International Journal of Digital Earth

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tjde20

Brazil Data Cube Workflow Engine: a tool for big Earth observation data processing

Vitor C. F. Gomes, Gilberto R. Queiroz, Karine R. Ferreira, Edzer Pebesma & Claudio C. F. Barbosa

To cite this article: Vitor C. F. Gomes, Gilberto R. Queiroz, Karine R. Ferreira, Edzer Pebesma & Claudio C. F. Barbosa (2024) Brazil Data Cube Workflow Engine: a tool for big Earth observation data processing, International Journal of Digital Earth, 17:1, 2313099, DOI: 10.1080/17538947.2024.2313099

To link to this article: https://doi.org/10.1080/17538947.2024.2313099

Published online: 09 Feb 2024.

INTERNATIONAL JOURNAL OF DIGITAL EARTH
2024, VOL. 17, NO. 1, 2313099
https://doi.org/10.1080/17538947.2024.2313099

Brazil Data Cube Workflow Engine: a tool for big Earth observation data processing

Vitor C. F. Gomes[a], Gilberto R. Queiroz[b], Karine R. Ferreira[b], Edzer Pebesma[c] and Claudio C. F. Barbosa[b]

[a]C4ISR Division, Institute for Advanced Studies (IEAv), São José dos Campos, Brazil; [b]Earth Observation and Geoinformatics Division, National Institute for Space Research (INPE), São José dos Campos, Brazil; [c]Institut für Geoinformatik, Westfälische Wilhelms-Universität, Münster, Germany

ABSTRACT

Earth Observation (EO) satellites have produced vast image collections that are freely accessible to society. However, handling these images often surpasses the capabilities of traditional hardware and software for EO data storage and processing, posing challenges for traditional Spatial Data Infrastructure (SDI). To overcome these challenges, innovative cloud computing and distributed systems have been developed, such as matrix databases, MapReduce systems, and web services. These technologies are now integrated into leading-edge platforms, forming a new generation of SDI for big EO data. These platforms have different characteristics in terms of governance, technologies, data access, infrastructure abstractions, data processing, and flexibility to extend their functionality. Our work contributes to the area of SDI for big EO data by proposing a server-side data-processing tool called Brazil Data Cube Workflow Engine (BDC-WE), based on workflow orchestration approach. BDC-WE provides a high-level interface using the openEO API for big EO data accessing and processing, allowing SDI maintainers to easily describe sequences of processes and integrate new algorithms. The architecture proposed in this study was implemented and the prototype was evaluated in two case studies described in this paper.

ARTICLE HISTORY
Received 12 September 2023
Accepted 25 January 2024

KEYWORDS
Big data; Earth observation data; spatial data infrastructure; openEO; workflow orchestration

1. Introduction

Earth observation (EO) data is crucial to map and comprehend the processes that occur on our planet, promoting significant advancements in monitoring environmental changes, risk detection, urban occupation, surface temperature, and food security (Brown 2016; Kamali Maskooni et al. 2021). EO data sets are important sources to measure global indicators of United Nations' Sustainable Development Goals (SDGs), including Indicator 15.3.1 on land degradation (Giuliani et al. 2020) or Indicators 11.3 and 11.7 on land use and land cover (Ekim and Sertel 2021). By extracting information from EO data, researchers and policymakers can formulate and implement effective policies for protecting the environment and managing natural resources.

Satellite observations and geospatial data are being produced and shared at an unprecedented rate with petabyte production on a daily basis (Soille et al. 2018). Storing, processing, and analyzing

CONTACT Vitor C. F. Gomes vitorvcfg@fab.mil.br, vconrado@gmail.com Institute for Advanced Studies, Trv Cel Av Jose A. A. do Amarante 01, São José dos Campos, SP, 12228-001, Brazil

these vast datasets pose significant technological challenges that limit the ability of EO scientists to take advantage of their potential. These datasets often exceed the storage, processing, and memory capacities of personal computers, leading users to utilize only a fraction of the available data for scientific research and operational applications (Camara et al. 2016; Müller, Bernard, and Brauner 2010; Xu et al. 2022). Thus, novel technological solutions are required to adequately store, process, disseminate, and analyze these big EO datasets.

Spatial Data Infrastructure (SDI) provides an environment that fosters the use, management, and production of geospatial data by allowing people and systems to interact with technology (Masser 2019). In recent years, SDIs have implemented technological components that adopt the standards proposed by the Open Geospatial Consortium (OGC) and International Organization for Standardization (ISO) to store, represent, disseminate, and process geospatial data. However, most current SDIs primarily focus on sharing and disseminating EO data as individual files through web portals and various protocols, such as HTTP, FTP, and SSH (Müller 2016).

1.1. Related big EO processing solutions

In the context of big EO data, managing, processing, and disseminating this enormous amount of data poses significant challenges for SDIs. This scenario demands more structured and precise research services, automated acquisition, spatiotemporal indexes, calibration, and availability processes, as well as the ability to process data sets in the server-side, without needing to move them across the network (Camara et al. 2016; Xu et al. 2022).

To address these challenges, the EO community has developed new technologies in the form of platforms for big EO data. Serving as computational solutions, they offer a range of functionalities for managing, storing, and accessing extensive EO data. These platforms allow server-side processing, eliminating the need to download massive amounts of EO datasets. In addition, they provide a certain level of data and processing abstractions that are useful to EO community users and researchers (Gomes, Queiroz, and Ferreira 2020). The integration of different types of technologies, Application Programming Interfaces (APIs), and web services results in a more comprehensive solution for managing and analyzing extensive EO data. Examples of platforms for big EO data are Open Data Cube (ODC) (Killough 2018), Google Earth Engine (GEE) (Gorelick et al. 2017), JRC Earth Observation Data and Processing Platform (JEODPP) (Soille et al. 2018), Sentinel Hub (SH) (Milcinski and Kolaric 2023), pipsCloud (Wang et al. 2018), SEPAL (FAO 2023) and openEO platform (Schramm et al. 2021). They adopt different data abstractions, standards, or technological solutions despite their similar functionalities.

Gomes, Queiroz, and Ferreira (2020) performed a review and comparative analysis of these platforms in relation to ten capabilities, including governance, infrastructure, data and processing abstractions, and extensibility. They pointed out that the greater the degree of abstraction delivered to the scientist, the greater the difficulty in providing flexibility in data-processing approaches. Platforms for big EO data need layers of abstractions that enable both data scientists and data production staff to express computations that exploit available computational resources. One possible alternative would be to provide scientists with a platform that provides two ways to perform server-side data processing. In the first form, an API with a high-level abstraction is made available for scientists to describe their analyzes in a manner equivalent to that provided by GEE or openEO. The second way allows new algorithms to be added to the platform. These algorithms would directly access the data and take advantage of the distributed processing capabilities provided by the platform.

In the Brazilian context, the Brazil Data Cube (BDC) project is an initiative of the National Institute for Space Research (INPE) to develop a platform for big EO data management and analysis. This project is producing 2 petabytes of Analysis-Ready Data (ARD) and multidimensional EO data cubes of satellite images Landsat-8/-9, Sentinel-2, CBERS-4/-4A and Amazonia for the entire

Brazilian territory (Ferreira et al. 2022, 2020). Besides that, it is developing a platform called Brazil Data Cube with services and tools to create, access, and analyze EO data cubes.

Currently, most of the ARD and EO data cubes of the BDC project are produced using two applications developed by the project team, *BDC Collection Builder* (Marujo et al. 2022) and *BDC Cube Builder* (Ferreira et al. 2022). These applications are configured by maintainers to discover and retrieve scenes from external providers of EO datasets, to index them in collections, to produce ARD and EO data cubes. These applications are configurable through the definition of processing workflows to enable the production of different types of products. This process is performed by defining a structure in JSON format that specifies the selection, parameterization, and chaining of a set of operations previously implemented in these tools (Marujo et al. 2022). There are two versions of these applications, one runs on AWS using Lambda services, and another runs on INPE's on-premise servers (Ferreira et al. 2022).

To analyze the ARD and EO data cubes, the BDC project team provides a JupyterHub environment for associated researchers. In this interactive environment, scientists can develop and run scripts to process and analyze EO data using the on-premise servers of the INPE's internal infrastructure. To produce land use and land cover maps, these scientists use the SITS (Satellite Image Time Series) R package. This package provides functions to produce land use and cover maps from image time series extracted from EO data cubes using machine and deep learning methods (Simoes et al. 2021). This package uses parallelization techniques to speed up the processing of datasets. However, there is no native support for large-scale processing of clusters of computers, as available in the *BDC Collection Builder* and *BDC Cube Builder* tools.

The ARD and EO data cubes of the BDC project can also be accessed and processed by the ODC framework. To integrate the BDC platform with the ODC framework, a tool for importing data was developed, and ODC modules were adapted to support the data produced by the BDC project (Gomes et al. 2021). As a result, this integration is made available to BDC users: (1) the ODC API in the BDC JupyterHub environment; (2) services for viewing metadata and data (*ODC datacube-explorer* and *ODC datacube-ows*); and (3) the *ODC Stats* tool, which allows parallel processing of scenes recorded in an ODC catalog.

Similar to the *BDC Cube Builder*, the *ODC Stats* is a command line tool that provides a set of previously defined statistical processing, but allows new processing functions to be added from the extension of the *Statistic* class and the implementation of two new methods: – *Measurements*, which provides a list of measurements that the class will produce, and – *compute*, which takes a *xarray.Dataset* and returns a *xarray.Dataset* with the computed measurements (Killough, Rizvi, and Lubawy 2021). Scene processing is described using a YAML file.

1.2. Our proposal

A common characteristic observed in all solutions in the previous subsection is the approach used to describe the processing tasks. The explicit or implicit use of Directed Acyclic Graphs (DAGs) to represent workflows has been observed in GEE, openEO, BDC Collection Builder, BDC Cube Builder, and ODC Stats. While BDC Collection Builder, BDC Cube Builder, and ODC Stats use text files to describe processing flows, GEE and openEO provide a higher-level interface, allowing the user to write code in a programming language (Javascript and Python for GEE and Javascript, Python, and R for openEO). In both cases, these scripts are converted into data structures that represent DAGs and are sent to run on the backend. Using this technique of sending code close to the data, known as the Moving Code approach (Müller 2016), EO platforms are evolving to integrate data ready for analysis and technologies for extracting information on the server side.

To address the BDC project demand for providing a system where users and developers can describe sequences of processes to be efficiently executed in the project server-side infrastructure, we propose a tool called BDC Workflow Engine (BDC-WE). This paper presents the software

archicture and the prototype of this tool. The BDC-WE uses DAGs as a core concept and integrates the openEO API to allow the submission and control of processes by the users. To take advantage of all computational power available in the INPE's infrastructure of the BDC project, this system uses technologies for orchestrating processes distributed in clusters of computers.

The remainder of this paper is organized as follows. In Section 2, we present the main concepts adopted and the proposed architecture. Section 3 describes the implementation of the BDC-WE prototype. Section 4 presents two study cases. In the first one, legacy processing flows from INPE's Mapaquali project were converted to DAGs to be processed in the BDC-WE. In the second case study, the processing flow of a land use and land cover classification application written in R language using SITS package was converted into DAGs and executed using the BDC-WE. Finally, in Section 5, we present the final considerations and the future steps that will be carried out.

2. BDC-WE: A tool for big EO processing

The BDC-WE architecture uses workflow as a central concept for the representation of processing described by the chaining of tasks. Direct Acyclic Graphs (DAGs) are used to represent these workflows. In this approach, each vertex of a graph represents a specific operation and the edges indicate the data dependency between each operation. The nomenclature defined by openEO is used as reference (Schramm et al. 2021). Vertices (tasks) are called *Process(es)* and chains of *Process* (DAG) are called *Process graph(s)*(*PGs*).

Figure 1 illustrates a simple example of *PG* with four *Processes*. This workflow, which illustrates data collection from an external provider, includes *Processes* for: (i) scene discovery from an external provider; (ii) downloading the scenes to a local repository; (iii) registration of new scenes in a metadata catalog; and (iv) publication of new scenes in a Web Map Service (WMS).

In the BDC-WE architecture, *Process* represents a meta-task, which means an operation class that does not have a functional core that actually performs the expected operation. BDC-WE uses an abstraction called *Resources* to configure the *Process* with the operations to be performed.

Figure 2 shows the BDC-WE architecture. Some elements of this diagram represent software artifacts (*Resources*, *Processing repository*, and *openEO Client*), tools for workflow orchestration and task execution (*Workflow Orchestrator* and *Workers*), (web)services (*openEO Backend*, *Rest API*, and *External Services*), and Graphical User Interfaces (GUIs) (*openEO Web Editor* and *Workflow Orchestrator Interfaces*).

There are three different kinds of actors that interact with a BDC-WE instance: (1) *Developers*: people responsible for deploying or maintaining a BDC-WE instance. They have technical knowledge for configuring BDC-WE and the other tools/services used; (2) *Experts*: people who master the topic of data products that are produced on the BDC-WE platform. They are responsible for the creation and parameterization of the algorithms that generate the data products; and (3) *Users*: people who make use of the services, APIs, or GUIs available in a BDC-WE instance. These actors do not need to have technical mastery of the inner workings of BDC-WE.

Resources are software artifacts (classes or functions) that abstract elements managed by the platform and that provide the implementations of the processing that will be performed. They must be accessible to *Workers* so that they can be instantiated and passed as a dependency on *Processes*. For instance, to record scenes in a local catalog, a new *Resource* can be implemented by extending an interface called `Catalog`. The use of *Resources* prevents the platform from having to know the

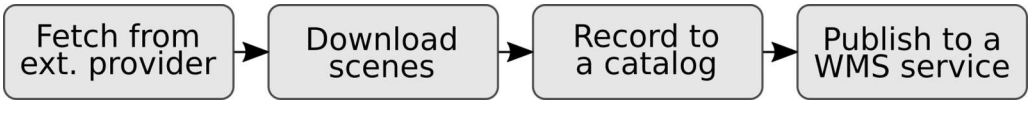

Figure 1. Process graph example.

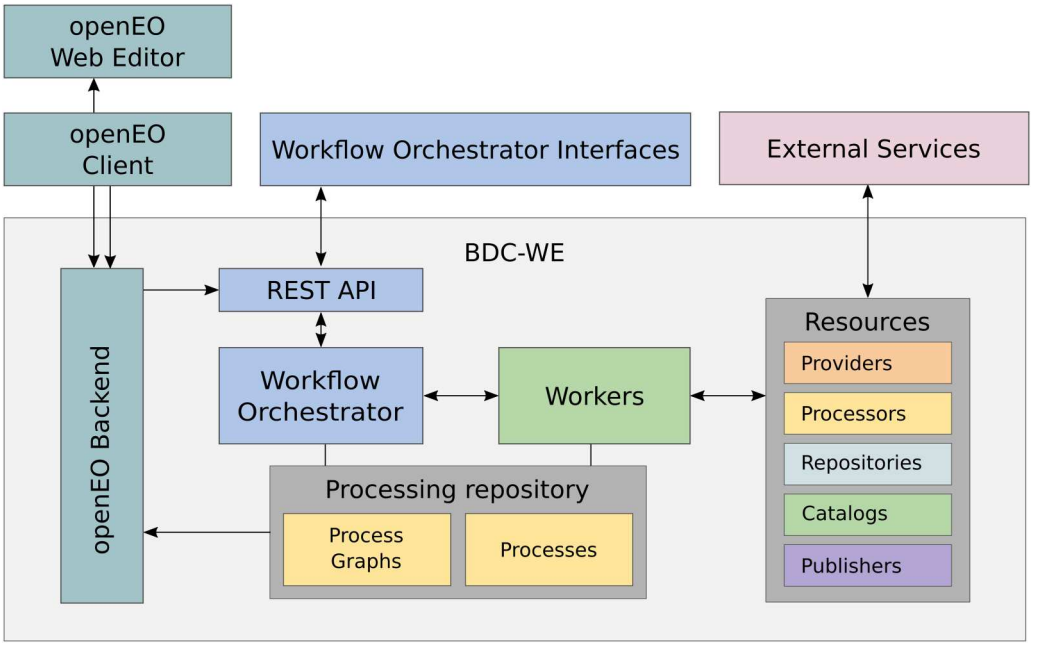

Figure 2. BDC-WE architecture.

algorithms that will be used and expands the opportunities for use in different applications, only observing that the specificities of each solution are integrated.

Currently, BDC-WE manages the following types of *Resources*:

- ***Provider***: represents an external provider. It has functions for searching scene metadata in an external catalog;
- ***Catalog***: represents a catalog of metadata that can be managed by BDC-WE. It provides functions for searching, adding, and removing scenes in a catalog;
- ***Processor***: represents a processing function that can be applied to a scene or a set of scenes;
- ***Publisher***: represents an external service to the platform. Provides functions for publishing (and unpublishing) a scene or a set of scenes;
- ***Repository***: represents a file system manager where data is retrieved or written. Provides functions to manage the structure of directories where the data will be stored; and
- ***Features***: represents a collection of vector data that can be used as input parameters to *Processors*. Provides a catalog of vector data.

The *Processing Repository* represents the repository with functions and classes that implement the available *Processes* and descriptions of *Process Graphs* available in the platform. *Developers* can add new *Processes* to the list of built-in *Processes* available in BDC-WE. *Workers* are responsible for executing *Processes*. The use of multiple *Workers* can increase the scalability of processing large volumes of data.

The *Workflow Orchestrator* (*WO*) is responsible for managing the execution of *Process Graphs* performed by *Workers*. It is responsible for loading the available *Process Graphs* and *Process*, checking whether the input and output dependencies between the *Process* are compatible, receiving execution requests from *Workflow Orchestrator Interfaces* or *openEO Backend* through *BDC-WE REST API*, and managing the execution of the workflow on *Workers*.

BDC-WE REST API is a module responsible for managing access and identifying users who will submit and monitor processing. For example, it is used to limit access to restricted data or the number of running processes by a *User*. The *openEO Backend* is responsible for providing a standardized API so that external clients to the BDC-WE can interact with the platform using available libraries or graphical interfaces, such as the openEO client (JavaScript, Python, and R) and the openEO Web Editor. The *openEO Backend* is responsible for making *Processes* available in the BDC-WE instance so that users can describe their *Process Graphs* using an openEO client. This backend also allows *Users* to invoke the *Process Graphs* to run and download the produced data.

Workflow Orchestrator Interfaces (*WOI*) represents interfaces that allow interaction with *WO*, allowing the configuration and execution of *Process Graphs*. They are mainly intended for *Developers*, as they require technical mastery of how BDC-WE works and provide more details about *Process Graphs* and processing executions. *WOI* can be used for scheduling recurring processing. The *External Services* represents the services used by the *Resources* in a BDC-WE instance, for instance an STAC Provider or an OGC WMS. In addition to *Resources*, which represent the resources that perform processing, BDC-WE provides a set of classes that represent processable elements. These data models have the necessary attributes to be managed by *WO* and can be extended by *Developers* to include other attributes or methods.

Figure 3 presents a class diagram supported by BDC-WE. The core element is *Scene*, which is extended to *LocalScene* and *RemoteScene*. A *LocalScene* represents a *Scene* that has a list of *Measurements* and methods to combine with other *LocalScenes* (`merge`) or to remove it (`remove`). A *Measurement* has a `path` to a file and a *MeasurementProperty*. A *MeasurementProperty* has at

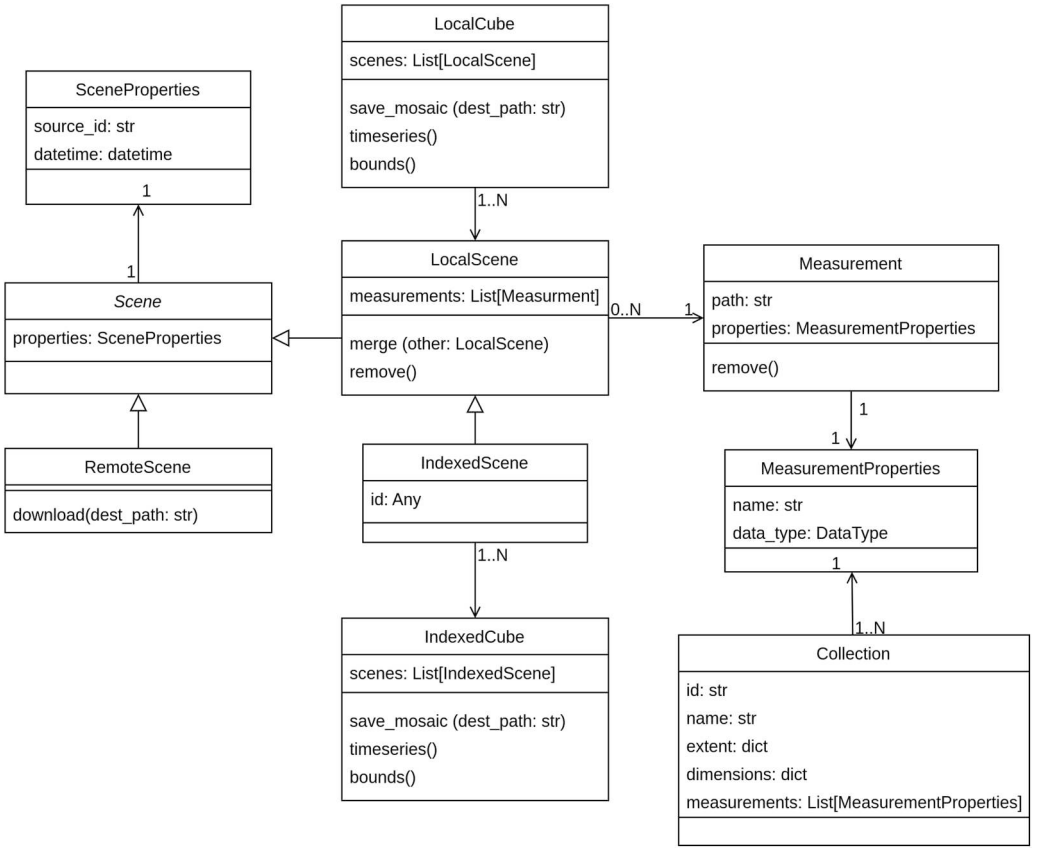

Figure 3. BDC-WE data abstraction model.

least two attributes: `name` and `data type`. A *RemoteScene* uses a method `download` that implements a way to download files from a *Scene* to the repository. An *IndexedScene* extends *LocalScene* by including a unique identifier for the *Catalog* in use. A *Collection* represents a set of *Scenes* that share the same *MeasurementProperties* types. A *Cube* is defined as a set of *Scenes*.

3. BDC-WE: implementation

To implement the architecture described in Section 2, a set of technologies were chosen as a way to accelerate the framework development process and to reuse open-source solutions that met the needs of the BDC-WE. The Python language was used as a reference, since it is used for the *BDC Collection Builder* and *BDC Cube Builder* and is also adopted by other platforms, such as Open Data Cube, openEO, and Google Earth Engine (Gomes, Queiroz, and Ferreira 2020).

The core element of the framework is the *Workflow Orchestrator*. In the ecosystem of processing data through workflows, there are a variety of open-source tools (Matskin et al. 2021), such as Apache Airflow (The Apache Software Foundation 2022), Argo (Argo 2022), Temporal (Temporal Technologies 2022), and Dagster (Elementl 2022).

Apache Airflow is a platform that allows the programmatic creation of workflows in Python and the scheduling and monitoring of executions (Harenslak and Ruiter 2021). *DAGs* are defined through the instantiation of `Operators` available on the platform. A *DAG* is created from the dependency configuration between `Operators`. Argo, however, has a higher granularity for *tasks*. This engine manages workflows in a Kubernetes environment, where each *task* is represented by the execution of a container and a workflow is defined through a YAML file.

Temporal is a platform for orchestrating workflows written in Go, Java, PHP, Python, or TypeScript codes. In Python, a *DAG* is represented by a class decorated with a decorator, *@workflow.defn*, and each *task* is represented by a method decorated with *@activity.defn*. Temporal is a platform that is still under development and does not have full support for some languages. The tool documentation is also under development (Temporal Technologies 2022).

Dagster is a platform for orchestrating workflows in Python. The elements that constitute the processing workflow are defined using the decorators available in the `dagster` package. A workflow *task* is a function decorated with the *@op* decorator, and a *DAG* is a function decorated with *@graph* called a task function. By identifying the *task* calling sequence, Dagster creates a *DAG* structure with data dependencies between the *tasks*. This structure is used during the orchestration of workflow execution. In Dagster, a workflow can run locally in serial or parallel modes using DASK, Celery, Docker, or Kubernetes. Triggering the execution of a DAG can be performed through a WEB interface, GraphQL API, Python code, or by configuring Schedulers or Sensors. Dagster also has the ability to export a DAG to run on Apache Airflow. Dagster provides a paid service to run DAGs in a private cloud environment (Elementl 2022).

Dagster was chosen for use as a *WO* in the BDC-WE system, because it allows the dynamic generation of *Process Graphs* and provides a GraphQL API for the interaction between external applications and the orchestrator. This API is necessary for the interaction between the *WO*, *WOI*, *openEO Backend*, and *BDC-WE Rest API*. In addition, Dagster has a web graphical interface that allows an easy configuration and monitoring of the processes. It also has a variety of execution modes that allows the execution of all the processing locally, in a single thread or in multiple threads, or the distribution of the processing, using Celery, Dask, and/or Kubernetes technologies. These features facilitate the debugging process and the transition between development and operational environments.

In Dagster, the central processing unit is called `Ops`, represented through functions with the *@op* decorator. In this way, all *Process* developed for BDC-WE use this decorator with the respective metadata to ensure compatibility verification of function input and output parameters. *Process Graphs* in Dagster are called `Graphs` and are defined using functions decorated with *@graph* and making calls to `Ops` functions.

The dynamic generation and configuration of these elements can be achieved through the classes available in the `dagster` package. Using this functionality, BDC-WE can create these elements from a JGF (Json Graph Format) file that describes a *Process Graph*. The objective of this feature is to facilitate the configuration and maintenance of the workflows managed by the BDC-WE. These files follow a high-level format, without requiring technical knowledge about how Dagster works. In Section 4, an example of the use of this functionality is presented.

BDC-WE-API was implemented through a REST service that intermediates requests from *WO* and *openEO Backend* to *WO*. REST requests are converted to GraphQL requests, which interact with the Dagster service. In the current phase of BDC-WE development, only access control via username and password is performed by *BDC-WE REST API*. The control of the number of running processes and the limitation of the *Process Graphs* available for each *User* are not implemented yet.

For the development of the openEO Backend, a template available in the repository (https://github.com/Open-EO/openeo-python-driver) of the developers of the openEO standard was used as a starting point. It implements the general REST request handling of the openEO API and dispatches the work to a pluggable openEO backend driver. Thus, a driver was developed to translate requests from openEO API requests into requests for *WO*. This driver uses the same *Resources* used by *Processors* running on *Workers*. `Resource Catalog`, for example, is used by developed driver to find available collections or find scenes to be delivered to users.

The *openEO backend* interacts with *WO* through the GraphQL API available in Dagster. For this, a Python client, `dagster_graphql_client`, was developed (https://github.com/vconrado/dagster_graphql_client). This client allows starting runs, tracking run status, retrieving results, and reloading *Process Graphs* available for running. This last feature, together with the possibility of generating new *Process Graphs* at runtime, provides great flexibility when creating new workflows. The implemented GraphQL client also has the possibility of controlling and managing executions through command line.

The BDC-WE is still a prototype and the *openEO Backend* developed still does not support all functions of the openEO standard. New functions are being incorporated according to the demand of Developers who use BDC-WE in INPE's internal projects.

A minimal working set of *Resources* is available in BDC-WE prototype. These built-in *Resources* can be used in operational applications or serve as a reference for other implementations. The built-in *Resource* `stac_provider` available in BDC-WE, for instance, extends the `RemoteScene` class to create the `StacRemoteScene` class, which implements the downloading of scenes listed in a STAC service. *Developers* can develop new *Resources* from the inheritance of base classes made available in BDC-WE. Table 1 presents a list of *Resources* currently available in BDC-WE.

Table 1. Resources available in the BDC-WE.

Name	Type	Description
stac_provider	Provider	Performs queries in a STAC service.
odc_catalog	Catalog	Performs insert, remove, and search metadata operations in an Open Data Cube catalog.
crop_scene_proc	Processor	Crops a Scene using GDAL.
reduce_time_proc	Processor	Performs time reduction operations (max, min, mean, median) on a cube.
download_proc	Processor	Download a remote scene via HTTP.
geoserver_pub	Publisher	Publishes a scene to an ImageMosaic Store on a Geoserver server.
ssh_pub	Publisher	Run a previously configured ssh command.
discord_pub	Publisher	Sends a message to a Discord channel.
slack_pub	Publisher	Send a message to a Slack channel.
file_system_repo	Repository	Performs the creation of folders in the file system using the metadata of the scenes.
protected_file_system_repo	Repository	Performs the creation of folders in the file system, blocking the other folders for read-only.
gdal_features	Features	Loads and serves vector data in formats supported by GDAL.

In addition to *Resources* and data models, BDC-WE prototype also provides a set of previously implemented *Processes* that can be used by *Developers* to build *Process Graphs*. With the currently available *Processes*, we believe that a large number of EO data processing applications demanded by INPE projects can be modeled because these *Processes* represent meta-tasks and that the code to be executed basically depends on the *Resources* used.

The *Processes* available in the framework are grouped into 4 types:

- ***Discovery***: *Processes* that allow discovery of the resources to be processed. The discovery can be performed in external services (*Provider Resource*) or in the catalog managed by BDC-WE (*Catalog Resource*). Discovery functions in *Providers* produce `RemoteScenes` while discovery functions in *Catalogs* produce `IndexedScenes`;
- ***Processing***: *Processes* that call a processing function (*Processor Resource*) which must receive a `LocalScene` or a `LocalCube` and will produce, respectively, a new `LocalScene` or a new `LocalCube`;
- ***Indexing***: *Process* that registers a `LocalScene` or *LocalCube* in the catalog managed by the BDC-WE (*Catalog Resource*). The indexing process takes a `LocalScene` or `LocalCube` and produces, respectively, an `IndexedScene` or a `IndexedCube`;
- ***Publishing***: *Process* which notifies an external service (*Publisher Resource*) about the creation or removal of a scene or set of `Scenes`. A *Publisher* receives a set of *IndexedScene* or an *IndexedCube* and must return an object of the same type received.

Table 2 presents the *Processes* currently available in the BDC-WE prototype. In this Table, the first column presents the name of the *Process*, the second the type, and the third the *Resources*

Table 2. Processes available in the BDC-WE.

Name	Type	Resources	Input	Output	Parameters
discovery_external	Discovery	Provider	–	List [RemoteScene]	bbox, start_date, end_date, collection, ignore_assets, limit, offset
discovery_external_by_feature	Discovery	Provider, Features	–	List [RemoteScene]	feature_id, start_date, end_date, collection, ignore_assets, limit, offset
discovery_not_processed	Discovery	Catalog	–	List [IndexedScene]	product, process_name, bbox, limit, offset
discovery_by_id	Discovery	Catalog	–	IndexedScene	id
discovery_cube	Discovery	Catalog	–	List [IndexedScene]	product, bbox, start_date, end_date
index_scene	Indexing	Catalog, Repository	LocalScene	IndexedScene	product, asset_keys
index_cube	Indexing	Catalog, Repository	LocalCube	IndexedCube	product, asset_keys
apply_scene	Processing	List [Processor], Repository	LocalScene	LocalScene	process_name, subpath, override, args
seq_apply_scene	Processing	List [Processor], Repository	LocalScene	LocalScene	process_name, subpath, override, remove_partials, args
apply_cube	Processing	List [Processor], Repository	LocalCube	LocalScube	process_name, subpath, override, args
publish_scenes	Publishing	List [Publisher], Repository	List [IndexedScene]	List [IndexedScene]	product, asset_keys
publish_cube	Publishing	List [Publisher], Repository	IndexedCube	IndexedCube	product, asset_keys

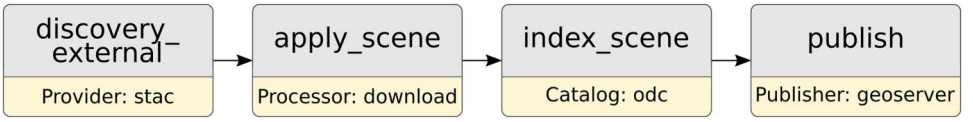

Figure 4. Process graph example with resources configuration.

used. The fourth and fifth columns present the types of object expected as the input and output by *Resource*, respectively. The last column of this table presents the parameters expected from *Processor*.

If necessary, *Developers* can implement new *Processes* and make them available to the platform through a configuration file. The *Processes* defined by *Developers* and those currently available in BDC-WE are loaded dynamically using the Python library `importlib`. *Developers* can use the same strategy to implement `Processor Resources` that receive as an argument the definition of a function that can be dynamically loaded during the execution of the Process. This approach is especially useful for implementing the User-Defined Functions (UDF) support specified by openEO.

Using the *Processes* and *Resources* available in BDC-WE, the *Process Graph* illustrated in Figure 1 can be configured according to the diagram presented in Figure 4. In this example, a STAC provider is being used as an external *RemoteScenes* provider, the *Processor* `download_proc` will be applied to each *RemoteScene* found, the `odc_catalog` *Catalog* will be used to index the *LocalScenes* and finally, the *IndexedScenes* will be published on a Geoserver server, using the *Publisher* `geoserver_pub`. Although this *Process Graph* illustrates a flow as if only a single *RemoteScene* was found in the first *Process*, BDC-WE allows the description of *Process Graphs* that execute, for example, the *Process* `apply_scene` (`download_proc`) in parallel for each *RemoteScene* returned by *Process* `discovery_external` (`stac_provider`). More details regarding the process description of *Process Graphs* are presented in Section 4.

3.1. BDC-WE boilerplate project

The architecture of the BDC-WE prototype with the chosen technologies is shown in Figure 5. To facilitate the process of deploying a BDC-WE instance, a preconfigured template project was developed to run BDC-WE with an Open Data Cube (ODC) catalog. This project has a basic example of

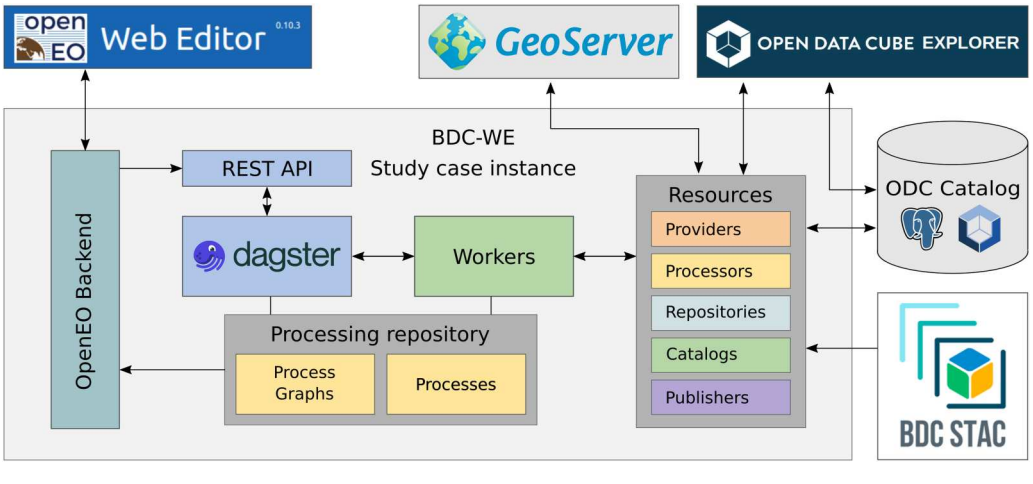

Figure 5. BDC-WE prototype architecture with the chosen technologies.

processing and a set of pre-configured services. To facilitate the deployment process, each service runs in a Docker container. These services have been grouped into four docker-compose files to facilitate the service management.

The main file, `docker-compose.yml`, has the minimum number of services for running BDC-WE: a PostgreSQL database, a RabbitMQ messaging service, and Dagster. The worker is configured in a separate file, `docker-compose .worker.yml`, to easily run on multiple servers. The third file is `docker-compose.odc.yml`, configures the following external services: datacube-explorer (STAC); Geoserver (WMS, WFS, WCS), nginx (as a file server); and a container with scripts to initialize the database and load the collections into the ODC base. The `docker-compose.openeo.yml` file configures the openEO backend and the openEO Web editor.

Using this complete project, it is possible to start a BDC-WE instance ready to work. Developers can also customize this template project to meet the specific needs of an application.

4. Case study

To evaluate the prototype of the BDC-WE, two case studies were conducted. The first case study, presented in Section 4.1, deals with the operationalization of the production of water quality indices of the MAPAQUALI project. The second, presented in Section 4.2, evaluates the use of BDC-WE implementation for land use and land cover classification using the SITS R package (Simoes et al. 2021). The second case study is useful to illustrate the use of the prototype with an application written in a language other than Python.

The BDC-WE boilerplate project was used as the starting point in both case studies. The Figure 5 illustrates the technologies and services used in both case studies. The ODC Catalog Database is a PostgreSQL instance configured with the schema used by ODC. The STAC service used by Workers is the publicly available BDC (Ferreira et al. 2020) instance. The ODC Explorer and Geoserver services were used for data dissemination using the STAC and OGC WMS standards.

4.1. Water quality indices from EO data

This case study was developed to validate the BDC-WE prototype in an operational environment and verify its applicability for producing EO data products. The MAPAQUALI project was selected for this case study. This project, being carried out at INPE's Aquatic Systems Instrumentation Laboratory (LabISA), has, among its objectives, the generation and availability of time series of the spatial distribution of water quality parameters (Lima et al. 2023; Lobo et al. 2021; Maciel et al. 2019, 2020, 2021): Chlorophyll-a, Cyanobacteria, Total Suspended Solids, Dissolved Colored Organic Matter (CDOM), an underwater light field through the diffuse attenuation coefficient (Kd), and alerts of bloom events (especially cyanobacteria). The MAPAQUALI project also demands that these water quality parameters be customized for new aquatic systems added to the system (LabISA 2022).

The MAPAQUALI project demands that a set of algorithms can be parameterized to generate and make available products for different areas of interest. To produce ARD, MAPAQUALI uses third-party scripts and algorithms written in Python produced by the project team. MAPAQUALI researchers also used this language to write scripts responsible for generating water quality parameter products. Most of these scripts receive the paths of the bands of a scene, the algorithm configuration parameters, and the path(s) of the file(s) of the product(s) that will be calculated as input parameters.

Figure 6 show the general data flow of the products calculated using the MAPAQUALI platform. The blue rectangles represent the data used or produced, whereas the gray rectangles illustrate the processing tasks performed.

From the collection of raw data (Landsat-8 OLI and Sentinel 2 L1C TOA) in an *External Provider*, a sequence of algorithms is applied to produce analysis-ready data (MAPAQUALI-ARD) used

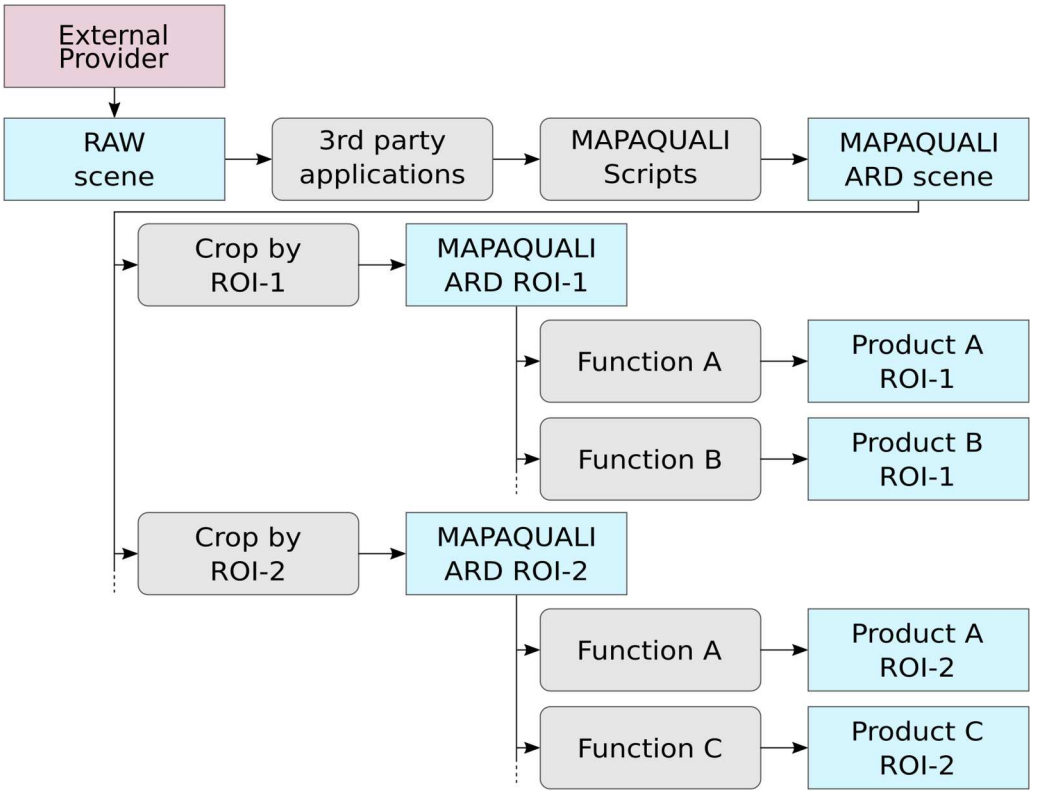

Figure 6. MAPAQUALI products generation dataflow.

as input for other water quality indicators algorithms. The second stage deals with the mapping of MAPAQUALI-ARD for the project's regions of interest (ROI). In the case of MAPAQUALI, these ROI were lakes, water reservoirs, and other aquatic systems. Finally, in the third step, the clipped ARDs were used as inputs to water quality indicator models developed by the LabISA research group. The same function that produces a water quality product can be used for different ROIs or be exclusive to a single ROI.

The STAC and OGC WMS services were chosen to disseminate data produced by the MAPAQUALI platform. The WMS is used to view the data on the web portal of the MAPAQUALI platform, whereas the STAC allows users to consult the catalog and download the products generated by the platform. As a metadata catalog, the Open Data Cube was chosen, since this framework meets the needs of the project and also provides the application datacube-explorer, which provides a STAC implementation and a visual interface for navigating between indexed collections in the catalog. For the WMS service, Geoserver (OSGeo 2022) was chosen, due to the previous experience of the MAPAQUALI team in the use and configuration of this server and the availability of a resource Publisher for Geoserver implemented by BDC-WE. In addition to publishing data in the WMS service, it was decided to publish messages in the Discord communication application. Thus, the MAPAQUALI team can monitor the generation of products more easily.

Briefly, the Resources selected for use in the case study of the MAPAQUALI project were: (i) stac (Provider); (ii) odc (Catalog); (iii) GDAL-Features (Features); and (iv) odc-stac and discord (Publishers). Wrapper functions were created as Processors for each legacy function previously developed by the MAPAQUALI team. The wrapper functions are responsible for receiving the

parameters in the format used by BDC-WE and passing them on to the legacy functions in the format they expect. Listing 1 presents an example of the recurring structure of wrapper functions used.

Processing Graphs (*PG*) were created from the identified data flow to perform processing. To facilitate reading, the *PG* will be presented, indicating the process used and resource(s) used in parentheses. For example, `index_scene` (Catalog: odc) indicates the use of the process index_scene configured with the Open Data Cube catalog. Arrows indicate the direction of data flow.

Listing 1. Wrapper function example.

```
def processor_a(in_scene: Scene, dest: Path, resources: dict, **kwargs) ->
    LocalScene:
    f = Path(dest, "prod_a.TIF")
    # call MAPAQUALI processing function
    function_a(f, in_scene.measurement("B01").path, kwargs["nodata"])
    return LocalScene (measurements=[Measurement (path=f, properties=
        MeasurementProperties(name="prod_a", data_type=DataType.float32))])
```

The *PGs* used in the case study of the MAPAQUALI platform are:

- **Collect Scenes**: `discovery_external` (Provider: stac) → `apply_scene` (Processor: download) → `index_scene` (Catalog: odc) → `publish` (Publishers: stac, geoserver, discord);
- **ARD**: `discovery_by_id` (Provider: odc) → `mq_scene` (Processor: download) → `index_scene` (Catalog: odc) → `publish` (Publishers: stac, geoserver, discord);
- **ARD ROI**: `discovery_by_id` (Provider: odc) → `crop` (Processor: crop, Features: GDAL-features) → `index_scene` (Catalog: odc) → `publish` (Publishers: stac, geoserver, discord); e
- **Product A**: `discovery_by_id` (Provider: odc) → `apply_scene` (Processor: processor_a) → `index_scene` (Catalog: odc) → `publish` (Publishers: stac, geoserver, discord); e

The *Processes* used are those listed in Table 2, while the *Resources* are those listed in Table 1. *Processor* mq_ard is a wrapper function that calls third-party scripts and MAPAQUALI functions to produce ARD data. *Process Graph* **Product A** represents the template used to generate the different products from the MAPAQUALI platform, while `processor_a` refers to a wrapper function that, for example, calls a function that calculates one of the indices of water quality developed by the project's researchers.

Listing 2 shows how the description of *PG* **Product A** is performed through a JSON file using the specification of the standard JSON Graph Format (Bargnesi et al. 2022).

Listing 2. Example of Process Graph description.

```
{"graph": {
"id": "process_a",
  "label": "Produce the water quality index A",
  "nodes": {
    "discovery_by_id": {
      "metadata": {
        "type": "discovery_by_id",
      "resources": {"provider": "odc"}}},
    "apply_scene": {
      "metadata": {
        "type": "apply_scene",
        "resources": {"processors": ["process_a"]}}},
    "index_scene": {
      "metadata": {
        "type": "index_scene",
        "resources": {"catalog": "odc"}}},
    "publish": {
      "metadata": {
        "type": "publish",
```

```
        "resources": {"publishers": ["stac","geoserver", "discord"]}}}
},
"edges": [
  { "source": "discovery_by_id",
    "target": "apply_scene",
    "relation": "map"},
  { "source": "apply_scene",
    "target": "index_scene",
    "relation": "map"},
  { "source": "index_scene",
    "target": "publish",
    "relation": "collect"}
]}
}
```

Initially, each node of *PG* is defined. The `type` attribute indicates the *Processor* to be invoked. Optionally, it is possible to define the `resources` that are used by the *Processor*. The dependencies between nodes are defined in the `edges` attribute of the `graph`. For each dependency, the origin and destination of the data and the type of relation (`map` or `collect`) were provided. The *Resources* used in a *PG* must be previously defined in a configuration file for BDC-WE to manage these artifacts.

Listing 3 illustrates the configuration of *Process* `process_a`, which does not demand any other *Resources* and will receive, in addition to the process interface parameters defined by BDC-WE, the argument `nodata`. These arguments are useful so that processing function parameters can be configured without being previously defined in wrapper functions.

Listing 3. Processors config example.

```
{
    "processors": {
      "process_a": {
        "path": "/path/to/wrappers.py",
        "function": "process_a",
        "resources": [],
        "args": { "nodata": 0.0 }
      }
      ...
    }
}
```

The configuration of data dependencies was performed using the `deps` attribute. Dependency `map:discovery_by_id` indicates that `apply_scene` is mapped (`map`) to each scene produced by `discovery_by_id`. On the other hand, the `collect:index_scene` dependency indicates that all scenes produced by `index_scene` are grouped into a list (`collect`) and then passed on to *Process* `publish`. When only one scene is produced by each *Process*, the processing flow is performed sequentially. On the other hand, if a *Process* produces a set of scenes, the next process can be performed in parallel. The concurrent execution of the *Processes* is managed by the *WO*. For example, running *PG* **Collect Scenes** that found two new scenes on the *External Provider* is illustrated by the diagram in Figure 7. The dependency of type `map` was used to map the multiple results of `discover_external` for each execution of `apply_scene`. The dependency of type `collect` performs grouping of results before invoking *Process* `publish`. The `map` type dependency can also be used between a *Process* that produces only one scene and one that consumes only one scene.

Schedulers are used for recurring execution of PGs. In the case of MAPAQUALI project, this feature is used for *PGs* of the **Collect Scene** type, which searches for new scenes from external providers daily.

For the initialization of the other PGs, the Sensor resource was used. This functionality is used in the following situations:

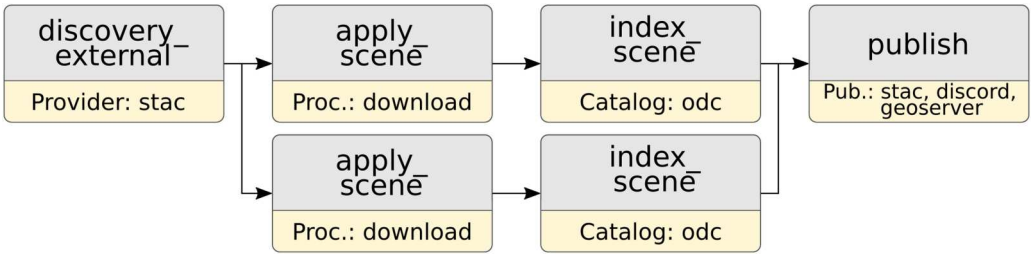

Figure 7. Process graph collect scenes diagram.

- *PG* **ARD** begins when a new scene is downloaded by *PG* **Collect Scenes**.
- *PG* **ARD ROI** when a new scene is produced by *PG* **ARD**; and
- *PGs* of type **Product A** begin when a new scene is produced by the *PG* **ARD ROI**.

The *PGs* used in this instance of MAPAQUALI's BDC-WE were also available for execution through the openEO interface. In this manner, researchers can execute algorithms with different parameters to carry out tests. The generated products can be downloaded to the researcher's desktop using openEO's API. These products are saved in a staging repository separate from the main repository. Likewise, the metadata of these products generated by researchers via the openEO API are not indexed in MAPAQUALI's main catalogue. This choice was motivated to avoid contamination of research data with products made available to the public. Figure 8 shows the openEO Web Editor interface for the BDC-WE MAPAQUALI instance.

4.2. Land use and land cover classification

In the BDC project, the R package SITS is used to produce land use and land cover maps from image time series extracted from EO data cubes using machine and deep learning methods (Simoes et al. 2021). The task of generating these maps is often divided into two phases: (1) training the machine and deep learning methods; and (2) classification using the model produced in the phase 1. In the training phase, labeled samples of a region of interest (ROI) are used to calibrate the predictive model. With this model, the scenes of this ROI, modeled as EO data cubes, are then classified.

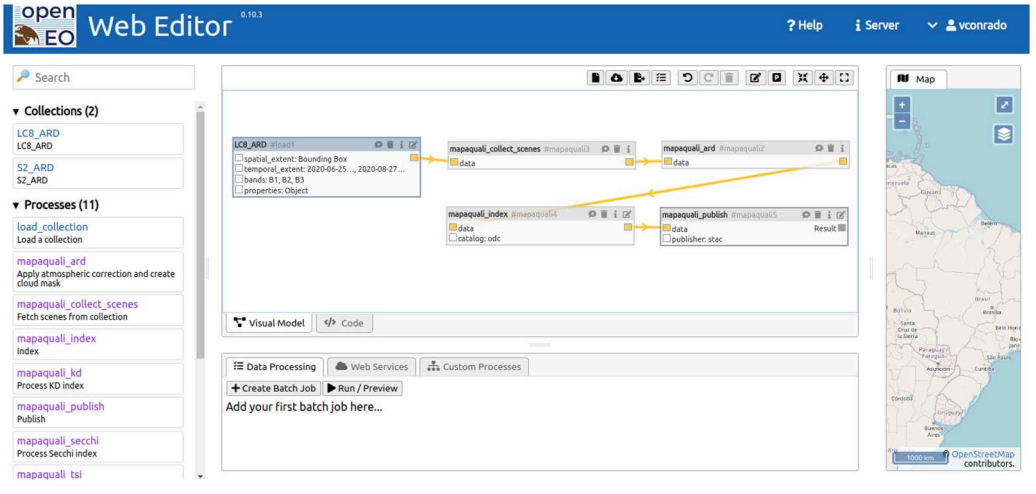

Figure 8. openEO Web Editor of the BDC-WE prototype in the MAPAQUALI case study.

For this case study, a *Process Graph* was created to classify the EO data cube, considering the existence of a previously calibrated predictive model. This classification phase consists in the following function invocation sequence of the SITS package:

(1) `sits_cube`: performs the query on the BDC STAC and defines one of a data cube for the region of interest;
(2) `sits_classify`: performs scene classification using a predictive model and the data cube produced in the previous step;
(3) `sits_smooth`: performs the smoothing of the classification performed in the previous step; and
(4) `sits_label_classification`: from the scene probability values, it converts to a label based on the highest probability of each pixel.

This algorithm was divided into two phases to run on BDC-WE, considering a *PG* type *Discovery* → *Process* → *Index* → *Publish*. In the search phase, a new Provider, SitsProvider, was implemented. SitsProvider's search method invokes the sits_cube function and produces a list of scenes to be processed. For this, a script was created in the R language, which receives the necessary parameters and invokes the `sits_cube` function. The data cube definition resulting from this function is then spatially split to define the smaller data cubes. This subdivision is performed by considering the grid used by the BDC for the collection. The purpose of this split was to make it easier to parallelize the sort run on each data cube. The data structure in R, representing the metadata of these smaller data cubes, was saved in `.rda` format. The `search` method of SitsProvider returns a list of objects of type *SitsRemoteScene* (which extends the *RemoteScene* class). Each of these objects has among its attributes the path to one of the data cubes generated by the script in R. This approach was used to represent an object produced in Python and processed in the R language.

The objects produced in the previous phase (discovery) were passed to a classified processor. This function follows the structure presented in listing 1, and calls a script in R called classify.R, which receives as input parameter the path of the predictive model and file *.rda* of the data cube. This script is responsible for performing classification, smoothing, and labeling of the pixels of each data cube. In addition, it returned the path of the sorted file. This path is for the processor to classify and create the scene object, which is returned to the *Process Graph*.

The classified scenes were then indexed into a STAC catalog and published to an OGC WMS service (Geoserver). Figure 9 illustrates the *Process Graph* used in the case study and Figure 10 shows the results of the classification performed in this study.

5. Final remarks and discussion

This paper presents the architecture of a system called BDC-WE, based on workflows for big EO data processing. This architecture allows the inclusion of new algorithms and provides a high-level interface for users, using the openEO API. These characteristics meet the needs and alternative solutions presented by Gomes, Queiroz, and Ferreira (2020). The main contributions of the proposed BDC-WE tool are: (1) the abstraction of EO data retrieval, processing, cataloging, and dissemination resources; (2) the definition of an interface to implement these resources; and (3) the

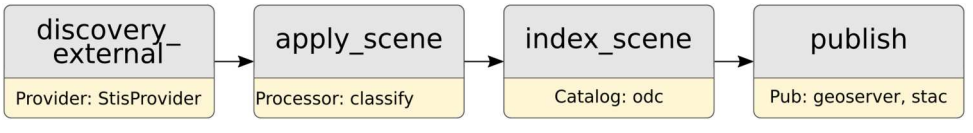

Figure 9. *Process graph* diagram for image classification with SITS.

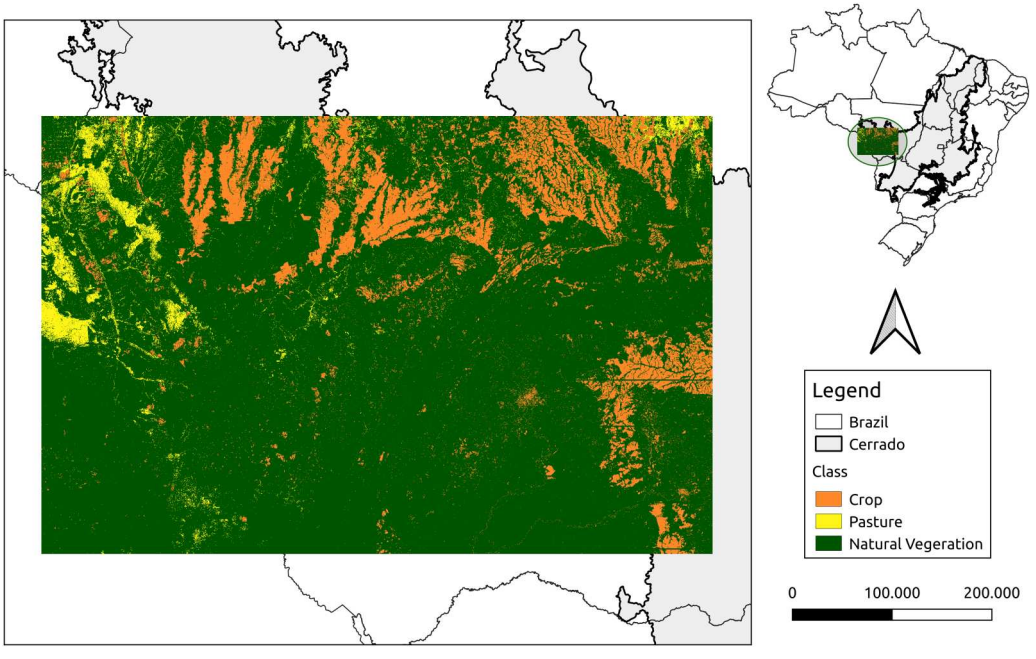

Figure 10. Land use and land cover map of the western region of the Cerrado biome produced using the SITS R package running on the BDC-WE prototype.

ability to dynamically load these resources into processing workflows. The presented solution reduces the level of complexity required to *Experts* to include new processing functions in a processing tool for large EO datasets. Furthermore, decoupling the access API from the processing functionalities allows other APIs to be integrated into the proposed architecture. To validate the proposed architecture, a prototype was developed and evaluated using two case studies.

The first case study showed us that the use of *Process* to represent meta-tasks made the process of creating *PGs* easier with algorithms previously developed by the MAPAQUALI project team. Through wrapper functions and the configuration of a *PG* with four *Process* (Discovery, Process, Index, and Publish), it was possible to produce most of the products of this project. Dagster's scheduling functionality was useful, allowing new scenes to be found daily from providers and processed by the configured *PG*. Currently, the MAPAQUALI project is using an exclusive instance of the BDC-WE prototype to produce water quality indices.

In the second use case, we observed that the decomposition of the algorithm for image classification into subtasks and the description through a *PG* facilitates the processing of massive data sets to produce land use and land cover maps using the SITS R package. The openEO API of the BDC-WE is used by users and developers to select a region of interest and to provide a file with a model previously trained by SITS. Then, BDC-WE executes the image classification *PG* distributing the *Process* to all available *Workers*.

These two case studies show us that the BDC-WE allowed applications, which were initially implemented to be executed sequentially or in parallel on a single machine, to easily gain processing scale. This is possible because of the description of these applications in *Processes*, which can be orchestrated by the BDC-WE. In this manner, once the application is modeled in the form of *PG*, new computational resources can be accommodated in the cluster to allow a gain in the processing scale, without the need for any change in *PG*.

The integration of openEO with *WO* through requests to the GraphQL API proved to be efficient because it is possible to access all the resources available in Dagster. This separation between *WO*

and the module responsible for processing requests allows, for example, new APIs, such as OGC WPS and WCPS (Aiordăchioaie and Baumann 2010), to be integrated or developed in the future without the need to change the way processing is carried out. The proposed architecture uses openEO as a way of exposing processing services to clients and does not intend to evaluate or compare its functions using other standards such as OGC WPS and WCPS. The architecture is structured in such a way as to allow other APIs to be able to be integrated into the processing solution. The advantage of using openEO as a high-level interface for BDC-WE is the availability of tools, such as the openEO Web Editor and clients in three different programming languages, and the possibility of future integration with other EO data processing platforms using this API.

Using a development-ready openEO Backend framework accelerated the integration of this API into BDC-WE. In particular, to make collections available, calls were made to methods already implemented by a *Resource* of type *Catalog*. However, the implementation of the entire set of operators available in this API requires considerable effort. Version 1.0 of the openEO API specifies 241 predefined processes. These predefined processes do not necessarily need to be implemented on all back-ends, requiring applications to perform queries (`listProcesses`) to check their availability (openEO 2023). For the integration of openEO with the BDC-WE prototype, the main goal was to validate the proposed architecture. Thus, the main operators crucial to the study cases were chosen for implementation, such as cross-band operation, time reduction (mean, maximum, minimum, and standard deviation), and invocation of predefined *PGs* specific to each use case. The inclusion of new openEO predefined processes in BDC-WE requires the implementation of a new *Process* that performs this function and the inclusion in the openEO Back-end of the invocation of a *PG* configured to execute the respective *Process*.

By adopting openEO API as the interface for the submission and control of processes by the users, the data produced in the BDC-WE tool is closer to being compatible with FAIR principles. Through the use of the STAC service, openEO provides a domain-relevant community standard that can provide rich metadata and provenance of available EO data.

The results presented in this work show the potential of the architecture proposed and of the prototype. For future work, we intend to advance in the individualized management of the use of computational resources, reproducibility, and code sharing. Regarding the management of the use of computational resources, openEO API defines endpoints for billing management, such as checking the credit available to the user, cost estimate for operations, and information on the costs of operations performed. However, this API does not define how these operations should be performed. In the BDC-WE architecture, the BDC-WE REST-API module that mediates all processing requests is responsible for these activities. A possible solution for this issue is the use of an approach inspired by the solution adopted by GEE, which limits the amount of RAM and CPU memory per processing. In the case of BDC-WE, the expectation is to limit the use of RAM by *Process* and use CPU time as a metric to be discounted from users' credits. The limits of memory and CPU used by a *Process* can be established in the execution of Docker containers and technology currently in use by BDC-WE.

Regarding code sharing, although the use of the openEO API facilitates this process in BDC-WE, it is still up to the researchers to manage the exchange of files among their peers. The ability to share the analyzes is the first step in the path to reproducibility (Ivie and Thain 2018). Carlos (2023)'s work presents a tool to assist in the process of managing research artifacts to ensure reproducible sharing, and should be considered as an important source of inspiration for including this capability in the BDC-WE.

In addition to these works, we intend to move forward with the implementation of BDC-WE through the implementation of all operators available in the openEO API, and automate the loading and availability of *PGs* through configuration files. As the implementation of this tool advances, our goal is to make BDC-WE the central tool for carrying out processing on the BDC platform, being responsible for both processing user analyzes and executing platform-specific applications, such as the *BDC Collection Builder* and *BDC Cube Builder*.

Disclosure statement

The authors report there are no competing interests to declare.

Funding

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code and by the Environmental Monitoring of Brazilian Biomes project (*Brazil Data Cube*), funded by the Amazon Fund through the financial collaboration of the Brazilian Development Bank (BNDES) and the Foundation for Science, Technology and Space Applications (FUNCATE), Process 17.2.0536.1 (ARS, AS, MP).

Data availability statement

The data that support the findings of this study are available from the corresponding author, VCFG, upon reasonable request.

References

Aiordăchioaie, Andrei, and Peter Baumann. 2010. "PetaScope: An Open-Source Implementation of the OGC WCS Geo Service Standards Suite." In *Scientific and Statistical Database Management*, edited by Michael Gertz and Bertram Ludäscher, 160–168. Berlin, Heidelberg: Springer Berlin Heidelberg.

The Apache Software Foundation. July, 2022. "Apache Airflow." Accessed July 20, 2022. https://airflow.apache.org/.

Argo. July, 2022. "Argo Project." Accessed July 20, 2022. https://argoproj.github.io/.

Bargnesi, Anthony, Anselmo DiFabio, William Hayes, Georgiy Shibaev, Cristophe Benz, Hugh Pyle, Erik Dao, and Travis Giggy. July, 2022. "JSON Graph Format (JGF)." Accessed July 20, 2022. https://jsongraphformat.info/.

Brown, Molly E.. 2016. "Remote Sensing Technology and Land Use Analysis in Food Security Assessment." *Journal of Land Use Science* 11 (6): 623–641. https://doi.org/10.1080/1747423X.2016.1195455.

Camara, Gilberto, Luiz Fernando Assis, Gilberto Ribeiro, Karine Reis Ferreira, Eduardo Llapa, and Lubia Vinhas. 2016. "Big Earth Observation Data Analytics: Matching Requirements to System Architectures." In Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, 6. ACM.New York, NY, USA

Carlos, Felipe Menino. May, 2023. "Storm: Platform to Support The Development of Reproducible and Collaborative Geospatial Applications." Accessed May 15, 2023. http://mtc-m21d.sid.inpe.br/col/sid.inpe.br/mtc-m21d/2023/01.04.23.36/doc/publicacao.pdf.

de Lima, Thainara Munhoz Alexandre, Claudia Giardino, Mariano Bresciani, Claudio Clemente Faria Barbosa, Alice Fabbretto, Andrea Pellegrino, and Felipe Nincao Begliomini. 2023. "Assessment of Estimated Phycocyanin and Chlorophyll-a Concentration From PRISMA and OLCI in Brazilian Inland Waters: A Comparison Between Semi-Analytical and Machine Learning Algorithms." *Remote Sensing*15 (5): 1299. https://doi.org/10.3390/rs15051299.

Ekim, Burak, and Elif Sertel. 2021. "Deep Neural Network Ensembles for Remote Sensing Land Cover and Land Use Classification." *International Journal of Digital Earth* 14 (12): 1868–1881. https://doi.org/10.1080/17538947.2021.1980125.

Elementl. July, 2022. "Dagster – Cloud-Native Orchestration of Data Pipelines." Accessed July 20, 2022. https://dagster.io/.

FAO. May, 2023. "SEPAL Repository." Accessed May 7, 2023. https://github.com/openforis/sepal/.

Ferreira, K. R., G. R. Queiroz, R. F. B. Marujo, and R. W. Costa. 2022. "Building Earth Observation Data Cubes on AWS." *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*XLIII-B3-2022:597–602. https://doi.org/10.5194/isprs-archives-XLIII-B3-2022-597-2022.

Ferreira, Karine R., Gilberto R. Queiroz, Lubia Vinhas, Rennan F. B. Marujo, Rolf E. O. Simoes, Michelle C. A. Picoli, Gilberto Camara, et al. 2020. "Earth Observation Data Cubes for Brazil: Requirements, Methodology and Products." *Remote Sensing* 12 (24): 4033. https://doi.org/10.3390/rs12244033.

Giuliani, Gregory, Paolo Mazzetti, Mattia Santoro, Stefano Nativi, Joost Van Bemmelen, Guido Colangeli, and Anthony Lehmann. 2020. "Knowledge Generation Using Satellite Earth Observations to Support Sustainable Development Goals (SDG): A Use Case on Land Degradation." *International Journal of Applied Earth Observation* 88:102068. https://doi.org/10.1016/j.jag.2020.102068.

Gomes, Vitor C. F., Felipe M. Carlos, Gilberto R. Queiroz, Karine R. Ferreira, and Rafael Santos. 2021. "Accessing and Processing Brazilian Earth Observation Data Cubes with the Open Data Cube Platform." *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*V-4-2021 (4): 153–159. https://doi.org/10.5194/isprs-annals-V-4-2021-153-2021.

Gomes, Vitor C. F., Gilberto R. Queiroz, and Karine R. Ferreira. 2020. "An Overview of Platforms for Big Earth Observation Data Management and Analysis." *Remote Sensing* 12 (8): 1253. https://doi.org/10.3390/rs12081253.

Gorelick, Noel, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. 2017. "Google Earth Engine: Planetary-scale Geospatial Analysis for Everyone." *Remote Sensing of Environment* 202 (2016): 18–27. http://dx.doi.org/10.1016/j.rse.2017.06.031.

Harenslak, Bas P., and Julian de Ruiter. 2021. *Data Pipelines with Apache Airflow*. Shelter Island, New York, USA: Manning.

Ivie, Peter, and Douglas Thain. 2018. "Reproducibility in Scientific Computing." *ACM Computing Surveys (CSUR)* 51 (3): 36.

Kamali Maskooni, Ehsan, Hossein Hashemi, Ronny Berndtsson, Peyman Daneshkar Arasteh, and Mohammad Kazemi. 2021. "Impact of Spatiotemporal Land-use and Land-cover Changes on Surface Urban Heat Islands in a Semiarid Region Using Landsat Data." *International Journal of Digital Earth*14 (2): 250–270. https://doi.org/10.1080/17538947.2020.1813210.

Killough, Brian. 2018. "Overview of the Open Data Cube Initiative." In IGARSS 2018 – 2018 IEEE International Geoscience and Remote Sensing Symposium, 8629–8632.Valencia, Spain

Killough, Brian, Syed Rizvi, and Andrew Lubawy. 2021. "Advancements in the Open Data Cube and the Use of Analysis Ready Data in the Cloud." In *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS*, 1793–1795. IEEE.

LabISA. July, 2022. "Mapaquali." Accessed July 20, 2022. http://www.dpi.inpe.br/labisa/project/mapaquali/.

Lobo, Felipe de Lucia, Gustavo Willy Nagel, Daniel Andrade Maciel, Lino Augusto Sander de Carvalho, Vitor Souza Martins, Cláudio Clemente Faria Barbosa, and Evlyn Márcia Leão de Moraes Novo. 2021. "AlgaeMAp: Algae Bloom Monitoring Application for Inland Waters in Latin America." *Remote Sensing* 13 (15): 2874. https://doi.org/10.3390/rs13152874.

Maciel, Daniel Andrade, Claudio Clemente Faria Barbosa, Evlyn Márcia Leão de Moraes Novo, Nagur Cherukuru, Vitor Souza Martins, Rogério Flores Júnior, Daniel Schaffer Jorge, Lino Augusto Sander de Carvalho, and Felipe Menino Carlos. 2020. "Mapping of Diffuse Attenuation Coefficient in Optically Complex Waters of Amazon Floodplain Lakes." *ISPRS Journal of Photogrammetry and Remote Sensing* 170:72–87. https://doi.org/10.1016/j.isprsjprs.2020.10.009.

Maciel, Daniel Andrade, Claudio Clemente Faria Barbosa, Evlyn Márcia Leão de Moraes Novo, Rogério Flores Júnior, and Felipe Nincao Begliomini. 2021. "Water Clarity in Brazilian Water Assessed Using Sentinel-2 and Machine Learning Methods." *ISPRS Journal of Photogrammetry and Remote Sensing* 182:134–152. https://doi.org/10.1016/j.isprsjprs.2021.10.009.

Maciel, Daniel, Evlyn Novo, Lino Sander de Carvalho, Cláudio Barbosa, Rogério Flores Júnior, and Felipe de Lucia Lobo. 2019. "Retrieving Total and Inorganic Suspended Sediments in Amazon Floodplain Lakes: A Multisensor Approach." *Remote Sensing* 11 (15): 1744. https://doi.org/10.3390/rs11151744.

Marujo, Rennan F. B., Karine R. Ferreira, Gilberto R. Queiroz, Raphael W. Costa, Jeferson S. Arcanjo, and Ricardo C. M. Souza. 2022. "Generating Analysis Ready Data Collections for Brazil." In IGARSS 2022 – 2022 IEEE International Geoscience and Remote Sensing Symposium, 6844–6847.Kuala Lumpur, Malaysia

Masser, Ian. 2019. "The Future of Spatial Data Infrastructures." In Geographic Information Systems to Spatial Data Infrastructure, 227–252. CRC Press.Boca Raton, Florida

Matskin, Mihhail, Shirin Tahmasebi, Amirhossein Layegh, Amir H. Payberah, Aleena Thomas, Nikolay Nikolov, and Dumitru Roman. 2021. "A Survey of Big Data Pipeline Orchestration Tools From The Perspective of The Datacloud Project." In Proceedings of the 23rd International Conference Data Analytics Management Data Intensive Domains (DAMDID/RCDL 2021), 63–78.Moscow, Russia

Milcinski, Grega, and Primoz Kolaric. 2023. "Sentinel Hub-Federated On-Demand ARD Generation." In EGU General Assembly 2023, 4160.Vienna, Austria

Müller, Matthias. 2016. "Service-Oriented Geoprocessing in Spatial Data Infrastructures." PhD diss., Technische Universität Dresden.

Müller, Matthias, Lars Bernard, and Johannes Brauner. 2010. "Moving Code in Spatial Data Infrastructures – Web Service Based Deployment of Geoprocessing Algorithms." *Transactions in GIS*14 (SUPPL. 1): 101–118. https://doi.org/10.1111/tgis.2010.14.issue-s1.

openEO. November, 2023. "openEO Processes (1.0)." Accessed November 5, 2023. https://openeo.org/documentation/1.0/processes.html.

OSGeo, S.. July, 2022. "Geoserver." Accessed July 20, 2022. https://geoserver.org/.

Schramm, Matthias, Edzer Pebesma, Milutin Milenković, Luca Foresta, Jeroen Dries, Alexander Jacob, Wolfgang Wagner, et al. 2021. "The OpenEO API–Harmonising the Use of Earth Observation Cloud Services Using Virtual Data Cube Functionalities." *Remote Sensing* 13 (6): 1125. https://doi.org/10.3390/rs13061125.

Simoes, Rolf, Gilberto Camara, Gilberto Queiroz, Felipe Souza, Pedro R. Andrade, Lorena Santos, Alexandre Carvalho, and Karine Ferreira. 2021. "Satellite Image Time Series Analysis for Big Earth Observation Data." *Remote Sensing* 13 (13): 2428. https://doi.org/10.3390/rs13132428.

Soille, P., A. Burger, D. De Marchi, P. Kempeneers, D. Rodriguez, V. Syrris, and V. Vasilev. 2018. "A Versatile Data-intensive Computing Platform for Information Retrieval From Big Geospatial Data." *Future Generation Computer Systems* 81:30–40. https://doi.org/10.1016/j.future.2017.11.007.

Temporal Technologies. July, 2022. "Temporal – Open Source Durable Execution Platform." Accessed July 20, 2022. https://temporal.io/.

Wang, Lizhe, Yan Ma, Jining Yan, Victor Chang, and Albert Y. Zomaya. 2018. "pipsCloud: High Performance Cloud Computing for Remote Sensing Big Data Management and Processing." *Future Generation Computer Systems* 78:353–368. https://doi.org/10.1016/j.future.2016.06.009.

Xu, Chen, Xiaoping Du, Xiangtao Fan, Gregory Giuliani, Zhongyang Hu, Wei Wang, Jie Liu, et al. 2022. "Cloud-based Storage and Computing for Remote Sensing Big Data: A Technical Review." *International Journal of Digital Earth* 15 (1): 1417–1445. https://doi.org/10.1080/17538947.2022.2115567.