OGC® DOCUMENT: 20-089R1

External identifier of this OGC® document: http://www.opengis.net/doc/BP/eoap/1.0

Open Geospatial Consortium

OGC BEST PRACTICE FOR EARTH OBSERVATION APPLICATION PACKAGE

BEST PRACTICE

PUBLISHED

Version: 1.0
Submission Date: 2021-08-27
Approval Date: 2021-11-13
Publication Date: 2021-12-14
Editor: Pedro Gonçalves

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I ABSTRACT

Platforms for the exploitation of Earth Observation (EO) 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. A fundamental principle of the platform operations concept is to move the EO data processing service’s user to the data and tools, as opposed to downloading, replicating, and exploiting data ‘at home’. In this scope, previous OGC activities 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.

This document defines the Best Practice to package and deploy Earth Observation Applications in an Exploitation Platform. The document is targeting the implementation, packaging and deployment of EO Applications in support of collaborative work processes between developers and platform owners.

The Best Practice includes recommendations for the application design patterns, package encoding, container and data interfaces for data stage-in and stage-out strategies focusing on three main viewpoints: Application, Package and Platform.

II KEYWORDS

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, EO, Application Package, Container, CWL, STAC

III PREFACE

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.

IV SECURITY CONSIDERATIONS

This document defines Best Practice to package and deploy an Application on a Platform that implies a trust relationship between the Application developer, the Application integrator, the Platform and the consumer.

No security considerations have been made for this Best Practice.

V SUBMITTING ORGANIZATIONS

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- European Space Agency
- Terradue
- CRIM
- CubeWerx Inc.
- 52°North GmbH
- SatCen
- Telespazio VEGA UK
- RHEA Group
- Pixalytics
- Solenix
- West University of Timisoara

VI SUBMITTERS

All questions regarding this submission should be directed to the editor or the submitters:

Name	**Representing**
Pedro Gonçalves *(editor)*	Terradue
Fabrice Brito	Terradue
Tom Landry	CRIM
Francis Charette-Migneault	CRIM
Richard Conway	Telespazio VEGA UK
Adrian Luna	European Union Satellite Centre

Omar Barrilero	European Union Satellite Centre
Panagiotis (Peter) A. Vretanos	CubeWerx Inc.
Cristiano Lopes	European Space Agency (ESA)
Antonio Romeo	RHEA Group
Paulo Sacramento	Solenix
Samantha Lavender	Pixalytics
Marian Neagul	West University of Timisoara

1

SCOPE

1 SCOPE

This document defines the Best Practice to package and deploy Earth Observation Applications in an Exploitation Platform.

The document is targeting the implementation, packaging and deployment of EO Applications in support of collaborative work processes between developers and platform owners. It supports developers that want to adapt and package their existing algorithms written in a specific language to be deployed in Earth Observation Exploitation Platforms and exposed through a Web Service endpoint, OGC API — Processes.

The Best Practice includes application design patterns, package encoding, container and data interfaces for data stage-in and stage-out strategies.

Section 6 introduces the information material about 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.

Sections 7, 8 and 9 present the normative material defining the best practices to implement an EO Application, to package an EO Application and to deploy the packaged EO Application.

2 CONFORMANCE

2 CONFORMANCE

This document defines the Best Practice for Earth Observation Application Packages targeting three standardization targets:

- Application — defines the best practices when implementing an EO Application
- Package — defines the best practices when packaging an EO Application
- Platform — defines the best practices when deploying a packaged EO Application in a platform

For the three standardization targets listed above, this Best Practice focuses on Earth Observation Applications that require the staging, input, and/or output of EO Products. The Best Practice also discusses the implications for the Application Package and hosting Platform.

In order to conform to this OGC Best Practice, an application developer shall choose to implement the following conformance classes:

- Conformance Class “Application”
- Conformance Class “Application Staged Inputs”
- Conformance Class “Application Staged Outputs”

In order to conform to this OGC Best Practice, and according to the Application Conformance Class, the application integrator shall implement the following conformance classes:

- Conformance Class “Application Package”
- Conformance Class “Application Package Staged Inputs”
- Conformance Class “Application Package Staged Outputs”

In order to conform to this OGC Best Practice, and according to the Application Package Conformance Class, a platform shall choose to implement the following conformance classes:

- Conformance Class “Platform”
- Conformance Class “Platform Staged Inputs”
- Conformance Class “Platform Staged Outputs”

Conformance with this Best Practice shall be checked using all the relevant tests specified in Annex A (normative) of this document. The framework, concepts, and methodology for testing, and the criteria to be achieved to claim conformance are specified in the OGC Compliance Testing Policies and Procedures and the OGC Compliance Testing web site.

All requirements-classes and conformance-classes described in this document are owned by the documents(s) identified.

3

NORMATIVE REFERENCES

3 NORMATIVE REFERENCES

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Arliss Whiteside Jim Greenwood : OGC 06-121r9, *OGC Web Service Common Implementation Specification*. Open Geospatial Consortium (2010). https://portal.ogc.org/files/?artifact_id=38867

OGC API — Processes — Part 1: Core Standard, 2021. https://docs.ogc.org/is/18-062r2/18-062r2.html

Commonwl.org: Common Workflow Language Specifications, https://w3id.org/cwl/

Radiant Earth Foundation: SpatioTemporal Asset Catalog specification, https://stacspec.org

4

TERMS AND DEFINITIONS

4 TERMS AND DEFINITIONS

This document uses the terms defined in OGC Policy Directive 49, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

This document uses the terms defined in Sub-clause 5.3 of OGC 06-121r9, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this standard.

For the purposes of this document, the following additional terms and definitions apply.

4.1. Application

A self-contained set of operations to be performed, typically to achieve a desired data manipulation, written in a specific language (e.g. Python, R, Java, C++, C#, IDL).

4.2. Application Package

A platform independent and self-contained representation of an Application, providing executables, metadata and dependencies such that it can be deployed to and executed within an Exploitation Platform.

4.3. Compute Platform

The Platform providing the computational resources for the execution of the Application.

4.4. Container

A container is a standard unit of software that packages up code and all its dependencies so that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

4.5. Exploitation Platform

An on-line system made of products, services and tools for exploitation of data.

4.6. Spatiotemporal Asset

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

4.7. GeoTIFF

A public domain metadata standard that 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.

4.8. HDF5

The Hierarchical Data Format version 5 (HDF5), is an open source file format that supports large, complex, heterogeneous data. HDF5 uses a “file directory” like structure that allows you to organize data within the file in many different structured ways, as you might do with files on your computer

4.9. JPEG2000

An image compression standard and coding system

4.10. SAFE

SAFE, the Standard Archive Format for Europe, is designed to act as a standard format for archiving and conveying Earth observation data within the European Space Agency (ESA) archiving facilities and, potentially, within the archiving facilities of cooperating agencies.

4.11. Processing Result

The Products produced as output of a Processing Service execution.

4.12. Processing Service

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

4.13. Processing Software

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

4.14. Products

Earth Observation data (commercial and non-commercial) and value-added data. It is assumed that the Exploitation Platform provides the data access mechanisms for an existing supply of Earth Observation Products.

5

CONVENTIONS

5 CONVENTIONS

This section provides details and examples for any conventions used in the document. Examples of conventions are symbols, abbreviations, use of XML schema, or special notes regarding how to read the document.

5.1. Identifiers

The normative provisions in this document are denoted by the URI

`http://www.opengis.net/spec/eoap-bp/1.0`

All requirements and conformance tests that appear in this document are denoted by partial URIs which are relative to this base.

5.2. Abbreviated terms

- API Application Programming Interface
- COG Cloud Optimized GeoTIFF
- CWL Common Workflow Language
- EO Earth Observation
- EP Exploitation Platform
- 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
- SLC Single Look Complex

- SNAP Sentinel Application Platform toolbox
- STAC SpatioTemporal Asset Catalog
- TBD To Be Determined
- TIFF Tagged Image File Format
- URL Uniform Resource Locator
- YAML YAML Ain’t Markup Language

6

COMPONENTS OVERVIEW

6 COMPONENTS OVERVIEW

6.1. Introduction

In recent years, Platforms for the Exploitation of Earth Observation (EO) 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 EO data in a growing number of fields in our society.

An EO exploitation platform is a collaborative virtual work environment providing the mechanisms to deliver EO data and the tools, processors, and ICT resources required to work with them, through one coherent set of interfaces. A fundamental principle of the platform operations concept is to move the EO data processing service's user to the data and tools, as opposed to downloading, replicating, and exploiting data 'at home'. Furthermore, platforms offer an environment which takes care of all data processing orchestration tasks and the availability of scalable computational resources offered by the Cloud shortens the time to market of applications.

In this scope, OGC Testbeds 13-16 initiated the drafting of an architecture to allow the deployment and execution of externally developed applications on Earth Observation (EO) data and processing platforms. During the OGC Innovation Program initiative OGC Earth Observation Applications Pilot, conducted between December 2019 and July 2020 (OGC 20-042, OGC 20-073), the participants explored and assessed the existent draft specifications, which addressed both application description and discovery, APIs for deployment, execution, and result access, as well as specifications for service chaining and workflow building. This document summarizes their findings and defines a Best Practice to package and deploy Earth Observation Applications in an Exploitation Platform.

In summary, the architecture as defined by the OGC Earth Observation Applications Pilot targets the following requirements:

- Decouple application developers from exploitation platform operators and from application consumers
- Allow application developers to make their applications available on any number of platforms with minimal modifications
- Allow application developers to focus on application development by minimizing platform specific particularities
- Enable exploitation platforms to virtually support any type of packaged EO application

The figure below provides a high-level view of the interactions between the actors and the architecture.

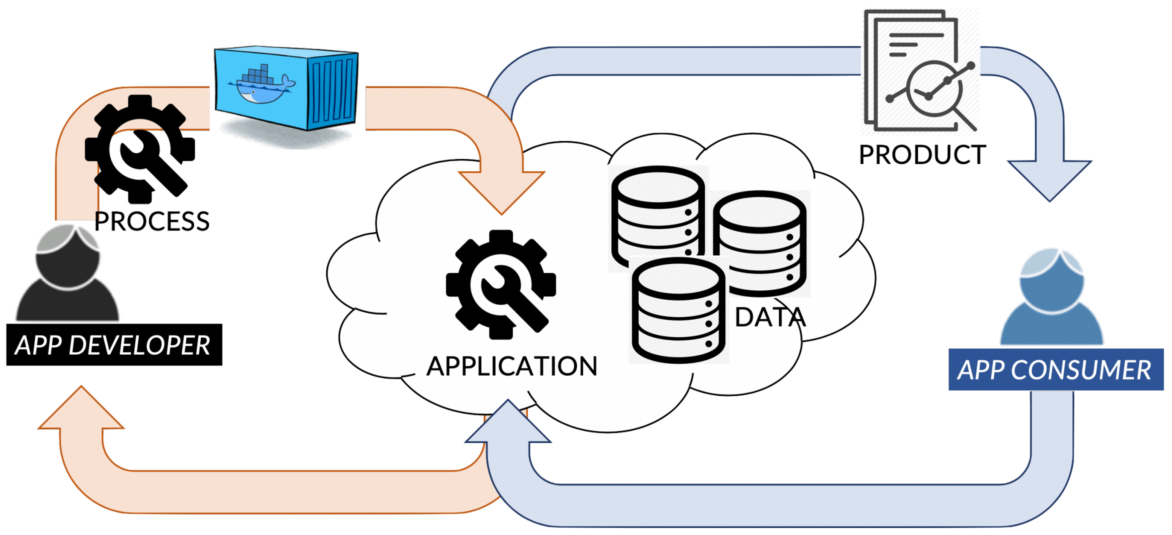


Figure 1 — Application developers and application consumers interacting with the cloud platform

Application developers on the left side develop their application and make it available in the form of an Application Package referencing one or more container images. The application developers register their application in a platform making it available to users.

Application consumers can discover applications available in the platform, request their execution with a given set of input products and parameters and obtain the resulting products in their platform user space.

6.2. Earth Observation Applications

Earth Observation Applications typically offer functions that perform data operations like processing / reprocessing, projection, visualization or analysis. The applications can be written in a variety of coding languages (e.g. Python, R, Java, C++, C#, shell scripts) and make use of specific software libraries (e.g. SNAP, GDAL, Orfeo Toolbox).

In the context of this Best Practice, the application is treated as a black-box that according to its application design pattern must comply with data stage-in and data stage-out mechanisms defined. Two main design patterns are identified: fan-in and fan-out. An application can combine these patterns in the nodes of a Directed Acyclic Graph (DAG).

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

The data driven application fan-in pattern refers to the execution of a data processing function that aggregates several input products.

The platform application accesses a list of input products, stages the input products making them available to the application execution block.

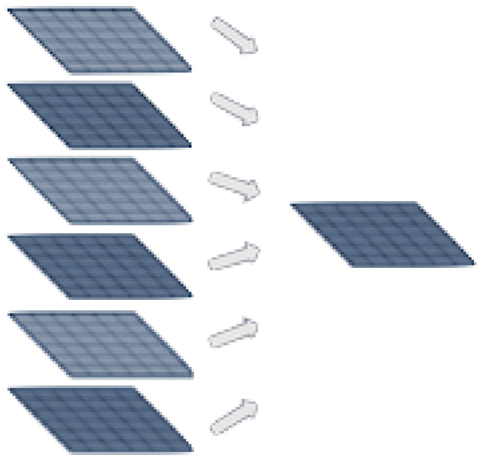


Figure 2 — Data-driven application with fan-in input references where an application processes the aggregates of n-input EO products

An application following this pattern must take in consideration that it will be invoked once with all the input products and is expected to create one output (but not necessarily a single file).

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

The data driven application fan-out pattern 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, stages each of the individual products making it available to the application execution block. The platform can apply different strategies to parallelize the execution of each individual product.

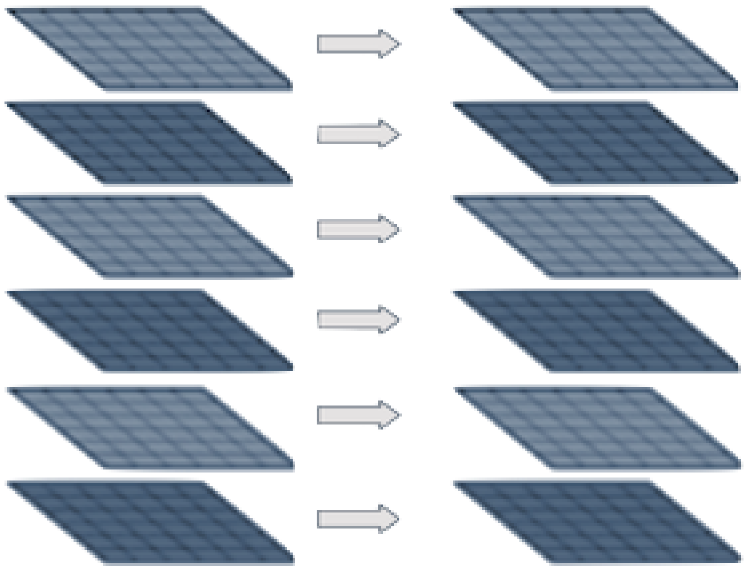


Figure 3 — Data-driven application with fan-out input references where an application processes several input EO products independently.

6.2.3. Staging Input and Output EO Products

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 the 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 the next figure.

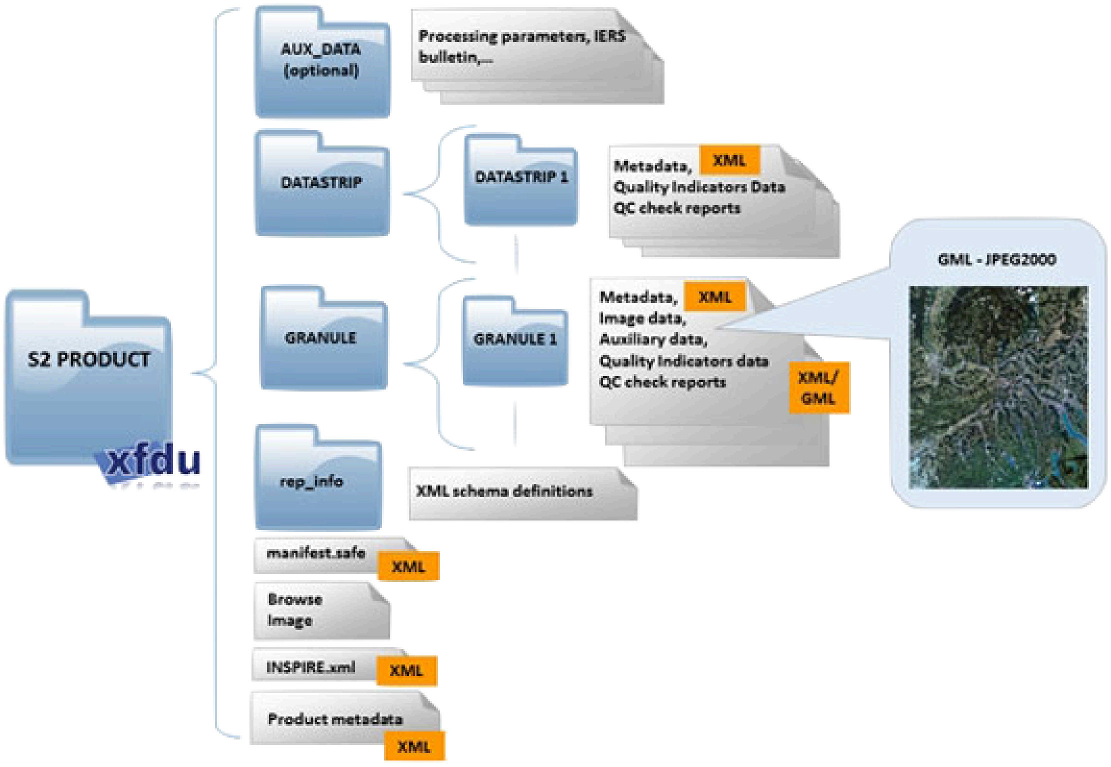


Figure 4 — SENTINEL-2 product physical format

A main concern application developers face is the different approaches through which the products are made available (i.e. stage-in) to the applications. For example, applications might find the same exact folder structure and return the folder root or the main XML manifest file or have the folder structure compressed in a single archive file.

In general, the onus of navigating the input folder directory and programmatically reacting to how the file was staged-in by the platform is on application and 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 places the resulting files in an output directory. The only information the platform might receive about the output files is the file media type (formerly known as "MIME-type") and is often missing critical information like spatial footprint, sub-items (e.g. masks, bands) and additional metadata (e.g. ground sample distance, orbit direction).

A good solution to represent the data manifest for input and output products is brought by the SpatioTemporal Asset Catalog (STAC).

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).

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, most significantly, it allows relative paths.

Most importantly the STAC specification can be implemented in a completely 'static' manner as flat local files located near the data enabling the application to access products assets (e.g. JPEG 2000 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).

This Best Practice selected a STAC Catalog with STAC Item files as the data manifests format, for application that require staging input data and/or output results.

6.3. Application Package

The Application Package is a document that describes the data processing application by providing information about the parameters, software item, executable, dependencies and metadata. This file document ensures that the application is fully portable among all supporting processing scenarios and supports automatic deployment in a Machine — To — Machine (M2M) scenario. Most importantly, the Application Package information model allows the deployment of the application as an OGC API — Processes (OGC 18-062) compliant web service.

The Application Package includes the following information:

- Reference to the executable block that implements the Application functionality
- Description of its input/output interface

The Application Package uses the Common Workflow Language (CWL) Workflow Description specification as an encoding to describe the Application, its parameters, command-line tools, their runtime environments, their arguments and their invocation within containers.

6.3.1. Common Workflow Language (CWL)

The CWL is a set of open standards 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.

The 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 or “DAG”)
- Command line tool(s)
- Parameter of the process
- Inputs/outputs

The CWL contains two main specifications. The Command Line Tool Description Specification that specifies the document schema and execution semantics for wrapping and executing command line tools and the Workflow Description Specification that specifies the document schema and execution semantics for composing workflows from components such as command line tools and other workflows. The CWL file is able to reference the application container images and also allow the definitions of the Application parameters, input/output interface and the overall process offering parameters.

Each input to a command line tool has a name and a type (e.g., File, string) and developers are encouraged to include documentation and labels for all components. Metadata about the command line tool descriptions can contain well-defined hints or mandatory requirements such as which software container to use or how much compute resources are required (memory, number of CPU cores, disk space, and/or the maximum time or deadline to complete the step or entire workflow.)

The CWL execution model is explicit: each command line tool’s runtime environment is explicit and any required elements must be specified by the CWL tool-description author (in contrast to hints, which are optional). Each tool invocation uses a separate working directory, populated according to the CWL tool description, e.g., with the input files explicitly specified by the workflow author. Some applications can expect particular filenames, directory layouts, and environment variables, and there are additional constructs in the CWL Command Line Tool standard to satisfy these needs.

The CWL standards use a declarative syntax, facilitating polylingual workflow tasks. By being explicit about the run-time environment and any use of software containers, the CWL standards enable portability and reuse while also providing a separation of concerns between workflow authors and workflow platforms.

The execution block (i.e. Application Artefact) describes the ‘software’ component that represents the execution unit in a specific container image 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 image can be parameterized or tailored.

A container image is an immutable, static file containing the dependencies for the creation of a container. These dependencies may include a single executable binary file, system libraries, system tools, environment variables, and other required platform settings (Cloud Native Glossary).

In overall, a container image describes a container environment whereas a container is an instance of that environment, ran by a container engine (e.g. Docker Engine). It is possible to run multiple containers from the same image, and all of them will contain the same software and configuration, as specified in the image.

In the scope of this Best Practice, the Application Package uses the Common Workflow Language (CWL) Workflow Description specification as encoding to describe the Application, its parameters, the command-line tools used, their arguments and their invocation within containers.

With the use of CWL Workflow Description Standard as encoding, the Application can also possibly yield several Application Packages that expose parameters and inputs in different flavors and execution patterns.

There are multiple community or commercially supported systems that support the CWL standards for executing workflows and a list of free and open-source implementations of the CWL standards are listed in Annex B.

6.3.2. Usage Scenarios

The application package provides a well-defined set of procedures to allow “build to run” operations. It covers five different usage scenarios from application testing, validation, and deployment to execution in production that enables:

- Alice to package an application
- Bob to script the execution of application
- Eric to deploy an application on platform Z
- Platform Z to accept the deployment of a new process
- Platform Z to execute a process with specific parameters

The scenarios cover the three “build to run” operations: Build, Deploy and Run.

Build

Alice builds a container image with her Application and command line tool(s) and respective runtime environments, publishes the container image on a repository and writes the Application Package document with a workflow that invokes the command line tool(s) included in the image.

For Alice, the Application Package is a portable and executable document that:

- References the container image
- Describes the input parameters of the Application
- Maps the input parameters to the command line tool(s) arguments

Bob wants to run Alice's application and scripts the application with different input parameters in his local machine. He uses two tools to script the execution: a container engine (e.g. Docker) and a CWL runner. The container image used is a controlled execution environment and it's the same used by Alice to build and test the application.

For Bob, the Application Package is a portable and executable document that:

- Is used by the CWL runner to mount the volumes for the inputs and outputs for the container run
- Is used by the CWL runner to invoke the CLI in a specified way with parameters passed as YAML.
- Isolates the process with no environments configured on Bob's machine

Deploy

Eric wants to deploy Alice's Application Package as a new Processing Service in Platform Z where he has authorization rights.

He uses the Application Package document to create an OGC API — Processes Transaction Extension request to dynamically add a process to a deployed OGC API — Processes server instance.

For Eric, the Application Package is a portable and executable document that:

- Maps the application input parameters to OGC API — Processes Input Parameters
- Identifies the container image to be deployed and corresponding execution unit
- Registers an application in a platform as a process within an OGC API — Processes server instance

Platform Z receives an OGC API — Processes deployment from Eric. The platform uses the Application Package CWL to create a new process in the OGC API — Processes server instance. For the Platform Z, the Application Package is a document that:

- Defines the process metadata (including the Input Parameters)
- Identifies the container image to be deployed and corresponding execution unit
- Creates a new process in the OGC API — Processes instance

Run

Platform Z receives an OGC API — Processes execution request for Eric's deployed process.

The platform uses the Application Package CWL to retrieve the specified container image, create the container, map the instantiated parameters with the execution values and execute the application.

For the Platform Z, the Application Package is a portable and executable document that:

- Describes the application metadata of the process
- Maps the OGC API — Processes input parameters to the application input parameters
- Identifies the container image to be deployed, the corresponding execution unit, to monitor the execution and retrieve the results

6.4. Platform

An Earth Observation Exploitation Platform provides interfaces, processing functions, tools and processing services invoked individually or utilized in workflows. Developers are able to test and execute their own applications, register them into the platform and make them available for exploitation by other users also individually or in their own workflows.

Connected from Client Portals, users of Earth Observation applications are able to find and exploit on the Platform the services matching their needs, out of a large offer gathered from multiple contributors.

Interfaced with Cloud Computing providers, Earth Observation Exploitation Platform executes the data processing tasks requested by users and retrieves the information produced for delivery back to the processing requester.

To support the application integration activities, a platform might also provide developers with an environment where they can integrate, build, test & debug their applications as part of their business use case. The ultimate goal of such an environment is to produce an Application Package. However, these steps are outside the scope of this Best Practice that focuses on the platform as the environment where the data processing application is registered and executed.

6.4.1. Platform Architecture

This best practice focuses on the scenario where an application is directly packaged as an Application Package, registered in a Platform and made available as an implementation of OGC API — Processes. The Web 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.

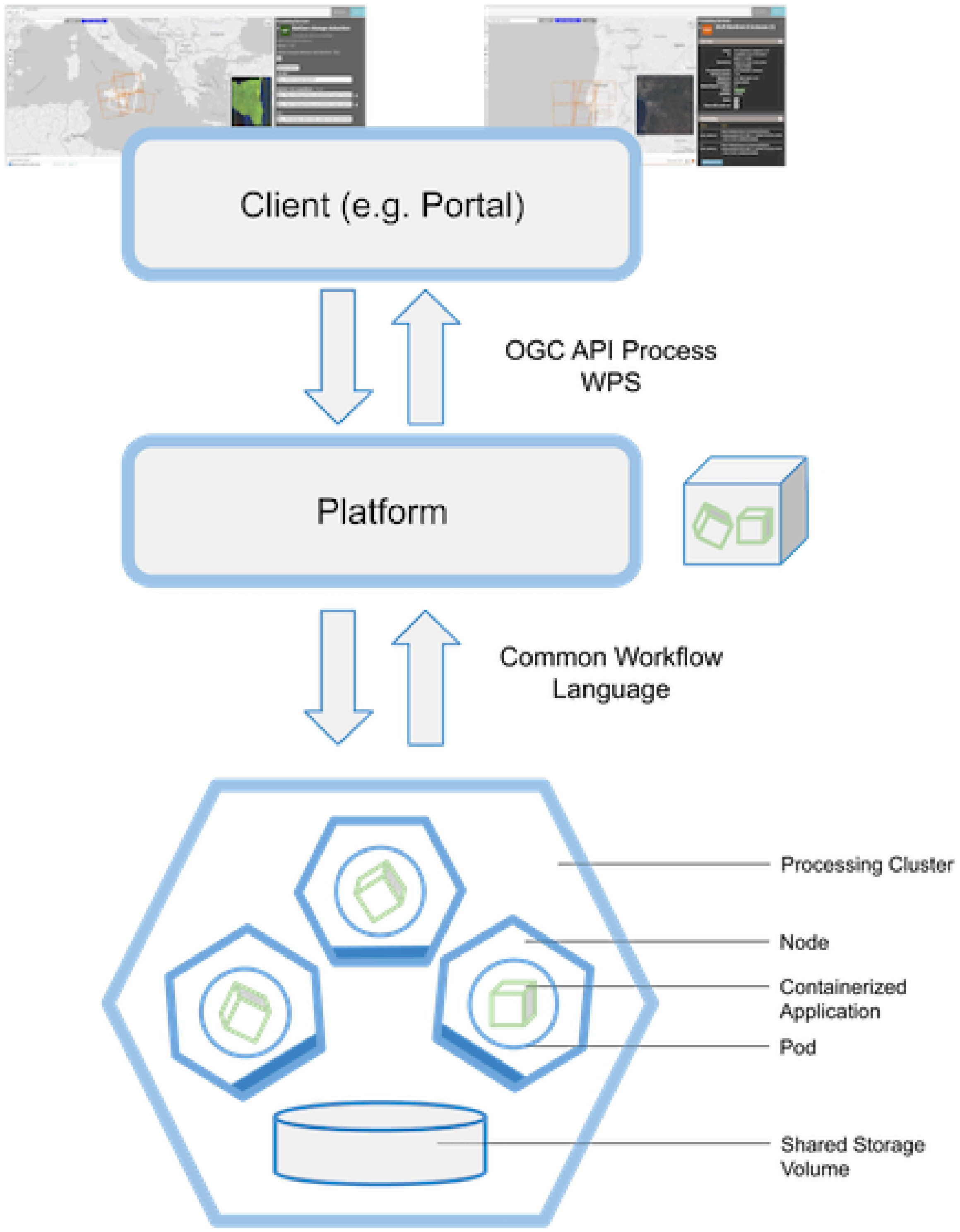


Figure 5 — Architecture overview of the Application execution in a Platform

In the context of this Best Practice, the main responsibilities of the Platform are to:

- Validate and accept an application deployment request (OGC API — Processes Transaction Extension)
- Validate and accept an execution request (OGC API — Processes)
- Submit the process execution to the processing cluster (CWL)
- Monitor the process execution (OGC API — Processes)
- Retrieve the processing results (OGC API — Processes)

For applications that require staged EO products and/or generate products that need to be staged-out the platform is responsible for the EO Product data flow management operations for:

- Data stage-in of the process input EO Product.
- Data stage-out of the process outputs.

6.4.2. EO Products Data Flow Management

This Best Practice addresses data flow management of the input and output EO Products files by defining rules for the data stage-in and data stage-out for Applications that require staged files and/or generate files that need to be staged-out.

Data stage-in is the process to retrieve the inputs and make these available for the processing. Processing inputs are provided as catalogue references and the Platform is responsible for translating those references into inputs available as files for the local processing.

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

For the data stage-in, the Platform creates a local STAC Catalog with a STAC Item whose Assets have an accessible href (either local or remote e.g. COG) as the input files manifest for the application.

For the data stage-out, the Application creates a local STAC Catalog as the output files manifest describing the results metadata and assets' location thus enabling the Platform to provide the processing results in the OGC API — Processes response.

7

APPLICATION BEST PRACTICE

7 APPLICATION BEST PRACTICE

The following section describes the Application Best Practice.

7.1. Overview

An Application that complies with the Best Practice for Earth Observation Application Package needs to be:

- Executable as a command-line tool.
- Delivered in a container image with all the necessary software, libraries and configuration files.

This section described the best practice for the command line tools, how to consider input data, output data and how to create a container.

7.2. Command Line

The Application is executed as a command-line interface (CLI) tool that runs as a non-interactive executable program: it receives input arguments, performs a computation, and terminates after producing some output.

The Application can have any number of command-line arguments.

When executed, the Application working directory is also the Application output directory. Any file created by the Application should be added under that directory.

```
#!/bin/bash

if [ "$#" -lt 3 ]
then
  echo "Usage: provide file, bbox and proj"
  exit 1
fi

# file to process
file=$1
# bbox processing argument
bbox=$2
# EPSG code used to express bbox coordinates
proj=$3

gdal_translate \
      -projwin \
      "$( echo $bbox | cut -d ',' -f 1)" \
      "$( echo $bbox | cut -d ',' -f 4)" \
      "$( echo $bbox | cut -d ',' -f 3)" \
      "$( echo $bbox | cut -d ',' -f 2)" \
      -projwin_srs \
      ${proj} \
      ${file} \
      cropped.tif
```

If the EO Products are already staged on the local computer this command-line could be executed with the parameters.

```
computer:~ user$ crop "S2B_53HPA_20210723_0_L2A/B02.tif" "136.522,-36.062,137.
027,-35.693" "EPSG:4326"
```

7.3. Container

The environment, libraries, binaries and configuration files necessary to execute the command-line tools need to be bundled in a container image.

The example below shows how Docker, one of the available container engine solutions to deliver software in containers, defines all the necessary commands to assemble an image.

```
FROM osgeo/gdal

RUN apt update && \
    apt-get install -y jq

ADD functions.sh /functions.sh

ADD crop /usr/bin/crop

RUN chmod +x /usr/bin/crop
```

Build the docker image with:

```
docker build -t crop_docker:0.1 .
```

Test the CLI with:

```
docker run --rm -it crop_docker:0.1 crop
```

7.3.1. EO Products as Input Data

To support Applications that required the input EO Products to be previously staged-in, this Best Practice recommends the usage of a STAC Catalog with STAC Item files as the format of the data manifest.

The command-line tool must have an argument that represents the path to the folder where the STAC Catalog file is located. The input products are defined by the STAC Catalog with one or more STAC Items (and associated STAC Assets) as input files for processing.

The Application is a wrapper command-line tool that reads the STAC Catalog, selects the input Item Assets href and executes another command-line tool taking as argument the asset href (i.e. path to local file).

```
#!/bin/bash

if [ "$#" -lt 3 ]
then
  echo "Usage: provide file, bbox and proj"
  exit 1
fi

## generic STAC functions
source /functions.sh

## processing arguments
in_dir=$1 # folder where the EO product is staged-in
bbox=$2 # bbox processing argument
proj=$3 # EPSG code used to express bbox coordinates

## Read the input STAC Catalog
catalog="${in_dir}/catalog.json"

# get the item path
item=$( get_items ${catalog} )

# get the B02 asset href (local path)
asset_href=$( get_asset ${item} B02 )

gdal_translate \
      -projwin \
      "$( echo $bbox | cut -d ',' -f 1)" \
      "$( echo $bbox | cut -d ',' -f 4)" \
      "$( echo $bbox | cut -d ',' -f 3)" \
      "$( echo $bbox | cut -d ',' -f 2)" \
      -projwin_srs \
      ${proj} \
      ${asset_href} \
      cropped.tif
```

The command-line tool reads the input EO product from the assets of the items included in the STAC Catalog file (catalog.json) in the specified directory.

The STAC items can be selected by the tool according to their respective metadata (e.g. bands, format, time).

Inside each STAC Item feature there are the corresponding STAC Assets for the product files (e.g. bands). The STAC Asset contains a link to the file associated with the STAC Item that can be downloaded or streamed (e.g. data, metadata, thumbnails) and can contain additional metadata.

```
{
  "id": "catalog",
  "stac_version": "1.0.0",
  "links": [
    {
      "type": "application/geo+json",
      "rel": "item",
      "href": "S2B_53HPA_20210723_0_L2A/S2B_53HPA_20210723_0_L2A.json"
    }
  ],
  "type": "Catalog",
  "description": "Root catalog"
}
```

```
{
  "stac_version": "1.0.0",
  "stac_extensions": [ "eo", "proj", "view"],
  "type": "Feature",
  "id": "S2B_53HPA_20210723_0_L2A",
  "geometry": {
    "type": "Polygon",
    "coordinates": [ [
        [ 136.11273785955868, -36.22788818051635],[ 136.09905192261127,
 -35.238096451039816],[ 137.30513468251897, -35.22113204961173],[ 137.
33381497932513, -36.21029815477051], [ 136.11273785955868, -36.22788818051635]
      ] ]
  },
  "properties": {
    "datetime": "2021-07-23T00:57:07Z",
    "platform": "sentinel-2b",
    "constellation": "sentinel-2",
    ...
  },
  "bbox": [ 136.09905192261127, -36.22788818051635, 137.33381497932513, -35.
22113204961173],
  "assets": {
    ...
    "B02": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Band 2 (blue)",
      "href": "B02.tif",
      "gsd": 10,
      "eo:bands": [
        {
          "name": "B02",
          "common_name": "blue",
          "center_wavelength": 0.4966,
          "full_width_half_max": 0.098
          }
      ],
      "proj:shape": [ 10980, 10980 ],
      "proj:transform": [ 10,  0, 600000, 0, -10, 6100000, 0, 0, 1 ],
      "file:size": 206117177
    },
    ...
  },
  "links": [
    {
      "type": "application/json",
      "rel": "canonical",
      "href": "https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-
l2a-cogs/53/H/PA/2021/7/S2B_53HPA_20210723_0_L2A/S2B_53HPA_20210723_0_L2A.json"
    },
    {
      "rel": "parent",
      "href": "../catalog.json"
    }
  ]
}
```

Below is another example of a shell script that retrieves the location of the “B02” band of the first STAC Item and executes the gdal_translate command line application.

```
# parse the catalog.json file and get the first STAC item
item="$( jq -r '.links | select(.. | .rel? == \"item\")[0].href' catalog.json)"

# get asset B02 href
asset_href=$( dirname ${item} )/$( cat ${item} | jq -r ".assets.B02.href" )

gdal_translate ${asset_href} output.png
```

7.4. EO Products as Output Data

An Application that creates EO product files that need to be staged-out must also create in the output directory a STAC Catalog that enumerates and documents the produced files.

Below is a *catalog.json* produced by the Application with the one result referenced as a STAC Item.

```
{
  "id": "catalog",
  "stac_version": "1.0.0",
  "type": "catalog",
  "description": "Result catalog",
  "links": [
    {
      "type": "application/geo+json",
      "rel": "item",
      "href": "result-item.json"
    }
  ]
}
```

The STAC Item files contain the corresponding STAC Assets with the results of the processing. Each STAC Asset contains a reference to the associated data (e.g. data, metadata, thumbnails).

```
{
  "id": "item_id",
  "stac_version": "1.0.0",
  "type": "Feature",
  "bbox": [ 136.522, -36.062, 137.027, -35.693
  ],
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [ 136.522, -36.062],
        [ 137.027, -36.062],
        [ 137.027, 137.027],
        [ 136.522, 137.027],
        [ 136.522, -36.062]
      ]
    ]
  },
  "properties": {
    "datetime": "2021-07-23T00:57:07Z",
    "gsd": 10
  },
  "assets": {
    "B02": {
      "role": [ "data"],
      "href": "./cropped.tif",
      "type": "image/tiff",
      "title": "Cropped B02 band"
    }
  }
}
```

The STAC Catalog created by the Application must include metadata elements defining properties as the start time & end time or the geographical footprint. These two elements are the minimum set of metadata elements to enable their discovery but depending on the context, the application can convey further elements.

It is assumed that all output files not referenced in the STAC Catalog local file are not relevant to the process and can be discarded by any subsequent action and thus not staged out.

Please note that, as with the input data, nothing hampers the Application to be a wrapper command-line tool that, after executing a command-line, creates the STAC Catalog referencing the output files.

The example below complements the previous examples with the creation of the STAC Catalog for the data output manifest.

```
#!/bin/bash

if [ "$#" -lt 3 ]
then
  echo "Usage: provide file, bbox and proj"
  exit 1
fi

## generic STAC functions
source /functions.sh

## processing arguments
in_dir=$1 # folder where the EO product is staged-in
bbox=$2 # bbox processing argument
proj=$3 # EPSG code used to express bbox coordinates

## Read the input STAC Catalog
catalog="${in_dir}/catalog.json"

# get the item path
item=$( get_items ${catalog} )

# get the B02 asset href (local path)
asset_href=$( get_asset ${item} B02 )

gdal_translate \
      -projwin \
      "$( echo $bbox | cut -d ',' -f 1)" \
      "$( echo $bbox | cut -d ',' -f 4)" \
      "$( echo $bbox | cut -d ',' -f 3)" \
      "$( echo $bbox | cut -d ',' -f 2)" \
      -projwin_srs \
      ${proj} \
      ${asset_href} \
      cropped.tif

## result as STAC
# get the properties from the input STAC item
# as these are the same for the output STAC item
datetime=$( get_item_property ${item} "datetime" )
gsd=$( get_item_property ${item} "gsd" )

# initialise a STAC item
init_item ${datetime} "${bbox}" "${gsd}" > result-item.json

# add an asset
add_asset result-item.json "B02" "./cropped.tif" "image/tiff" "Cropped B02
 band"

# initialise the output catalog
init_catalog > catalog.json

# add the item to the catalog
add_item catalog.json result-item.json
```

7.5. Requirement Classes

7.5.1. Requirements Class “Application”

This class contains the requirements for any Application to comply with the Best Practice for Earth Observation Application Package.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/app	
Target Type	Application
Dependency	
Requirement 1	req/app/cmd-line The Application SHALL be a non-interactive executable as a command-line application.
Requirement 2	req/app/container The environment, libraries, binaries, executable and configuration files necessary to execute the Application SHALL be bundled in a container image.

Requirement 3	req/app/registry The Application container image SHALL be accessible in a container registry.

7.5.2. Requirements Class “Application Staged Inputs”

This class contains the requirements for an Application that requires staged files as input.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/app-stage-in	
Target Type	Application
Dependency	Requirements Class “Application” SpatioTemporal Asset Catalog

Requirement 4	req/app/stac-input An Application input argument that requires staged EO product files SHALL be defined as an argument that points to a folder where a STAC Catalog, named catalog.json, contains a list of one or more STAC Items and associated STAC Assets referencing the files.

7.5.3. Requirements Class “Application Staged Outputs”

This class contains the requirements for an Application that creates files that need to be staged-out.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/app-stage-out	
Target Type	Application
Dependency	Requirements Class “Application” + SpatioTemporal Asset Catalog

Requirement 5	req/app/stac-out

	An Application that creates EO product files to be stage-out SHALL generate a valid STAC Catalog, named catalog.json, and include the STAC Item(s) and corresponding STAC Assets pointing to the results of the processing.
	rec/app/stac-out-metadata
Requirement 6	The STAC Catalog created by the Application SHALL include metadata elements for each STAC Item with at least their spatial (geometry, box) and temporal (datetime) properties.

8

PACKAGE BEST PRACTICE

8 PACKAGE BEST PRACTICE

This section describes the Package Best Practice for EO Applications.

8.1. Overview

A Package that complies with the Best Practice for Earth Observation Application Package needs to:

- Be a valid CWL document with a single *Workflow* Class and at least one *CommandLineTool* Class
- Define the command-line and respective arguments and container for each *CommandLineTool*
- Define the Application parameters
- Define the Application Design Pattern
- Define the requirements for runtime environment

The *Workflow* class steps field orchestrates the execution of the application command line and retrieves all the outputs of the processing steps.

8.2. CWL Document

The CWL Document references the Application parameters with the class *Workflow* and the command lines tools and arguments with the *CommandLineTool* classes.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...

- class: CommandLineTool
  id: crop-cl
  ...
```

8.3. Command-Line Tool

As stated previously, the command-line tool is a non-interactive executable program that reads some input, performs a computation, and terminates after producing some output.

The *CommandLineTool* class defines the actual interface of the command-line tool and its arguments according to the CWL *CommandLineTool* standard.

The CWL explicitly supports the use of software container technologies, such as Docker or Singularity, to enable portability of the underlying analysis tools. The Application Package needs to explicitly provide for each command-line tool the container requirements defining the container image needed.

The field *DockerRequirement* indicates that the component should be run in a container, and specifies how to fetch or build the image.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...

- class: CommandLineTool
  id: crop-cl
  requirements:
    DockerRequirement:
      dockerPull: docker.io/terradue/crop_container
  baseCommand: crop
  arguments: []
  inputs:
  ...
  outputs:
  ...
```

The field *inputs* defines the list of input parameters of the command-line that control how to run the tool. Each parameter has an *id* for the name of parameter, and a *type* field describing what types of values are valid for that parameter (e.g. string, int, double, null, File, Directory, Any). Additionally, if there are command-line bindings not directly associated with input parameters (e.g. fixed values or environment run-time values), the field *arguments* can also be used.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...

- class: CommandLineTool
  id: crop-cl
  ...
  baseCommand: crop
  arguments: []
  inputs:
    ...
    bbox:
      type: string
      inputBinding:
        position: 2
    epsg:
      type: string
      inputBinding:
        position: 3
  outputs:
  ...
```

When the command-line is executed under CWL, the starting working directory is the designated output directory. The underlying tool or script records its results in the form of files created in the output directory.

All the outputs of the command line tool are retrieved at this level.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...

- class: CommandLineTool
  id: crop-cl
  ...
  outputs:
    cropped_tif:
      outputBinding:
        glob: .
      type: Directory
...
```

8.3.1. Staging Input and Output EO Products

A command-line tool that requires staged EO product files must have an input field of the type *Directory* that will convey the path to the folder. When mounting the environment, this path is used for the data stage-in and STAC Catalog file location.

A command-line tool that generates products that need to be staged-out it must have the results of the type *Directory* and collect all the files available.

The example below defines a *CommandLineTool* class called *crop-cl* that maps to a command line application called *crop*, that accepts three inputs: *product*, *bbox* and *epsg*, and that is available in a container named *docker.io/terradue/crop_container*.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...

- class: CommandLineTool
  id: crop-cl
  requirements:
    DockerRequirement:
      dockerPull: docker.io/terradue/crop_container
  baseCommand: crop
  arguments: []
  inputs:
    product:
      type: Directory
      inputBinding:
        position: 1
    band:
      type: string
      inputBinding:
        position: 2
    bbox:
      type: string
      inputBinding:
        position: 3
    epsg:
      type: string
      inputBinding:
        position: 4
  outputs:
    cropped_tif:
      outputBinding:
        glob: .
      type: Directory
...
```

8.4. Application

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

The *Workflow* class includes four basic blocks: identification, inputs, steps and outputs.

For the identification block, the *Workflow* class supports the definition of a unique identifier (*id*), a short human-readable title (*label*) and a long human-readable description (*doc*) of the Application.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  label: Sentinel-2 band crop
  doc: This application crops a band from a Copernicus Sentinel-2 product using
 GDAL
  ...

- class: CommandLineTool
  id: crop-cl
  ...
```

For the inputs, the *Workflow* class supports the definition of the input parameters of the process. Each input parameter has a corresponding identifier (the field's name), title (*label*), abstract (*doc*) and a type (*type*) that is mandatory according to the CWL Workflow specification.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...
  inputs:
    ...
    bbox:
      type: string
      label: bounding box
      doc: Area of interest expressed as a bounding box
    proj:
      type: string
      label: EPSG code
      doc: Projection EPSG code for the bounding box
      default: "EPSG:4326"
  steps:
  ...
  outputs:
  ...
  - class: CommandLineTool
  id: crop-cl
  ...
```

The workflow is managed by the *steps* field of the *Workflow* class that links the corresponding parameters with arguments of the command-line class defined in the previous section.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 band crop
  doc: This application crops a Sentinel-2 band
  id: s2-cropper
  inputs:
    product:
    ...
    band:
    ...
    bbox:
    ...
    proj:
    ...
  steps:
    node_crop:
      run: "#crop-cl"
      in:
        product: product
        band: band
        bbox: bbox
        epsg: proj
      out:
        - cropped_tif
  outputs:
  ...

- class: CommandLineTool
  id: crop-cl
  ...
  inputs:
    product:
       ...
    band:
       ...
    bbox:
       ...
    epsg:
       ...
  outputs:
```

```
  ...
```

The previous workflow can be visualized as shown in the next figure.

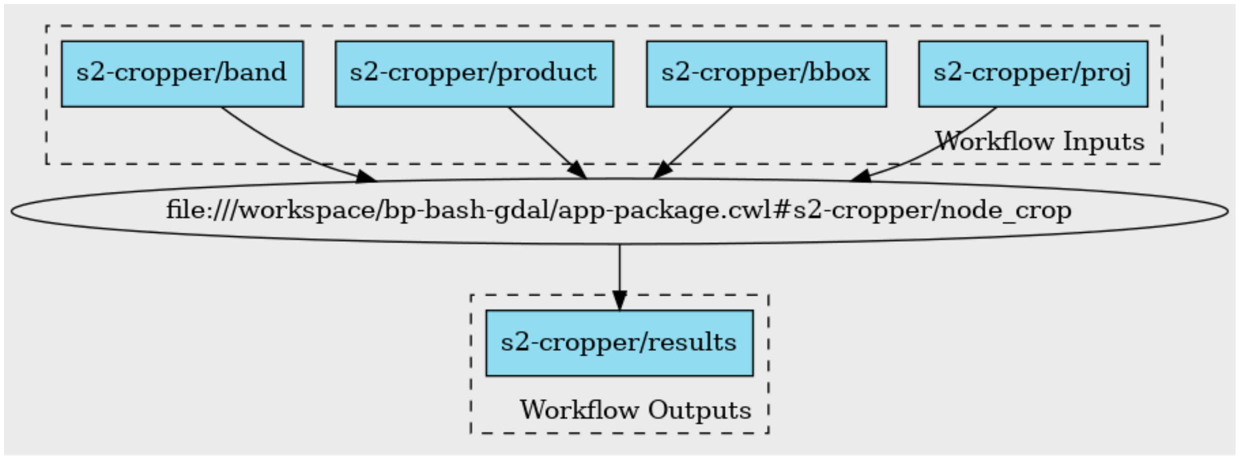


Figure 6 — Workflow diagram for the a simple Application with four inputs parameters and one execution block

For the outputs, the *Workflow* class includes the *outputs* section. This is a list of output fields where each field consists of an identifier and a data type.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...
  inputs:
  ...
  steps:
  ...
  outputs:
    results:
      outputSource:
      - node_crop/cropped_tif
      type: Directory

  - class: CommandLineTool
  id: crop-cl
  ...
```

8.4.1. Staging Input and Output EO Products

An Application that includes command-line tools that require staged EO products must have an *inputs* field of the type *Directory* that will convey the path where the data will be staged-in and STAC Catalog file located.

An Application that generates products that need to be staged-out it must have a field *outputs* of the type *Directory* and collect the available files and STAC Catalog.

```
cwlVersion: v1.0
$graph:
- class: Workflow
```

```
  id: s2-cropper
  label: Sentinel-2 band crop
  doc: This application crops a Sentinel-2 band
  inputs:
    product:
      type: Directory
      label: Sentinel-2 inputs
      doc: Sentinel-2 Level-1C or Level-2A input reference
   band:
      type: string
      label: Sentinel-2 band
      doc: Sentinel-2 band to crop (e.g. B02)
    bbox:
      ...
    proj:
      ...
  steps:
    node_crop:
      run: "#crop-cl"
      in:
        product: product
        band: band
        bbox: bbox
        epsg: proj
      out:
        - cropped_tif
  outputs:
    results:
      outputSource:
      - node_crop/cropped_tif
      type: Directory

- class: CommandLineTool
  id: crop-cl
  ...
```

Together with the cropped TIFF file, the application produces a STAC Catalog file with the list of items produced.

```
{
  "id": "catalog",
  "stac_version": "1.0.0",
  "type": "catalog",
  "description": "Result catalog",
  "links": [
    {
      "type": "application/geo+json",
      "rel": "item",
      "href": "result-item.json"
    }
  ]
}
```

The application also produces a STAC item describing the output product.

```
{
  "id": "item_id",
  "stac_version": "1.0.0",
  "type": "Feature",
  "bbox": [ 136.522, -36.062, 137.027, -35.693 ],
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [ [ 136.522, -36.062 ],
        [ 137.027, -36.062 ],
        [ 137.027, -35.693 ],
        [ 136.522, -35.693 ],
        [ 136.522, -36.062 ] ]
    ]
  },
  "properties": {
    "datetime": "2021-07-23T00:57:07Z",
    "gsd": 10
  },
  "assets": {
    "B02": {
      "role": [ "data" ],
      "href": "./cropped.tif",
      "type": "image/tiff",
      "title": "Cropped B02 band"
    }
  }
}
```

All the necessary files for this example (shell scripts, Docker container, Application Package and execution parameters) are included in Annex D.

8.5. Application Pattern

The fan-in application design pattern, shown in the previous examples, is the simplest case where all the EO inputs are used for the single processing. This pattern is the default behavior of a CWL workflow execution step.

For the fan-out application design pattern to be expressed in CWL, it is necessary to use the *ScatterFeatureRequirement* requirement. This requirement tells the CWL 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.

For the fan-out design pattern the *Workflow* class needs to change the type of input to an array (i.e. a *string* with square brackets *string[]*), and add a new requirement with the *ScatterFeatureRequirement* field to change the specific step as shown below where the input parameter *band* was changed to *bands* for the fan-out behavior.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 band crop
  doc: This application crops a Sentinel-2 band
  id: s2-cropper

  requirements:
  - class: ScatterFeatureRequirement

  inputs:
    ...
    bands:
      type: string[]
      label: Sentinel-2 bands
      doc: Sentinel-2 list of bands to crop
    ...
  outputs:
    results:
      outputSource:
      - node_crop/cropped_tif
      type: Directory[]

  steps:
    node_crop:
      run: "#crop-cl"
      in:
        product: product
        band: bands
        bbox: bbox
        epsg: proj
      out:
        - cropped_tif
      scatter: band
      scatterMethod: dotproduct

- class: CommandLineTool
  id: crop-cl
  ...
  inputs:
    ...
    band:
      type: string
      inputBinding:
        position: 2
    ...
  outputs:
    ...
...
```

In the definition above the fan-out pattern is applied to the *bands* workflow field is mapped to a *band* parameter at step level.

The field *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* field. 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.

The full Application Package file is included in Annex D.

8.6. Extended Workflows

The *Workflow* class can orchestrate one or more command-line applications in a directed acyclic graph describing a sequence of operations that transform input data sets to output.

The example below shows a new tool, called *composite* that creates a RGB image from three inputs for the red, green and blue channels.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  ...

- class: CommandLineTool
  id: composite-cl
  requirements:
    DockerRequirement:
      dockerPull: docker.io/terradue/composite-container
    InlineJavascriptRequirement: {}
  baseCommand: composite
  arguments:
  - $( inputs.tifs[0].path )
  - $( inputs.tifs[1].path )
  - $( inputs.tifs[2].path )
  inputs:
    tifs:
      type: File[]
    lineage:
      type: Directory
      inputBinding:
        position: 4
  outputs:
    rgb_composite:
      outputBinding:
        glob: .
      type: Directory
...
```

Adding this tool to a new workflow of two command-line tools it is possible to define an application that accepts a Sentinel-2 product, selects the bands, crops them and creates a composite. Below is the interface for the application with input product, the red, green and blue bands together with the bounding box and projection.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 RGB composite
  doc: This application generates a Sentinel-2 RGB composite over an area of
 interest
  id: s2-compositer
  inputs:
    product:
      type: Directory
      label: Sentinel-2 inputs
      doc: Sentinel-2 Level-1C or Level-2A input reference
    red:
      type: string
      label: red channel
      doc: Sentinel-2 band for red channel
    green:
      type: string
      label: green channel
      doc: Sentinel-2 band for green channel
    blue:
      type: string
      label: blue channel
      doc: Sentinel-2 band for blue channel
    bbox:
      type: string
      label: bounding box
      doc: Area of interest expressed as a bounding box
    proj:
      type: string
      label: EPSG code
      doc: Projection EPSG code for the bounding box coordinates
      default: "EPSG:4326"
  outputs:
    ...
- class: CommandLineTool
  id: crop-cl
  ...
- class: CommandLineTool
  id: composite-cl
  ...
...
```

The workflow orchestration is managed by the *steps* field where the *crop* tool extracts an area of a Sentinel-2 product then the *composite* tool that creates an image of the product with the selection of three bands for the red, green and blue channels.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-compositer
  ...
  inputs:
    ...
  outputs:
    results:
      outputSource:
      - node_composite/rgb_composite
      type: Directory
  steps:
    node_crop:
      run: "#crop-cl"
      in:
        product: product
        band: [red, green, blue]
        bbox: bbox
        epsg: proj
      out:
        - cropped_tif
      scatter: band
      scatterMethod: dotproduct
    node_composite:
      run: "#composite-cl"
      in:
        tifs:
          source:  node_crop/cropped_tif
        lineage: product
      out:
        - rgb_composite

- class: CommandLineTool
  id: crop-cl
  ...
  outputs:
    cropped_tif:
      outputBinding:
        glob: '*.tif'
      type: File

- class: CommandLineTool
  id: composite-cl
  ...
  outputs:
    rgb_composite:
      outputBinding:
        glob: .
      type: Directory

...
```

This workflow can be visualized as shown in the next figure.

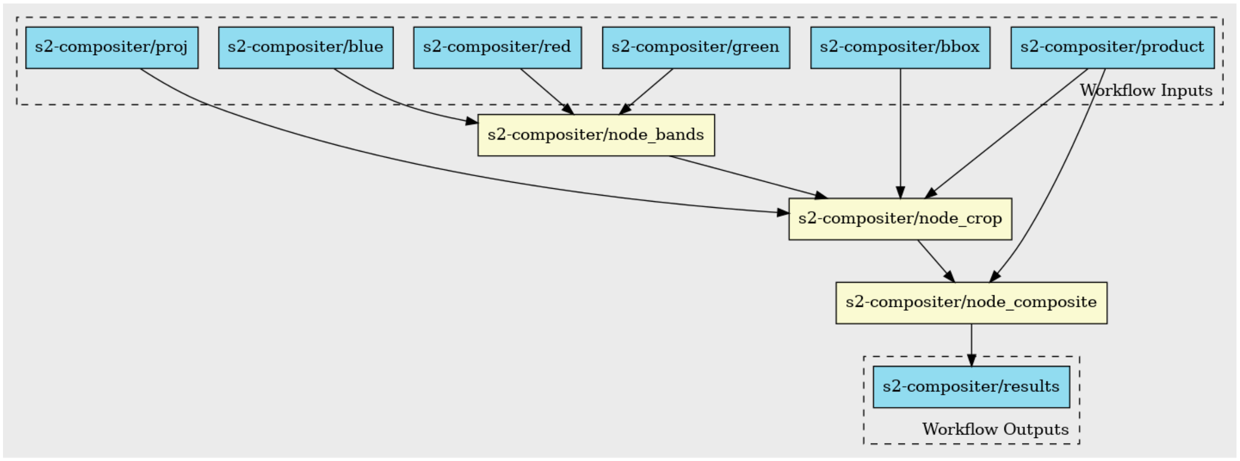


Figure 7 — Workflow diagram for the Application with six inputs parameters and two execution blocks

To execute the previous workflow with multiple input EO products it is only necessary to create a parent workflow that will replicate the input parameters and an array of input products. Together with the scatter requirements this workflow will process multiple composite images.

This workflow below takes a list of products as input and invokes a two-step sub-workflow that crops (using scatter over the bands) and creates a composite.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-composites
  label: Sentinel-2 RGB composites
  doc: This application generates a Sentinel-2 RGB composite over an area of
 interest with selected bands
  requirements:
  - class: SubworkflowFeatureRequirement
  - class: ScatterFeatureRequirement
  inputs:
    products:
      type: Directory[]
      label: Sentinel-2 inputs
      doc: Sentinel-2 Level-1C or Level-2A input references
    red:
      type: string
      label: red channel
      doc: Sentinel-2 band for red channel
    green:
      type: string
      label: green channel
      doc: Sentinel-2 band for green channel
    blue:
      type: string
      label: blue channel
      doc: Sentinel-2 band for blue channel
    bbox:
      type: string
      label: bounding box
      doc: Area of interest expressed as a bounding bbox
    proj:
      type: string
      label: EPSG code
      doc: Projection EPSG code for the bounding box coordinates
      default: "EPSG:4326"

  outputs:
    wf_results:
      outputSource:
      - node_rgb/results
      type: Directory[]

  steps:
    node_rgb:
      run: "#s2-compositer"
      in:
        product: products
        red: red
        green: green
        blue: blue
        bbox: bbox
        proj: proj
      out:
      - results
      scatter: product
      scatterMethod: dotproduct

- class: Workflow
  id: s2-compositer
  label: Sentinel-2 RGB composite
  doc: This sub-workflow generates a Sentinel-2 RGB composite over an area of
 interest

- class: CommandLineTool
  id: crop-cl
  ...
- class: CommandLineTool
```

```
  id: composite-cl
  ...
```

The full Application Package file is included in Annex D.

8.7. Application Additional Metadata

The Application Package can include additional metadata in CWL descriptions and developers should provide a minimal amount of authorship information to encourage correct citation.

It is recommended to include additional metadata in the Application Package using schema.org class *Person* to define the author and contributions and properties like *citation*, *codeRepository*, *dateCreated* and *license* as seen in the next example.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...
- class: CommandLineTool
  id: crop-cl
  ...
$namespaces:
  s: https://schema.org/
s:softwareVersion: 1.0.0
schemas:
- http://schema.org/version/9.0/schemaorg-current-http.rdf
```

It is recommended that concepts from schema.org are used whenever possible and linked with their RDF encoding. Table 1 lists the selected elements recommended by the Best Practice for Earth Observation Application Package.

Table 1 — Application Package additional Metadata elements

NAME	DESCRIPTION	ELEMENT	MANDATORY
author	The main author of the Application Package	https://schema.org/author	NO
citation	A citation or reference to a publication, web page, scholarly article, etc.	https://schema.org/citation	NO
codeRepository	Link to the repository where the Application code is located (e.g. SVN, github).	https://schema.org/codeRepository	NO
contributor	A secondary contributor to the Application Package	https://schema.org/contributor	NO

NAME	DESCRIPTION	ELEMENT	MANDATORY
dateCreated	The date on which the Application Package was created.	https://schema.org/dateCreated	NO
keywords	Keywords used to describe this application. Multiple entries in a keywords list are delimited by commas.	https://schema.org/keywords	NO
license	An URL to the license document that applies to this application.	https://schema.org/license	NO
releaseNotes	Description of what changed in this version.	https://schema.org/releaseNotes	NO
version	The version of the Application Package.	https://schema.org/version	YES

8.8. 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.

If appropriate the Application Package can define resources for the runtime environment with *ResourceRequirement* class either at the level of each *CommandLineTool* classes or at the level of the *Workflow* class (that will be inherited to all *CommandLineTool* classes)

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...
  requirements:
    ResourceRequirement:
      ramMin: 10240
      coresMin: 3

- class: CommandLineTool
  id: crop-cl
  ...
```

8.9. Requirement Classes

8.9.1. Requirements Class “Application Package”

This class contains the requirements for any Application Package to comply with the Best Practice for Earth Observation Application Package.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/app-pck/	
Target Type	Application Package
Dependency	Requirements Class “Application” Common Workflow Language

Requirement 7	req/app-pck/cwl The Application Package SHALL be a valid CWL document with a “Workflow” class and one or more “CommandLineTool” classes.
Requirement 8	req/app-pck/clt The Application Package CWL *CommandLineTool* classes SHALL contain the following elements: — Identifier (“id”)

	— Command line name (“baseCommand”) — Input parameters (“inputs”) — Environment requirements (“requirements”) — Docker information (“DockerRequirement”)
Requirement 9	req/app-pck/wf The Application Package CWL *Workflow* class SHALL contain the following elements: — Identifier (“id”) — Title (“label”) — Abstract (“doc”)
Requirement 10	req/app-pck/wf-inputs The Application Package CWL *Workflow* class “inputs” fields SHALL contain the following elements: — Identifier (“id”) — Title (“label”) — Abstract (“doc”)
Requirement 11	req/app-pck/metadata The Application Package CWL Workclass classes SHALL include additional metadata as defined in Table 1
Recommendation 1	rec/app-pck/fan-out For applications with the fan-out design pattern, the Application Package CWL Workclass class SHOULD include the “ScatterFeatureRequirement” class in the “requirements” section and include the “scatterMethod” in the corresponding input of the step.

8.9.2. Requirements Class “Application Package Staged Inputs”

This class contains the requirements of an Application Package when packaging an Application that requires staged files as input.

Requirements Class
http://www.opengis.net/spec/eoap-bp/1.0/req/app-pck-stage-in/

Target Type	Application Package
Dependency	Requirements Class “Application Package” Requirements Class “Application Staged Inputs”

Requirement 12	req/app-pck-stage-in/clt-stac All input parameters of the CWL ComandLineTool that require the staging of EO products SHALL be of type *Directory*.

Requirement 13	req/app-pck-stage-in/wf-stac Input parameters of the CWL Workflow that require the staging of EO products SHALL be of type *Directory*.

8.9.3. Requirements Class “Application Package Staged Outputs”

This class contains the requirements of an Application Package when packaging an Application that creates files that need to be staged-out.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/app-pck-stage-out/	
Target Type	Application Package
Dependency	Requirements Class “Application Package” Requirements Class “Application Staged Outputs”

Requirement 14	req/app-pck-stage-out/output-stac The outputs field of the CommandLineTool that requires the stage-out of EO products SHALL retrieve all the files produced in the working directory.

9

PLATFORM BEST PRACTICE

9 PLATFORM BEST PRACTICE

The following section describes the Platform Best Practice.

9.1. Overview

A Platform that complies with the Best Practice for Earth Observation Application Package needs to provide a mechanism to deploy the Application Package and a mechanism to execute the process defined by the Application Package (i.e. create a new job) with specific parameters.

For the deployment, the platform needs to:

- Accept a Post request with an Application Package (OGC API — Processes)
- Translate Application Package metadata to create a new process in the OGC API — Processes instance
- Translate Application Package Workflow Inputs defined in the CWL document as OGC API — Processes parameters
- Create a new Process offering in the OGC API — Processes

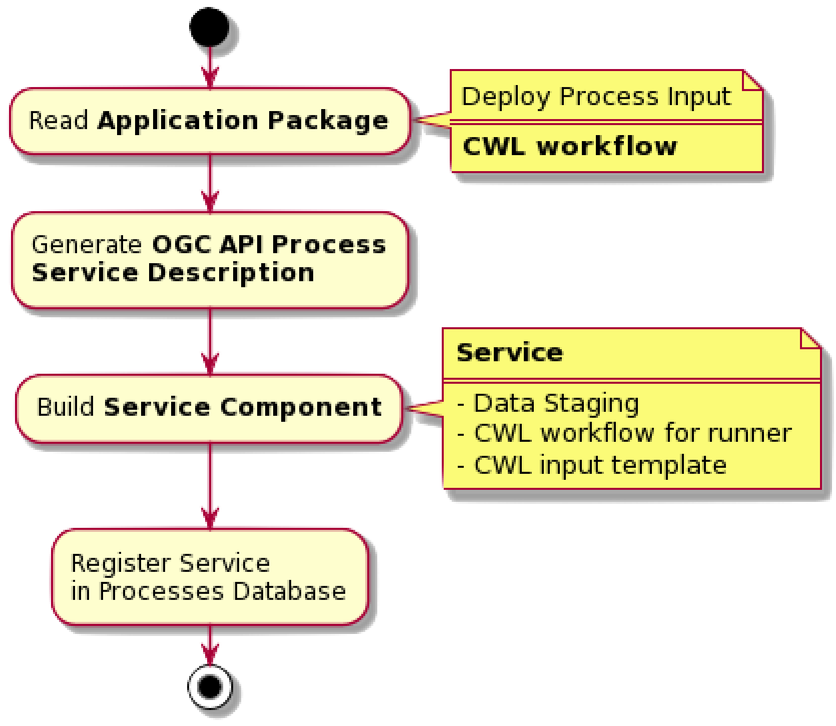


Figure 8 — Platform steps for the EO Application Package deployment

For the execution, the platform needs to:

- Translate OGC API — Processes execute parameters to the Workflow Inputs defined in the Application Package (CWL document)
- If applicable, execute the data stage-in for the input EO products
- Orchestrate and execute CWL
- Translate output to OGC API process outputs

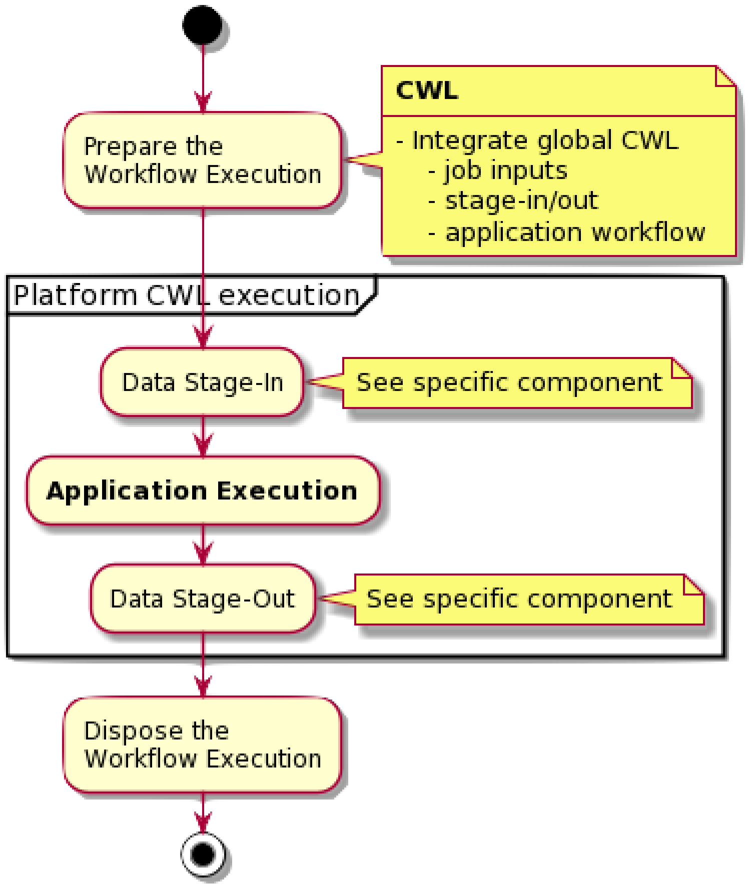


Figure 9 — Platform steps for the EO Application Package execution

A possible diagram of such a platform is shown in the next figure where the Platform presents an OGC API Processes interface for a CWL Conformant Executor that performs the service request in a processing cluster.

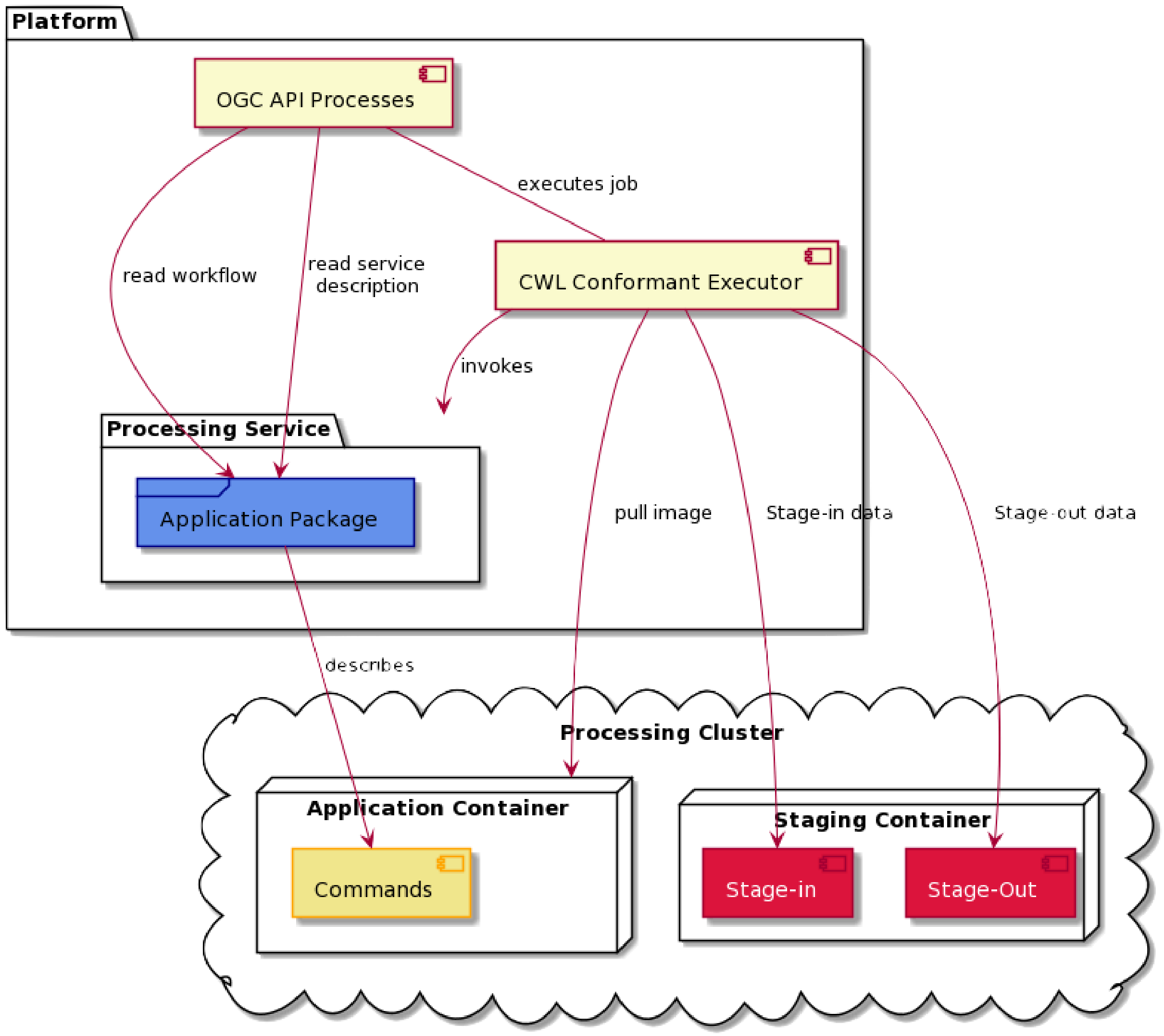


Figure 10 — High level diagram of Platform Components executing an EO Application Package

9.2. Process Description

The Application Package defined in the previous section provides the document that formally defines the inputs, outputs and other necessary metadata about a process that is to be deployed through the API.

For the Application Package, the *Workflow* class level is the interface used to map the parameters of the OGC API — Processes Web Service and respective mapping with the exposed service. The CWL *Workflow* class defines the overall processing service and includes two main sections: one for Service definition and the other for Service parameters.

As shown in the next example there is a main section for the Workflow class (extended with the schema.org classes) and two additional sections for the inputs and outputs. These fields are used to comply with the required information for an OGC process description.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  label: This application crops a Sentinel-2 band
  doc: This application crops a band from a Copernicus Sentinel-2 product using
 GDAL
  ...
  inputs:
    ...
  outputs:
  ...
s:version: 1.0
s:keywords: Sentinel-2, Copernicus

$namespaces:
 s: https://schema.org/
$schemas:
 - http://schema.org/version/latest/schemaorg-current-http.rdf
```

The Application Package properties defined in CWL are mapped according to the process description of the OGC API Processes as shown below.

```
{
  "id": "s2-cropper",
  "title": "This application crops a Sentinel-2 band",
  "description": "This application crops a band from a Copernicus Sentinel-2
 product using GDAL",
  "keywords": [ "Sentinel-2", "Copernicus" ],
  "inputs": [
    ...
  ],
  "outputs": [
    ...
  ],
  "version": 1.1,
  "jobControlOptions": [ "async-execute"],
  "outputTransmission": [ "reference" ],
  "links": [
    ..
  ]
}
```

The *Workflow* fields *id*, *label* and *doc* describe respectively the Process Description elements *id*, *title* and *description* as shown in Table 2.

Table 2 — Mapping the Workflow class fields to OGC API Processes

FIELD	DESCRIPTION	TYPE	REQUIRED	OGC API PROCESSES
id	The unique identifier for this object.	string	Required	id
label	A short, human-readable label of this object.	string	Optional	title

FIELD	DESCRIPTION	TYPE	REQUIRED	OGC API PROCESSES
doc	A documentation string for this object, or an array of strings that should be concatenated.	string	Optional	description
s:keywords	Keywords used to describe this Application. Multiple entries in a keywords list are delimited by commas.	string or array of strings	Optional	keywords
s:version	The version of the Application Package.	string	Required	version

9.3. Input Parameters

The input parameters of the application package are defined on the inputs section and are used to map the input parameters defined in the CWL to the OGC process description as used in the OGC API — Processes implementation. The process description language may use JSON Schema fragments to define the input and output parameters of a process.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: s2-cropper
  ...
  inputs:
    ...
    band:
      type: string
      label: Sentinel-2 band to crop
      doc: Sentinel-2 band to crop (e.g. B02)
  ...
```

The workflow call properties are mapped accordingly to the process description of the OGC API Processes as shown below.

```
{
  "id": "s2-cropper",
  ...
    "inputs": {
      ...
      "band": {
        "title": "Sentinel-2 band to crop ",
        "description": "Sentinel-2 band to crop (e.g. B02)"
        "schema": {
          "type": "string"
        }
      },
      ...
    },
  ...
  "outputs": {
    ...
  },
  ...
}
```

The following sections describe the rules for mapping the input parameters.

9.3.1. Single CWL type Parameter

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

Table 3 — Mapping CWL types to OGC API Process Input elements

CWL TYPE	DESCRIPTION	OGC API PROCESSES
null	no value	No mapping
boolean	A binary value	type=boolean
int	32-bit signed integer	type=integer
long	64-bit signed integer	type=integer
float	single precision (32-bit) IEEE 754 floating-point number	type=number
double	double precision (64-bit) IEEE 754 floating-point number	type=number
string	Unicode character sequence	type=string
enum	List of possible values	See Enumeration Parameters
File	File object	Complex data type (see Inputs by reference)
Directory	Directory object	Complex data type (see Input EO Products)

9.3.2. Optional 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 OGC API Processes with nullable equal to true.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: any-service
  ...
  inputs:
    ...
    my_parameter:
      label: Title of integer parameter
      doc: Abstract describing integer parameter
      type: int?
      default: 100
  ...
```

With the corresponding Process Description.

```
{
  ...
    "inputs": {
      ...
      "my_parameter": {
        "title": "Title of integer parameter  ",
        "description": "Abstract describing integer parameter"
        "schema": {
          "type": "integer"
          "nullable": "true"
        }
      },
      ...
    }
  ...
}
```

9.3.3. Array Parameters

There are two ways to specify an array parameter in CWL. First is to provide the “type” field with “type: array” and items defining the valid data types that may appear in the array.

Alternatively, square brackets [] may be added after the type name to indicate that the input parameter is an array of that type.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: any-service
  ...
  inputs:
    ...
    my_array_parameter:
      label: Title of integer array parameter
      doc: Abstract describing integer array parameter
      type: int[]
  ...
```

The array will be represented as shown below in the OGC Process Description.

```
{
  ...
    "inputs": {
      ...
      "my_array_parameter": {
      "title": "Title of integer array parameter",
      "description": "Abstract describing integer array parameter",
      "schema": {
        "type": "array",
        "minItems": 1,
        "maxItems": unbounded,
        "items": {
          "type": "integer"
          }
        }
      },
      ...
    }
  ...
}
```

9.3.4. Enumeration Parameters

The CWL enumeration type specifies value definitions into a LiteralData Type in the OGC API parameters.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: any-service
  ...
  inputs:
    ...
    string_with_options_parameter:
      label: Title of string_with_options_parameter
      doc: This parameter accepts a list of possible values
      type:
        type: enum
        Symbols: [‘option1’, ‘options2’, ‘option3’]
  ...
```

With the corresponding OGC Process Description.

```
{
  ...
    "inputs": {
      ...
      "string_with_options_parameter": {
        "title": "Title of string_with_options_parameter",
        "description": "This parameter accepts a list of possible values",
        "schema": {
          "type": "string",
            "enum": [
              "option1",
              "option2",
              "option3"
            ]
        }
      },
      ...
    }
  ...
}
```

9.3.5. Inputs by reference (File)

The workflow parameters of the type “File” correspond to files that need to be staged on a file system made available for the command line.

In the following example, the command line was extended with a new auxiliary file parameter. In this definition the file needs to be available for the application to execute.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  id: any-service
  ...
  inputs:
    ...
    aux_file:
      label: Auxiliary File
      doc: This is an auxiliary file needed for the processing
      type: File
      format: text/xml
  ...
```

The corresponding Process Description will contain the auxiliary file as an input.

```
 {
  ...
    "inputs": {
      ...
      "aux_file" : {
        "title": "Auxiliary File",
        "description": "This is an auxiliary file needed for the processing",
        "minOccurs": "1", "maxOccurs": "1",
        "schema": {
          "type" : "string",
          "contentMediaType" : "text/xml"
        }
      },
      ...
    }
  ...
 }
```

These parameters must be managed by the platform as physical files in the processing runtime environment.

At job submission, the inputs passed as references (as HTTP link, S3 link, etc.) must be fetched and made available for processing by executing the CWL document.

9.3.6. Input EO Products (Directory)

The input parameters of the CWL Workflow representing the EO products have the type “Directory” and must be mapped to a GeoJSON feature collection file with STAC Items.

```
{
  ...
  "inputs": {
    ...
    "input_reference" : {
      ...
      "formats": [
        {
          "mimeType": "application/json",
          "schema" : "https://raw.githubusercontent.com/stac-extensions/single-
file-stac/main/json-schema/schema.json"
          "default": true
        },
        {
          "mimeType": "application/geo+json",
          "schema" : "https://raw.githubusercontent.com/stac-extensions/single-
file-stac/main/json-schema/schema.json"
        }
      ]
    ...
    }
  ...
  }
}
```

The platform may accept alternative formats for the EO products (e.g. Atom feed) but in either case it must translate them to a STAC Catalog file with STAC Items when executing the CWL document.

The platform may adopt a strategy to download and stage-in the files defined in the STAC assets but will need to update their respective addresses.

9.4. Data Flow Management

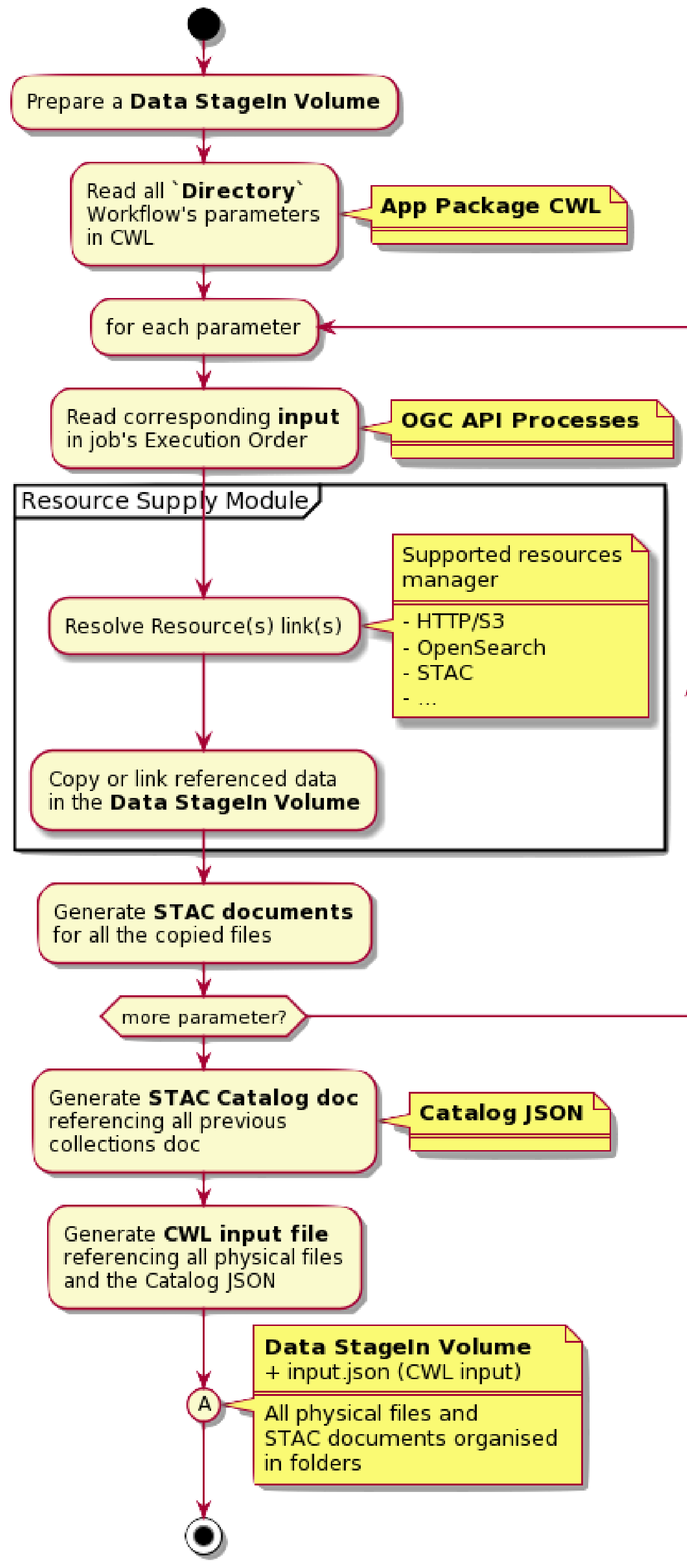
The Platform is responsible for the data flow management by using a local catalogue encoded using the SpatioTemporal Asset Catalog (STAC) specification as a data manifest for application inputs and outputs. The local catalogue provides knowledge about the input and output files data contents like spatial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

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).

9.4.1. Data Stage-In

The processing inputs are provided as EO Catalogue references and the Platform is responsible for translating those references into inputs available for the local processing.

For each File or Directory parameter, the platform resolves the resources, stages the data for processing. The full steps of the Platform for data stage-in are described in the Figure below.



Prepare a **Data StageIn Volume**
Read all `Directory` Workflow's parameters in CWL
App Package CWL
for each parameter
Read corresponding **input** in job's Execution Order
OGC API Processes
Resource Supply Module
Supported resources manager
- HTTP/S3
- OpenSearch
- STAC
- ...
Resolve Resource(s) link(s)
Copy or link referenced data in the **Data StageIn Volume**
Generate **STAC documents** for all the copied files
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 11 — Platform steps for data stage-in from the CWL service definition and OGC API Processes execution request

9.4.2. Data Stage-Out

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

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

The application execution provides a local STAC Catalog including items and assets describing the results generated. This local STAC Catalog is the application results manifest.

The Platform takes the STAC Catalog (catalog.json) file as the generated results entry point and uses the href links to items and assets and uses the found metadata and physical files to push them to a persistent storage and/or catalog and finally provide the OGC API Processes execution response.

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

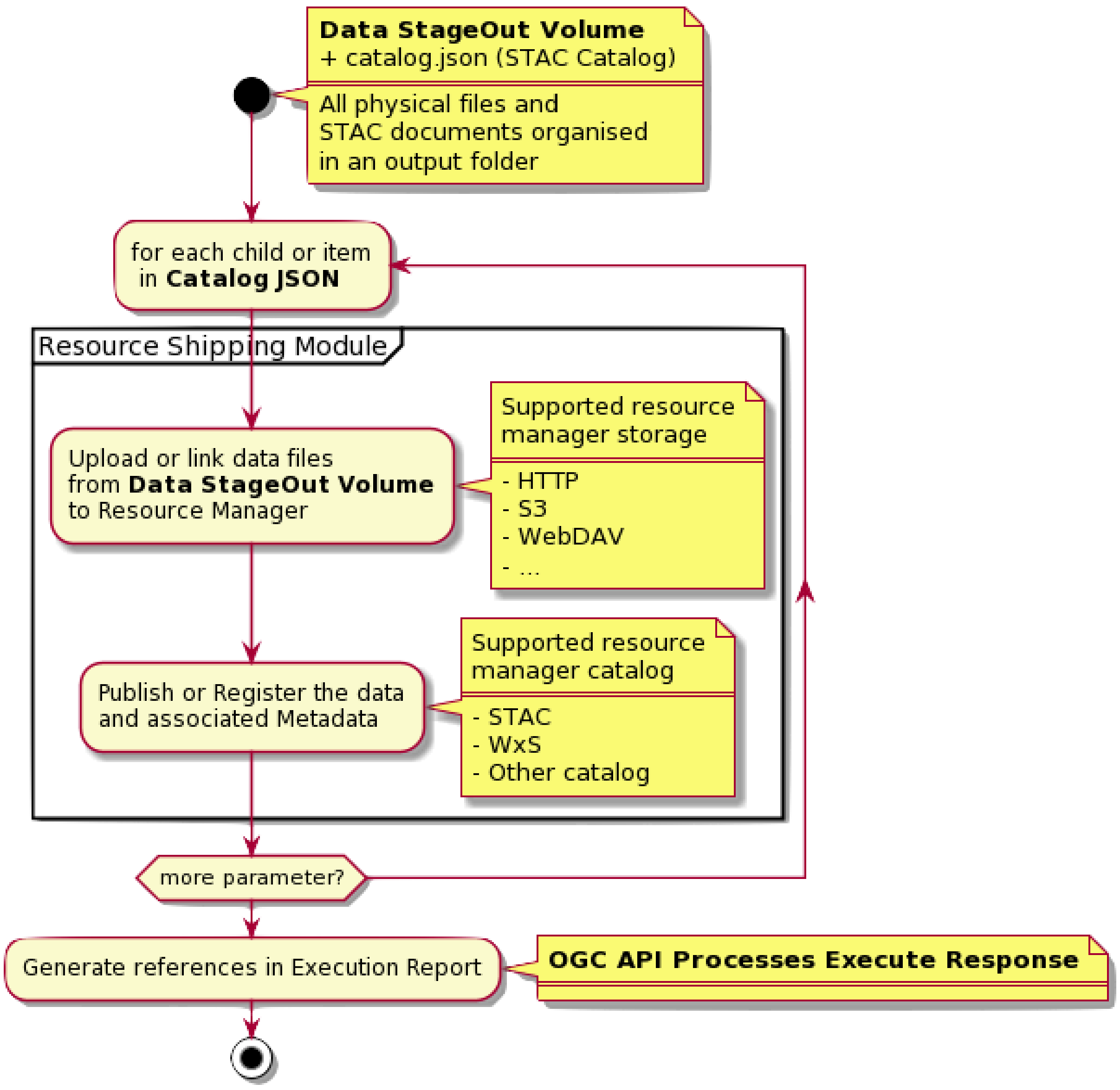


Figure 12 — Platform steps for data stage-out application execution for the the OGC API Processes response

9.5. Requirement Classes

9.5.1. Requirements Class “Platform”

This class contains the requirements for any Platform to comply with the Best Practice for Earth Observation Application Package when deploying an Application Package and executing the process defined by the Application Package with specific parameters as defined by the OGC API Processes Specification (OGC 18-062r2)

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/plt/	
Target Type	Application Package
Dependency	Requirements Class “Application Package” OGC API — Processes — Part 1: Core

Requirement 15	req/plt/api The Platform SHALL deploy the Application Package as a new process in a web service according to the OGC API Processes specification. In particular, the Platform SHALL map the top elements of the Application Package Workflow class to the OGC API Processes elements as defined in Table 2.
Requirement 16	req/plt/inputs The Platform SHALL map workflow input parameters types for the OGC API Processes input description types defined in Table 3. The Platform SHALL treat as optional all input parameters that the type has the suffix “?”. The Platform SHALL treat as array all input parameters that are defined as such in the CWL document (type array or have a double square brackets “[]” suffix). The Platform SHALL treat as enumerations all input parameters that are defined as such in the CWL document (type *enum*).
Requirement 17	req/plt/file The Platform MUST must map all inputs of type *File* as an input by reference and make them available as local files when executing the application.

9.5.2. Requirements Class “Platform Staged Inputs”

This class contains the requirements of a Platform to comply with the Best Practice for Earth Observation Application Package when deploying an Application Package and executing the process defined by the Application Package with specific parameters as defined by the OGC API Processes Specification (OGC 18-062r2) that requires staged files as input.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/plt-stage-in/	
Target Type	Platform
Dependency	Requirements Class “Platform” Requirements Class “Application Package Staged Inputs”

Requirement 18	req/plt-stage-in/input-stac The Platform SHALL map workflow input parameters of type “Directory” to a GeoJSON feature collection with STAC Items in the Process description.

Requirement 19	req/plt-stage-in/stac-stage The Platform SHALL stage-in all the files present on the STAC file provided by the input parameters of type “Directory”. A STAC Catalog file SHALL also be created with STAC Items referencing the files made available during execution.

9.5.3. Requirements Class “Platform Staged Outputs”

This class contains the requirements of a Platform to comply with the Best Practice for Earth Observation Application Package when deploying an Application Package and executing the process defined by the Application Package with specific parameters as defined by the OGC API Processes Specification (OGC 18-062r2) that requires staged files as output.

Requirements Class	
http://www.opengis.net/spec/eoap-bp/1.0/req/plt-stage-out/	
Target Type	Platform
Dependency	Requirements Class “Platform” Requirements Class “Application Package Staged Outputs”

	req/plt-stage-out/stac-stage
Requirement 20	The Platform SHALL stage-out all the files present on the STAC file created by the Application and create A STAC Catalog with all the outputs of the processing .

A

ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE (NORMATIVE)

A ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE (NORMATIVE)

A.1. Conformance Class “Application”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/app	
Target Type	Application
Requirement Class	Requirements Class “Application”

A.1.1. Command-Line Application

Abstract Test 1	**/conf/app/cmd-line**
Test Purpose	Validate that the Application is a non-interactive executable as a command-line application.
Requirement	req/app/cmd-line
Test Method	Execute the application and validate that no-interactive input is required

A.1.2. Container

Abstract Test 2	**/conf/app/container**

Test Purpose	Validate that all the environment, libraries, binaries, executable and configuration files necessary to execute the Application are bundled in a container image.
Requirement	req/app/container
Test Method	Test the container image and validate the successful execution.

Abstract Test 3	**/conf/app/registry**
Test Purpose	Validate that the Application container is accessible in a container registry.
Requirement	req/app/registry
Test Method	Execute the command to download the container image from a public or private registry.

A.2. Conformance Class “Application Staged Inputs”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/app-stage-in	
Target Type	Application
Requirement Class	Requirements Class “Application Staged Inputs”

A.2.1. STAC Catalog Input

Abstract Test 4	**/conf/app/stac-input**
Test Purpose	Validate that the Application requires staged EO product files listed in a STAC Catalog, named catalog.json that contains a list of one or more STAC Items and associated STAC Assets referencing the files.
Requirement	req/app/stac-input

Test Method	Execute the application with STAC Catalog input and validate the successful execution.

A.3. Conformance Class “Application Staged Outputs”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/app-stage-out	
Target Type	Application
Requirement Class	Requirements Class “Application Staged Outputs”

A.3.1. STAC Catalog Ouput

Abstract Test 5	**/conf/app/stac-out**
Test Purpose	Validate that the Application creates a valid STAC Catalog, named catalog.json, and include the STAC Item(s) and corresponding STAC Assets pointing to the results of the processing.
Requirement	req/app/stac-out
Test Method	Execute the Application and validate that a valid STAC Catalog is created with the STAC Items listing the results of the processing.

A.3.2. STAC Metadata

Abstract Test 6	**/conf/app/stac-out-metadata**
Test Purpose	Validate that the STAC Catalog created by the Application includes the necessary metadata elements .
Requirement	req/app/stac-input

Test Method	Validate that the STAC Catalog created by the Application contains for each STAC Item their spatial (geometry, box) and temporal (datetime) properties.

A.4. Conformance Class “Application Package”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/app-pck	
Target Type	Application Package
Requirement Class	Requirements Class “Application Package”

Abstract Test 7	**/conf/app-pck/cwl**
Test Purpose	Validate that the Application Package is a valid CWL and contains the expected fields.
Requirement	req/app-pck/cwl req/app-pck/clt req/app-pck/wf req/app-pck/wf-inputs req/app-pck/metadata
Test Method	Verify that the Application Pakcage is a valid CWL file and contains all the required *Workflow*, *CommandLineTool*, *inputs* and metadata fields.

A.5. Conformance Class “Application Package Staged Inputs”

Conformance Class
http://www.opengis.net/spec/eoap-bp/1.0/conf/app-pck-stage-in

Target Type	Application Package
Requirement Class	Requirements Class “Application Package Staged Inputs”

Abstract Test 8	**/conf/app-pck/stage-in**
Test Purpose	Validate that the Application Package is correctly defining an Application that requires staged files as input.
Requirement	req/app-pck-stage-in/clt-stac req/app-pck-stage-in/wf-stac
Test Method	Verify that the Application Pakcage *inputs* that require the staging of files in the *Workflow* and *CommandLineTool* classes are of type *Directory*.

A.6. Conformance Class “Application Package Staged Outputs”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/app-pck-stage-out	
Target Type	Application Package
Requirement Class	Requirements Class “Application Package Staged Outputs”

Abstract Test 9	**/conf/app-pck/stage-out**
Test Purpose	Validate that the Application Package is correctly defining an Application that requires staged files as output.
Requirement	req/app-pck-stage-out/output-stac
Test Method	Verify that the Application Pakcage main *Workflow* class is retrieve all the results and STAC Catalog.

A.7. Conformance Class “Platform”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/plt	
Target Type	Application Package
Requirement Class	Requirements Class “Platform”

Abstract Test 10	**/conf/plt/deploy**
Test Purpose	Validate that the Application Package is correctly deployed on the Platform.
Requirement	req/plt/api req/plt/inputs req/plt/file
Test Method	Verify that the Application Pakcage deployed on the Platform as a new Process and verify that the Process Description contains all the necessary elements.

A.8. Conformance Class “Platform Staged Inputs”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/plt-stage-in	
Target Type	Application Package
Requirement Class	Requirements Class “Platform Staged Inputs”

Abstract Test 11	**/conf/plt-stage-in/input-stac**

Test Purpose	Validate that the input parameters for stage-in files are mapped to GeoJSON feature collection with STAC Items in the Process description.
Requirement	req/plt-stage-in/input-stac
Test Method	Verify that the Process Description of the deployed Application Package the inputs parameters for stage-in products have the correct format and schema.

Abstract Test 12	**/conf/plt-stage-in/stac-stage**
Test Purpose	Validate that the files listed on the STAC Catalog are staged and made available for the Application execution.
Requirement	req/plt-stage-in/input-stac-stage
Test Method	Submit a test execution of the Application and verify that all the files listed in the STAC Catalog are made available.

A.9. Conformance Class “Platform Staged Outputs”

Conformance Class	
http://www.opengis.net/spec/eoap-bp/1.0/conf/plt-stage-out	
Target Type	Application Package
Requirement Class	Requirements Class “Platform Staged Outputs”

Abstract Test 13	**/conf/plt-stage-out/stac-stage**
Test Purpose	Validate that the files listed produced during the execution of the Application are staged-out.
Requirement	req/plt-stage-out/stac-stage
Test Method	Submit a test execution of the Application and verify that the result is a STAC Catalog and all the files listed are made available.

B

ANNEX B (INFORMATIVE) FREE AND OPEN-SOURCE CWL IMPLEMENTATIONS

ANNEX B (INFORMATIVE) FREE AND OPEN-SOURCE CWL IMPLEMENTATIONS

The following table lists free and open-source implementations of the CWL standards.

IMPLEMENTATION	PLATFORM
CWLTOOL	Linux, macOS, Windows (via WSL 2) local execution only
ARVADOS	in the cloud on AWS, Azure and GCP, on premise & hybrid clusters using Slurm
TOIL	AWS, Azure, GCP, Grid Engine, HTCondor, LSF, Mesos, OpenStack, Slurm, PBS/Torque also local execution on Linux, macOS, MS Windows (via WSL 2)
CWL-AIRFLOW	Local execution on Linux, OS X or via dedicated Airflow enabled cluster.
REANA	Kubernetes, CERN OpenStack, OpenStack Magnum
CALRISSIAN	CWL implementation designed to run inside a Kubernetes cluster
STREAMFLOW	StreamFlow framework is a container-native Workflow Management System

C

ANNEX C (INFORMATIVE) STAC EXAMPLES

C

ANNEX C (INFORMATIVE) STAC EXAMPLES

```
{
  "stac_version": "1.0.0",
  "stac_extensions": [ "eo", "proj", "view"],
  "type": "Feature",
  "id": "S2B_53HPA_20210723_0_L2A",
  "geometry": {
    "type": "Polygon",
    "coordinates": [ [
        [ 136.11273785955868, -36.22788818051635],[ 136.09905192261127,
 -35.238096451039816],[ 137.30513468251897, -35.22113204961173],[ 137.
33381497932513, -36.21029815477051], [ 136.11273785955868, -36.22788818051635]
      ] ]
  },
  "properties": {
    "datetime": "2021-07-23T00:57:07Z",
    "platform": "sentinel-2b",
    "constellation": "sentinel-2",
    "instruments": ["msi"],
    "gsd": 10,
    "view:off_nadir": 0,
    "proj:epsg": 32753,
    "sentinel:utm_zone": 53,
    "sentinel:latitude_band": "H",
    "sentinel:grid_square": "PA",
    "sentinel:sequence": "0",
    "sentinel:product_id": "S2B_MSIL2A_20210723T004709_N0301_R102_T53HPA_
20210723T022813",
    "sentinel:data_coverage": 100,
    "eo:cloud_cover": 9.52,
    "sentinel:valid_cloud_cover": true,
    "created": "2021-07-23T04:02:10.55Z",
    "updated": "2021-07-23T04:02:10.55Z"
  },
  "bbox": [ 136.09905192261127, -36.22788818051635, 137.33381497932513, -35.
22113204961173],
  "assets": {
    "thumbnail": {
      "type": "image/png",
      "roles": ["thumbnail"],
      "title": "Thumbnail",
      "href": "preview.jpg",
      "file:size": 123551
    },
    "overview": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": ["overview"],
      "title": "True color image",
      "href": "L2A_PVI.tif",
      "gsd": 10,
      "eo:bands": [
        {
          "name": "B04", "common_name": "red",
          "center_wavelength": 0.6645, "full_width_half_max": 0.038
        },
        { "name": "B03", "common_name": "green", "center_wavelength": 0.
56,"full_width_half_max": 0.045 },
        {
          "name": "B02",
          "common_name": "blue",
          "center_wavelength": 0.4966,
          "full_width_half_max": 0.098
        }
      ],
      "proj:shape": [ 343, 343],
      "proj:transform": [ 320, 0, 600000, 0, -320, 6100000, 0, 0, 1 ],
      "file:size": 273345
    },
    "info": {
      "type": "application/json",
      "roles": [ "metadata"],
      "title": "Original JSON metadata",
      "href": "tileInfo.json",
      "file:size": 1491
    },
    "metadata": {
      "type": "application/xml",
      "roles": [ "metadata"],
      "title": "Original XML metadata",
      "href": "metadata.xml",
      "file:size": 631511
    },
    "visual": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "overview" ],
      "title": "True color image",
      "href": "TCI.tif",
      "gsd": 10,
      "eo:bands": [
        { "name": "B04","common_name": "red", "center_wavelength": 0.6645,
 "full_width_half_max": 0.038},
        { "name": "B03", "common_name": "green", "center_wavelength": 0.56,
 "full_width_half_max": 0.045},
        { "name": "B02", "common_name": "blue", "center_wavelength": 0.4966,
 "full_width_half_max": 0.098}
      ],
      "proj:shape": [ 10980, 10980 ],
      "proj:transform": [ 10, 0, 600000, 0, -10, 6100000, 0, 0, 1 ],
      "file:size": 242697374
    },
    "B01": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data"],
      "title": "Band 1 (coastal)",
      "href": "B01.tif",
      "gsd": 60,
      "eo:bands": [ { "name": "B01", "common_name": "coastal", "center_
wavelength": 0.4439, "full_width_half_max": 0.027 } ],
      "proj:shape": [ 1830, 1830 ],
      "proj:transform": [60, 0, 600000, 0, -60, 6100000, 0, 0, 1],
      "file:size": 6156076
    },
    "B02": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Band 2 (blue)",
      "href": "B02.tif",
      "gsd": 10,
      "eo:bands": [ { "name": "B02", "common_name": "blue", "center_
wavelength": 0.4966, "full_width_half_max": 0.098 } ],
      "proj:shape": [ 10980, 10980 ],
      "proj:transform": [ 10,  0, 600000, 0, -10, 6100000, 0, 0, 1 ],
      "file:size": 206117177
    },
    "B03": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Band 3 (green)",
      "href": "B03.tif",
      "gsd": 10,
      "eo:bands": [ { "name": "B03", "common_name": "green", "center_
wavelength": 0.56, "full_width_half_max": 0.045 } ],
      "proj:shape": [ 10980, 10980 ],
      "proj:transform": [ 10, 0, 600000, 0, -10, 6100000, 0, 0, 1 ],
      "file:size": 201505523
    },
    "B04": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data"  ],
      "title": "Band 4 (red)",
      "href": "B04.tif",
      "gsd": 10,
      "eo:bands": [ { "name": "B04", "common_name": "red", "center_wavelength":
 0.6645, "full_width_half_max": 0.038 } ],
      "proj:shape": [ 10980, 10980 ],
      "proj:transform": [ 10, 0, 600000, 0, -10, 6100000, 0, 0, 1 ],
      "file:size": 192143151
    },
    "B05": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Band 5",
      "href": "B05.tif",
      "gsd": 20,
      "eo:bands": [ { "name": "B05", "center_wavelength": 0.7039, "full_width_
half_max": 0.019 } ],
      "proj:shape": [5490,5490],
      "proj:transform": [20,0,600000,0,-20,6100000,0,0,1],
      "file:size": 49644807
    },
    "B06": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": ["data"],
      "title": "Band 6",
      "href": "B06.tif",
      "gsd": 20,
      "eo:bands": [{ "name": "B06", "center_wavelength": 0.7402," full_width_
half_max": 0.018 }],
      "proj:shape": [ 5490,5490 ] ,
      "proj:transform": [ 20, 0, 600000, 0, -20, 6100000, 0, 0, 1 ],
      "file:size": 46787561
    },
    "B07": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [  "data"  ],
      "title": "Band 7",
        "href": "B07.tif",
        "gsd": 20,
        "eo:bands": [ { "name": "B07", "center_wavelength": 0.7825, "full_width_
  half_max": 0.028 }
        ],
        "proj:shape": [ 5490, 5490 ],
        "proj:transform": [ 20, 0, 600000, 0, -20, 6100000, 0, 0, 1 ],
        "file:size": 49482530
      },
      "B08": {
        "type": "image/tiff; profile=cloud-optimized; application=geotiff",
        "roles": [ "data" ],
        "title": "Band 8 (nir)",
        "href": "B08.tif",
        "gsd": 10,
        "eo:bands": [ { "name": "B08", "common_name": "nir", "center_wavelength":
   0.8351, "full_width_half_max": 0.145 } ],
        "proj:shape": [ 10980, 10980 ],
        "proj:transform": [ 10, 0, 600000, 0, -10, 6100000, 0, 0, 1 ],
        "file:size": 176605232
      },
      "B8A": {
        "type": "image/tiff; profile=cloud-optimized; application=geotiff",
        "roles": [ "data" ],
        "title": "Band 8A",
        "href": "B8A.tif",
        "gsd": 20,
        "eo:bands": [ { "name": "B8A", "center_wavelength": 0.8648, "full_width_
  half_max": 0.033 } ],
        "proj:shape": [ 5490, 5490 ],
        "proj:transform": [ 20, 0, 600000, 0, -20, 6100000, 0, 0, 1 ],
        "file:size": 46667884
      },
      "B09": {
        "type": "image/tiff; profile=cloud-optimized; application=geotiff",
        "roles": [ "data"],
        "title": "Band 9",
        "href": "B09.tif",
        "gsd": 60,
        "eo:bands": [ { "name": "B09", "center_wavelength": 0.945, "full_width_
  half_max": 0.026 } ],
        "proj:shape": [ 1830, 1830 ],
        "proj:transform": [ 60, 0, 600000, 0, -60, 6100000, 0, 0, 1 ],
        "file:size": 4854967
      },
      "B11": {
        "type": "image/tiff; profile=cloud-optimized; application=geotiff",
        "roles": [ "data" ],
        "title": "Band 11 (swir16)",
        "href": "B11.tif",
        "gsd": 20,
        "eo:bands": [ { "name": "B11", "common_name": "swir16", "center_
  wavelength": 1.6137, "full_width_half_max": 0.143 } ],
        "proj:shape": [ 5490, 5490  ],
        "proj:transform": [ 20, 0, 600000, 0, -20, 6100000, 0, 0, 1 ],
        "file:size": 50874908
      },
      "B12": {
        "type": "image/tiff; profile=cloud-optimized; application=geotiff",
        "roles": [ "data" ],
        "title": "Band 12 (swir22)",
        "href": "B12.tif",
        "gsd": 20,
      "eo:bands": [ { "name": "B12", "common_name": "swir22", "center_
wavelength": 2.22024, "full_width_half_max": 0.242 } ],
      "proj:shape": [ 5490, 5490 ],
      "proj:transform": [ 20, 0, 600000, 0, -20, 6100000, 0, 0, 1 ],
      "file:size": 50482209
    },
    "AOT": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Aerosol Optical Thickness (AOT)",
      "href": "AOT.tif",
      "proj:shape": [ 1830, 1830 ],
      "proj:transform": [ 60, 0, 600000, 0, -60, 6100000, 0, 0, 1 ],
      "file:size": 135032
    },
    "WVP": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Water Vapour (WVP)",
      "href": "WVP.tif",
      "proj:shape": [ 10980, 10980 ],
      "proj:transform": [ 10, 0, 600000, 0, -10, 6100000, 0, 0, 1 ],
      "file:size": 11314834
    },
    "SCL": {
      "type": "image/tiff; profile=cloud-optimized; application=geotiff",
      "roles": [ "data" ],
      "title": "Scene Classification Map (SCL)",
      "href": "SCL.tif",
      "proj:shape": [ 5490, 5490 ],
      "proj:transform": [ 20, 0, 600000, 0, -20, 6100000, 0, 0, 1 ],
      "file:size": 2850898
    }
  },
  "links": [
    {
      "type": "application/json",
      "rel": "canonical",
      "href": "https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-
l2a-cogs/53/H/PA/2021/7/S2B_53HPA_20210723_0_L2A/S2B_53HPA_20210723_0_L2A.json"
    },
    {
      "rel": "parent",
      "href": "../catalog.json"
    }
  ]
}
```

D

ANNEX D (INFORMATIVE) APPLICATION EXAMPLES

D ANNEX D (INFORMATIVE) APPLICATION EXAMPLES

D.1. Crop Application Example

This example includes all the files for an application that crops a band from a Sentinel-2 product. It includes the shell scripts (crop and functions.sh), docker container (Dockerfile.composite), the Application Package as a CWL file, and execution parameters as a YAML file (params.yml).

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 band crop
  doc: This application crops a Sentinel-2 band
  id: s2-cropper

  inputs:
    product:
      type: Directory
      label: Sentinel-2 inputs
      doc: Sentinel-2 Level-1C or Level-2A input reference
    band:
      type: string
      label: Sentinel-2 band
      doc: Sentinel-2 band to crop (e.g. B02)
    bbox:
      type: string
      label: bounding box
      doc: Area of interest expressed as a bounding bbox
    proj:
      type: string
      label: EPSG code
      doc: Projection EPSG code for the bounding box
      default: "EPSG:4326"

  outputs:
    results:
      outputSource:
      - node_crop/cropped_tif
      type: Directory

  steps:

    node_crop:

      run: "#crop-cl"
      in:
        product: product
        band: band
        bbox: bbox
        epsg: proj

      out:
        - cropped_tif


- class: CommandLineTool

  id: crop-cl

  requirements:
    DockerRequirement:
      dockerPull: docker.io/terradue/crop-container

  baseCommand: crop
  arguments: []

  inputs:
    product:
      type: Directory
      inputBinding:
        position: 1
    band:
      type: string
      inputBinding:
        position: 2
    bbox:
      type: string
      inputBinding:
        position: 3
    epsg:
      type: string
      inputBinding:
        position: 4

  outputs:
    cropped_tif:
      outputBinding:
        glob: .
      type: Directory

$namespaces:
  s: https://schema.org/
s:softwareVersion: 1.0.0
schemas:
- http://schema.org/version/9.0/schemaorg-current-http.rdf
```

app-package.cwl

```
#!/bin/bash

source /functions.sh

## processing arguments
in_dir=$1 # folder where the EO product is staged-in
band=$2 # asset key to crop
bbox=$3 # bbox processing argument
proj=$4 # EPSG code used to express bbox coordinates
```

```
## Read the input STAC catalog
# STAC catalog path
catalog="${in_dir}/catalog.json"

# get the item path
item=$( get_items ${catalog} )

# get the B02 asset href (local path)
asset_href=$( get_asset ${item} ${band} )

gdal_translate \
      -projwin \
      "$( echo $bbox | cut -d ',' -f 1)" \
      "$( echo $bbox | cut -d ',' -f 4)" \
      "$( echo $bbox | cut -d ',' -f 3)" \
      "$( echo $bbox | cut -d ',' -f 2)" \
      -projwin_srs \
      ${proj} \
      ${asset_href} \
      ${band}_cropped.tif


## result as STAC
# get the properties from the input STAC item
# as these are the same for the output STAC item
datetime=$( get_item_property ${item} "datetime" )
gsd=$( get_item_property ${item} "gsd" )

# initialise a STAC item
init_item ${datetime} "${bbox}" "${gsd}" > result-item.json

# add an asset
add_asset result-item.json "${band}" "./cropped.tif" "image/tiff" "Cropped
 ${band} band" "data"

# initialise the output catalog
init_catalog > catalog.json

# add the item
add_item catalog.json result-item.json
```

crop

```
function get_items {
    # returns the list of items in catalog
    # args:
    #  STAC catalog
    catalog=$1
    for item in $( jq -r '.links[] | select(.. | .rel? == "item") | .href'
 ${catalog} )
    do
        echo $( dirname ${catalog})/${item}
    done
}
export -f get_items

function get_asset {
    # returns the asset href
    # args:
    #  STAC item
    #  asset key
    item=$1
    asset=$2
    echo $( dirname $item)/$( jq --arg asset $asset -r ".assets.$asset.href"
 $item )
}
export -f get_asset

function get_item_property {
    # return an item property value
    # args:
    #  STAC item
    #  property key
    item=$1
    property=$2
    echo $( jq --arg property $property ".properties.$property" $item )
}
export -f get_item_property

function init_catalog {
    # returns a STAC catalog without items
    echo '{}' |
    jq '.["id"]="catalog"' |
    jq '.["stac_version"]="1.0.0"' |
    jq '.["type"]="catalog"' |
    jq '.["description"]="Result catalog"' |
    jq '.["links"]=[]'
}
export -f init_catalog

function add_item {
    # adds an item to a STAC catalog
    # args:
    #  STAC catalog
    #  STAC item href
    catalog=$1
    item=$2
    jq -e \
        --arg item ${item} \
        '.links += [{ "type":"application/geo+json", "href":$item}]' ${catalog}
 > ${catalog}.tmp && mv ${catalog}.tmp ${catalog}
}
export -f add_item

function init_item {
    datetime=$1
    bbox=$2
    gsd=$3
    echo '{}' |
    jq '.["id"]="item_id"' | # set the item id
    jq '.["stac_version"]="1.0.0"' |
    jq '.["type"]="Feature"' | # set the item type
    jq --arg c "$( echo $bbox | cut -d ',' -f 1)" '.bbox[0]=$c' | # set the
 bbox elements
    jq --arg c "$( echo $bbox | cut -d ',' -f 2)" '.bbox[1]=$c' |
    jq --arg c "$( echo $bbox | cut -d ',' -f 3)" '.bbox[2]=$c' |
    jq --arg c "$( echo $bbox | cut -d ',' -f 4)" '.bbox[3]=$c' |
    jq '.bbox[] |= tonumber' | # convert the bbox to number
    jq '.["geometry"].type="Polygon"' | # set the geometry type
    jq '.["geometry"].coordinates=[]' |
    jq '.["geometry"].coordinates[0]=[]' |
    jq --arg min_lon "$( echo $bbox | cut -d ',' -f 1)" \
    --arg min_lat "$( echo $bbox | cut -d ',' -f 2)" \
    --arg max_lon "$( echo $bbox | cut -d ',' -f 3)" \
    --arg max_lat "$( echo $bbox | cut -d ',' -f 4)" \
    '.["geometry"].coordinates[0][0]=[$min_lon | tonumber, $min_lat | tonumber]
  | .["geometry"].coordinates[0][1]=[$max_lon | tonumber, $min_lat | tonumber]
  | .["geometry"].coordinates[0][2]=[$max_lon | tonumber, $max_lat | tonumber]
  | .["geometry"].coordinates[0][3]=[$min_lon | tonumber, $max_lat | tonumber]
  | .["geometry"].coordinates[0][4]=[$min_lon | tonumber, $min_lat | tonumber]'
  | # set the geojson Polygon coordinates
    jq --arg dt ${datetime} '.properties.datetime=$dt' | # set the datetime
    jq --arg gsd "${gsd}" '.properties.gsd=$gsd' | # set the gsd
    jq '.properties.gsd |= tonumber' | # convert the gsd to number
    jq -r '.["assets"]={}'
}
export -f init_item

function add_asset {
    # adds an asset to a STAC item
    # args:
    #  STAC item
    #  asset key
    #  asset href
    #  asset mime-type
    #  asset title
    #  asset role
    item=$1
    asset_key=$2
    href="$3"
    type="$4"
    title="$5"
    role="$6"
    jq -e -r \
        --arg asset_key $asset_key \
        --arg href ${href} \
        --arg type "${type}" \
        --arg title "${title}" \
        --arg role "${role}" \
        '.assets += { ($asset_key) : { "role":[$role], "href":$href, "type":
$type, "title":$title}}' ${item} > ${item}.tmp && mv ${item}.tmp ${item}
}
export -f add_asset
```

functions.sh

```
FROM osgeo/gdal

RUN apt update && \
    apt-get install -y jq

ADD functions.sh /functions.sh

ADD composite /usr/bin/composite

RUN chmod +x /usr/bin/composite
```

Dockerfile.composite

```
product:
  class: Directory
  path: /tmp/docker_tmpoj8r5t9a
band: B02
bbox: "136.522,-36.062,137.027,-35.693"
proj: "EPSG:4326"
```

params.yml

D.2. Scatter Crop Application Example

This section extends the previous example with an Application Package that scatters the processing from an array of input values.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 product crop
  doc: This application crops bands from a Sentinel-2 product
  id: s2-cropper

  requirements:
  - class: ScatterFeatureRequirement

  inputs:
    product:
      type: Directory
      label: Sentinel-2 input
      doc: Sentinel-2 Level-1C or Level-2A input reference
    bands:
      type: string[]
      label: Sentinel-2 bands
      doc: Sentinel-2 list of bands to crop
    bbox:
      type: string
      label: bounding box
      doc: Area of interest expressed as a bounding box
    proj:
      type: string
      label: EPSG code
      doc: Projection EPSG code for the bounding box
      default: "EPSG:4326"

  outputs:
    results:
      outputSource:
      - node_crop/cropped_tif
      type: Directory[]

  steps:

    node_crop:

      run: "#crop-cl"

      in:
        product: product
        band: bands
        bbox: bbox
        epsg: proj

      out:
        - cropped_tif

      scatter: band
      scatterMethod: dotproduct

- class: CommandLineTool
```

```
  id: crop-cl

  requirements:
    DockerRequirement:
      dockerPull: docker.io/terradue/crop-container

  baseCommand: crop
  arguments: []

  inputs:
    product:
      type: Directory
      inputBinding:
        position: 1
    band:
      type: string
      inputBinding:
        position: 2
    bbox:
      type: string
      inputBinding:
        position: 3
    epsg:
      type: string
      inputBinding:
        position: 4

  outputs:
    cropped_tif:
      outputBinding:
        glob: .
      type: Directory

$namespaces:
  s: https://schema.org/
s:softwareVersion: 1.0.0
schemas:
- http://schema.org/version/9.0/schemaorg-current-http.rdf
```

app-package-scatter.cwl

D.3. Composite two-step Workflow Example

This section extends the previous example with an Application Package that is a two-step workflow that crops (using scatter over the bands) and creates a composite image.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 RGB composite
  doc: This application generates a Sentinel-2 RGB composite over an area of
 interest
  id: s2-compositer
  requirements:
  - class: ScatterFeatureRequirement
  - class: InlineJavascriptRequirement
  - class: MultipleInputFeatureRequirement
```

```
    inputs:
      product:
        type: Directory
        label: Sentinel-2 input
        doc: Sentinel-2 Level-1C or Level-2A input reference
      red:
        type: string
        label: red channel
        doc: Sentinel-2 band for red channel
      green:
        type: string
        label: green channel
        doc: Sentinel-2 band for green channel
      blue:
        type: string
        label: blue channel
        doc: Sentinel-2 band for blue channel
      bbox:
        type: string
        label: bounding box
        doc: Area of interest expressed as a bounding bbox
      proj:
        type: string
        label: EPSG code
        doc: Projection EPSG code for the bounding box coordinates
        default: "EPSG:4326"
    outputs:
      results:
        outputSource:
        - node_composite/rgb_composite
        type: Directory
    steps:
      node_crop:
        run: "#crop-cl"
        in:
          product: product
          band: [red, green, blue]
          bbox: bbox
          epsg: proj
        out:
          - cropped_tif
        scatter: band
        scatterMethod: dotproduct
      node_composite:
        run: "#composite-cl"
        in:
          tifs:
            source:  node_crop/cropped_tif
          lineage: product
        out:
          - rgb_composite

  - class: CommandLineTool
    id: crop-cl
    requirements:
      DockerRequirement:
        dockerPull: docker.io/terradue/crop-container
    baseCommand: crop
    arguments: []
    inputs:
      product:
        type: Directory
        inputBinding:
          position: 1
      band:
        type: string
        inputBinding:
          position: 2
      bbox:
        type: string
        inputBinding:
          position: 3
      epsg:
        type: string
        inputBinding:
          position: 4
    outputs:
      cropped_tif:
        outputBinding:
          glob: '*.tif'
        type: File

  - class: CommandLineTool
    id: composite-cl
    requirements:
      DockerRequirement:
        dockerPull: docker.io/terradue/composite-container
      InlineJavascriptRequirement: {}
    baseCommand: composite
    arguments:
    - $( inputs.tifs[0].path )
    - $( inputs.tifs[1].path )
    - $( inputs.tifs[2].path )
    inputs:
      tifs:
        type: File[]
      lineage:
        type: Directory
        inputBinding:
          position: 4
    outputs:
      rgb_composite:
        outputBinding:
          glob: .
        type: Directory

$namespaces:
  s: https://schema.org/
s:softwareVersion: 1.0.0
schemas:
- http://schema.org/version/9.0/schemaorg-current-http.rdf
```

app-package-two-steps-rgb.cwl

```
#!/bin/bash

## processing arguments
red_channel=$1 # tif file for composite red channel
green_channel=$2 # tif file for composite green channel
blue_channel=$3 # tif file for composite blue channel
in_dir=$4 # input STAC to retrieve the metadata (folder where the EO product is
 staged-in)
bbox=$5 # area of interest

gdalbuildvrt \
      -separate composite.vrt \
      ${red_channel} \
      ${green_channel} \
      ${blue_channel}

gdal_translate \
      composite.vrt \
      composite.tif

gdal_translate \
      -ot Byte \
      composite.vrt \
      composite-preview.tif

rm -f composite.vrt

## result as STAC
source /functions.sh
## Read the input STAC catalog
# STAC catalog path
catalog="${in_dir}/catalog.json"

# get the item path
item=$( get_items ${catalog} )

# get the properties from the input STAC item
# as these are the same for the output STAC item
datetime=$( get_item_property ${item} "datetime" )
gsd=$( get_item_property ${item} "gsd" )

# initialise a STAC item
init_item ${datetime} "${bbox}" "${gsd}" > composite-item.json

# add an asset
add_asset composite-item.json "composite" "./composite.tif" "image/tiff" "RGB
 composite" "data"

# add an asset
add_asset composite-item.json "preview" "./composite-preview.tif" "image/tiff"
 "RGB composite preview" "preview"

# initialise the output catalog
init_catalog > catalog.json

# add the item
add_item catalog.json composite-item.json
```

composite

```
FROM osgeo/gdal

RUN apt update && \
    apt-get install -y jq

ADD functions.sh /functions.sh

ADD composite /usr/bin/composite

RUN chmod +x /usr/bin/composite
```

Dockerfile.composite

```
product:
  class: Directory
  path: /tmp/docker_tmpoj8r5t9a
red: B04
green: B03
blue: B02
bbox: "136.522,-36.062,137.027,-35.693"
proj: "EPSG:4326"
```

params-rgb.yml

D.4. Multiple Inputs Composite Two-step Workflow Example

This section extends the previous example by creating a workflow that takes a list of products as input and invokes a two-step sub-workflow that crops (using scatter over the bands) and creates a composite.

```
cwlVersion: v1.0
$graph:
- class: Workflow
  label: Sentinel-2 RGB composites
  doc: This application generates a Sentinel-2 RGB composite over an area of
 interest with selected bands
  id: s2-composites
  requirements:
  - class: SubworkflowFeatureRequirement
  - class: ScatterFeatureRequirement
  inputs:
    products:
      type: Directory[]
      label: Sentinel-2 inputs
      doc: Sentinel-2 Level-1C or Level-2A input references
    red:
      type: string
      label: red channel
      doc: Sentinel-2 band for red channel
    green:
      type: string
      label: green channel
      doc: Sentinel-2 band for green channel
    blue:
      type: string
      label: blue channel
      doc: Sentinel-2 band for blue channel
    bbox:
      type: string
      label: bounding box
      doc: Area of interest expressed as a bounding bbox
    proj:
      type: string
      label: EPSG code
      doc: Projection EPSG code for the bounding box coordinates
      default: "EPSG:4326"
  outputs:
    wf_results:
      outputSource:
      - node_rgb/results
```

```
        type: Directory[]
    steps:
      node_rgb:
        run: "#s2-compositer"
        in:
          product: products
          red: red
          green: green
          blue: blue
          bbox: bbox
          proj: proj
        out:
        - results
        scatter: product
        scatterMethod: dotproduct

  - class: Workflow
    label: Sentinel-2 RGB composite
    doc: This sub-workflow generates a Sentinel-2 RGB composite over an area of
   interest
    id: s2-compositer
    requirements:
    - class: ScatterFeatureRequirement
    - class: InlineJavascriptRequirement
    - class: MultipleInputFeatureRequirement
    inputs:
      product:
        type: Directory
        label: Sentinel-2 input
        doc: Sentinel-2 Level-1C or Level-2A input reference
      red:
        type: string
        label: red channel
        doc: Sentinel-2 band for red channel
      green:
        type: string
        label: green channel
        doc: Sentinel-2 band for green channel
      blue:
        type: string
        label: blue channel
        doc: Sentinel-2 band for blue channel
      bbox:
        type: string
        label: bounding box
        doc: Area of interest expressed as a bounding bbox
      proj:
        type: string
        label: EPSG code
        doc: Projection EPSG code for the bounding box coordinates
        default: "EPSG:4326"
    outputs:
      results:
        outputSource:
        - node_composite/rgb_composite
        type: Directory
    steps:
      node_crop:
        run: "#crop-cl"
        in:
          product: product
          band: [red, green, blue]
          bbox: bbox
          epsg: proj
        out:
          - cropped_tif
        scatter: band
        scatterMethod: dotproduct
      node_composite:
        run: "#composite-cl"
        in:
          tifs:
            source:  node_crop/cropped_tif
          lineage: product
        out:
          - rgb_composite

  - class: CommandLineTool
    id: crop-cl
    requirements:
      DockerRequirement:
        dockerPull: docker.io/terradue/crop-container
    baseCommand: crop
    arguments: []
    inputs:
      product:
        type: Directory
        inputBinding:
          position: 1
      band:
        type: string
        inputBinding:
          position: 2
      bbox:
        type: string
        inputBinding:
          position: 3
      epsg:
        type: string
        inputBinding:
          position: 4
    outputs:
      cropped_tif:
        outputBinding:
          glob: '*.tif'
        type: File

  - class: CommandLineTool
    id: composite-cl
    requirements:
      DockerRequirement:
        dockerPull: docker.io/terradue/composite-container
      InlineJavascriptRequirement: {}
    baseCommand: composite
    arguments:
    - $( inputs.tifs[0].path )
    - $( inputs.tifs[1].path )
    - $( inputs.tifs[2].path )
    inputs:
      tifs:
        type: File[]
      lineage:
        type: Directory
        inputBinding:
          position: 4
    outputs:
      rgb_composite:
        outputBinding:
          glob: .
        type: Directory

  $namespaces:
    s: https://schema.org/
  s:softwareVersion: 1.0.0
  schemas:
  - http://schema.org/version/9.0/schemaorg-current-http.rdf
```

app-package-multiple-products.cwl

E

ANNEX E (INFORMATIVE) REVISION HISTORY

E

ANNEX E (INFORMATIVE) REVISION HISTORY

Table E.1

DATE	RELEASE	EDITOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
2020-11-10	0.1	Pedro Gonçalves	all	initial version
2021-03-19	0.2	Pedro Gonçalves	all	consolidated draft
2021-05-03	0.3	Pedro Gonçalves	all	improved for DWG evaluations
2021-08-21	1.0	Pedro Gonçalves	all	improved with comments and suggestion received added requeriments and conformance classes

BIBLIOGRAPHY

BIBLIOGRAPHY

1. *Cloud Native Glossary — CNCF Business Value Subcommittee (BVS)*, 2021, https://glossary.cncf.io/

2. Commonwl.org: *Common Workflow Language (CWL) Command Line Tool Description v1.2*, 2020, https://www.commonwl.org/v1.2/CommandLineTool.html

3. Commonwl.org: *Common Workflow Language (CWL) Workflow Description v1.2*, 2020, https://www.commonwl.org/v1.2/Workflow.html

4. Pedro Gonçalves: OGC 20-042, *OGC Earth Observations Applications Pilot: Terradue Engineering Report*. Open Geospatial Consortium (2020). http://docs.ogc.org/per/20-042.html

5. Ingo Simonis: OGC 20-073, *OGC Earth Observation Applications Pilot: Summary Engineering Report*. Open Geospatial Consortium (2020). https://docs.ogc.org/per/20-073.html