

OGC® DOCUMENT: 24-037

External identifier of this OGC® document: <http://www.opengis.net/doc/PER/t20-D021>



Open
Geospatial
Consortium

OGC TESTBED 20 GDC USABILITY TESTING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2025-02-13

Approval Date: 2025-03-06

Publication Date: YYYY-MM-DD

Editor: Peter Zellner, Jonas Eberle

Notice: This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is *not an official position* of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard.

Further, any OGC Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

License Agreement

Use of this document is subject to the license agreement at <https://www.ogc.org/license>

Copyright notice

Copyright © 2025 Open Geospatial Consortium

To obtain additional rights of use, visit <https://www.ogc.org/legal>

Note

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.

CONTENTS

- I. OVERVIEWv
- II. EXECUTIVE SUMMARYv
- III. KEYWORDSvi
- IV. CONTRIBUTORSvi
- V. FUTURE OUTLOOKvii
- VI. VALUE PROPOSITIONvii
- 1. INTRODUCTION2
 - 1.1. Aims3
 - 1.2. Objectives3
- 2. TOPICS6
 - 2.1. Usability Testing Setup6
 - 2.2. Usability Testing Outcomes14
- 3. OUTLOOK23
- 4. SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS25
- BIBLIOGRAPHY27
- ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS29
- ANNEX B (NORMATIVE) INTEGRATION TESTS31
 - B.1. STAC to openEO31
 - B.2. STAC to OGC API – Processes32
 - B.3. OGC API Coverages to openEO35
 - B.4. OGC API – Coverages to OGC API – Processes36

LIST OF TABLES

- Table – Table Contributorsvi
- Table 1 – Overview of implemented processing APIs, inputs and outputs.11

Table 2 – Overview of integration tests on the implementation level.	13
---	----



OVERVIEW

The initial idea behind the OGC Testbed-20 GeoDataCube (GDC) activity was to homogenize existing OGC API Standards that can be used for GDC access and processing (OGC API – Coverages, OGC API – Processes and openEO) into a single draft OGC API – GeoDataCubes (GDC API) Standard. The Testbed 20 outcomes showed that the integration and co-existence of existing OGC API Standards is more promising than defining a new standard. This means that existing implementation solutions can be combined and/or integrated to create a new processing chain. The enabler for this kind of integration is interoperability between existing API implementations, which is the ability to package results from one API endpoint response and use this response package as input to another API endpoint. The missing “bridge” is the ability to seamlessly “pass” the inputs and outputs between different solutions because they are not harmonized and do not support seamless interoperability.

Therefore, the Testbed-20 GDC usability testing focused on identifying the gaps that hinder interoperability between existing solutions. Useability testing consisted of a set of integration tests between the considered Testbed 20 solutions and their available implementations. Testing was performed by the Testbed 20 participants. The integration tests were conducted on five backends provided by the Testbed participants (Ecere, Eurac Research, CRIM, Ellipsis Drive, MMS). Each backend offered different implementations of the existing OGC API Standards. The testing focused on the interoperability between the API endpoints. As such, the relevant capabilities were request and response mechanisms and how they could be combined to enhance integration and interoperability.

The goal was to close these integration bridges by suggesting best practices and conversion approaches to increase interoperability. Outcomes from the testing are documented as recommendations in this Report for the further development and improvement of GDC access and processing.



EXECUTIVE SUMMARY

The goal of the OGC Testbed-20 GeoDataCube (GDC) initiative was to investigate streamlining APIs for GDC access and processing, such as implementations of the OGC API – Coverages, OGC API – Processes Standards and the OGC openEO and STAC Community Standards. The original hypothesis was to define a draft standard that unified aspects of these standards. However, integrating and co-existing with the existing OGC API baseline standards proved more promising than creating a new standard. The key enabler is interoperability – the seamless transfer of data between API implementations. Usability and integration testing identified gaps in input and output mechanisms, with integration tests across five backends (Ecere, Eurac Research, CRIM, Ellipsis Drive, MMS) highlighting the need for best practices and conversion methods.

Focusing on access mechanism alignment enhances user experience by fostering interoperability, enabling users to integrate and switch between different API processing

chains. Developers benefit from common references, reducing custom solutions and simplifying conversions between protocols such as STAC and OGC API – Coverages implementations. Expanding tests to more implementations will further enhance API interoperability and support future standardization efforts within the GDC ecosystem.



KEYWORDS

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

ogc, geodatacube, usability testing, integration testing, interoperability, application-to-the-cloud, testbed, platform



CONTRIBUTORS

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

Table — Table Contributors

NAME	ORGANIZATION	ROLE
Peter Zellner	German Aerospace Center	Editor
Jonas Eberle	German Aerospace Center	Editor
Matthias Mohr	Matthias Mohr – Softwareentwicklung	Contributor
Jérôme Jacovella-St-Louis	Ecere Corporation	Contributor
Francis Charette Migneault	CRIM	Contributor
Daniel van der Maas	Ellipsis Drive	Contributor
Michele Claus	Eurac Research	Contributor



FUTURE OUTLOOK

Ideally, the integration tests could be expanded to other OGC API Standards implementations after the completion of Testbed 20 (April 2025). Such future work would enable the OGC to gather more insights as to how other implementations of existing OGC API Standards interpret access mechanisms and where further alignment is needed. The results of the integration tests focused on facilitating future initiatives for increasing interoperability and standardization in the field of GDCs. An example of such an activity would be to align the SpatioTemporal Asset Catalog (STAC) metadata generation into best practices that GDC API Standards and implementations can adhere to.



VALUE PROPOSITION

The results of the OGC Testbed 20 GDC usability testing contribute to the groundwork for harmonization of implementations of OGC API Standards in the GDC information domain. In this OGC Report, the current state of interoperability is presented and discusses the need to improve the definition of input and output mechanisms. Aligning input and output mechanisms for different OGC API Standard's implementation instances inherently creates interoperability by ensuring that different implementations of the same standard can communicate, allowing users to chain and interchange between different API implementation instances (endpoints). Developers will profit from aligned input and output mechanisms because then they will have a common reference to develop and test against and will not need to invent custom solutions. Also, the development of conversion methods between different access mechanisms will benefit from this alignment (e.g., STAC to OGC API — Coverages or vice versa). Finally, the results of the usability testing will facilitate the harmonization of access and processing GDC related API implementations in the future.



1

INTRODUCTION

The impact of climate change on Earth presents complex challenges in Earth System Sciences (ESS). At the same time, there is a rapid increase in the volume and variety of ESS data monitoring the planet. Although cloud computing platforms offer solutions by hosting data and processing services, this landscape is evolving quickly and remains fragmented ([Sudmanns2019]). The goal of the OGC Testbed 20 GDC task was to contribute to the standardization of cloud data access, processing, and outcomes by defining a standardized GeoDataCube API. Such an API standard would help address the difficulties of adapting to diverse platform technologies and promote the adoption of the FAIR principles (Findability, Accessibility, Interoperability, and Reusability) ([Wilkinson2016]). The long-term goal is for users to perform their analyses using a client in their preferred programming language, enabling the creation of standardized workflows that are portable and reusable across cloud platforms, with results that can be published directly in accordance with the FAIR principles.

Since the cloud platform landscape is diverse, evolving, and offers many access points, it is difficult to get a comprehensive overview of users and their needs. Immediate needs are usually addressed through platform forums or helpdesks, where users can report specific problems. However, this information is widely scattered. Some efforts have been made to collect user perspectives on the usefulness and usability of cloud platforms through questionnaires ([Wagemann2021a], [Wagemann2021b], [DiLeo2024], [DiLeo2023], [Zellner2024]). Additionally, the recommendations from the OGC Testbed-19 GeoDataCubes Usability Testing ([Jacob2024]) are available for comparison.

Wagemann et al. ([Wagemann2021a], [Wagemann2021b]) examined user perspectives on cloud-based services for Big Earth data, identifying the need to align current user practices (download and process) with the evolving strategies of major data organizations (cloud data access and processing). A web-based survey revealed strong interest in cloud migration but identified key barriers: limited cloud literacy, security concerns, and unclear cost structures. Users showed a preference for cloud-based systems such as “bring code to the data” over user-centric platforms that are overly engineered and highly abstracted, despite low adoption rates.

To address these challenges, the authors recommend targeting researchers and data scientists rather than decision-makers, as scientists and researchers are the actual users. They emphasize the importance of capacity-building to bridge skill gaps—aligning with the findings of Zellner et al. ([Zellner2024])—to improve trust in cloud services and showcase the benefits of cloud-based solutions. Promoting interoperability through adherence to established standards is identified as one of the most critical steps to support broader adoption and enhance user confidence.

Di Leo et al. ([DiLeo2023]) analyzed over 150 Earth Observation (EO) cloud platforms from a user-centric perspective, identifying key obstacles such as platform fragmentation, steep learning curves, limited service discoverability, and non-reusable workflows. They emphasize the need for open licenses for data, the adoption of open standards, and standardized APIs to improve interoperability. Usability testing and co-design with users are critical for enhancing platform functionality, with user involvement driving improvements that lead to greater uptake and impact of the services offered.

In a follow-up study, Di Leo et al. ([DiLeo2024]) classified user requirements into ten dimensions of the EO value chain by studying policy-relevant use cases carried out on EO cloud platforms. The top priority is interoperability, followed by accessibility and discoverability. Data must be assessed in accordance with the FAIR and TRUST principles. The self-assessment guidelines suggest evaluating interoperability through compliance with standards, the ability to integrate data and services from different sources, and semantic consistency.

The Committee on Earth Observation Satellites (CEOS) is working on an Interoperability Handbook (Version 2) to address the topics mentioned above by defining a set of recommendations that cloud platform and data providers should follow.

1.1. Aims

The OGC Testbed-20 GDC API usability testing aimed to gain further insights into the state of interoperability between different OGC API implementations for accessing and processing GDC, and to identify areas needing improvement. During the Testbed-20 activities, it became evident that, particularly for GeoDataCube access mechanisms, alignment and implementation need to be addressed to enhance interoperability. This Engineering Report contributes to the limited number of efforts undertaken to shed light on the interoperability of access mechanisms across various GDC-related API implementations within the field of Earth system science cloud computing. The OGC Testbed-19 Usability Testing GeoDataCubes Engineering Report [OGC 23-047] ([Jacob2024]) focused on the technical capabilities a GDC API should support (e.g., averaging over polygons, visualizing selected bands). In contrast, the Testbed-20 GDC API usability testing concentrated on the interoperability between implementations.

1.2. Objectives

The specific objectives of the Testbed-20 usability testing were:

- *Understand the shortcomings of the draft GDC API [OGC 24-035] ([Eberle2025]) and the needs of the users to efficiently reach their goals for accessing and processing content maintained in a GDC.
- *Gather feedback on the state of the interoperability of existing GDC API implementations and their current access mechanisms.
- *Derive focus areas, pathways, and concrete advice for the further development of the OGC API – GeoDataCube Standard (e.g., define best practices for STAC metadata creation).

Furthermore, the user needs identified by the referenced studies were addressed as follows:

- *Increase interoperability: Demonstrating that the same access mechanisms work across different cloud backend implementations and that results from different GDC API endpoints can be combined.

*Knowledge gap in cloud technologies: The usability testing was designed to showcase which access mechanisms exist for different GDC API endpoints and how they should be used.

The results of usability testing target expert users (scientists, developers, data scientists): The results of the usability testing are primarily aimed at expert users, who are encouraged to co-design the interoperability of existing GDC API implementations by contributing to best practices.



2 TOPICS

2.1. Usability Testing Setup ---

2.1.1. Subject and Scope of Testing

The initial objective of the Testbed-20 GDC initiative was to standardize access to and processing of GDCs by integrating existing OGC API Standards—including OGC API — Coverages, OGC API — Processes, and openEO—into a unified draft specification: the OGC API — GeoDataCubes (GDC API) Standard. However, the results of the Testbed-20 GDC Task indicated that, rather than developing a new standard, the integration and harmonization of existing implementations of OGC API Standards for accessing and processing GDC content currently present a more viable approach. This approach enables the combination of existing solutions to form a seamless processing chain.

A critical factor in achieving this integration is interoperability, which allows data outputs from one API to be used as inputs for another. At present, this interoperability remains a challenge, as the expected inputs and produced outputs across various solutions are not fully harmonized, preventing seamless interaction. As stated above, “Aligning input and output mechanisms for different OGC API Standard implementation instances inherently creates interoperability by ensuring that different implementations of the same standard can communicate, allowing users to chain and interchange between different API implementation instances (endpoints).”

Consequently, the Testbed-20 GDC usability testing was designed to identify gaps that hinder interoperability between and among existing solutions. This involved a series of integration tests conducted across the available implementations of applicable API Standards. These Testbed-20 participants performed these tests using Jupyter notebooks, which are included in the annex. The findings and conclusions are detailed in the following sections.

The use cases UC-1: Combination and visualization of meteorological and EO data, and UC-2: Basic processing on meteorological and EO data, are covered in the deliverable D020: GDC API Profile Engineering Report OGC 24-035. The provenance of data sources and processing was not explicitly subject to usability testing. Please refer to Testbed-20 GDC Deliverable D144: Provenance OGC 24-036.

2.1.2. Overview of GDC Processing APIs

This section describes existing capabilities to get data in and out of the OGC APIs available in Testbed-20 and openEO. A brief description of these access mechanisms is important since they naturally enable the data exchange between API implementation instances in a processing chain.

Therefore, an understanding of the available methods is essential to facilitate harmonization activities.

2.1.2.1. openEO

This section outlines the existing capabilities for accessing and exchanging data through the OGC APIs available in Testbed-20 and openEO.

2.1.2.1.1. Input

- load_collection
 - “Loads a collection from the current back-end by its id and returns it as a processable data cube. The data that is added to the data cube can be restricted with the parameters `spatial_extent`, `temporal_extent`, `bands` and `properties`.”
 - This is the way openEO implementations load the collections they expose internally (this could be STAC, internal OGC API implementation instances or other formats).
- load_stac (experimental)
 - “Loads data from a static STAC catalog or a STAC API Collection and returns the data as a processable data cube. A batch job result can be loaded by providing a reference to it.”
 - Input parameter URL: “The URL to a static STAC catalog (STAC Item, STAC Collection, or STAC Catalog) or a specific STAC Collection that supports filter items and download assets. This includes batch job results, that are compliant with the STAC specification. For external URLs, authentication details such as API keys or tokens may need to be included in the URL. Batch job results can be specified in two ways:
 - For Batch job results at the same back end, a URL pointing to the corresponding batch job results endpoint should be provided. The URL usually ends with `/jobs/{id}/results` and `{id}` is the corresponding batch job ID.
 - For external results, a signed URL must be provided. Not all back-ends support signed URLs, which are provided as a link with the link relation `canonical` in the batch job result metadata.”
 - The definition of how the STAC-compliant metadata is defined is not available.
- load_url (experimental)
 - “Loads a file from a URL (supported protocols: HTTP and HTTPS).”
 - Input parameter URL: “The URL to read from. Authentication details such as API keys or tokens may need to be included in the URL.”

- Input parameter *format*: “The file format to use when loading the data. It must be one of the values that the server reports as supported input file formats, which usually correspond to the short GDAL/OGR codes.”
- This is a very generic form of transferring data. It could be a possible bridge to ingest data via OGC API — Coverage URLs. Due to the lack of defined parameters it would probably be more promising to define a special process like `load_ogcapi_collection`.
- load_uploaded_files (experimental)
 - “Loads one or more user-uploaded files from the server-side workspace of the authenticated user and returns them as a single data cube.”
 - This option is very user specific and customizable to specific needs.

`load_stac` and `load_url` are potential openEO input mechanisms that could be used to build bridges to other APIs. At the time of the Testbed GDC activity (ending April 2025), both processes were in an experimental stage. `load_stac` supports reading arbitrary STAC input. To be interoperable the process would benefit from a standardized method for describing the STAC items or collections that are fed into the process. This approach would enable the underlying implementation to know how to interpret the data. This capability is not yet in place and needs to be worked on by the community. The same is true for `load_url` which can potentially support reading arbitrary data (e.g. via OGC API — Coverages URLs). A custom openEO process could possibly be implemented to read content via an OGC API — Coverages endpoint. To make such development effort worthwhile, a dedicated focus on how the integration could work in detail is required.

2.1.2.1.2. Output

- save_result
 - “Makes the processed data available in the given file format...”
 - Synchronous processing: Does not provide STAC metadata. The data is sent to the client as a direct response to the request.
 - Asynchronous processing (batch jobs): The data are stored on the back end. STAC-compatible metadata is usually made available with the processed data. This is not required due to legacy reasons.
 - The STAC-compatible metadata (item or collection) is not currently compliant with any metadata standard.
- secondary webservice

- On-demand access to data using other web service protocols.
- This enables defining a processing workflow once, while allowing trigger processing for different areas of interest.
- An implementation of the OGC API — Coverages Standard could be a secondary web service.

`save_result` enables delivering STAC-compatible metadata when used with asynchronous processing. This is not mandatory and currently there is no standardized way of describing the STAC items or collections that are delivered. There are some best practices for STAC metadata generation available that should be taken into account when working towards best practices (e.g., [stac-spec](#), [stac projection extension](#), [odc stac](#))

2.1.2.2. OGC API — Processes

The OGC API — Processes Standard is a multipart standard.

- [Part 1: Core](#) describes the description of processes and the ability submit execution requests for synchronous or asynchronous processing.
- [Part 2: Deploy, Replace, Undeploy](#) supports deploying and managing custom application packages to create new processes which can be executed using implementations of the OGC API — Processes — Part 1 Standard.
- [Part 3: Workflows](#) defines [collection input](#) and [nested processes](#) as additional inputs to processes allowing defining ad-hoc workflows (assemble parameterized processes that are either built-in or already deployed), while also defining [collection output](#) as a mechanism to trigger on-demand processing from implementations of the OGC API [data access mechanisms](#). Implementations supporting the [input/output field modifiers](#) (also defined in Part 3) allow the integration of [OGC Common Query Language \(CQL2\)](#) expressions to perform additional processing of inputs and outputs besides the operations performed by the available processes.

The following subsections describe how data can be ingested into a data cube using an OGC API — Processes endpoint and the processing results can be returned.

2.1.2.2.1. Input

The Processes — Part 1 Standard defines requirements that support the following input types inside execution requests:

- Link to data from as URL in an `href`
- Embedded data as JSON values (base64-encoded for binary data)

Processes — Part 3 adds support for these additional input types:

- Collection Input supports referencing an OGC API collection from which the processing engine can retrieve data as needed using one or more supported data access mechanisms (e.g., OGC API — Coverages, OGC API — Tiles, STAC).
- Outputs from nested processes either from the same local deployment or a remote implementation of the OGC API — Processes Standard.

An OGC API — Processes — Part 3 implementation supporting Collection Input needs to support at least one OGC API data access mechanism as a client.

2.1.2.2.2. Output

The OGC API — Processes — Part 1 Standard defines requirements for returning results either as the payload of the post request for synchronous execution or through links to the output data from a JSON document describing the results.

The OGC API Processes — Part 3 Standard adds support for collection output allowing a client to request results for a specific Area/Time/Resolution/Field of interest triggering processing on-demand. This is done by instantiating a virtual OGC API collection as a result of submitting the execution request via a http post method. Subsequent client operations to retrieve data from that virtual collection using an OGC API data access mechanism (Coverages, Tiles using raw tiled data, DGGS) are no different from accessing a regular OGC API collection. For server-side visualization of the results the virtual collection may support OGC API — Maps or OGC API — Tiles (map tiles) implementations.

An implementation supporting Collection Output needs to support at least one OGC API data access mechanism for its virtual collections.

The more data access mechanisms are implemented (e.g., Coverages, Tiles, DGGS, EDR) for collection input/output (and 2DTMS for Tiles, DGGRS for DGGS), the more interoperable the implementation will be with various clients / servers / deployments (the same data access mechanism, and 2DTMS for Tiles / DGGRS for DGGS, needs to be supported on each end for a successful connection).

2.1.2.3. OGC API — Coverages

The candidate OGC API — Coverages Standard is a multipart Standard.

- Part 1: Core defines the ability to request data for specific Area/Time/Resolution/Fields of interest
- Part 2: Filtering, deriving and aggregating fields enables processing defined by OGC Common Query Language (CQL2) expressions

2.1.2.3.1. Input

In addition to the originating collection associated with the request (/collections/{collectionId}), Part 2 introduces a joinCollections query parameter allowing clients to reference other local or remote OGC API collections as inputs.

2.1.2.3.2. Output

OGC API – Coverages enabled clients make synchronous HTTP GET requests to retrieve data of interest, with servers usually limiting the maximum amount of data which can be returned for a single request. The format of the data cube output response is selected using HTTP content negotiation through an Accept: request header.

2.1.2.4. Available access mechanisms in Testbed-20

Table 1 shows which GDC processing APIs and associated access mechanisms were used in Testbed-20.

Table 1 – Overview of implemented processing APIs, inputs and outputs.

IMPLEM	PROCESSING API			INPUT			OUTPUT			
	OGC API – Processes	open EO API	OGC API – Coverages Part 2	STAC	OGC API – Coverages (collection input)	OGC API – Processes – Part 1 (href or embedded process)	OGC API – Processes – Part 3 (remote process)	STAC (Output)	OGC API – Coverages (collection output)	OGC API – Processes – Part 1
CRIM	##			##	##	##	##		##	##
Ecere	##		##		##	##	##		##	##
Ellipsis Drive	##			##				##		
Eurac		##		##				##	##	
MMS		##		##				##		

2.1.3. Different approaches: STAC and OGC API – Coverages

The overview of the access mechanisms based on openEO and on OGC API Standards show that they function differently. This section explicitly describes these different concepts .

The STAC specification is a common language to describe geospatial information. It describes datasets that exist, for example datasets that reside on a server. The STAC API takes a request (e.g. space, time and bands) and returns the available elements that fall into the criteria and returns them as a list of STAC items, with each item containing one or more asset with an associated URL. Clients wishing to retrieve the actual data must perform additional GET requests on the assets linked from within STAC items, ideally using HTTP range requests for formats such as Cloud-Optimized GeoTIFF (COG) supporting overviews and tiles to retrieve only the portion of interest from the asset(s). In many cases further processing can happen on this list, including mosaicking (combining the list elements into one product) and cropping (cutting out the exact spatial extent of request). This step is often done by software/clients (e.g. odc-stac).

***The draft OGC API – Coverages Standard+** defines requirements for a data access mechanism. In OGC API – Coverages, a coverage is defined as “a function which returns values from its range for any direct position within its domain”. A Coverages client can request data for a specific area, time, resolution and/or field(s) of interest. A client that implements the OGC API – Coverages Standard issues a request that “tells” the backend server how to prepare/process the data according to the request and returns a response as requested. The result returned to the client normally includes metadata describing the output data cube. As a data access mechanism, the API can be used in conjunction with an implementation of OGC API – Processes – Part 3: Workflows for a processing engine to request data as needed from a remote collection (collection input), as well for clients to request data from a processing engine through an OGC API – Coverages compliant endpoint from a virtual collection triggering processing on-demand (collection output).

For considering integration and interoperability between these data access mechanisms (or the APIs) it is important to be aware of this difference. Generally speaking, STAC describes objects that already exist, while an implementation of the OGC API – Coverages Standard creates the requested products specifically on-demand.

2.1.4. Testing Procedure

The interoperability tests carried out depict the current state of interoperability between implementations of the API Standards considered for Testbed-20. The tests focused on their available access mechanisms and excluded any additional interoperability at (processing) API level. The questions to be answered were: Are there connections between the API implementations at an input and output level and can these integration points be used to take data accessed via one API endpoint and pass the data to another API endpoint for additional use of processing?

Within Testbed-20 GDC Task, following were the general combinations of access mechanisms that were tested for interoperability:


- STAC to openEO
- STAC to OGC API – Processes
- OGC API – Coverages to openEO
- OGC API – Coverages to OGC API – Processes

At the implementation level, the combinations shown in Table 2 were tested. The table shows which transfer mechanisms (STAC, OGC Coverages) have been used to ingest data into a processing API enabled process workflow (openEO, OGC APIs) via which functionality and for which specific implementations the interoperability was tested.

Table 2 – Overview of integration tests on the implementation level.

TRANSFER MECHANISM IN	PROCESSING API	SOURCE INPUT	IMPLEMENTATION PROCESSING	FUNCTIONALITY	STATUS
STAC	openEO	Eurac	Eurac	load_stac()	##
STAC	openEO	GEE	Eurac	load_stac()	##
STAC	openEO	CDSE STAC	Eurac	load_stac()	#
STAC	openEO	CDSE processing results	Eurac	load_stac()	#
STAC	openEO	CDSE STAC	CDSE	load_stac()	#
STAC	openEO	CDSE processing results	CDSE	load_stac()	##
STAC	openEO	GEE	CDSE	load_stac()	##
STAC	openEO	Eurac	CDSE	load_stac()	##
STAC	OGC API – Processes	GEE	EllipsisDrive	post	##
STAC	OGC API – Processes	Eurac	EllipsisDrive	post	##
STAC	OGC API – Processes	CDSE	EllipsisDrive	post	#
STAC	OGC API – Processes	Element84	CRIM	custom CWL or Collection_input	##

TRANSFER MECHANISM IN	PROCESSING API	SOURCE INPUT	IMPLEMENTATION PROCESSING	FUNCTIONALITY	STATUS
OGC API – Coverages	openEO	Ecere	CDSE	load_url()	#
OGC API – Coverages	OGC API – Processes	Ecere	CRIM	Collection_input	#
OGC API – Coverages	OGC API – Processes	Ecere	Ecere	Collection_input	##
OGC API – Coverages	OGC API – Processes	Eurac	CRIM	Collection_input	##

 # Copernicus Data Space Ecosystem (CDSE) developers were not part of the Testbed-20. It cannot be guaranteed that their resources (STAC, openEO) have been used exactly as intended.

The notebooks containing the integration tests are available in Annex B.

2.2. Usability Testing Outcomes

2.2.1. Results

The results of the usability testing are described in this section. Results contain the outcomes of the integration tests. Keep in mind that this is a snapshot in time and that many of the implementations are constantly evolving. If developers had been involved in the experiments, some of the errors could probably be handled directly. Nevertheless, the aim of performing these tests was to get a perspective of how a user would interact the deployed solutions as of early 2025.

2.2.1.1. STAC to openEO

- Eurac to Eurac
 - Tested whether a STAC resource created by Eurac's openEO backend can be ingested into Eurac's openEO backend via `load_stac`.
 - Tested the interoperability within a single openEO implementation.
 - Since there is no standard for how STAC-compatible metadata created by openEO should look, it is not a given that the same backend can ingest its own resulting STAC resources.
 - The data loaded correctly. ##

- The implementation of `load_stac` needs the `band` parameter to be specified to work. Complete STAC collections cannot yet be loaded.
- Google Earth Engine (GEE) to Eurac
 - Tested whether a STAC resource created by GEE's openEO backend can be ingested into Eurac's openEO backend via `load_stac`.
 - Tested the interoperability between two independent openEO implementations.
 - The data loaded correctly. ##
 - Before this workflow was successful, iterations on the format of the STAC collection were performed. This shows the need for aligned STAC metadata best practices.
- CDSE STAC to Eurac
 - Tested whether a STAC resource from CDSE's STAC catalogue can be ingested into Eurac's openEO backend via `load_stac`.
 - Tested the interoperability between two independent openEO implementations.
 - Two URLs to different STAC Catalogs exposed via CDSE were tested.
 - The data could not be loaded. #
 - Error: Failed to extract cube metadata from STAC URL `https://stac.dataspace.copernicus.eu/v1/collections/sentinel-2-l1c/items/S2B_MSIL1C_20250202T103149_N0511_R108_T33VVG_20250202T122435 ...` `ValueError: Invalid band name/index 'B04'. Valid names: []`
- CDSE processing results to Eurac
 - Tested if a STAC resource created by CDSE's openEO backend can be ingested into Eurac's openEO backend via `load_stac`.
 - tested the interoperability between two independent openEO implementations.
 - The data could not be loaded. #
 - Error: `OpenEoApiError: [500] 500: No converter for [class org.openeo.spring.model.Error] with preset Content-Type 'null'`
- CDSE STAC to CDSE
 - Tested whether a STAC resource from CDSE's STAC catalogue can be ingested into CDSE's openEO backend via `load_stac`.
 - Tested the interoperability within a single openEO implementation.

- Since there is no standardized specification of how STAC-compatible metadata created by openEO is structured, it is not given, that the same backend can ingest the STAC resources it has created itself.
- The data could not be loaded. #
- Error: Failed to extract cube metadata from STAC URL <https://catalogue.dataspace.copernicus.eu/stac/collections/SENTINEL-2> ... ValueError: Invalid band name/index 'B04'. Valid names: []
- CDSE processing results to CDSE
 - Tested whether a STAC resource created by CDSE's openEO backend can be ingested into CDSE's openEO backend via `load_stac`.
 - Tested the interoperability within a single openEO implementation.
 - The data was loaded correctly. ##
- GEE to CDSE
 - Tested whether a STAC resource created by Eurac's openEO backend can be ingested into CDSE's openEO backend via `load_stac`.
 - Tested the interoperability between two independent openEO implementations.
 - The data could not be loaded. #
 - Error: OpenEoApiPlainError: [502] Bad Gateway
- Eurac to CDSE
 - Tested whether a STAC resource created by Eurac's openEO backend can be ingested into CDSE's openEO backend via `load_stac`.
 - Tested the interoperability between two independent openEO implementations.
 - The data was loaded correctly. ##

2.2.1.2. STAC to OGC API – Processes

- GEE to Ellipsis Drive
 - Tested whether if a STAC resource created by GEE's openEO backend can be ingested into Ellipsis Drive's OGC Processes implementation via a post method.
 - Ellipsis Drive reads STAC assets and creates the metadata necessary for reading on-the-fly, it does not rely on the STAC metadata. Ellipsis Drive needs STAC items, STAC collections cannot be read directly.

- The data loaded correctly. STAC metadata was created for the results. ##
- Eurac to Ellipsis Drive
 - Tested whether if a STAC resource created by Eurac's openEO backend can be ingested into Ellipsis Drive's OGC Processes implementation via a post method.
 - The Eurac STAC metadata URL had to be extended to the item level.
 - The data loaded correctly. STAC metadata was created for the results. ##
- CDSE to Ellipsis Drive
 - Tested whether a STAC resource from CDSE's STAC catalogue can be ingested into Ellipsis Drive's OGC Processes implementation via a post method.
 - The Eurac STAC metadata URL had to be extended to the item level.
 - The data could not be loaded. #
 - Error: ValueError: {'status': 'error', 'id': 'ad483d51-c777-4eac-ae18-5ec263a2316d', 'output': 'invalid stac url'}
- Element84 to Computer Research Institute of Montreal (CRIM)
 - Tested whether a STAC resource from Element84's STAC catalogue can be ingested into CRIM's OGC API – Processes implementation via `Collection_input` or a custom-made CWL process.
 - The echo-files process serves as a simple passthrough operation. It shows how the collection is resolved as input, and for which each resolved STAC Item within the search criteria is forwarded to the process. After that point, it depends on the process CWL definition how to employ those STAC Items.
 - The data loaded correctly. The data were used in the Vegetation Health Index (VHI) use case. ##

2.2.1.3. OGC API – Coverages to openEO

- Ecere to CDSE
 - Tested whether a OGC Coverage created by Ecere's OGC API can be ingested into CDSE's openEO backend via `load_url`.
 - Tested the interoperability between an OGC API – Coverages implementation and openEO.
 - The data could not be loaded. #

- The `load_url` process from openEO is very generic and not trimmed to read the response package from an OGC API – Coverages endpoint.

2.2.1.4. OGC API – Coverages to OGC API – Processes

- Ecere to CRIM
 - Tested whether an OGC Coverage created by Ecere's OGC API can be ingested into CRIM's OGC API – Processes implementation via `collection_input`.
 - Tested the interoperability between an OGC API – Coverages implementation and an OGC API – Processes implementation.
 - The data could not be loaded. #
 - There is a downstream dependency issue in the library OWSLib that CRIM is currently fixing ([OWSLib pull request](#), [Weaver client pull request](#)).
- Ecere to Ecere
 - Tested whether an OGC Coverage created by Ecere's OGC API can be ingested into a separate implementation of Ecere's OGC API – Processes via `collection_input`.
 - Tested the interoperability between an OGC API – Coverages implementation and OGC API – Processes implementation.
 - The data loaded correctly. ##
 - This capability was demonstrated in OGC Testbed-19 [Section 6.1. Ecere Technology Integration Experiments](#) (Jacob et al., 2024).
- Eurac to Ecere
 - Tested whether a OGC Coverage created by Eurac's OGC API can be ingested into Ecere's OGC API – Processes implementation via `collection_input`.
 - Tested the interoperability between OGC API – Coverages and OGC API – Processes.
 - The data was loaded correctly. ##
 - This capability was demonstrated in OGC Testbed-19 [Section 6.1. Ecere Technology Integration Experiments](#) (Jacob et al., 2024).

2.2.2. Discussion

An increasing number of cloud platforms are emerging. To use these platforms efficiently, the interoperability between them must be improved. The aim of the usability integration tests

was to demonstrate how users interact with different GDC API implementations and how they would transfer data between these implementations.

The general state of interoperability between GDC API implementations is still low. There are no standardized transfer mechanisms between the implementations of the processing APIs openEO and OGC API – Processes and implementations of the access mechanisms STAC and OGC API – Coverages. Naturally, openEO uses STAC and implementations of OGC API Standards use OGC API – Coverages. When using two different implementations of the same processing API, it is not a given that the output of another implementation of the same API Standard can read the produced data. When switching between different APIs this interoperability gap increases (for example between implementations of openEO to OGC API – Processes). There are custom solutions that show that interoperability between the two solutions is technically feasible. Furthermore, there are first efforts to align processes such as openEO's `load_stac`.

There are many examples of widely used STAC collections (e.g., Sentinel-2 by Element84 or Microsoft Planetary Computer). Unfortunately, their STAC metadata is not aligned. Therefore developing generic tools that read from these collections is difficult. The challenges encountered with some of these STAC collections were described in the Testbed-20 API Profile Engineering Report [Section Annex D.2. Issues encountered accessing STAC API deployments](#) (Eberle et al., 2025).

STAC to openEO: Of the ten experiments that were conducted, five resulted in successful workflows. This shows that the process `load_stac` has good potential to increase interoperability between different openEO implementations. Since STAC metadata can contain a plethora of different information, implementing processes that read arbitrary STAC metadata is difficult. Therefore, the implementations of `load_stac` requires making some assumptions of what STAC metadata to expect. Currently, changing the code (e.g., filtering for bands or not) can break the process or implementations can be optimized to tread their own STAC metadata. As a consequence, more work needs to be dedicated to standardize the way STAC metadata is provided. In the future, there are more STAC Catalogs and implementations that can be tested (e.g., RadiantEarth, Destination Earth, Microsoft Planetary Computer, CDSE, openEO platform).

The STAC specification defines a widely used metadata description. The agreement of how the metadata is described within STAC should be further aligned for interoperable machine readability. There are some examples of best practices for STAC metadata generation that show how software can be developed that is able to read STAC metadata in a consistent and interoperable manner.

STAC to OGC API – Processes: Four experiments were conducted to test the interoperability between STAC implementations and OGC API – Processes implementations. Three experiments were successful and one failed. In Testbed-20 the implementations of how to read STAC content from an OGC API – Processes endpoint are very different. Ellipsis Drive's solution relies largely on the asset present in the STAC metadata to extract the crucial metadata by reading from the file. CRIM tested two solutions:

1. Employ a custom-made CWL process, such as `select-products-sentinel2`, that performs the necessary requests and response parsing. This can be useful if additional operations unsupported by built-in strategies are required.
2. Employ a OGC API – Processes `Collection_input`, which knows how to interact with some data-access mechanisms. This is useful to quickly chain processes

together in a nested fashion but has more limited capabilities than a custom script.

Since it is a common use case to read from publicly available STAC catalogs, there are probably more implementations that are developing custom solutions.

OGC API – Coverages to openEO: One experiment was conducted to test the interoperability between an OGC API – Coverages implementation and an openEO implementation. OpenEO's `load_url` was used, though it is probably not the best way to achieve interoperability. A more promising solution would be to design a new openEO process `load_ogcapi_collection`.

OGC API – Coverages to OGC API – Processes: One experiment was conducted to test the interoperability between an OGC API – Coverages implementation and an OGC API – Processes implementation. CRIM is actively working on solutions to read OGC API Coverages response packages. Furthermore, there were Testbed-19 tests (Jacob et al., 2024) between Eurac's OGC API – Coverages endpoint and Ecere's OGC API – Processes implementation. Ecere also demonstrated interoperability on two backend independent implementations.

2.2.3. Recommendations

The following recommendations are based on discussions of the results of the above experiments. The recommendations suggest a way forward for the future development in the field of interoperable API implementations for GeoDataCube content access and processing.

1. Start by aligning data access mechanisms between GDC APIs
 - Currently the evidence from the experiments suggest that aligning existing GDC API Standards into one generic GDC API Standard will be difficult.
 - Therefore, the current focus should be put on enabling the existing GDC focused API implementations to easily exchange data.
 - Doing so will increase interoperability for users.
2. Define STAC best practices
 - STAC is a widely used language to define metadata.
 - The current challenge is to align *how* this metadata is described. Some initial examples exist (e.g., [stac-spec](#), [stac projection extension](#), [odc stac](#)).
 - This will enable developers to develop generic access mechanisms, which will increase interoperability between solutions.
3. Develop access mechanism bridges

3.1. Define an openEO process to ingest a response package from an OGC API – Coverages endpoint.

- Currently there is no specific processes to read a response package from an OGC API — Coverages endpoint.
- The process `load_url` seems to be too generic.
- A specific openEO process should be defined (e.g., `load_ogcapi_collection`)

3.2. Define a generic solution for how STAC metadata resources can be ingested into an OGC API — Processes workflow.

- Define OGC API — Processes Collection Input for STAC Collection/API

3.3 openEO outputting OGC API — Coverages through secondary services

- OpenEO supports being able to set up secondary services.
- How can an implementation of OGC APIs / WxS enabled applications such as GeoServer “pull” on openEO to execute a process graph on-demand?
- Ideally, connect multiple different access mechanisms (e.g., OGC API — Coverages, Tiles, Maps; WMTS, WMS) to openEO backends. This could be a worthwhile effort to try to define a standard set of parameters for data servers (e.g., GeoServer) to pull raw data from openEO backends (i.e., a “multiple” secondary service type) or allow reuse of the same process graph with different access mechanisms.



3

OUTLOOK

This Testbed 20 GDC usability and integration testing was a starting point to capture the status of interoperability between different GDC API implementations. The next steps should adopt the lessons learned and further concentrate on the interoperability of data rather than harmonization of the deployed API implementations. These are the initial ideas for how to increase interoperability. These should be further jointly developed, implemented, and showcased.

A potential set-up for further activities would benefit from:

- openEO servers:
 - Ability to ingest data from OGC API instances `load_ogcapi_collection`; and
 - Ability to set up a Web Service pulling raw data from openEO process graphs, supporting one or more OGC API endpoint access mechanisms.
- OGC API Data Servers (Coverages, Tiles and/or DGGS):
 - possibly with additional implementation of STAC at `/collections/{collectionId}/items`; and
 - possibly with Coverages Scenes prototype at `/collections/{collectionId}/scenes`.
- OGC API Processing Servers:
 - Ability to ingest data from an OGC API structured collection (“Collection Input”); and
 - Ability to set up virtual collections (supporting Coverages, Tiles and/or DGGS).

With the rise of Earth Science Services (ESS) cloud platforms and the growing number of projects centered around them (such as EOEPKA+, APEX, and Destination Earth), it is becoming increasingly important to track the decisions made within these projects and related communities (e.g., OGC API Standards, Pangeo, openEO) and to foster greater collaboration among them.

A key scenario that underscores the need for cooperation is the integration of Zarr with STAC. While there have been discussions about treating Zarr as a STAC Catalog, no consensus has been reached. Currently, Destination Earth Collections are served as Zarr files, with STAC metadata provided only at the collection level. More significantly, the new Copernicus data format, EOPF, will rely on Zarr for the Sentinel missions. The EOPF Sample Service project will need to determine how Zarr files should be described in STAC metadata down to the asset level.

To prevent further fragmentation—and given the widespread use of Sentinel data—it is essential to align the STAC best practices mentioned above with the decisions made by the EOPF Sample Service project.



4

SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS

4

SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS

During the course of this project, a thorough review was conducted to identify any potential security, privacy, and ethical concerns. After careful evaluation, it was determined that none of these considerations were relevant to the scope and nature of this project. Therefore, no specific measures or actions were required in these areas.



BIBLIOGRAPHY





BIBLIOGRAPHY

- [1] Wilkinson, M. et al. (2016) 'The FAIR Guiding Principles for scientific data management and stewardship', Sci Data 3, 160018. doi: <https://doi.org/10.1038/sdata.2016.18>.
- [2] Sudmanns, M. et al. (2019) 'Big Earth data: disruptive changes in Earth observation data management and analysis?', International Journal of Digital Earth, 13(7), pp. 832–850. doi: 10.1080/17538947.2019.1585976.
- [3] Wagemann, J. et al. (2021) 'A user perspective on future cloud-based services for Big Earth data', International Journal of Digital Earth, 14(12), pp. 1758–1774. doi: 10.1080/17538947.2021.1982031.
- [4] Wagemann, J. et al. (2021) 'Users of open Big Earth data – An analysis of the current state', Computers & Geosciences, Volume 157, 104916, doi: <https://doi.org/10.1016/j.cageo.2021.104916>.
- [5] Di Leo, M. et al. (2023) 'DIGITAL EARTH OBSERVATION INFRASTRUCTURES AND INITIATIVES: A REVIEW FRAMEWORK BASED ON OPEN PRINCIPLES', Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., XLVIII-4/W7-2023, 33–40, doi: <https://doi.org/10.5194/isprs-archives-XLVIII-4-W7-2023-33-2023>.
- [6] Di Leo, M. et al. (2024) 'Self-Assessment Framework for Earth Observation Platforms from User Experience', Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., XLVIII-4/W12-2024, 43–49, doi: <https://doi.org/10.5194/isprs-archives-XLVIII-4-W12-2024-43-2024>.
- [7] Jacob A., et al. (2024) 'OGC 23-047: OGC Testbed-19 GeoDataCubes Engineering Report', <http://www.opengis.net/doc/PER/T19-D011>.
- [8] Zellner, P. J., et al. (2024) 'MOOC Cubes and Clouds – Cloud Native Open Data Sciences for Earth Observation', Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., XLVIII-4/W12-2024, 157–162, doi: <https://doi.org/10.5194/isprs-archives-XLVIII-4-W12-2024-157-2024>.
- [9] Eberle J., et al. (2025) 'OGC 24-035: OGC Testbed 20 GeoDataCube (GDC) API Profile Report', <http://www.opengis.net/doc/PER/t20-D020>.
- [10] Eberle J., et al. (2025) 'OGC 24-036: OGC Testbed 20 GDC Provenance Demonstration Report', <http://www.opengis.net/doc/PER/t20-D022>.



ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS



ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

API	Application Programming Interface
CDSE	Copernicus Dataspace Ecosystem
CEOS	Committee on Earth Observation Satellites
CRIM	Computer Research Institute of Montreal
ESS	Earth System Sciences
GDC	GeoDataCube
GEE	Google Earth Engine
STAC	SpatioTemporal Asset Catalog



ANNEX B (NORMATIVE) INTEGRATION TESTS

B

ANNEX B (NORMATIVE) INTEGRATION TESTS

Below is the code that was used to test the interoperability between the different access mechanisms and GDC processing API implementations. In case of failure, the respective error messages were captured in the results section.

B.1. STAC to openEO

The following URLs to STAC metadata resources have been tested.

```
url_eurac = "link:++https://stac.eurac.edu/collections/SENTINEL2_L2A_SAMPLE_2+  
+[]" "  
url_gee = "link:++https://earthengine.openeo.org/v1.0/results/c08dc17428fde51ea7e  
1332eec2abd06e74188924e6c773257b4fb00aee0a308++[]" "  
url_cdse = "link:++https://stac.dataspace.copernicus.eu/v1/collections/sentinel-  
2-l1c/items/S2B_MSIL1C_20250202T103149_N0511_R108_T33VVG_20250202T122435++[]" "  
url_cdse = "link:++https://stac.dataspace.copernicus.eu/v1/collections/sentinel-  
2-l1c++[]" "  
url_cdse = "link:++https://catalogue.dataspace.copernicus.eu/stac/collections/  
SENTINEL-2++[]" "
```

Listing B.1

The following backends were tested. Both need authentication.

Eurac's development openEO backend.

```
conn = openeo.connect('link:++https://dev.openeo.eurac.edu/++[]')  
conn.authenticate_basic(username="", password="")
```

Listing B.2

CDSE's openEO backend.

```
conn = openeo.connect('link:++https://openeo.dataspace.copernicus.eu/++[]')  
conn.authenticate_oidc()
```

Listing B.3

This code snippet was used to test the interoperability between the STAC metadata and the openEO implementations mentioned above.

```
# import necessary libs  
import openeo
```

```

import xarray as xr

# create an openEO process graph for loading the data
url = url_eurac # set to one of the URLs above.
band = ['B04'] # can be left empty if the full range of bands should be loaded
(not yet supported by some implementations)
cube = conn.load_stac(url = url, bands = band)

# download directly to variable and check if bits arrive
res = cube.download(format="netCDF")
res[0] # check if bits arrive

# download to file
res = cube.download("res.nc", format="netCDF")
tst = xr.open_dataset("res.nc")

# inspect the result
tst

```

Listing B.4

B.2. STAC to OGC API – Processes

B.2.1. Ellipsis Drive

The following code snippets were used to test the interoperability between STAC metadata and the Ellipsis Drive OGC API – Processes implementation.

The same URLs as for STAC to openEO were tested.

```

url_eurac = "link:++https://stac.eurac.edu/collections/SENTINEL2_L2A_SAMPLE_2+
+[]"
url_gee = "link:++https://earthengine.openeo.org/v1.0/results/c08dc17428fde51ea7e
1332eec2abd06e74188924e6c773257b4fb00aee0a308++[]"
url_cdse = "link:++https://stac.dataspace.copernicus.eu/v1/collections/sentinel-
2-l1c/items/S2B_MSIL1C_20250202T103149_N0511_R108_T33VVG_20250202T122435++[]"
url_cdse = "link:++https://stac.dataspace.copernicus.eu/v1/collections/sentinel-
2-l1c++[]"
url_cdse = "link:++https://catalogue.dataspace.copernicus.eu/stac/collections/
SENTINEL-2++[]"

```

Listing B.5

The libraries and functions used.

```

import requests
import urllib
import json
import time
import ellipsis as el

def get_call(endpoint, token=None, body=None, headers={}):
    call = url + endpoint
    if type(body) != type(None):

```

```

        for k in body.keys():
            if type(body[k]) != type('x'):
                body[k] = json.dumps(body[k])

        body = urllib.parse.urlencode(body)
        call = call + '?' + body
        #print(f"endpoint: {call}")
        if token is not None:
            headers['Authorization'] = f"Bearer {token}"
        return requests.get(call, headers=headers).json()

def post_call(endpoint, token=None, body=None, headers={}):
    call = url + endpoint
    #print(f"endpoint: {call}")
    if token is not None:
        headers['Authorization'] = f"Bearer {token}"
    return requests.post(call, json=body, headers=headers)

```

Listing B.6

Authentication is needed.

```
token = el.account.logIn('', '')
```

Listing B.7

The actual test. The process is loading from the STAC metadata and multiplying by two. Then a new STAC item is created as the result.

```

processId = 'ffbc47fd-0777-4b92-bf99-62e37aac3ed3' # multiply by 2

body = {'stacUrl': url_gee} # change to accroding URL here

while True:
    time.sleep(1)
    print('waiting for job to finish')
    job = get_call('/ogc/processes/jobs', token=token)[-1]
    if job['status'] != 'processing':
        break

if job['status'] == 'error':
    raise ValueError(job)
else:
    print('created new STAC item: link:++https://api.ellipsis-drive.com/v3/ogc/stac/collection/' + job['output'])
    url_ed_stac = 'link:++https://api.ellipsis-drive.com/v3/ogc/stac/collection/' + job['output']
    while el.path.get(pathId= job['output'], token=token)['raster']['timestamps'][0]['status'] == 'activating':
        time.sleep(1)
        print('waiting for visualisation to finish')
    print('you can view the result here: link:++https://app.ellipsis-drive.com/view?pathId=' + job['output'])

```

Listing B.8

B.2.2. CRIM

The following code snippets were used to test the interoperability between STAC metadata and the CRIM OGC API — Processes implementation. These snippets are excerpts from the full VHI workflow CRIM produced. Two approaches were tested.

In order to list available items in the STAC collection that match the search criteria, there are two possible approaches.

1. Employ a custom-made CWL process, such as `select-products-sentinel2`, that performs the necessary requests and response parsing. This can be useful if additional operations unsupported by built-in strategies are required.
2. Employ a OGC API — Processes collection input, which knows how to interact with some data-access mechanisms. This is useful to quickly chain processes together in a nested fashion, but has more limited capabilities than a custom script.

Only the second approach is captured here, since in terms of general interoperability it is more promising.

OGC API — Processes collection approach: For this demo, CRIM employed the `echo-files` process that is a simple passthrough operation. This way, how the collection was resolved as input could be observed, and for which each resolved STAC Item within the search criteria was forwarded to the process. After that point, it would depend on the process CWL definition how to employ those STAC Items.

```
result = weaver_client.execute(
    process_id="echo-files",
    inputs={
        "files": {
            # in this case, we interact directly with the STAC API
            # therefore, search parameters must be provided following its
            interface
                "collection": IMAGERY_COLLECTION_URL,
                "datetime": "2020-01-01T00:00:00Z/2020-01-02T00:00:00Z", # smaller
            range just for demonstration
                "bbox": BBOX,
                # guide Weaver about the collection access-mechanism and retrieve
            only items, not assets
                "format": "stac-items",
                # GeoJSON is enforced since requesting STAC Items directly (rather
            than underlying Assets)
                # however, this could be used to retrieve Assets by media-type in
            the context of another process
                "type": "application/geo+json",
            }
        }
    }
```

```
result
```

Listing B.9

```
OperationResult(success=True, code=201, message="Job successfully submitted to  
processing queue. Execution should begin when resources are available.")  
{  
    "description": "Job successfully submitted to processing queue. Execution  
should begin when resources are available.",  
    "jobID": "67c49a19-d4dd-4bd4-b88a-1e0fe9766d81",  
    "processID": "echo-files",  
    "status": "accepted",  
    "location": "link:++https://hirondelle.crim.ca/weaver/processes/echo-files/  
jobs/67c49a19-d4dd-4bd4-b88a-1e0fe9766d81++[]"  
}
```

Listing B.10

Below is the list of resolved STAC Items after resolution of the collection input. Using CWL, it would then be possible to process the resolved STAC Items into the relevant data calculations according to available bands.

```
weaver_client.results(result.headers["Location"])
```

Listing B.11

```
OperationResult(success=True, code=200, message="Listing job results.")  
{  
    "files": [  
        {  
            "href": "link:++https://hirondelle.crim.ca/wpsoutputs/weaver/users/18/  
67c49a19-d4dd-4bd4-b88a-1e0fe9766d81/files/S2B_33TUL_20200101_1_L2A.geojson++[]",  
            "type": "application/geo+json"  
        },  
        ..  
        {  
            "href": "link:++https://hirondelle.crim.ca/wpsoutputs/weaver/users/18/  
67c49a19-d4dd-4bd4-b88a-1e0fe9766d81/files/S2B_33TXN_20200101_0_L2A.geojson++[]",  
            "type": "application/geo+json"  
        }  
    ]  
}
```

Listing B.12

B.3. OGC API Coverages to openEO

Testing interoperability between the Ecere OGC API Coverages/Processes backends output Coverage URL fed into CDSE's openEO backend using the function `load_url()`.

```
conn = openeo.connect('link:++https://openeo.dataspace.copernicus.eu/++[]')  
conn.authenticate_oidc()
```

```
coverages_url = "link:++https://maps.gnosis.earth/ogcapi/collections/T20-VHI:  
MonthlyRef_2017_2020:VHI_2020/coverage.tiff++[]"  
cube = conn.load_url(url = coverages_url, format = "GTiff")
```

```

res = cube.download(format = "GTiff")

res = cube.download("res.tif", format = "GTiff")
tst = xr.open_dataset("res.tif")

# inspect the result
tst

```

Listing B.13

B.4. OGC API – Coverages to OGC API – Processes

The following code snippets were used to test the interoperability between Eceres OGC API – Coverages endpoint and the CRIM OGC API – Processes implementation.

Install the CRIM clients.

```

%pip install git+https://github.com/crim-ca/weaver
%pip install git+https://github.com/Ouranosinc/requests-magpie

```

Listing B.14

Load libraries.

```

import os
from requests_magpie import MagpieAuth
from weaver.cli import WeaverClient

```

Listing B.15

Authenticate.

```

MAGPIE_USERNAME = os.getenv("MAGPIE_USERNAME")
MAGPIE_PASSWORD = os.getenv("MAGPIE_PASSWORD")
MAGPIE_URL = f"{CRIM_SERVER_URL}/magpie"
MAGPIE_AUTH = MagpieAuth(MAGPIE_URL, MAGPIE_USERNAME, MAGPIE_PASSWORD)

WEAVER_URL = "link:++http://localhost:4002++[]"
WEAVER_CLIENT = WeaverClient(WEAVER_URL)#, auth=MAGPIE_AUTH)

```

Listing B.16

Settings for the OGC API – Coverages request from Ecere.

```

ECERE_SERVER_URL = "link:++https://maps.gnosis.earth/ogcapi++[]"
ECERE_COLLECTION_ID = "sentinel2-l2a"
ECERE_COVERAGE_SUBSET = {"Lat": [46.45, 46.50], "Lon": [11.30, 11.35]}
ECERE_COVERAGE_DATETIME = "2019-05-01/2019-05-31"
ECERE_COVERAGE_PROPERTIES = ["B08"]
ECERE_COVERAGE_PARAMETERS = {"scale-factor": 2}

```

Listing B.17

Define plotting function and settings.

```

def hex2rgb(color_hex):
    color_hex = color_hex.lstrip('#')

```

```

        return tuple(int(color_hex[i:i+2], 16) for i in (0, 2, 4))

COLOR_LEVELS = [0, 1000, 2000, 3000, 4000, 5000]
COLOR_HEX = ['#03051a', '#4c1d4b', '#a3195b', '#e8403e', '#f69b71', '#faebdd']
COLOR_RGB = [hex2rgb(color) for color in COLOR_HEX]
COLOR_PLOT = [[level, color] for level, color in zip(COLOR_LEVELS, COLOR_RGB)]

```

Listing B.18

Execute the request and processing. It is selecting specific bands and plotting the map.

```

result = WEAVER_CLIENT.execute(
    "plot-image",
    inputs={
        "input_image": {
            "collection": f"{ECERE_SERVER_URL}/collections/{ECERE_COLLECTION_
ID}",
            "format": "ogc-coverage-collection",
            "type": "image/tiff; application=geotiff",
            "subset": ECERE_COVERAGE_SUBSET,
            "datetime": ECERE_COVERAGE_DATETIME,
            "properties": ECERE_COVERAGE_PROPERTIES,
            **ECERE_COVERAGE_PARAMETERS,
        },
        "color_scale": str(COLOR_PLOT),
        "plot_title": "sentinel-2 Near-Infrared (May 2019)",
    },
    monitor=True,
    timeout=30,
    interval=5,
)

result

```

Listing B.19