place the resulting files in the output directory. The only information the ADES has about the output files is the file media type (formerly known as “MIME-type”).

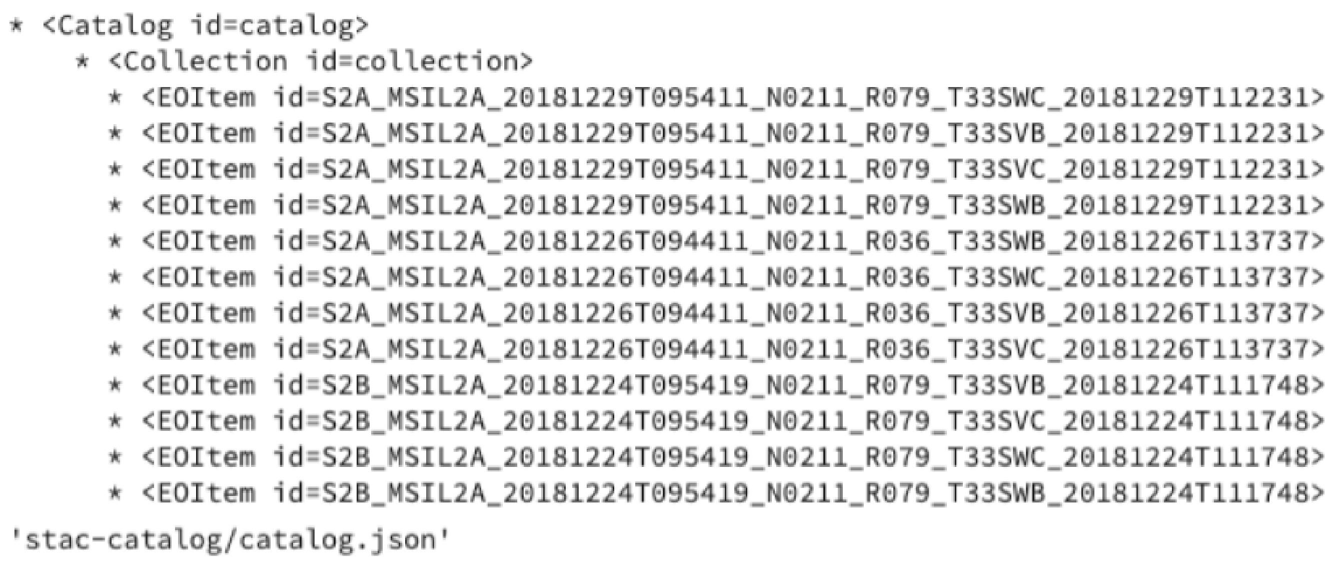
To summarize, in previous testbeds the ADES had limited knowledge about the input and output files respectively their data contents. Thus, ADES was often missing critical information like spatial footprint, sub-items (e.g. masks, bands) and additional metadata (e.g. ground sample distance, orbit direction). To respond to this issue, the activities during this Pilot activity showed the potential to follow an approach that uses a local catalogue encoded using the SpatioTemporal Asset Catalog (STAC) specification as a data manifest for application input and output [2].

The STAC specification standardizes the way geospatial assets are exposed online and queried. A 'spatiotemporal asset' is any file that represents information about the earth captured in a certain space and time (e.g. satellites, planes, drones, balloons). Most importantly the STAC specification can be implemented in a completely 'static' manner as flat files, enabling data publishers to expose their data by simply publishing one or several JSON files drilling down from collection (e.g. Sentinel-2), item (e.g. Sentinel-2 product) and assets (e.g. JPG2000 band file, auxiliary data, browse) with a relative path (something that was not possible using OpenSearch as defined by OGC 13-026r8, OGC 13-032r8).

The STAC specification defines several objects:

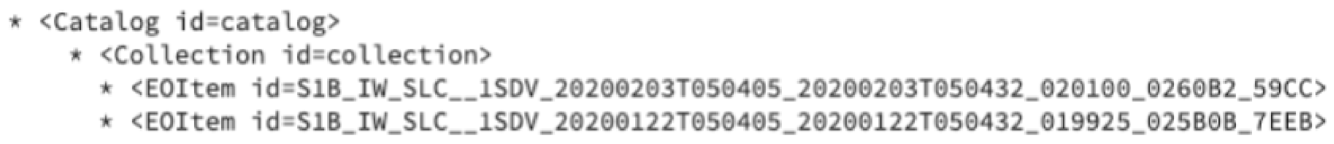
- STAC Catalog: STAC Catalog is a collection of STAC Items or other STAC Catalogs (sub-catalogs). The division of sub-catalogs is transparently managed by links to ease online browsing.
- STAC Collection: extends the STAC Catalog with additional fields to describe a whole set of STAC Items that share properties and metadata. STAC Collections are meant to be compatible with OGC API - Features Collections (OGC 17-069r3).
- STAC Item: a GeoJSON Feature with additional fields (e.g. time, geo), links to related entities and STAC Assets.
- STAC Asset: is an object that contains a link to data associated with the STAC Item that can be downloaded or streamed (e.g. data, metadata, thumbnails) and can contain additional metadata. Similar to *atom:link* it has properties like href, title, description, type and roles; but allows relative paths.

The STAC specification supports the ADES stage-in operation for all the identified application patterns using EO data catalog references as inputs. A local STAC catalog for the fan-in (Figure 6) and pairs (Figure 8) patterns or several local STAC catalogs with a single item for the fan-out pattern (Figure 7).



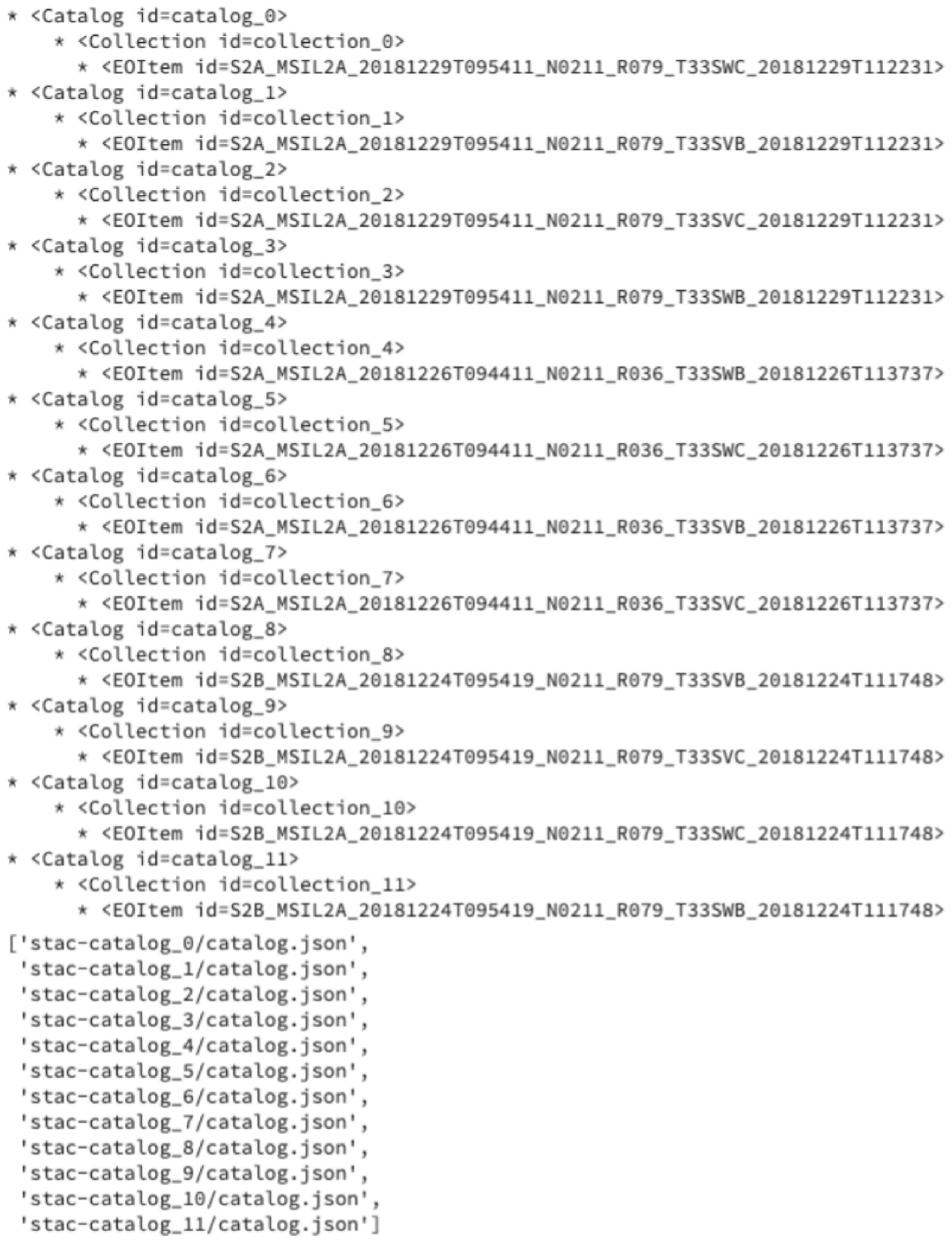
```
* <Catalog id=catalog>
    * <Collection id=collection>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SWC_20181229T112231>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SVB_20181229T112231>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SVC_20181229T112231>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SWB_20181229T112231>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SWB_20181226T113737>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SWC_20181226T113737>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SVB_20181226T113737>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SVC_20181226T113737>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SVB_20181224T111748>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SVC_20181224T111748>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SWC_20181224T111748>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SWB_20181224T111748>
'stac-catalog/catalog.json'
```

Figure 6. STAC catalog structure for the fan-in pattern where each item can have one or more file as assets



```
* <Catalog id=catalog>
    * <Collection id=collection>
      * <EOItem id=S1B_IW_SLC__1SDV_20200203T050405_20200203T050432_020100_0260B2_59CC>
      * <EOItem id=S1B_IW_SLC__1SDV_20200122T050405_20200122T050432_019925_025B0B_7EEB>
```

Figure 7. STAC catalog structure for the input pairs pattern where each item can have one or more file as assets



```
* <Catalog id=catalog_0>
    * <Collection id=collection_0>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SWC_20181229T112231>
* <Catalog id=catalog_1>
    * <Collection id=collection_1>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SVB_20181229T112231>
* <Catalog id=catalog_2>
    * <Collection id=collection_2>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SVC_20181229T112231>
* <Catalog id=catalog_3>
    * <Collection id=collection_3>
      * <EOItem id=S2A_MSIL2A_20181229T095411_N0211_R079_T33SWB_20181229T112231>
* <Catalog id=catalog_4>
    * <Collection id=collection_4>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SWB_20181226T113737>
* <Catalog id=catalog_5>
    * <Collection id=collection_5>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SWC_20181226T113737>
* <Catalog id=catalog_6>
    * <Collection id=collection_6>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SVB_20181226T113737>
* <Catalog id=catalog_7>
    * <Collection id=collection_7>
      * <EOItem id=S2A_MSIL2A_20181226T094411_N0211_R036_T33SVC_20181226T113737>
* <Catalog id=catalog_8>
    * <Collection id=collection_8>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SVB_20181224T111748>
* <Catalog id=catalog_9>
    * <Collection id=collection_9>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SVC_20181224T111748>
* <Catalog id=catalog_10>
    * <Collection id=collection_10>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SWC_20181224T111748>
* <Catalog id=catalog_11>
    * <Collection id=collection_11>
      * <EOItem id=S2B_MSIL2A_20181224T095419_N0211_R079_T33SWB_20181224T111748>
['stac-catalog_0/catalog.json',
 'stac-catalog_1/catalog.json',
 'stac-catalog_2/catalog.json',
 'stac-catalog_3/catalog.json',
 'stac-catalog_4/catalog.json',
 'stac-catalog_5/catalog.json',
 'stac-catalog_6/catalog.json',
 'stac-catalog_7/catalog.json',
 'stac-catalog_8/catalog.json',
 'stac-catalog_9/catalog.json',
 'stac-catalog_10/catalog.json',
 'stac-catalog_11/catalog.json']
```

Figure 8. STAC catalog structure for the fan-out pattern where each item can have one or more file as assets

In each item, there can be multiple assets expressed in relative paths enabling the application to navigate on the mounted volume. The use of STAC in the ADES data stage-in and stage-out follows the following steps:

- ADES stages the enclosures associated to the input catalog references
- ADES creates a local catalog using STAC up to the files (assets) in the staged-product(s) (items)
- Application uses the STAC catalog as an input parameter to access, as needed, to the different assets and their metadata
- Application creates a STAC catalog describing the application output products including assets (files) and metadata.

Chapter 7. Earth Observation Applications

This section explains the concepts from the developer viewpoint. It describes how the Application Package is specified.

7.1. Application Descriptor

The Application Package in the Exploitation Platforms context is required to:

- Reference the executable unit that implements the application functionality
- Describe its input/output interface
- List additional required or optional resources that impact the application execution, such as catalogue endpoints, base map layers, etc.

The Common Workflow Language (CWL) targets data-intensive processing scenarios and makes these portable and scalable across platforms capable of interpreting and execute the processes by describing:

- A runtime environment
- A Workflow (Directed Acyclic Graph (DAG))
- Command line tool(s)
- Parameter of the process
- Inputs/outputs

CWL is used as the application package descriptor. It covers the following elements necessary to describe the application:

- Workflow DAG orchestrating the steps in order to map workflow input/output with steps input/output
- Steps describing a command line with their input/output
- CWL specification extensions that may be used to provide the additional information elements

The application package is thus composed of a CWL file with the role of the application descriptor. The container reference is included in the CWL using the CWL `DockerRequirement`.

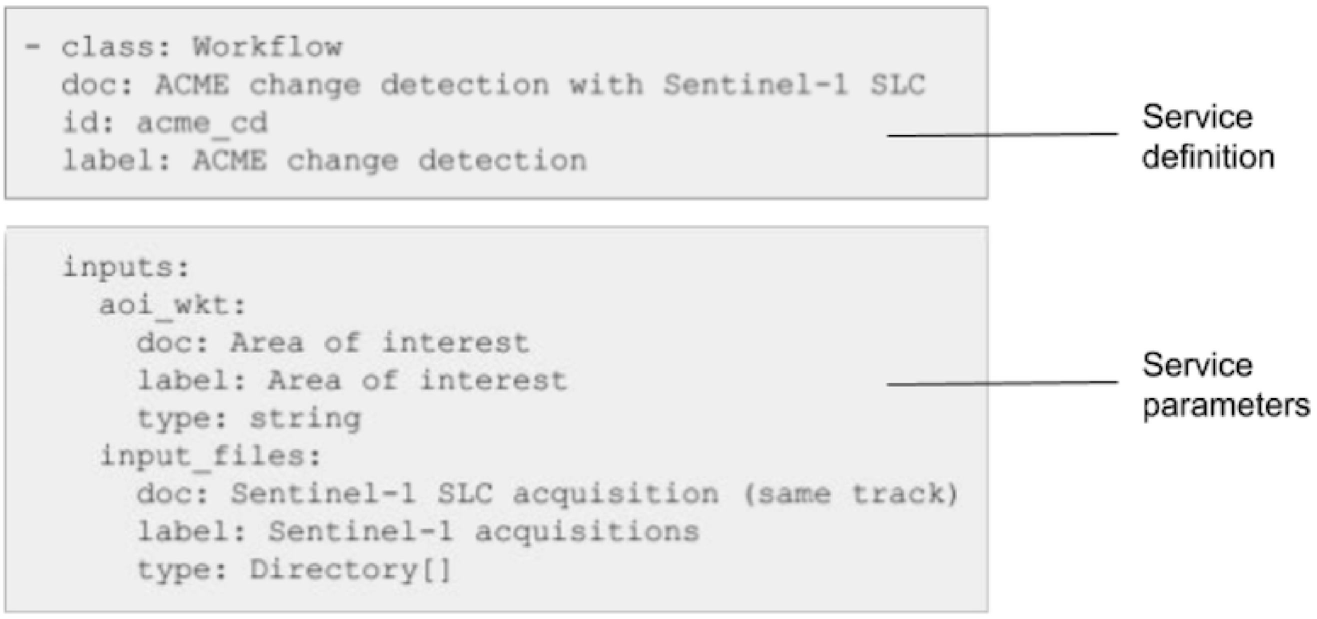
The CWL has two main classes: `Workflow` and `CommandLineTool`.

The `Workflow` class defines an analysis task represented by a directed graph describing a sequence of operations that transform an input data set to output.

The `CommandLineTool` class defines a non-interactive executable program that reads some input, performs a computation, and terminates after producing some output.

For the application package, the `Workflow` class level is the interface used to map the parameters of the exposed WPS service and respective mapping with the exposed service. The `CommandLineTool` class defines the actual interface of the application.

The CWL Workflow class defines the overall processing service. As seen in Figure 9 there are two main sections: one for Service definition and the other for Service parameters.



```
- class: Workflow
  doc: ACME change detection with Sentinel-1 SLC
  id: acme_cd
  label: ACME change detection
```

```
  inputs:
    aoi_wkt:
      doc: Area of interest
      label: Area of interest
      type: string
    input_files:
      doc: Sentinel-1 SLC acquisition (same track)
      label: Sentinel-1 acquisitions
      type: Directory[]
```

Figure 9. CWL Workflow class defining the processing service

The ADES offers an entry point to deploy the application package. Different protocols exist to perform the application deployment such as WPS-T that defines transactions.

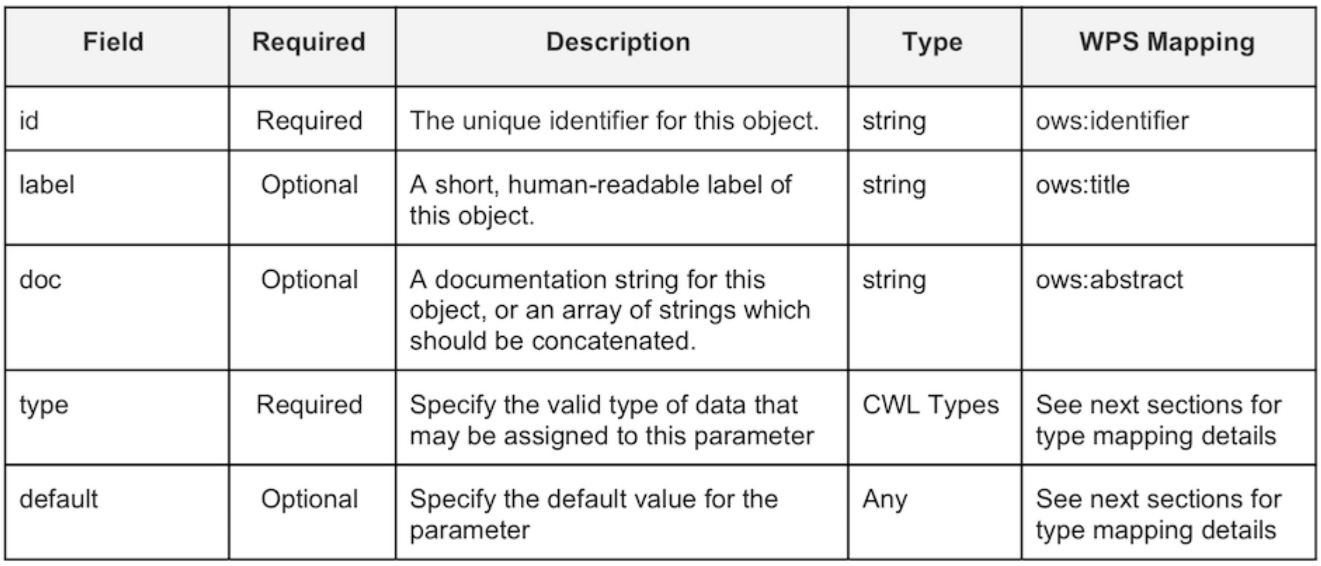
Regardless of the deployment mechanism, CWL is used to convey the information to describe the web service inputs/outputs using WPS concepts. The mapping between CWL and WPS follows the rules described in this section.

The Workflow level is the entry point to the overall CWL workflow and thus the interface used to map the inputs/outputs. The inputs/outputs description defined in the command line or workflow steps are not taken into account directly since they are linked between steps or bound at workflow level as foreseen by the CWL specification. Indeed, at the web processing service level, only the Workflow input and output parameters are exposed. This section describes the mapping rules to describe inputs and outputs in the WPS data model from the possible inputs/outputs definition from the CWL document.

7.1.1. Common rules for Parameters Mapping

The following table lists the CWL elements of an input/output definition and their corresponding elements in the WPS data model.

Table 1 - Mapping common CWL elements to the WPS data model

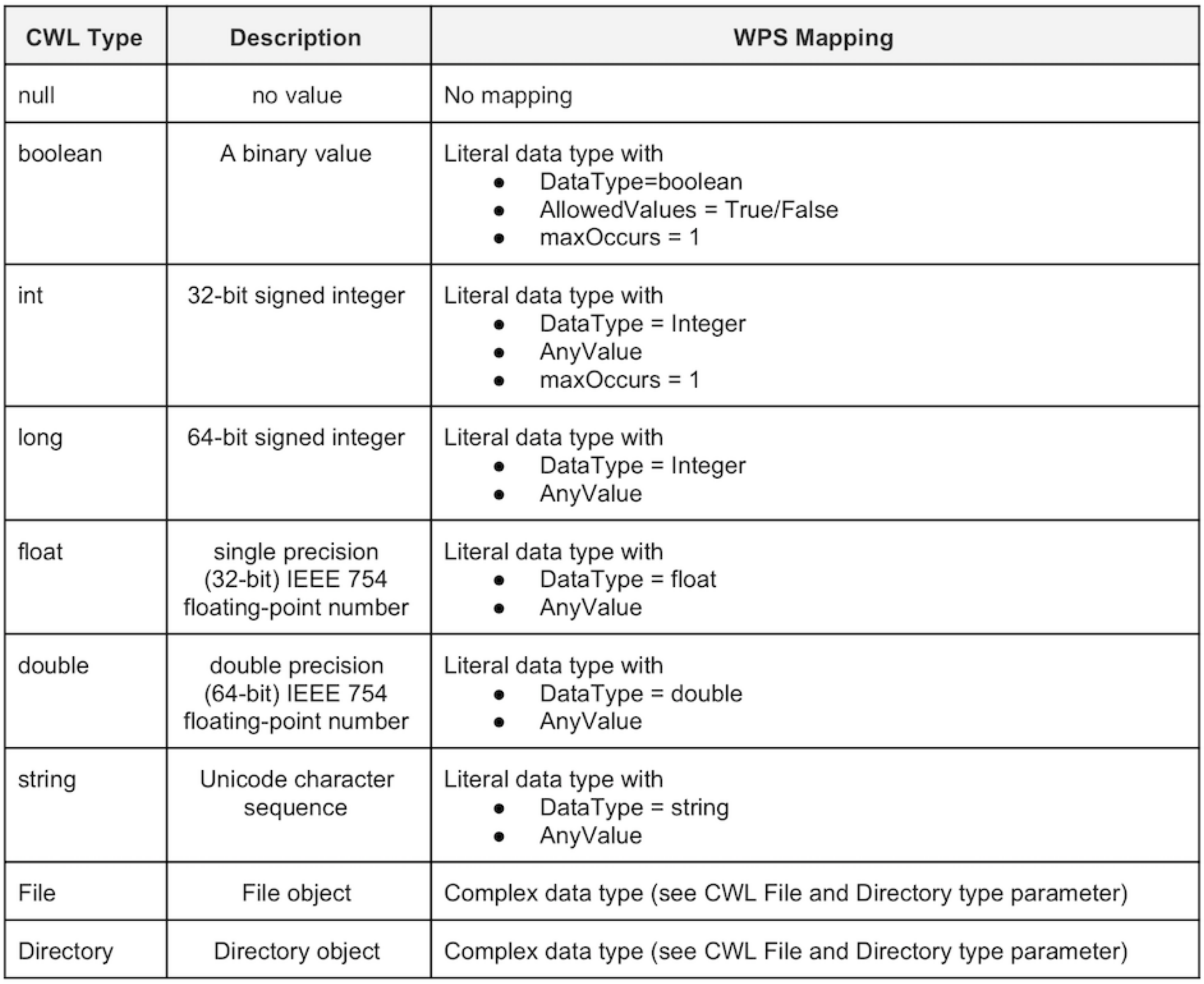


Field	Required	Description	Type	WPS Mapping
id	Required	The unique identifier for this object.	string	ows:identifier
label	Optional	A short, human-readable label of this object.	string	ows:title
doc	Optional	A documentation string for this object, or an array of strings which should be concatenated.	string	ows:abstract
type	Required	Specify the valid type of data that may be assigned to this parameter	CWL Types	See next sections for type mapping details
default	Optional	Specify the default value for the parameter	Any	See next sections for type mapping details

7.1.2. Single CWL type Parameter

The CWL types are mapped to a `LiteralData` Type in WPS with the specific rules for each type defined in the following table.

Table 2 - Mapping CWL types to WPS elements

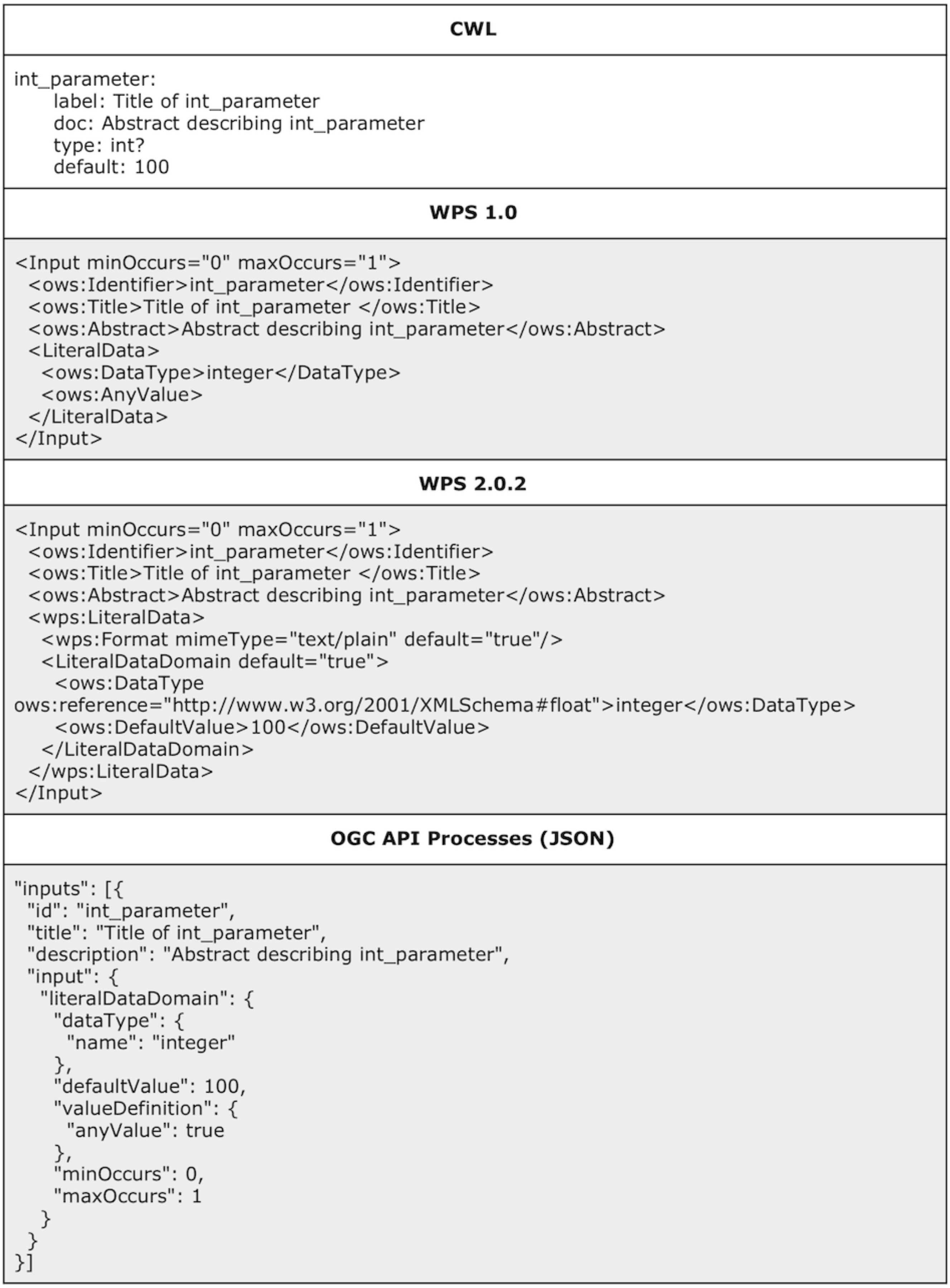


CWL Type	Description	WPS Mapping
null	no value	No mapping
boolean	A binary value	Literal data type with ● DataType=boolean ● AllowedValues = True/False ● maxOccurs = 1
int	32-bit signed integer	Literal data type with ● DataType = Integer ● AnyValue ● maxOccurs = 1
long	64-bit signed integer	Literal data type with ● DataType = Integer ● AnyValue
float	single precision (32-bit) IEEE 754 floating-point number	Literal data type with ● DataType = float ● AnyValue
double	double precision (64-bit) IEEE 754 floating-point number	Literal data type with ● DataType = double ● AnyValue
string	Unicode character sequence	Literal data type with ● DataType = string ● AnyValue
File	File object	Complex data type (see CWL File and Directory type parameter)
Directory	Directory object	Complex data type (see CWL File and Directory type parameter)

7.1.3. Optional and Mandatory Parameters

In CWL, each type can be suffixed with a '?' indicating that the parameter is optional. This parameter specification is mapped to the corresponding specification in WPS/OGC API Processes with `minOccurs="0"`.

Table 3 - Example of optional parameter in CWL, WPS 1.0/2.0.2, OGC API Processes



CWL

```
int_parameter:
      label: Title of int_parameter
      doc: Abstract describing int_parameter
      type: int?
      default: 100
```

WPS 1.0

```
<Input minOccurs="0" maxOccurs="1">
  <ows:Identifier>int_parameter</ows:Identifier>
  <ows:Title>Title of int_parameter </ows:Title>
  <ows:Abstract>Abstract describing int_parameter</ows:Abstract>
  <LiteralData>
    <ows:DataType>integer</DataType>
    <ows:AnyValue>
  </LiteralData>
</Input>
```

WPS 2.0.2

```
<Input minOccurs="0" maxOccurs="1">
  <ows:Identifier>int_parameter</ows:Identifier>
  <ows:Title>Title of int_parameter </ows:Title>
  <ows:Abstract>Abstract describing int_parameter</ows:Abstract>
  <wps:LiteralData>
    <wps:Format mimeType="text/plain" default="true"/>
    <LiteralDataDomain default="true">
      <ows:DataType
ows:reference="http://www.w3.org/2001/XMLSchema#float">integer</ows:DataType>
      <ows:DefaultValue>100</ows:DefaultValue>
    </LiteralDataDomain>
  </wps:LiteralData>
</Input>
```

OGC API Processes (JSON)

```
"inputs": [{
  "id": "int_parameter",
  "title": "Title of int_parameter",
  "description": "Abstract describing int_parameter",
  "input": {
    "literalDataDomain": {
      "dataType": {
        "name": "integer"
      },
      "defaultValue": 100,
      "valueDefinition": {
        "anyValue": true
      },
      "minOccurs": 0,
      "maxOccurs": 1
    }
  }
}]
```

In CWL, each type can be wrapped with brackets (‘[]’) indicating that the parameter is an input array. This parameter specification must be mapped to the corresponding specification in WPS/OGC

API Processes with `maxOccurs > 1`.

7.1.4. Enumeration Parameters

The CWL enumeration type specifies value definitions into a `LiteralData` Type in WPS.

Table 4 - Example of enumeration with options parameter in CWL, WPS 1.0/2.0.2, OGC API Processes

CWL

```
string_with_options_parameter:
  label: Title of string_with_options_parameter
  type:
    type: enum
    Symbols: [‘option1’, ‘options2’, ‘option3’]
```

WPS 1.0

```
<Input minOccurs="1" maxOccurs="1">
  <ows:Identifier>string_with_options_parameter</ows:Identifier>
  <ows:Title>Title of string_with_options_parameter </ows:Title>
  <LiteralData>
    <ows:DataType>string</DataType>
    <AllowedValues xmlns="http://www.opengis.net/ows/1.1">
      <Value>option1</Value>
      <Value>option2</Value>
      <Value>option3</Value>
     </AllowedValues>
  </LiteralData>
</Input>
```

WPS 2.0.2

```
<Input minOccurs="1" maxOccurs="1">
  <ows:Identifier>string_with_options_parameter</ows:Identifier>
  <ows:Title>Title of string_with_options_parameter </ows:Title>
  <wps:LiteralData>
    <wps:Format mimeType="text/plain" default="true"/>
    <LiteralDataDomain default="true">
      <ows:DataType
ows:reference="http://www.w3.org/2001/XMLSchema#float">string</ows:DataType>
      <ows:AllowedValues>
        <ows:Value>option1</ows:Value>
        <ows:Value>option2</ows:Value>
        <ows:Value>option3</ows:Value>
        </ows:AllowedValues>
    </LiteralDataDomain>
  </wps:LiteralData>
</Input>
```

OGC API Processes (JSON)

```
"inputs": [{
  "id": "string_with_options_parameter",
  "title": "Title of string_with_options_parameter",
  "input": {
    "literalDataDomain": {
      "dataType": {
        "name": "string"
      },
      "valueDefinition": {
        "allowedValues": [ "option1", "option2", "option3" ]
      },
      "minOccurs": 1,
      "maxOccurs": 1
  }
}
```

7.1.5. Array Parameters

When the type is an array of types, the WPS literal data can describe multiple domains (at least for WPS 2.0 and OGC API). If `null` type is in the array of type, the parameter is optional. This parameter specification must be mapped to the corresponding specification in WPS/OGC API Processes with `minOccurs="0"`.

Table 5 - Example of array of strings optional Parameter in CWL, WPS 1.0/2.0.2, OGC API Processes

CWL

Both cwl specifications below are valid

```
array_of_types_parameter:
  label: Title of array_of_types_parameter
  type:
    type: array
    items:
      - null
      - string
```

```
array_of_types_parameter:
  label: Title of array_of_types_parameter
  type: string[]?
```

WPS 1.0

```
<Input minOccurs="0" maxOccurs="99">
  <ows:Identifier>array_of_types_parameter</ows:Identifier>
  <ows:Title>Title of array_of_types_parameter </ows:Title>
  <LiteralData>
    <ows:DataType>string</DataType>
    <ows:AnyValue>
  </LiteralData>
</Input>
```

WPS 2.0.2

```
<Input minOccurs="0" maxOccurs="99">
  <ows:Identifier>array_of_types_parameter</ows:Identifier>
  <ows:Title>Title of array_of_types_parameter </ows:Title>
  <wps:LiteralData>
    <wps:Format mimeType="text/plain" default="true"/>
    <LiteralDataDomain default="true">
      <ows:DataType
ows:reference="http://www.w3.org/2001/XMLSchema#float">string</ows:DataType>
    </LiteralDataDomain>
  </wps:LiteralData>
</Input>
```

OGC API Processes (JSON)

```
"inputs": [{
  "id": "array_of_types_parameter",
"title": "Title of array_of_types_parameter",
"input": {
  "literalDataDomain": {
    "dataType": {
      "name": "string"
    },
    "valueDefinition": {
      "anyValue": true
    },
    "minOccurs": 0,
    "maxOccurs": 99
  }
}
```

7.1.6. File and Directory Parameters

In CWL, the `File` or `Directory` types correspond to files on a file system and thus require a specific mapping according to the ADES implementation exposing the WPS.

Indeed, the ADES manages physical files in the processing runtime environment. At job submission, the inputs may be passed as references (HTTP link, S3 link, OpenSearch URL, STAC catalog, etc.) to a resource manager and these are resolved and fetched by the ADES making them available for processing as specified in the CWL using the `File` or `Directory` type.

For Directory parameters, the interface between the ADES and the application process is a STAC local catalog (catalog.json and all its downstream items and assets including their associated staged files) referencing all the data staged-in corresponding collections as specified initially in the CWL.

The `File` or `Directory` types are thus mapped to these staged resources.

The example in Table 6, the ADES is able to handle the following resource description document or references:

- OpenSearch description document with possible search endpoints for input products
- Atom feed with entries containing the input files (enclosures)
- STAC catalogue with features describing assets as input files

Table 6 - Example of File Parameter in CWL, WPS 1.0/2.0.2, OGC API Processes

CWL

file_parameter:
    label: Title of file_parameter
    doc: Abstract describing file_parameter
    type: File

WPS 1.0
<Input minOccurs="0" maxOccurs="1"> <ows:Identifier>int_parameter</ows:Identifier> <ows:Title>Title of int_parameter </ows:Title> <ows:Abstract>Abstract describing int_parameter</ows:Abstract> <ComplexData>   <Supported>   <Format><MimeType>application/opensearchdescription+xml</MimeType></Format>   <Format><MimeType>application/atom+xml</MimeType></Format>   <Format><MimeType>application/geo+json; profile=stac</MimeType></Format>   </Supported> </ComplexData> </Input>
WPS 2.0.2
<Input minOccurs="0" maxOccurs="1"> <ows:Identifier>int_parameter</ows:Identifier> <ows:Title>Title of int_parameter </ows:Title> <ows:Abstract>Abstract describing int_parameter</ows:Abstract> <wps:ComplexData>   <wps:Format mimeType="application/opensearchdescription+xml" encoding="raw" />   <wps:Format mimeType="application/atom+xml" encoding="raw"/>   <wps:Format mimeType="application/geo+json; profile=stac-item" encoding="raw"/> </wps:ComplexData> </Input>
OGC API Processes (JSON)

"inputs": [{
 "id": "int_parameter",
 "title": "Title of int_parameter",
 "description": "Abstract describing int_parameter",
 "input": {
  "complexData": {
   "format": [{
    "name": "OpenSearch Description",
    “Type”: "application/opensearchdescription+xml"},
   {
    "name": "Atom feed",
    “Type”: "application/atom+xml"},
   {
    "name": "Stac Item",
    “Type”: "application/geo+json; profile=stac-item"}],
   "minOccurs": 0,
   "maxOccurs": 1
  }
 }
}]

7.1.7. Resources for the runtime environment

CWL provides a mechanism for expressing runtime environment resource requirements with the simple rule:

- `min` is the minimum amount of a resource that must be reserved to schedule a job. If "min" cannot be satisfied, the job should not be run.
- `max` is the maximum amount of a resource that the job shall be permitted to use. If a node has sufficient resources, multiple jobs may be scheduled on a single node provided each job's "max" resource requirements are met. If a job attempts to exceed its "max" resource allocation, an implementation may deny additional resources, which may result in job failure.

Hardware resources are expressed with the CWL `ResourceRequirement` allowing the definition of:

- `coresMin` for the minimum reserved number of CPU cores
- `coresMax` for the maximum reserved number of CPU cores
- `ramMin` for the minimum reserved RAM in mebibytes
- `ramMax` for the maximum reserved RAM in mebibytes

This definition covers most of the application resource requirements needs.

7.1.8. Application Design Patterns

The fan-in application design pattern is the simplest case realized with CWL as it is the default behavior of a CWL workflow execution step.

The fan-out application design pattern is expressed in the CWL using the `ScatterFeatureRequirement`. This feature tells the runner that the application will run multiple times over a list of inputs. The workflow then takes the input(s) as an array and will run the specified step(s) on each element of the array as if it were a single input.

The `ScatterFeatureRequirement` is defined at the CWL `Workflow` class with:

```
cwlVersion: v1.0
  class: Workflow
    requirements:
      ScatterFeatureRequirement: {}
```

The complete definition of the ScatterFeatureRequirement is provided below:

```
- class: ScatterFeatureRequirement
 steps:
   node_1:
     in:
       inp1: input_reference
       inp2: tiling_level
     out:
     - results
     run: '#clt'
     scatter: inp1
     scatterMethod: dotproduct
```

In the definition above the fan-out pattern is applied to the *input_reference* workflow parameter, which needs to be mapped to an input parameter at step level (in the definition above as *inp1*). The keyword `scatter` is used to define which step input parameter is scattered in the workflow step requirements and the fan-out method is defined with the `scatterMethod` keyword. Its value is one of: dotproduct, nested_crossproduct, or flat_crossproduct:

- `dotproduct` specifies that each of the input arrays are aligned and one element taken from each array to construct each job. It is an error if all input arrays are not the same length.
- `nested_crossproduct` specifies the Cartesian product of the inputs, producing a job for every combination of the scattered inputs. The output must be nested arrays for each level of scattering, in the order that the input arrays are listed in the scatter field.
- `flat_crossproduct` specifies the Cartesian product of the inputs, producing a job for every combination of the scattered inputs. The output arrays must be flattened to a single level, but otherwise listed in the order that the input arrays are listed in the scatter field.

A complete example is provided below.

```
- class: Workflow
  doc: This service takes as input a Sentinel-3 SLSTR Level 2 (SL_2_LST____) product
    on DESCENDING pass and does the reprojection and tiling
  id: slstr-tiling
  inputs:
    input_reference:
      doc: This service takes as input a Sentinel-3 SLSTR Level 2 (SL_2_LST____) product
        on DESCENDING pass
      label: Sentinel-3 SLSTR Level-2 (SL_2_LST____ descending pass)
      stac:collection: input_reference
      type: Directory[]
    tiling_level:
      doc: Tiling level
      label: Tiling level
      type: string
  label: Sentinel-3 SLSTR Level-2 reprojection and tiling
  outputs:
  - id: wf_outputs
    outputSource:
    - node_1/results
    type:
      items: Directory
      type: array
  requirements:
  - class: ScatterFeatureRequirement
  steps:
    node_1:
      in:
        inp1: input_reference
        inp2: tiling_level
      out:
      - results
      run: '#clt'
      scatter: inp1
      scatterMethod: dotproduct
$namespaces:
  stac: http://www.me.net/stac/cwl/extension
cwlVersion: v1.0
```

The parameter driven application design pattern may choose to scatter on an input parameter list of values and the method is exactly the same as the fan-out concept explained above.

7.2. Data Flow Management

The ADES is responsible for the data flow management by using a local catalogue encoded according to the Spatio Temporal Asset Catalog (STAC) specification as a data manifest for application inputs and outputs. The local catalogue provides knowledge about the input and output files data contents like spatial footprint, sub-items (e.g. masks, bands) and additional metadata.

This section describes the strategy to data stage-in, locally retrieving the inputs products for the processing, and data stage-out, making the outputs of the processing available for the subsequent steps (locally or on external systems).

7.2.1. Data Stage-In

The processing inputs are provided as EO Catalogue references and the ADES is responsible to translate those references into inputs available for local processing.

ADES resolves the resources for each File or Directory parameter, stages the data for processing and creates a STAC file to describe the available inputs. The full steps of the ADES for data stage-in are described in the Figure 10.

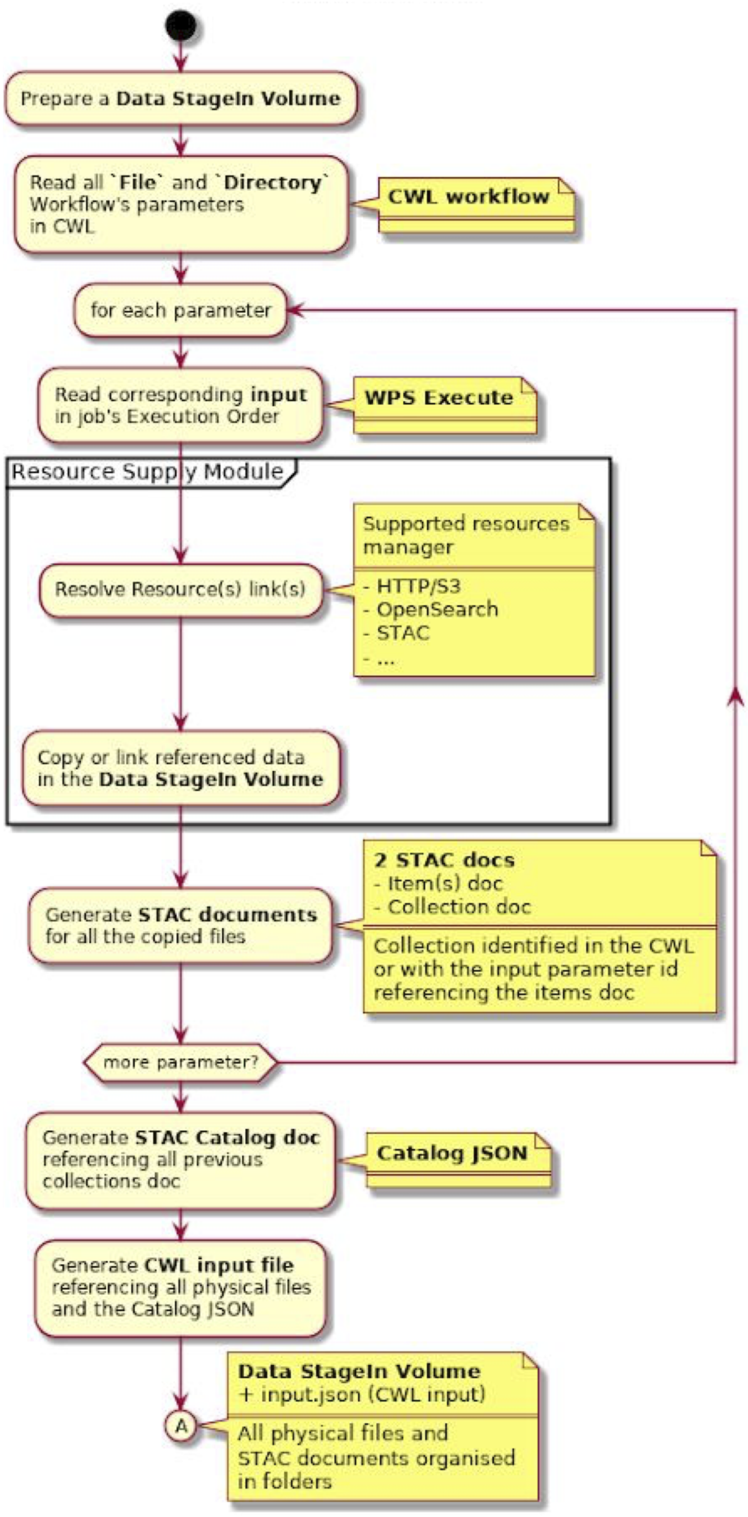
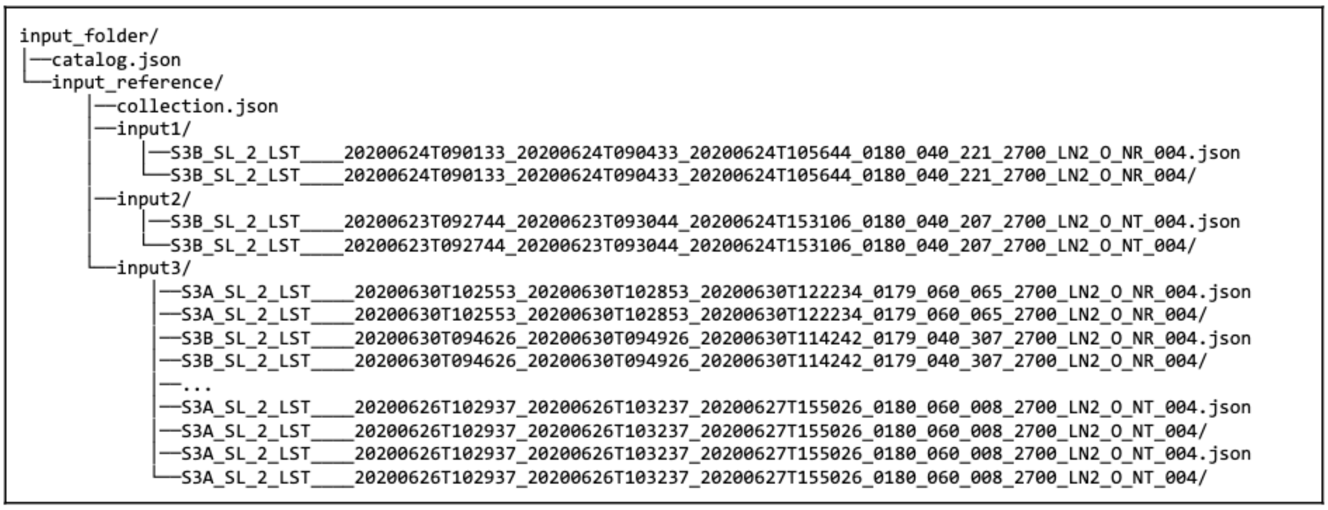

Prepare a Data StageIn Volume
Read all `File` and `Directory` Workflow's parameters in CWL
CWL workflow
for each parameter
Read corresponding input in job's Execution Order
WPS Execute
Resource Supply Module
Resolve Resource(s) link(s)
Supported resources manager
- HTTP/S3
- OpenSearch
- STAC
- ...
Copy or link referenced data in the Data StageIn Volume
Generate STAC documents for all the copied files
2 STAC docs
- Item(s) doc
- Collection doc
Collection identified in the CWL or with the input parameter id referencing the items doc
more parameter?
Generate STAC Catalog doc referencing all previous collections doc
Catalog JSON
Generate CWL input file referencing all physical files and the Catalog JSON
A
Data StageIn Volume
+ input.json (CWL input)
All physical files and STAC documents organised in folders

Figure 10. ADES workflow steps for data stage-in from the CWL service definition and WPS execution request

As an example, let’s consider an execution request with 3 URLs for a catalogue. Each input parameter is a link to a catalogue that will provide a feed containing one or more products. The ADES must be able to consume these URLs and handle the overall staging operation.

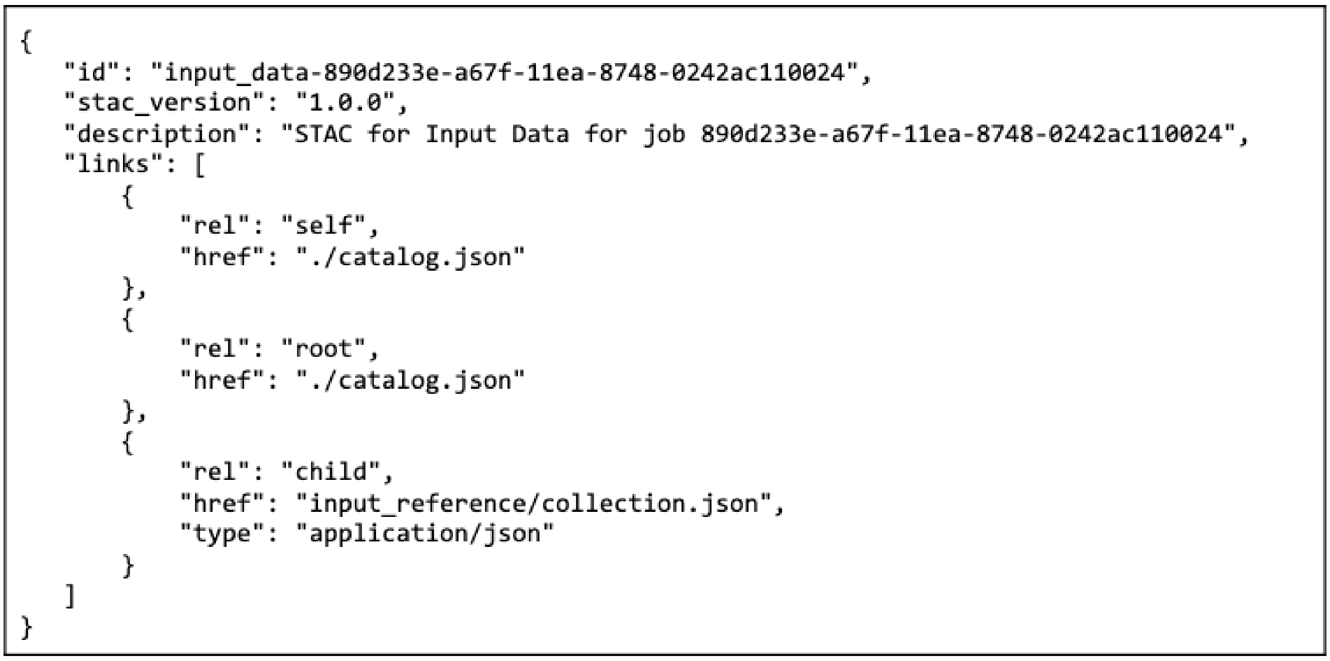
The ADES performs the data stage-in and creates a STAC catalog that describes for the application the contents of the input products and their metadata as shown in the figure below.



```
input_folder/
├─catalog.json
└─input_reference/
        ├─collection.json
        ├─input1/
        │       ├─S3B_SL_2_LST____20200624T090133_20200624T090433_20200624T105644_0180_040_221_2700_LN2_O_NR_004.json
        │       └─S3B_SL_2_LST____20200624T090133_20200624T090433_20200624T105644_0180_040_221_2700_LN2_O_NR_004/
        ├─input2/
        │       ├─S3B_SL_2_LST____20200623T092744_20200623T093044_20200624T153106_0180_040_207_2700_LN2_O_NT_004.json
        │       └─S3B_SL_2_LST____20200623T092744_20200623T093044_20200624T153106_0180_040_207_2700_LN2_O_NT_004/
        └─input3/
                ├─S3A_SL_2_LST____20200630T102553_20200630T102853_20200630T122234_0179_060_065_2700_LN2_O_NR_004.json
                ├─S3A_SL_2_LST____20200630T102553_20200630T102853_20200630T122234_0179_060_065_2700_LN2_O_NR_004/
                ├─S3B_SL_2_LST____20200630T094626_20200630T094926_20200630T114242_0179_040_307_2700_LN2_O_NR_004.json
                ├─S3B_SL_2_LST____20200630T094626_20200630T094926_20200630T114242_0179_040_307_2700_LN2_O_NR_004/
                ├─...
                ├─S3A_SL_2_LST____20200626T102937_20200626T103237_20200627T155026_0180_060_008_2700_LN2_O_NT_004.json
                ├─S3A_SL_2_LST____20200626T102937_20200626T103237_20200627T155026_0180_060_008_2700_LN2_O_NT_004/
                ├─S3A_SL_2_LST____20200626T102937_20200626T103237_20200627T155026_0180_060_008_2700_LN2_O_NT_004.json
                └─S3A_SL_2_LST____20200626T102937_20200626T103237_20200627T155026_0180_060_008_2700_LN2_O_NT_004/
```

Figure 11. STAC catalog hierarchy describing the input products

The input data folder is organized following common rules and conventions. A catalog file named `catalog.json` must be present in the root folder. This catalog is the starting point to route to all collections, items and assets. In our example, Figure 12 shows the `catalog.json` file.



```
{
   "id": "input_data-890d233e-a67f-11ea-8748-0242ac110024",
   "stac_version": "1.0.0",
   "description": "STAC for Input Data for job 890d233e-a67f-11ea-8748-0242ac110024",
   "links": [
       {
           "rel": "self",
           "href": "./catalog.json"
       },
       {
           "rel": "root",
           "href": "./catalog.json"
       },
       {
           "rel": "child",
           "href": "input_reference/collection.json",
           "type": "application/json"
       }
   ]
}
```

Figure 12. STAC catalog root folder to describe the input products

Besides the usual *self* and *root* link, the catalog file must reference a child for each input representing an input data. This link must point to a collection file.

For each input parameter, a collection file is referenced by the root catalog file. This file is a summary for the items found in the input parameter references. The `collection.json` file for the `input_reference` parameter of the previous example is shown below and includes all the bands information present in each product.

```
{
   "id": "input_parameter",
   "stac_version": "1.0.0",
   "description": "A collection for input input_parameter",
   "title": "input_parameter input",
   "extent": {
       "spatial": { "bbox": [[-21.2975, 7.56004, 22.2513, 20.9598 ]] },
       "temporal": { "interval": [["2020-06-23T09:27:43Z","2020-07-01T10:02:42Z"]] }
   },
   "license": "CC-BY-SA-4.0",
   "stac_extensions": [ "eo" ],
   "properties": {
       "eo:bands": [
              {"name": "LST_ancillary_ds",
               "description": "LST ancillary measurement dataset"},
              {"name": "LST_in",
               "description": "LST_in Data Set"},
              ....
              {"name": "met_tx",
               "description": "Meteorological parameters regridded onto the 16km tie points"},
              {"name": "time_in",
               "description": "Time annotations for the 1 KM grid"}
       ],
       "eo:gsd": [ 1000 ],
       "eo:platform": [
           "sentinel-3a",
           "sentinel-3b"
       ],
       "eo:instrument": "SLSTR"
   },
   "links": [
       {
           "rel": "item",
           "href":
"./input1/S3B_SL_2_LST____20200624T090133_20200624T090433_20200624T105644_0180_040_221_2700_LN2_O_NR
_004/S3B_SL_2_LST____20200624T090133_20200624T090433_20200624T105644_0180_040_221_2700_LN2_O_NR_004.
json",
           "type": "application/json"},
         ...
       {
           "rel": "item",
           "href":
"./input3/S3B_SL_2_LST____20200624T104232_20200624T104532_20200625T161613_0179_040_222_2700_LN2_O_NT
_004/S3B_SL_2_LST____20200624T104232_20200624T104532_20200625T161613_0179_040_222_2700_LN2_O_NT_004.
json",
           "type": "application/json"
       },
       {
           "rel": "root",
           "href": "../catalog.json",
           "type": "application/json"
       },
       {
           "rel": "parent",
           "href": "../catalog.json",
           "type": "application/json"
       }
   ]
}
```

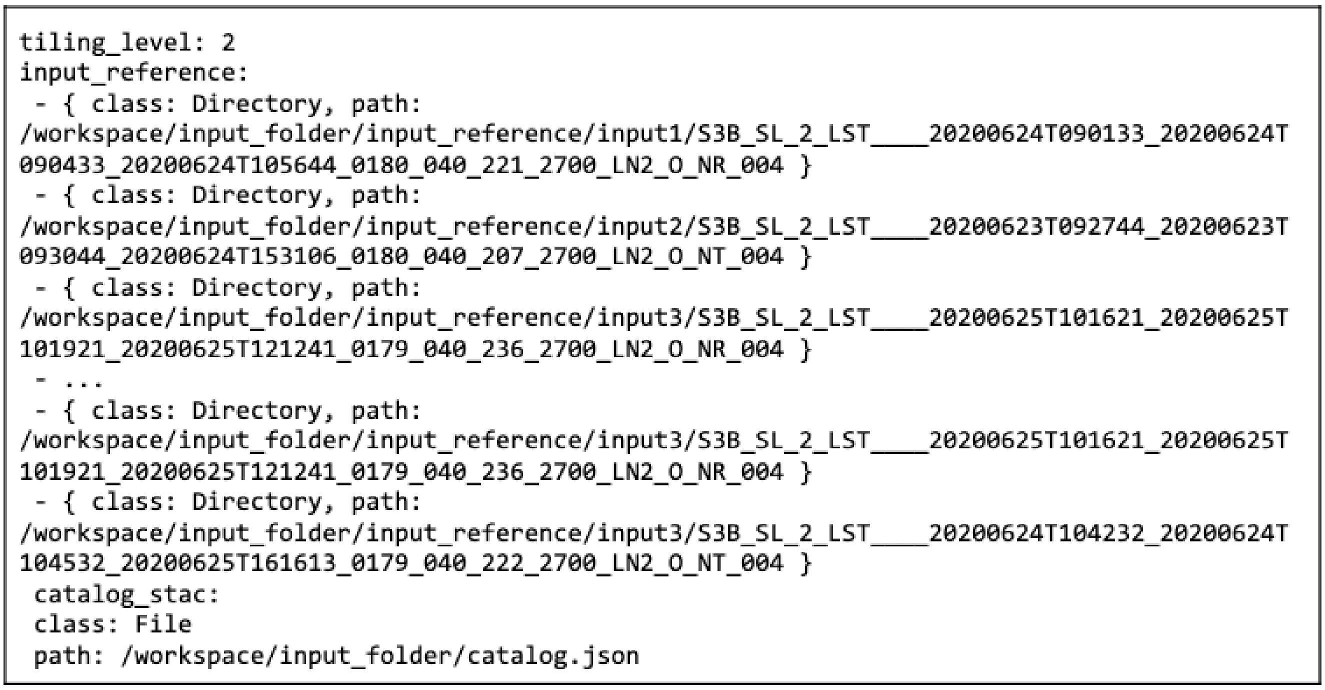
Figure 13. STAC catalog root folder to describe the input products

The STAC collection file links the items of the input parameter in a separate dataset item file along with the data itself. The STAC collection file may contain summaries of items sharing common properties.

The STAC item *File* is a GeoJSON file that describes each dataset with its properties and assets. The ADES should resolve special cases like compressed files by describing those assets and extracting more information from the dataset and referencing them in the item file.

Data is now ready to be processed and the ADES ensures that the input data folder is properly mounted on each cluster node and accessible from the container that executes the processing.

A preliminary step before launching effectively the CWL execution is to prepare the input file for the `cwl-runner` tool as an input YAML document as shown in the Figure below.



```
tiling_level: 2
input_reference:
 - { class: Directory, path:
/workspace/input_folder/input_reference/input1/S3B_SL_2_LST____20200624T090133_20200624T
090433_20200624T105644_0180_040_221_2700_LN2_O_NR_004 }
 - { class: Directory, path:
/workspace/input_folder/input_reference/input2/S3B_SL_2_LST____20200623T092744_20200623T
093044_20200624T153106_0180_040_207_2700_LN2_O_NT_004 }
 - { class: Directory, path:
/workspace/input_folder/input_reference/input3/S3B_SL_2_LST____20200625T101621_20200625T
101921_20200625T121241_0179_040_236_2700_LN2_O_NR_004 }
 - ...
 - { class: Directory, path:
/workspace/input_folder/input_reference/input3/S3B_SL_2_LST____20200625T101621_20200625T
101921_20200625T121241_0179_040_236_2700_LN2_O_NR_004 }
 - { class: Directory, path:
/workspace/input_folder/input_reference/input3/S3B_SL_2_LST____20200624T104232_20200624T
104532_20200625T161613_0179_040_222_2700_LN2_O_NT_004 }
 catalog_stac:
 class: File
 path: /workspace/input_folder/catalog.json
```

Figure 14. YAML file with the input products to be executed by the `cwl-runner` tool

7.2.2. Data Stage-out

The data stage-out is the ADES operation of retrieving the application execution results and pushing them to a persistent storage.

The application descriptor uses CWL statements to define what results are pushed from the execution container to the ADES local filesystem. A typical CWL execution engine also provides a simple manifest with the list of generated results.

Application results may be used as inputs to other node(s) of a wider directed acyclic graph defined by the EMS or as final results to be consumed by the end-user.

The application results must be complemented by metadata elements as the start time & end time or the geographical footprint. These two elements are the minimum set of metadata elements to enable later discovery in a catalog. In the majority of the cases, this minimal set is not sufficient for subsequent workflow nodes to understand what they are being provided with.

With such a context, the application may provide a local STAC catalog including collections, items and assets describing the generated results. The STAC data model is rich and extendable and covers the large majority of requirements. This local STAC catalog is thus seen as an application results manifest.

The ADES takes the STAC `catalog.json` file as the generated results entry point and uses the href links to collections (these are optional), items and assets and uses the found metadata and physical files to push them to a persistent storage and push the metadata to the resource manager and finally provide the WPS execution response.

The diagram below shows how the ADES does the steps explained above.

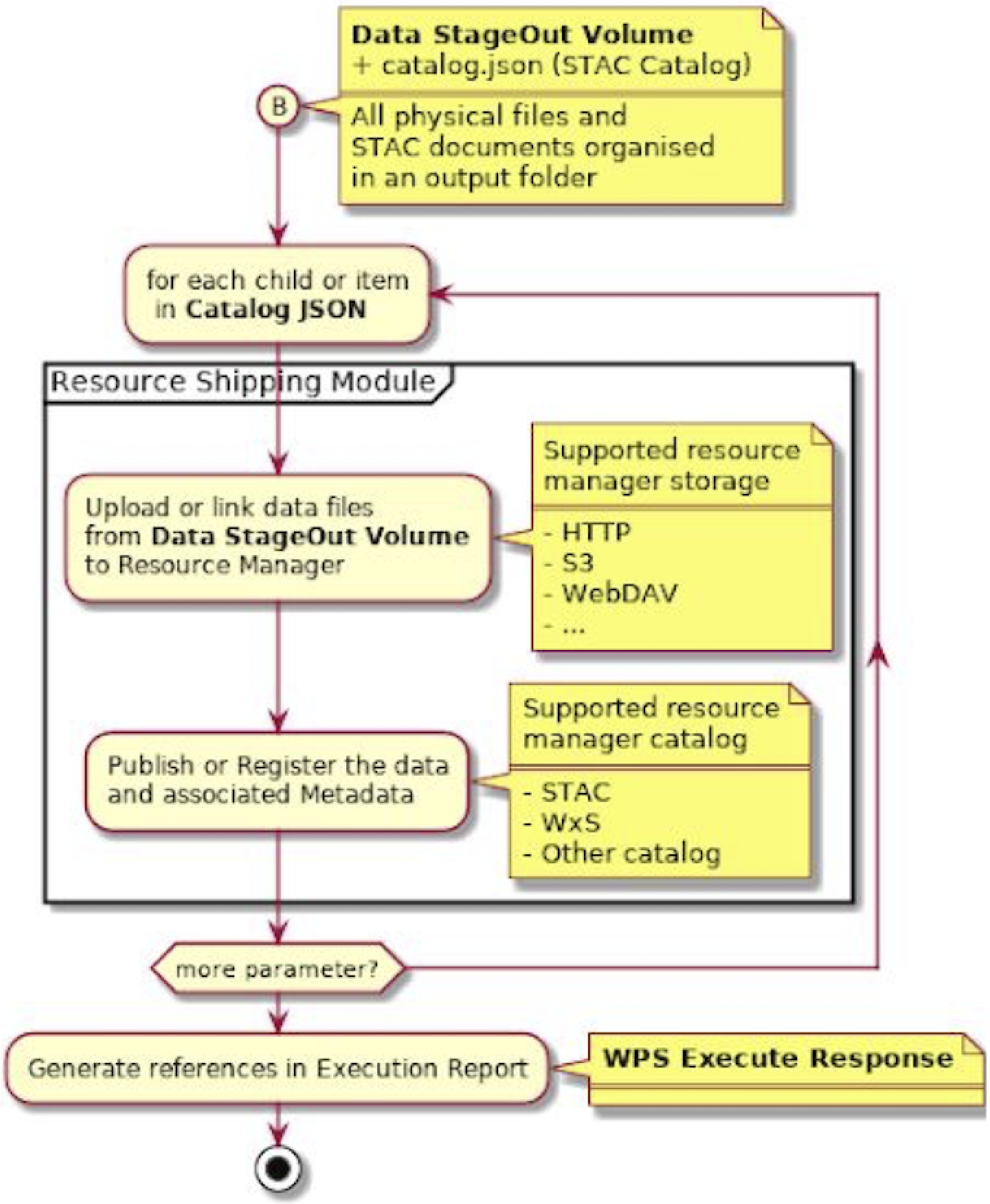

Data StageOut Volume
+ catalog.json (STAC Catalog)
All physical files and STAC documents organised in an output folder
B
for each child or item in Catalog JSON
Resource Shipping Module
Upload or link data files from Data StageOut Volume to Resource Manager
Supported resource manager storage
- HTTP
- S3
- WebDAV
- ...
Publish or Register the data and associated Metadata
Supported resource manager catalog
- STAC
- WxS
- Other catalog
more parameter?
Generate references in Execution Report
WPS Execute Response

Figure 15. ADES workflow steps for data stage-in from the CWL service definition and WPS execution request

Chapter 8. Pilot Applications

This Section presents the EO Applications brought by the different application developers to the OGC Earth Observation Applications Pilot and documents all the steps for their deployment and execution. The experiences gained by working closely together with developers challenged the platform approach and guided the evolutions proposed in the previous sections.

8.1. EU SatCen Sentinel-1 Change Detection

The European Union Satellite Centre (SatCen) brought an application taking as input an interferometric pair of *Sentinel-1 Single Look Complex* acquisition. The app produces the interferometric coherence between the two acquisitions and the backscatter for each of the acquisitions as results.

Sentinel-1 Level-1 Single Look Complex (SLC) products are images in the slant range by azimuth imaging plane, in the image plane of satellite data acquisition. Each image pixel is represented by a complex (I and Q) magnitude value and therefore contains both amplitude and phase information. The *Interferometric Wide* swath (IW) SLC product contains one image per sub-swath per polarization channel, for a total of three or six images. Each sub-swath image consists of a series of bursts, where each burst was processed as a separate SLC image.

The coherence is the fixed relationship between waves in a beam of electromagnetic (EM) radiation. Two wave trains of EM radiation are coherent when they are in phase. That is, they vibrate in unison. In terms of the application to things like radar, the term coherence is also used to describe systems that preserve the phase of the received signal. In an interferogram, coherence is a measure of correlation. It ranges from 0, where there is no useful information in the interferogram; to 1, where there is no noise in the interferogram (a perfect interferogram).

The backscatter is the portion of the outgoing radar signal that the target redirects directly back towards the radar antenna. Backscattering is the process by which backscatter is formed. The scattering cross section in the direction toward the radar is called the backscattering cross section; the usual notation is the symbol sigma. It is a measure of the reflective strength of a radar target. The normalized measure of the radar return from a distributed target is called the backscatter coefficient, or sigma nought, and is defined as per unit area on the ground.

SatCen's application uses ESA's *Sentinel Application Platform* toolbox (SNAP) to derive the coherence and the backscatter and the *Geospatial Data Abstraction Library* (GDAL) to generate the RGB composite combining the outputs produced by SNAP.

This application implements the fan-in pattern where the two Sentinel-1 SLC inputs are processed in a single process and the input data is staged-in. The processing requires a computing environment with at least 14GB of RAM as declared by SatCen.

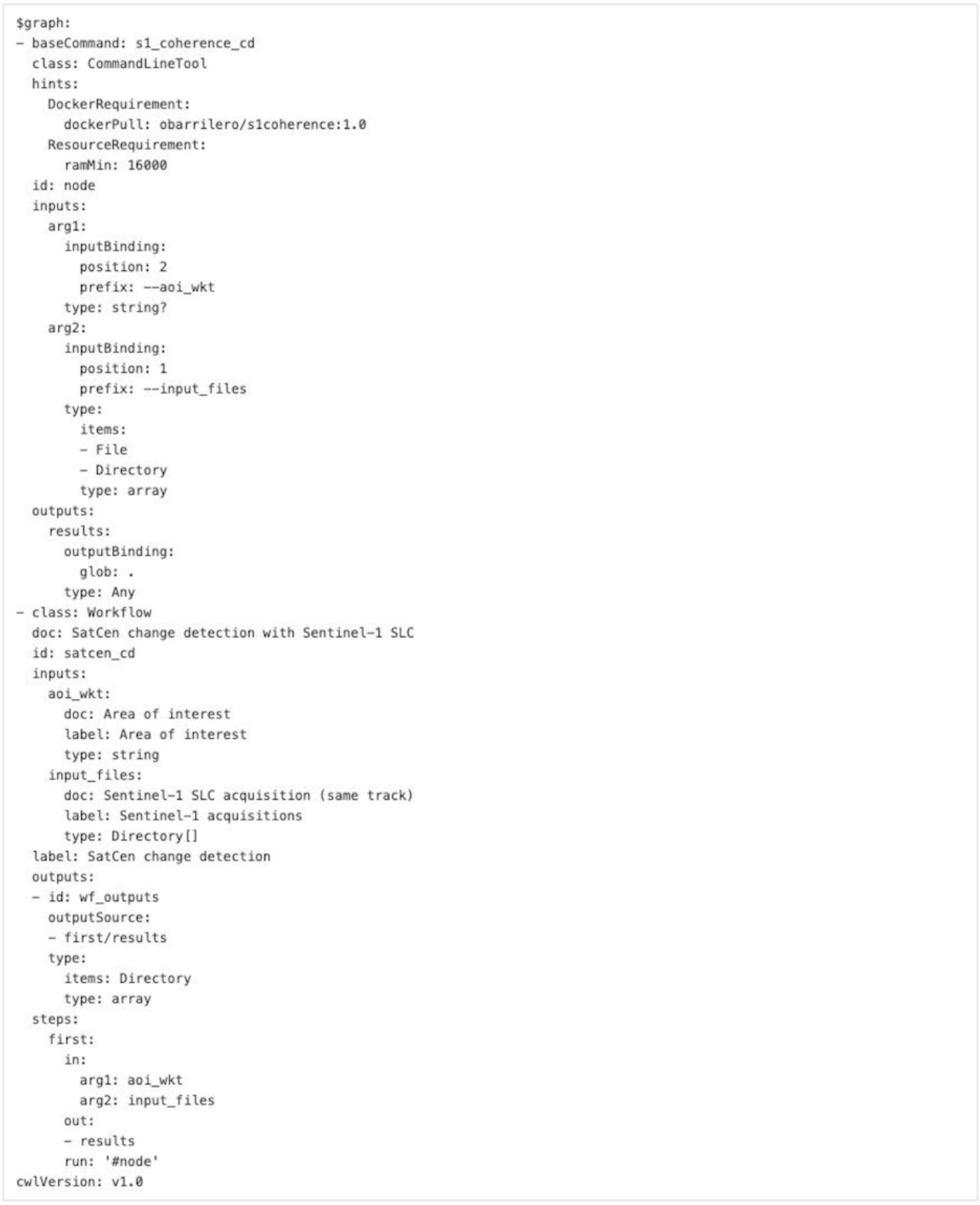
SatCen provided a simple CWL file with the *CommandLineTool* class including the *DockerRequirement* hint providing the Docker image to use.

Terradue updated the provided CWL to include:

- A CWL Workflow class to allow providing a title and a description to the application

- The definition of the Workflow inputs to allow the definition of a title and description for each of the Workflow inputs
- The specification of the minimum RAM for the container with the *ResourceRequirement* CWL requirement.

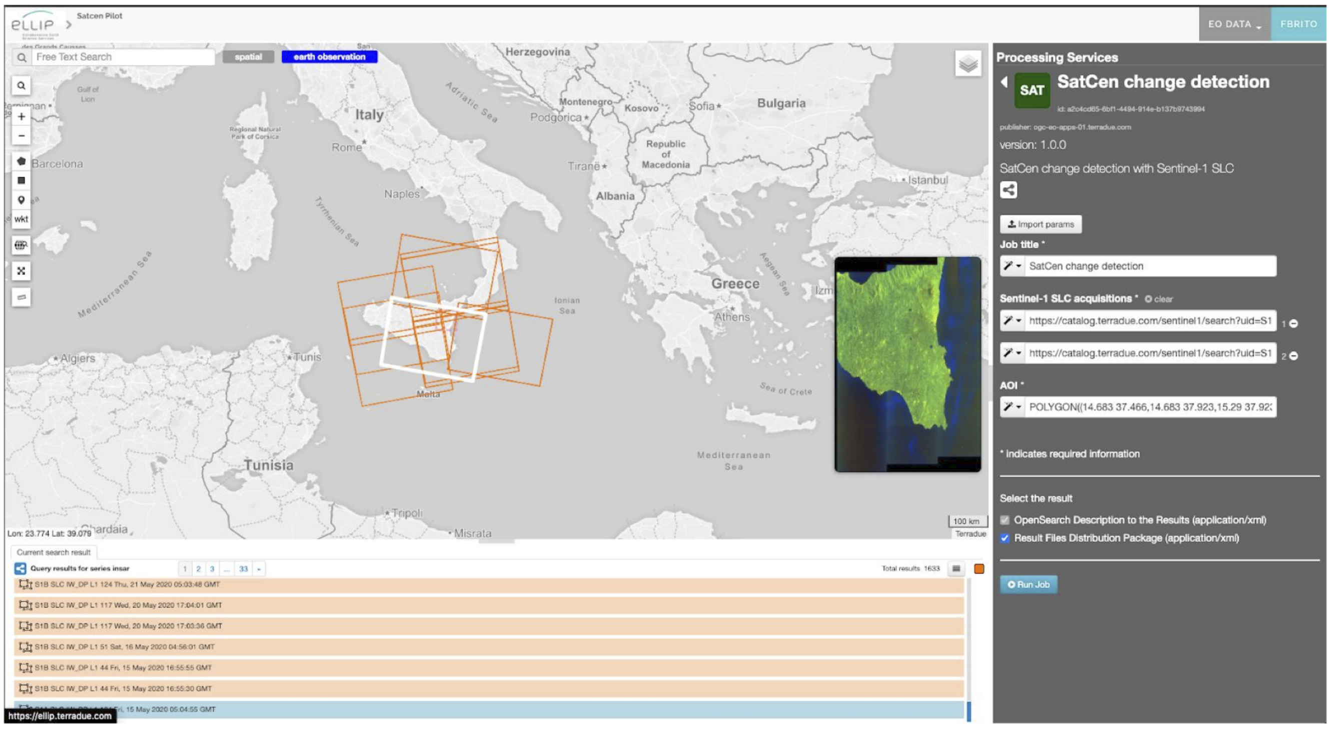
The final CWL is listed in the figure below.



```
$graph:
- baseCommand: s1_coherence_cd
  class: CommandLineTool
  hints:
    DockerRequirement:
      dockerPull: obarrilero/s1coherence:1.0
    ResourceRequirement:
      ramMin: 16000
  id: node
  inputs:
    arg1:
      inputBinding:
        position: 2
        prefix: --aoi_wkt
      type: string?
    arg2:
      inputBinding:
        position: 1
        prefix: --input_files
      type:
        items:
        - File
        - Directory
        type: array
  outputs:
    results:
      outputBinding:
        glob: .
      type: Any
- class: Workflow
  doc: SatCen change detection with Sentinel-1 SLC
  id: satcen_cd
  inputs:
    aoi_wkt:
      doc: Area of interest
      label: Area of interest
      type: string
    input_files:
      doc: Sentinel-1 SLC acquisition (same track)
      label: Sentinel-1 acquisitions
      type: Directory[]
  label: SatCen change detection
  outputs:
  - id: wf_outputs
    outputSource:
    - first/results
    type:
      items: Directory
      type: array
  steps:
    first:
      in:
        arg1: aoi_wkt
        arg2: input_files
      out:
      - results
      run: '#node'
cwlVersion: v1.0
```

Figure 16. Application Package File for the SatCen Application in CWL

The application was then deployed on Ellip Platform and exposed on a Web based graphical interface as shown in the figure below.



Processing Services

SatCen change detection

id: a2c4cd65-6bf1-4494-914e-b137b9743994

publisher: ogc-eo-apps-01.terradue.com

version: 1.0.0

SatCen change detection with Sentinel-1 SLC

Import params

Job title *: SatCen change detection

Sentinel-1 SLC acquisitions * clear

1: https://catalog.terradue.com/sentinel1/search?uid=S1

2: https://catalog.terradue.com/sentinel1/search?uid=S1

AOI *: POLYGON((14.683 37.466,14.683 37.923,15.29 37.92

* indicates required information

Select the result

- [✓] OpenSearch Description to the Results (application/xml)
- [✓] Result Files Distribution Package (application/xml)

Run Job

Current search result

Query results for series insar — 1 2 3 ... 33 — Total results 1633

S1B SLC IW_DP L1 124 Thu, 21 May 2020 05:03:48 GMT
S1B SLC IW_DP L1 117 Wed, 20 May 2020 17:04:01 GMT
S1B SLC IW_DP L1 117 Wed, 20 May 2020 17:03:36 GMT
S1B SLC IW_DP L1 51 Sat, 16 May 2020 04:56:01 GMT
S1B SLC IW_DP L1 44 Fri, 15 May 2020 16:55:55 GMT
S1B SLC IW_DP L1 44 Fri, 15 May 2020 16:55:30 GMT
Fri, 15 May 2020 05:04:55 GMT

Figure 17. SatCen Application deployed in Ellip Platform

The interaction with SatCen allowed us to consolidate the concept of using the CWL language:

- As an application descriptor and thus conveying the information required by a Platform to generate and deploy a Web Processing Service
- As the language to express directed acyclic graph (DAG) and thus build and describe complex EO applications such as SatCen's
- As the language to declare computing environment requirements such as the minimum RAM
- As the language to describe the application inputs and outputs

Nevertheless, the application defined by SatCen for the EO Apps Pilot is a simplified subset of a real-life scenario.

During the Pilot, we've used a single pair of Sentinel-1 SLC. The inputs could be up to four when using Sentinel-1 A and B over some areas of the world to obtain the change detection with 6 days apart instead of 12 days. In this case, the acquisitions of the same day may have to be assembled before they're processed by the interferometric chain.

During the Pilot, we've processed a single swath. The swaths to process may be IW1, IW2 and IW3 (or a combination of these). We've also processed a single polarization VV. SatCen uses this polarization so we could skip that as a parameter.

So in terms of an efficient workflow that reduces moving around large sets of data, we would define a workflow running on the ADES with several steps or directed acyclic graph nodes. We would have a first node that generates processing sets split by swath and master and slave combinations to do the pre-processing in a second node. This second node would then pre-process these sets (fan-out) optionally applying the *SliceAssembly* operator. A third node and finale would then do the TOPSAR-Merge and generate the Coherence/backscatter products.

This approach can be achieved using a CWL that defines these steps in a workflow.

8.2. DLR TimeScan Sentinel-2 Vegetation indexes

The German space agency (DLR) brought an application taking as input one or more Sentinel-2 Level 1C products to produce a list of vegetation indexes.

SENTINEL-2 is a wide-swath, high-resolution, multi-spectral imaging mission with its Multispectral Instrument (MSI) sampling 13 spectral bands: four bands at 10 meters, six bands at 20 meters and three bands at 60 meters spatial resolution.

The DLR TimeScan Processing Framework utilizes a novel approach to exploit large multi temporal satellite time series for a variety of applications like urban monitoring and land cover classification. DLR’s goal in the OGC EO Apps Pilot is thus to adapt the TimeScan processors to an Application Package that is deployable and executable in an EMS/ADES infrastructure.

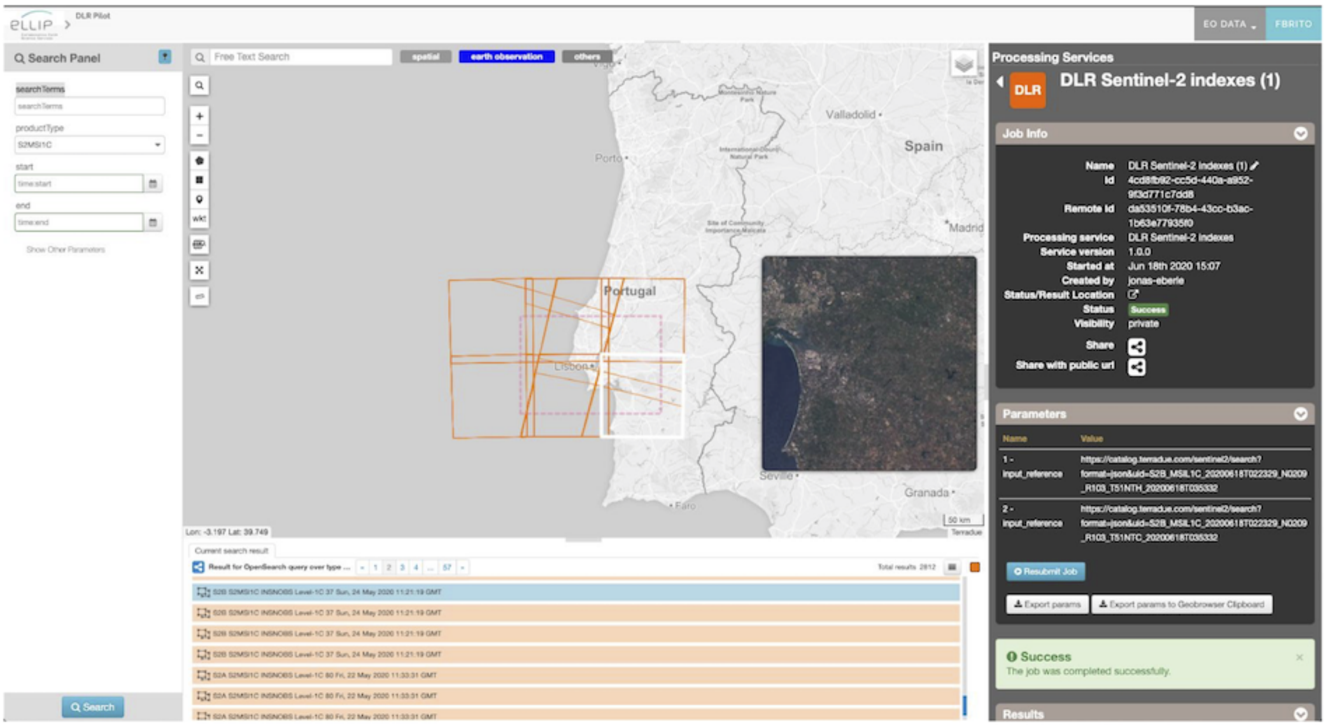
The provided application implemented the Sentinel-2 pre-processing module as a processing graph in the ESA SNAP Toolbox including a reader interpreting the multi-resolution input, a resampling operator rescaling all bands to 20m spatial resolution, and the SNAP *Sentinel2.Idepix* operator generating a flag band with cloud identification, whereas *BandMaths* operators compute the vegetation indices.

As done by SatCen, the DLR provided a CWL file as the application package. Terradue applied the same enhancements to allow deploying the DLR application on the Ellip Platform as a WPS processing service:

- A CWL Workflow class to allow providing a title and a description to the application
- The definition of the Workflow inputs to allow the definition of a title and description for each of the Workflow inputs
- The *ScatterFeatureRequirement* CWL requirement to instruct the ADES that the Sentinel-2 inputs can be processed in parallel and independently (fan-out pattern)

Extending the CWL lessons learned with the SatCen case, the interaction with DLR allowed us to consolidate the concept of using the CWL language to express the fan-out pattern *ScatterFeatureRequirement* CWL requirement to process inputs in parallel and independently and thus reducing the processing time of large stacks of Sentinel-2 data required by the TimeScan processing scenarios.

The DLR team was provided access to a Web based graphical interface to submit processing jobs as shown in the figure below.



S2B S2MSI1C INSNOBS Level-1C 37 Sun, 24 May 2020 11:21:19 GMT
S2B S2MSI1C INSNOBS Level-1C 37 Sun, 24 May 2020 11:21:19 GMT
S2B S2MSI1C INSNOBS Level-1C 37 Sun, 24 May 2020 11:21:19 GMT
S2B S2MSI1C INSNOBS Level-1C 37 Sun, 24 May 2020 11:21:19 GMT
S2A S2MSI1C INSNOBS Level-1C 80 Fri, 22 May 2020 11:33:31 GMT
S2A S2MSI1C INSNOBS Level-1C 80 Fri, 22 May 2020 11:33:31 GMT
S2A S2MSI1C INSNOBS Level-1C 80 Fri, 22 May 2020 11:33:31 GMT

DLR Sentinel-2 indexes (1)

Job Info

Name	DLR Sentinel-2 indexes (1)
Id	4cd8fb92-cc5d-440a-a952-9f3d771c7dd8
Remote Id	da53510f-78b4-43cc-b3ac-1b63e77935f0
Processing service	DLR Sentinel-2 indexes
Service version	1.0.0
Started at	Jun 18th 2020 15:07
Created by	jonas-eberle
Status/Result Location	
Status	Success
Visibility	private
Share	
Share with public url	

Parameters

Name	Value
1 - input_reference	https://catalog.terradue.com/sentinel2/search?format=json&uid=S2B_MSIL1C_20200618T022329_N0209_R103_T51NTH_20200618T035332
2 - input_reference	https://catalog.terradue.com/sentinel2/search?format=json&uid=S2B_MSIL1C_20200618T022329_N0209_R103_T51NTC_20200618T035332

Success
The job was completed successfully.

Figure 18. DLR Application deployed in Ellip Platform

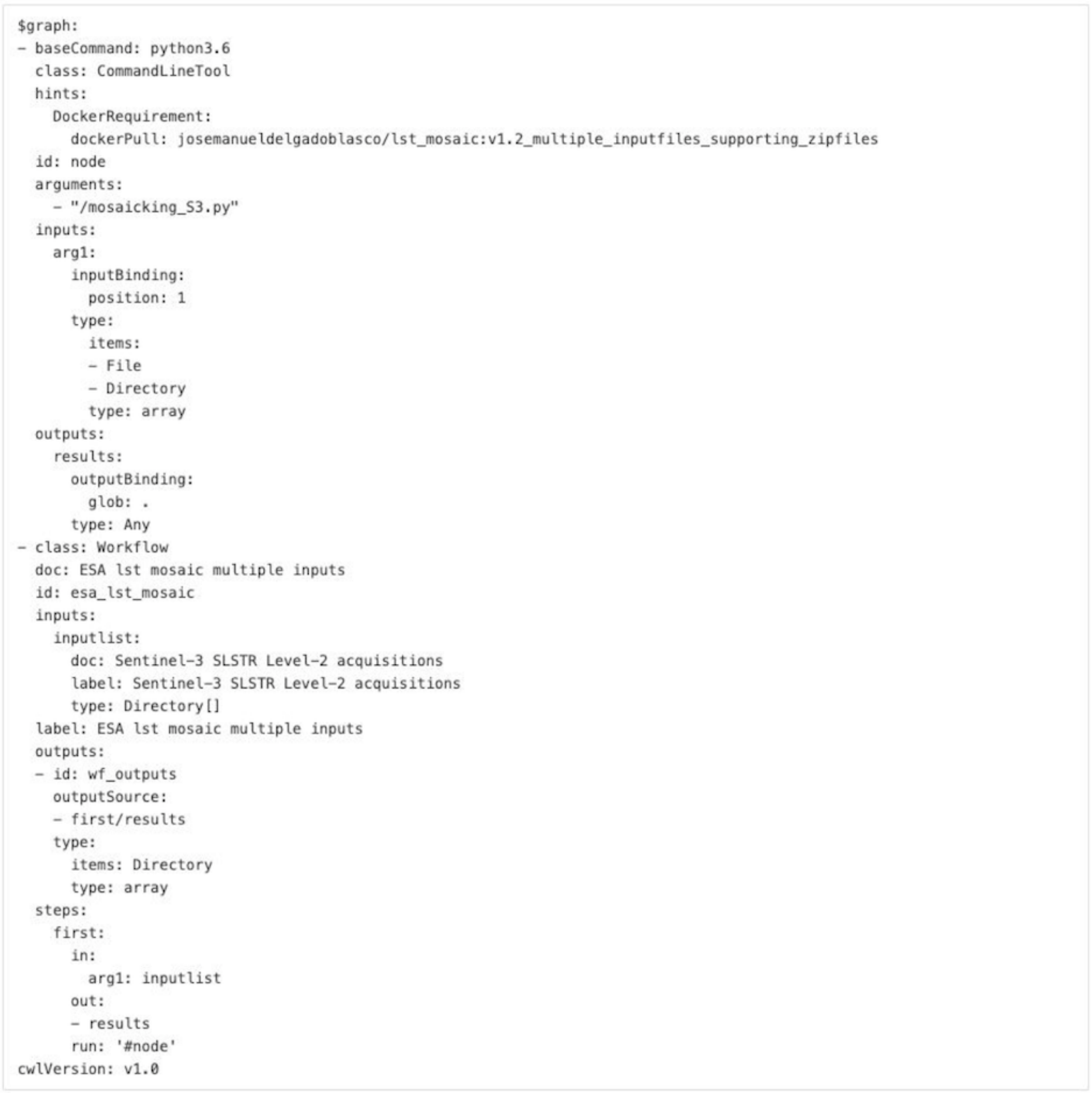
8.3. ESA Sentinel-3 SLSTR Land Surface Temperature

The European Space Agency (ESA), supported by Rhea, created an application taking as input a number of Sentinel-3 SLSTR Level-2 Land products to produce a temporal and spatial aggregation of the Land Surface Temperature (LST).

The principal aim of the SLSTR instrument mission on-board SENTINEL-3 is to maintain continuity with the (A)ATSR series of instruments (on-board the ERS-2 and Envisat satellites). The (A)ATSR provided a reference Sea Surface Temperature dataset for other satellite missions and the SLSTR design is based on the reuse of AATSR concepts with existing and qualified technologies.

The application takes as inputs gridded Land Surface Temperature generated on the wide 1 km measurement grid to aggregate the observations in time and space producing an LST mosaic.

ESA also provided a simple CWL with the `CommandLineTool` CWL class only. As such, Terradue applied the same changes to the provided CWL to yield the script below.



```
$graph:
- baseCommand: python3.6
  class: CommandLineTool
  hints:
    DockerRequirement:
      dockerPull: josemanueldelgadoblasco/lst_mosaic:v1.2_multiple_inputfiles_supporting_zipfiles
  id: node
  arguments:
    - "/mosaicking_S3.py"
  inputs:
    arg1:
      inputBinding:
        position: 1
      type:
        items:
        - File
        - Directory
        type: array
  outputs:
    results:
      outputBinding:
        glob: .
      type: Any
- class: Workflow
  doc: ESA lst mosaic multiple inputs
  id: esa_lst_mosaic
  inputs:
    inputlist:
      doc: Sentinel-3 SLSTR Level-2 acquisitions
      label: Sentinel-3 SLSTR Level-2 acquisitions
      type: Directory[]
  label: ESA lst mosaic multiple inputs
  outputs:
  - id: wf_outputs
    outputSource:
    - first/results
    type:
      items: Directory
      type: array
  steps:
    first:
      in:
        arg1: inputlist
      out:
      - results
      run: '#node'
cwlVersion: v1.0
```

Figure 19. Application Package File for the ESA Application in CWL

As for the SatCen and DLR, ESA was provided with the access to a web-based interface to interact with the deployed WPS processing service.

The interaction with ESA's application allowed us to define how an application provider is asked to test his application before providing it for deployment on an exploitation platform. The method is quite simple: using the `cwltool`, a CWL execution tool and the set of processing parameters defined in a YAML file, the application provider can run his/her application locally and thus have a reference execution that can be used for validation purposes on the platform in an acceptance phase for example. This testing step is also a valid strategy to identify potential not recommended practices in creating the Docker container (root execution for instance).

This facet of the CWL as a mechanism to describe the application, define how to execute it, describe the inputs/outputs and finally validate the docker execution as if on an exploitation platform provided an additional confirmation that CWL may be the sole required element of an application package to deploy it on an ADES as a WPS service.

8.4. Food Security applications

Terradue brought the EO Apps Pilot a set of applications derived from the World Food Programme (WFP) Vulnerability Analysis and Mapping (VAM) network that addresses critical aspects of the food security:

- Food availability: The amount of food physically available to a household or at national level
- Access to food: The physical and economical ability of a household to acquire adequate amounts of food
- Food utilization: The intra-household use of the food accessible and the individual's ability to absorb and use those nutrients.

The use-cases are focused on seasonal monitoring for tracking growing season and climate & vegetation time series analysis with:

- Vegetation performance
- Normalized Difference Vegetation Index (NDVI) smoothed yearly time series
- NDVI differences
- Drought monitoring
- Synthetic-Aperture Radar (SAR) backscatter average for growing season

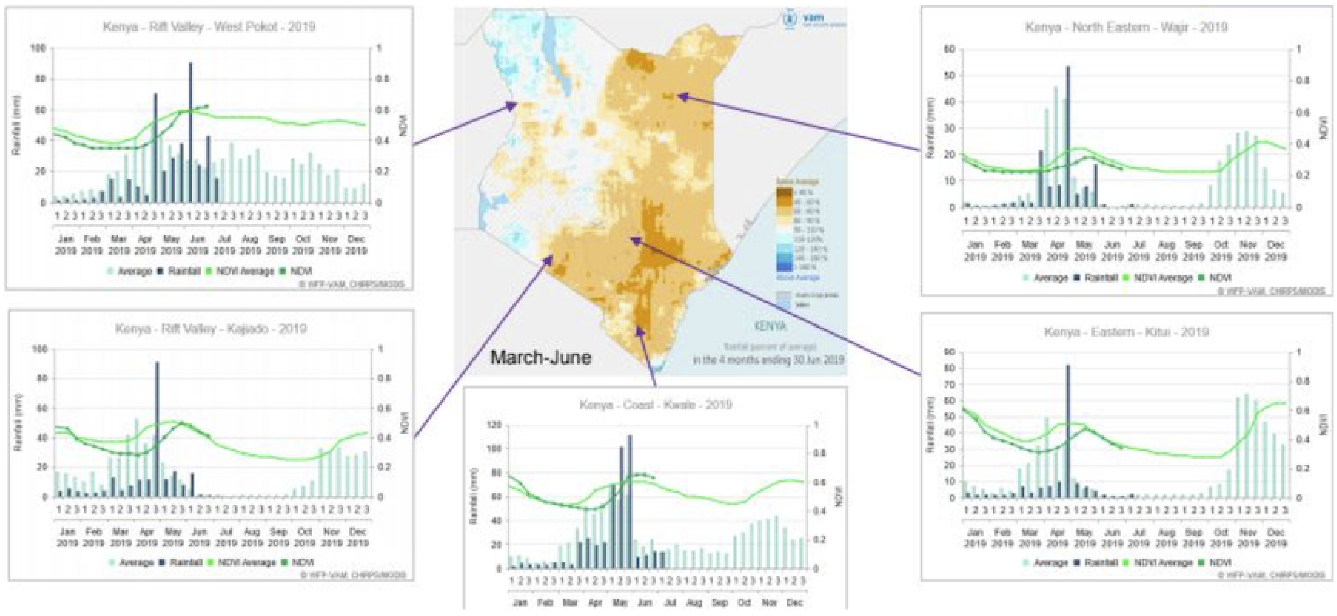


Figure 20. Example of seasonal monitoring for Food security application

The derived applications are depicted in the figure below.

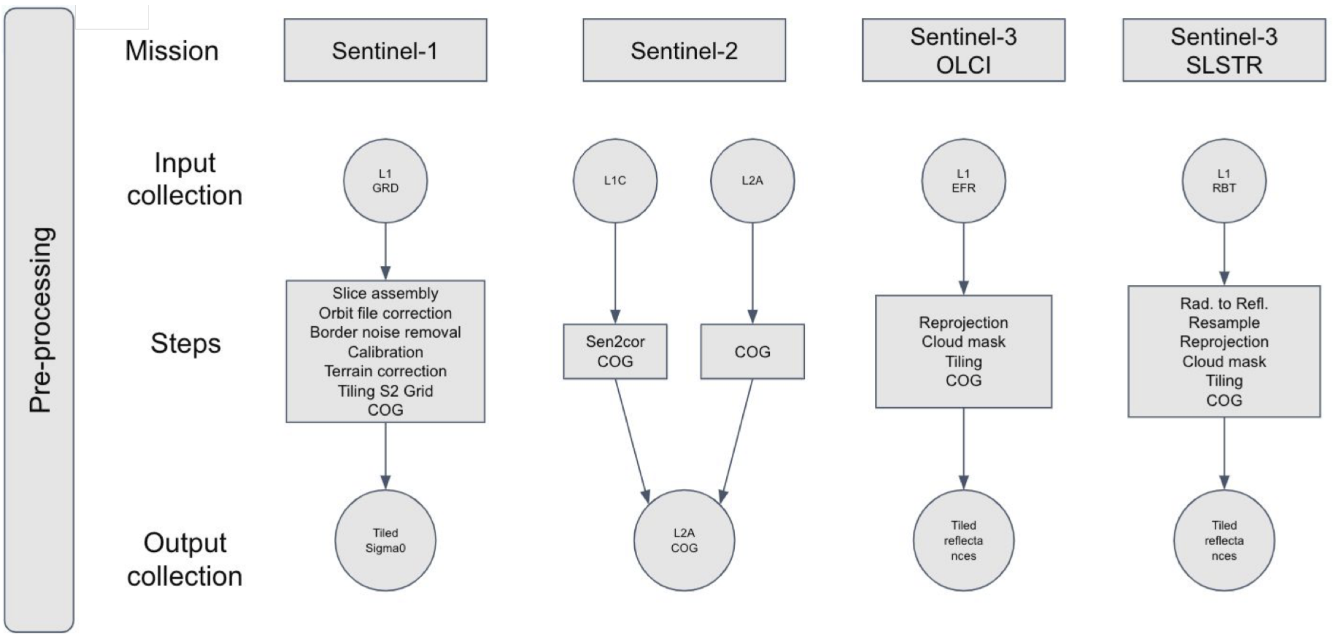


Figure 21. Processing steps for the Food security application

We describe thoroughly the Sentinel-3 SLSTR pre-processing as the remaining applications implement the same concepts.

As an example of the applications provided, the next paragraphs describe the Sentinel-3 SLSTR reprojection and tiling that takes as input one or more Sentinel-3 SLSTR Level-2 products to:

- Reproject the data to EPSG:4326 [http://www.opengis.net/def/crs/EPSG/0/4326]
- Create tiles according to the OGC tiling scheme used in the WMTS specification

These tiles are then post-processed in another application that applies the Whitaker smoothing filter to, on the one hand, fill the temporal and spatial gaps and on the other smooth the time-series.

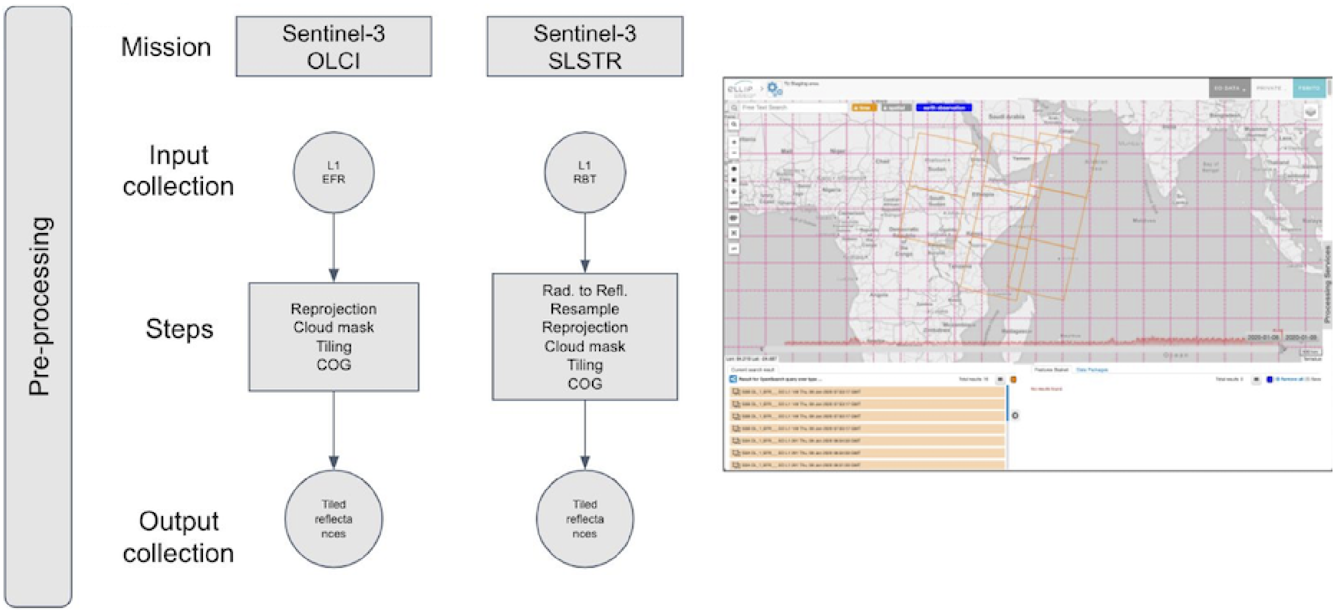
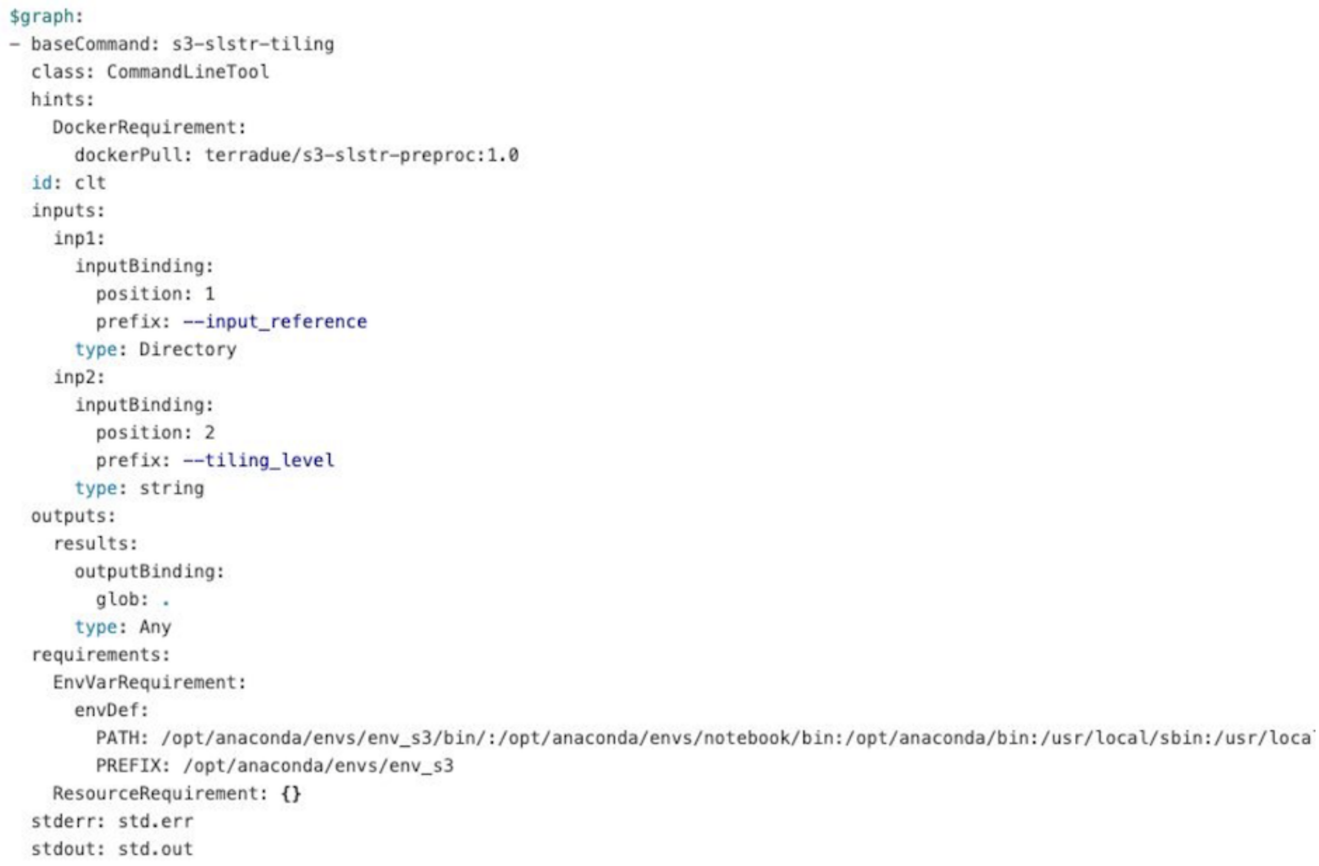


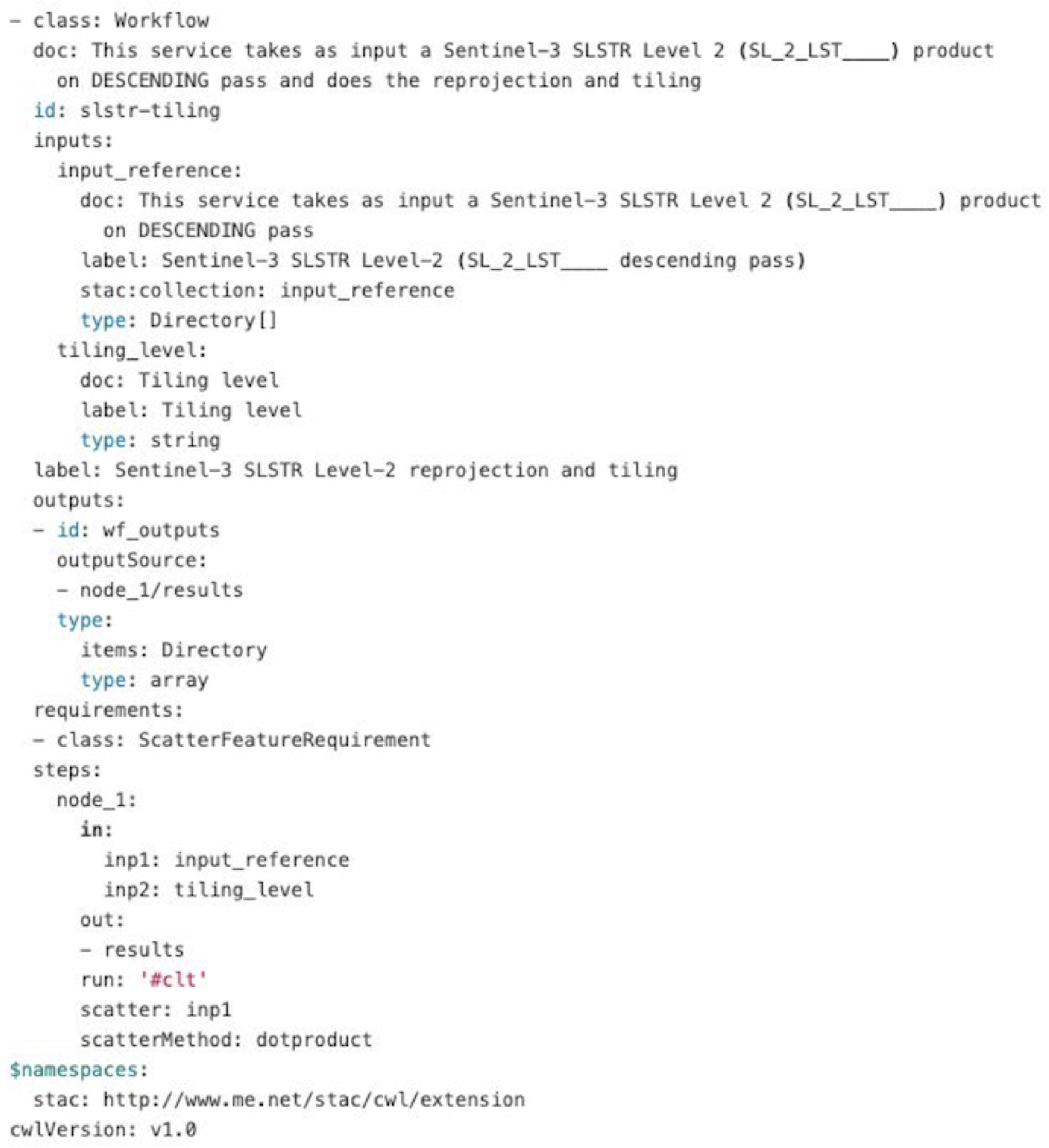
Figure 22. Processing Steps for the Sentinel-3 SLSTR Reprojection

The Sentinel-3 SLSTR reprojection and tiling CWL file is shown in the figure below.



```
$graph:
- baseCommand: s3-slstr-tiling
  class: CommandLineTool
  hints:
    DockerRequirement:
      dockerPull: terradue/s3-slstr-preproc:1.0
  id: clt
  inputs:
    inp1:
      inputBinding:
        position: 1
        prefix: --input_reference
      type: Directory
    inp2:
      inputBinding:
        position: 2
        prefix: --tiling_level
      type: string
  outputs:
    results:
      outputBinding:
        glob: .
      type: Any
  requirements:
    EnvVarRequirement:
      envDef:
        PATH: /opt/anaconda/envs/env_s3/bin/:/opt/anaconda/envs/notebook/bin:/opt/anaconda/bin:/usr/local/sbin:/usr/loca
        PREFIX: /opt/anaconda/envs/env_s3
    ResourceRequirement: {}
  stderr: std.err
  stdout: std.out
```

Figure 23. Application Package File for Sentinel-3 SLSTR reprojection in CWL



```
- class: Workflow
  doc: This service takes as input a Sentinel-3 SLSTR Level 2 (SL_2_LST____) product
    on DESCENDING pass and does the reprojection and tiling
  id: slstr-tiling
  inputs:
    input_reference:
      doc: This service takes as input a Sentinel-3 SLSTR Level 2 (SL_2_LST____) product
        on DESCENDING pass
      label: Sentinel-3 SLSTR Level-2 (SL_2_LST____ descending pass)
      stac:collection: input_reference
      type: Directory[]
    tiling_level:
      doc: Tiling level
      label: Tiling level
      type: string
  label: Sentinel-3 SLSTR Level-2 reprojection and tiling
  outputs:
  - id: wf_outputs
    outputSource:
    - node_1/results
    type:
      items: Directory
      type: array
  requirements:
  - class: ScatterFeatureRequirement
  steps:
    node_1:
      in:
        inp1: input_reference
        inp2: tiling_level
      out:
      - results
      run: '#clt'
      scatter: inp1
      scatterMethod: dotproduct
$namespaces:
  stac: http://www.me.net/stac/cwl/extension
cwlVersion: v1.0
```

Figure 24. Application Package File for Sentinel-3 SLSTR Tiling in CWL

This CWL was extended to declare that the application expects a STAC local catalog with the Sentinel-3 staged data included in a STAC collection named *input_reference*.

This type of extension in a CWL is foreseen and provides additional degrees of freedom to express concepts such as the expected input format for the application.

During the EO Apps Pilot, we presented STAC as a potential solution to be the input and output manifests between the ADES and the application. STAC provides several advantages to application developers as data is staged and described as a local STAC catalog that describes a catalog, one or more collections, the items in the collections and the assets in the item. The assets describe the content at band level or the metadata and thus avoiding developers to implement find, grep, glob strategies to identify the GeoTIFFs or XML manifests.

The drawback in terms of application interoperability experiment is that the Terradue applications have only been deployed on Ellip as the other platforms did not implement the support for STAC during the Pilot.

Chapter 9. Conclusions

Previous OGC Testbeds initiated the design of an application package for Earth Observation Applications in distributed Cloud Platforms. The application package must provide information about the software item, executable, metadata and dependencies such that it can be deployed and executed within an Exploitation Platform in a service compliant with the OGC API Process specification. The activity performed during this OGC Earth Observation Applications Pilot evaluated and tested the maturity of this architecture in real world scenarios.

Terradue responded to the invitation with its own Ellip Platform that explores the benefit of Cloud technologies for large scale processing of EO data. During this pilot we were able to evaluate the maturity of the architecture with applications brought by several developers that work with data from Earth observation satellites.

These developers brought different views and requirements in terms of data discovery and processing to challenge the application package architecture and platform readiness

Application developers were able to create containers with their runtime environment and dependencies. The overall package was described using the Common Workflow Language where the developer was able to create single or multiple nodes applications without the need to specify new properties. Data Flow Management between the nodes was achieved by using a local catalogue encoded using STAC, the SpatioTemporal Asset Catalog specification, as a data manifest for application inputs and outputs metadata.

By taking into consideration the viewpoints of both EO platforms and EO application developers, the challenges faced during this pilot consolidated important aspects of the architecture, with the use of CWL to describe the application packaging and STAC to facilitate the data stage-in and stage-out, all within the OGC API Process context.

With the support of several application developers, we confronted the defined architecture with their applications and proposed solutions to several issues, mainly :

- Application package
- Data flow management
- Application scalability

The following subsections contain the conclusions of our work that mitigated these issues and present some recommendations for future activities.

9.1. Application Package

The Application Package targets algorithms (developed by a third-party developer) that are to be deployed in a Cloud Platform. During this pilot, we extended the use of CWL to reference the application containers (e.g. Dockers) and also to allow the definitions of the application parameters, input/output interface and the overall process offering parameters.

Providing a single CWL file showed to reduce the complexity and facilitate the communication between the application providers and the platforms. The CWL file provides the information model for the full application definition from input cardinality and constraint to memory, CPU or storage requirements.

From application developers just wrapping their code on a Workflow class, to Expert users that want to fully control the application workflow, CWL showed a great potential for portability and reproducibility. CWL also demonstrated the importance of independent testing and validation when deploying the applications in a heterogeneous environment of platforms.

As CWL is widely used and there are several backends to support the testing of application packages at local or cluster level there is no major obstacle to its adoption. Documentation about the language is freely available, support forums are quite active and the main patterns for EO application design were explored during this pilot activity. The implemented examples provide a basic template for future applications substantially reducing the implementation effort.

In terms of security, the CWL execution engine can detect situations where the containers are executed as a root user and abort the processes. This situation should be avoided and must be part of the application developers guidelines for building docker images to be deployed on platforms.

As hardware requirements fall outside the scope of the OGC WPS specification, the application package needs also to be able to allow developers to describe their application and thus to convey the information about the runtime environment to the ADES. The approach followed during this Pilot demonstrated how CWL allows the definition of the hardware requirements independently of the container-orchestration system (e.g. Kubernetes) and additional software requirements can be specified as the developer raises the complexity of the application.

9.2. Data flow management

A major issue discussed, but not fully addressed, on previous OGC Testbeds (13, 14 and 15) relates to the approach by the ADES to provide the data stage-in of product input references specified on the WPS request and stage-out the results of the application.

In previous testbeds, the ADES had limited knowledge about the input and output files data contents missing critical information like spatial footprint, sub-items (e.g. masks, bands) and additional metadata (e.g. ground sample distance, orbit direction).

In this pilot we decided to support the data flow management by using a local catalogue encoded using the SpatioTemporal Asset Catalog (STAC) specification as a data manifest for application inputs and outputs. The local catalogue provides knowledge about the input and output files data contents like spatial footprint, sub-items (e.g. masks, bands) and additional metadata.

The usage of CWL together with STAC to support the data stage-in allows the transparent definition of the parameters that convey resources to be made available to the application.

9.3. Application Scalability

In previous OGC testbeds, the majority of implementation examples focused on a single node machine and didn't fully explore the application scalability.

In this pilot, the possibility to directly specify the parallel processing potential of the application at the CWL Workflow class level with the use of CWL requirement `ScatterFeatureRequirement` allowed the support of the data driven application fan-out patterns.

These applications demonstrated the execution of a data processing function that processes concurrently several products generating independent output for each input. As such, parallelization and scalability of the application can be transparently managed by the ADES cluster (e.g. Kubernetes) taking into consideration the available computing resources not only for multiple requests but also the specific application internal workflows.

9.4. Recommendations for Future Work

To continue the activities explored in this pilot, it is important to support the new Transactions extension of the OGC API - Processes. This extension, currently being addressed by the working group, will define the behavior of an implementation that supports the ability to dynamically add, update or remove processes from a deployed *OGC API-Processes* instance.

In this ER we demonstrated that the usage of the Common Workflow Language can define the application package and allows the developer to define single or multiple nodes applications without the need to specify new properties. CWL provides a simple way of pointing to an application container (e.g. Docker) together with the definitions of the application parameters, input/output interface and the overall process offering parameters. It is our recommendation that the Transactions extension use CWL to define the application package.

We recommend that a Best Practice should be created to further document the use of Common Workflow Language for application packages. The proposed best practice should extend the application design patterns discussed in this ER and provide guidelines on how to improve the automatic generation of CWL (e.g. from *Jupyter Notebooks*). The design or creation of specific wizard templates that would facilitate the creation of application packages is also recommended.

We also recommend producing a Best Practice for the data flow management using STAC local catalogues as the input and output manifests between the ADES and the hosted application and between the ADES and the EMS.

Appendix A: Revision History

Table 1. Revision History

Date	**Editor**	**Release**	**Primary clauses modified**	**Descriptions**
June 13, 2020	P. Gonçalves	.1	all	initial TOC
July 2, 2020	P. Gonçalves	.2	all	TOC review
July 9, 2020	P. Gonçalves	.3	6, 7	inputs for EO platforms
July 10, 2020	P. Gonçalves	.4	8	inputs for EO applications
July 14, 2020	P. Gonçalves	1	9	update on conclusion
August 6, 2020	I. Simonis	1.1	all	full review, editorial changes only
October 22, 2020	G. Hobona	1.2	all	final staff review

Appendix B: Bibliography

[1] Commonwl.org: Common Workflow Language Specifications, v1.1, https://www.commonwl.org/v1.1/, (2019).

[2] Radiant Earth Foundation: SpatioTemporal Asset Catalog specification, https://stacspec.org, (2020).