Data identification and process monitoring for reproducible earth observation research

Bernhard Gößwein
TU Wien
Vienna, Austria

Tomasz Miksa
TU Wien & SBA Research
Vienna, Austria

Andreas Rauber
TU Wien
Vienna, Austria

Wolfgang Wagner
TU Wien
Vienna, Austria

***Abstract*—Earth observation researchers use specialised computing services for satellite image processing offered by various data backends. The source of data is often the same, for example Sentinel-2 satellites operated by Copernicus, but the way how data is pre-processed, corrected, updated, and later analysed may differ among the backends. Backends often lack mechanisms for data versioning, for example, data corrections are not tracked. Furthermore, an evolving software stack used for data processing remains a black box to researchers. Researchers have no means to identify why executions of the same code deliver different results. This hinders reproducibility of earth observation experiments. In this paper, we present how infrastructure of existing earth observation data backends can be modified to support reproducibility. The proposed extensions are based on recommendations of the Research Data Alliance regarding data identification and the VFramework for automated process provenance documentation. We implemented these extensions at the Earth Observation Data Centre, a partner in the openEO consortium. We evaluated the solution on a variety of usage scenarios, providing also performance and storage measures to evaluate the impact of the modifications. The results indicate reproducibility can be supported with minimal performance and storage overhead.**

I. Introduction

Earth Observation (EO) data consists mostly of satellite images. Similar to other eScience disciplines, data is too big to be downloaded for local analysis. The solution is to store it in high-performance computational backends, process it there, and browse the results or download resulting figures or numbers later [1]. Such an approach addresses performance issues, but does not allow researchers to take full control over the environment in which their experiments are executed. The backends act as black boxes to outsiders with no possibility of getting information on environment configuration, e.g. software libraries and their versions. Studies in different domains show that the computational environment can have impact on reproducibility of scientific experiments and must be documented in order to ensure reproducibility [2] [3] [4]. Still the vast majority of backend providers do not share such environment information.

Another problem deals with a precise identification of data used for processing. EO backends in Europe usually obtain data from the same source, for example from the services and data bubs of the European Earth Observation (ESA) Programme Copernicus. The ESA releases data updates and corrections in cases when one of the instruments used for observation was wrongly calibrated or broken and raw data had to be processed again. This data is then distributed to backend operators. Usually there is no versioning mechanism for data. Researchers do not know which version of data was used in their study, i.e. whether they were using a version before or after some specific modifications. This leads to the problem that scientists are not able to precisely identify and cite the input data of their experiments, which hinders reproducibility and in turn undermines trust in results. Both the computational environment and the input data constitute the context of an experiment. The Research Data Alliance (RDA) has identified 14 general rules [5] for the identification of data used in computation that allows to cite and retrieve the precise subset and version of data that existed at a certain point in time. The VFramework [4] and context model [6] were proposed to automatically document environments in which computational workflows execute and to enable their comparison. The openEO project [7] works on creating a common EO interface to enable interoperability of EO backends by allowing researchers to run experiments on different backends without reimplementing their code. By bringing these three elements together a scientific infrastructure is created that allows automatically documented and reproducible experiments to be executed with minimal overhead to the infrastructure and researchers performing their studies.

In this paper, we present this solution improving reproducibility of EO experiments executed at openEO compliant backends. We follow the RDA recommendations for data identification and present how data provided by backends is made identifiable by assigning identifiers to subset queries made by researchers. We discuss which specific information must be captured, which interfaces must be modified, and which software components must be implemented. We also show how jobs being executed at backends can be captured and compared using the VFramework to determine whether any differences in software dependencies among two executions exist. We implemented our solution at the Earth Observation Data Centre for Water Resources Monitoring (EODC), a 20 Petabyte storage infrastructure connected to the Vienna Scientific Cluster[1] HPC system. For the evaluation we simulated use cases representing updates of data and changes in the backend environment. We also measured the performance and storage impact on the backend, which turned out to be minimal.

[1] http://vsc.ac.at/systems/vsc-3/

978-1-7281-2451-3/19/$31.00 ©2019 IEEE
DOI 10.1109/eScience.2019.00011

IEEE computer society

The remainder of this paper is structured as follows: Section II presents related work that is a basis of our solution and provides earth observation context. Section III presents the architecture of the proposed solution. Section IV focuses on the implementation of the prototype at the EODC backend. Section V presents methods offered to researchers enhancing reproducibility. Section VI presents a typical EO workflow using our implementation. Section VII describes the experimental evaluation followed by conclusions in Section VIII.

II. Related Work

A. Reproducibility in geoscience

In [8] reproducibility of scientific papers in geoscience is tested by analysing more than 400 papers. Authors conclude that only for half of the analysed publications identical results can be achieved. The solution described in this paper facilitates the assessment of experiments' reproducibility by precisely identifying input data used in computation, as well as automatically documenting the execution environment.

Geoscience Papers of the Future (GPF) [9] is an initiative to encourage geoscientists to publish papers together with the associated digital products of their research. This means that a paper would include:

1) documentation of datasets, including descriptions, unique identifiers, and availability in public repositories;
2) documentation of software, including pre-processing of data and visualization steps, described with data unique identifiers and pointers to public code repositories;
3) documentation of the provenance and workflow for each figure or result.

Our solution aids in providing information requested above by precisely identifying datasets and software (including version and libraries) used by backends to compute results. This information is collected automatically and can be accessed by users any time using the same API as they use for the implementation of their experiments.

B. Recommendations for data identification

The Research Data Alliance (RDA)[2] is an international body issuing recommendations to help remove barriers in data sharing. Recommendations are based on a community consensus elaborated within working groups. The Data Citation working group[3] has identified 14 rules [5] for identification of data used in computation that allows to:

- identify and cite arbitrary views of data, from a single record to an entire data set in a precise, machine-actionable manner,
- cite and retrieve that data as it existed at a certain point in time, whether the database is static or highly dynamic.

The rules are independent of the implementation technology and require that:

- data is stored in a versioned and timestamped manner,
- data is identified by persistent identifiers (PIDs) assigned to timestamped queries that can be re-executed against the timestamped data store.

The recommendations provide a generic set of rules independent of an application domain. So far they have been implemented in settings ranging from atomic and molecular data [10], climate change [11] to health policy planning [12]. We use them also in our solution to make data provided by backends identifiable, by assigning PIDs to queries identifying the subsest of data selected by researchers for their analyses.

C. Process execution monitoring

The solution described in this paper follows the VFramework [4] that can verify and validate workflow re-executions. It allows to identify whether any differences in software dependencies among two executions of the same workflow exist. The VFramework uses the context model [6] to document the environment and thus enables a comparison of workflow executions without having to access both environments at the same time. We compare context models of workflow executions (experiments submitted to run on backends) to verify whether the workflow re-execution was obtained in a compliant way [13]. Figure 1 gives an overview of the framework. Context includes not only high level description of workflow steps and services, but also technical details on infrastructure, including software and data.

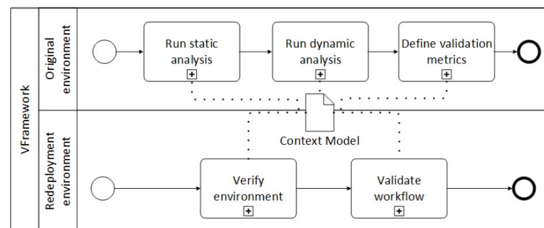

Fig. 1: VFramework: Framework for verification and validation of re-executed workflows [13].

The importance of preserving software dependencies to enable reproducibility of scientific computing was highlighted in [14]. Similar to the context model, authors propose to capture software dependencies and provenance of process executions. They introduce the PRUNE environment that automates this process and hides its complexity from researchers, but requires them to change their existing processing pipelines. In our solution we extend the existing backend infrastructure to add reproducibility supporting functionality and elaborate client side interfaces to provide additional functionality. Thus, we do not change the way how researchers work, but introduce modifications for existing environments to improve the reproducibility of experiments.

D. Earth observation and openEO

EO data is too big to be downloaded locally for analysis. The solution is to store this data in computational backends, process it there, and browse the results or download resulting

[2]https://rd-alliance.org
[3]https://rd-alliance.org/groups/data-citation-wg.html

figures or numbers later. Fig. 2 presents a typical usage scenario in which an earth observation scientist uses a backend for processing data in an experiment (hereinafter referred to as ”job”).

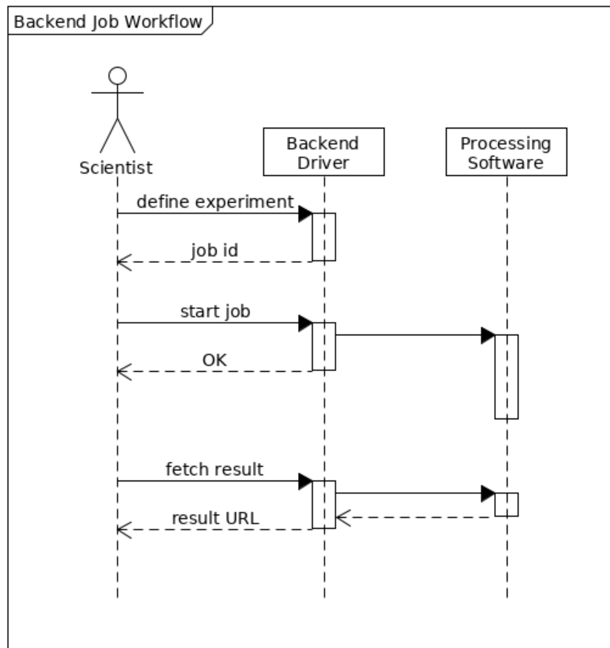

Fig. 2: Typical workflow of an experiment execution on an EO backend.

The European Commission Horizon 2020 project openEO [7] works on creating a unified earth observation backend interface that decouples clients from backends, so that code written for one backend can be executed on other backends as well. As a result, researchers can switch between different backends, without having to re-implement their code. To describe jobs and input data openEO uses process graphs. They are defined as JSON objects, that have a tree-like structure to define the processes that are submitted by the client to the backend for processing.

Fig. 3 shows an example process graph. In the center of the figure the process *get_collection* can be seen, which defines the satellite data that is used. It functions as the input for the filter processes *filter_bbox* and *filter_daterange*, which define the temporal and spatial extent. The two outermost processes *NDVI* (Normalized Difference Vegetation Index) and *min_time* (Taking the minimum value of each pixel of the resulting timeseries) are then applied to the filtered data [7].

The tendency to make experiments FAIR [15] fits with openEO, since it creates trust and makes the experiments reuseable. In this paper we describe how we extended the openEO specification and propose how our solution can be implemented at an openEO compliant backend, i.e. EODC backend. Thus, we provide a general solution that can be adopted by other backends compliant to the openEO standard.

```
{
  "process_graph":{
    "imagery":{
      "imagery":{
        "extent":[
          "2017-01-01",
          "2017-01-31"
        ],
        "imagery":{
          "extent":{
            "north":49.041469,
            "east":17.171631,
            "west":9.497681,
            "south":46.517296,
            "crs":"EPSG:32632"
          },
          "imagery":{
            "process_id":"get_collection",
            "name":"s2a_prd_msil1c"
          },
          "process_id":"filter_bbox"
        },
        "process_id":"filter_daterange"
      },
      "nir":"B08",
      "process_id":"NDVI",
      "red":"B04"
    },
    "process_id":"min_time"
  }
}
```

Fig. 3: Example process graph according to the openEO core API version 0.3.1.

III. Architecture

In this section we describe the architecture of our solution, i.e. components that must be added to the openEO compliant backends to support data identification and process execution capturing.

The architecture of openEO has 3 major elements:

- *core API* – Application Programming Interface which specifies a common set of methods exposed by backends.
- *EO clients* – client modules written in different programming languages (e.g. Python, R, Java) that allow researchers to define their processes and execute them on selected backends.
- *backend drivers* – software modules implementing the functionality described by the core API that are deployed at backends. They act as endpoints to which EO clients connect.

Our solution adds methods to the EO clients and core API. It also requires an implementation of additional functionalities by the backend. Fig. 4 shows a simplified model of a backend – client communication. White represents existing components. Green depicts new components added by us to an existing backend.

A scientist defines an experiment by using a *Client* application. It creates a process graph and transmits it to a selected backend, where a new job is created. The *Client* is then used to start the execution of the job and to retrieve results (cf. Fig. 2).

The *Data Query* component gets filter arguments of the process graph and translates them into an internally used data query language. It executes the query and forwards received data to the *Process Execution* component. The *Query Handler* receives the query and its resulting data.

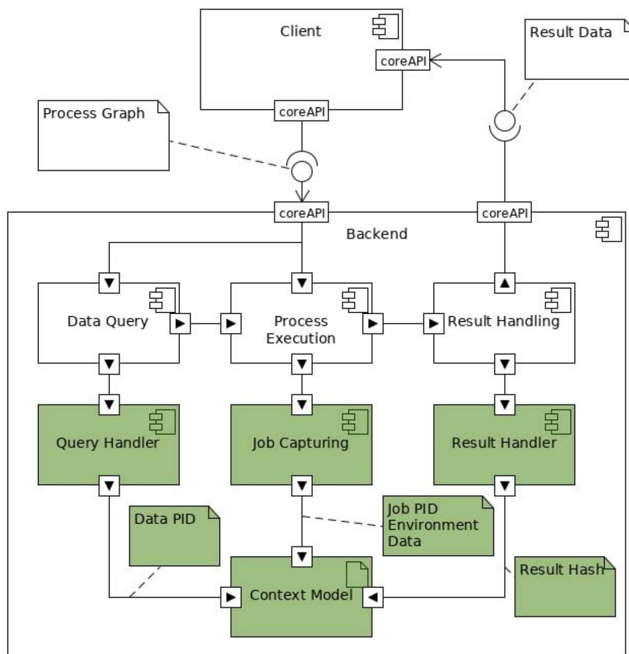

Fig. 4: Overview of the extensions to the backend.

The *Process Execution* component receives a process graph from the *Client* application and an input data from the *Data Query* component and executes the job. It provides the results to the *Result Handling* component and provides job environment data for the *Job Capturing* component.

The *Result Handling* component receives results from the *Process Execution* component and persists all data about the job and its result, so that the *Result Handler* component can use it. In the meantime it sends results back to the *Client* application. The final data is available for a few days, then the backend erases it. The *Query Handler* component offers data identification for the backend. It receives the query and its result from the *Data Query* component. It checks if there is already a persistent identifier (PID, e.g. a DOI or URI) for the given query and query result combination. If so, it returns the existing query PID, otherwise a new PID is generated and the query is stored. It adds the query PID to the *Context Model* as persistent input data identifier. Other than the *Result Handler* component, the *Query Handler* manages to have no duplicates for a query PID, whereas the *Result Handler* just creates a checksum of results to make it comparable. Therefore, the *same* input data leads to the same *query PID*, hence we use the term data PID subsequently.

The *Job Capturing* component is responsible for identifying code (version) and its components used at the backend. It assigns a PID to code used for processing, which is a result of translation of the submitted process graph into a code natively supported by the backend. Additionally, the component captures data about the environment in which the job is executed. The data consists of static and dynamic information. Static information contains environment information that is independent of the job definition. Dynamic information is dependent on the job definition and is collected at runtime. The following static backend data is collected:

1) **Backend version**
   The version of the backend is used to identify a certain backend configuration. The configuration contains the environment information of the backend e.g. docker container description files and the executed code inside of the docker container. It has to be unique and different on every change of the backend configuration. The backend version is needed to identify the state of the backend and therefore the environment of the job execution.
2) **Code identification**
   Identifier of the code deployed at the backend. It is needed to identify the code used for the execution of the jobs.
3) **API version**
   Version of openEO API supported by the backend. A different API version may lead to different results of job executions and is therefore part of the environment information.
4) **Publication timestamp**
   Timestamp since when the backend version is in place, so that the active backend version of a specific time can be resolved.

The following dynamic data as given below must be captured:

1) **Input data persistent identifier**
   This element is the output of the *Query Handler* component, the persistent identifier of the used query. The query represents the input data of the job execution. By re-executing the same query, the original input data of the job can be recreated. It is needed to identify the input data of the job.
2) **Backend version of the execution**
   The version of the backend during the execution of the job. Connects the dynamic job context model with the static backend environment. This information is needed to be able to reconstruct the active backend version during the job execution.
3) **Programming language**
   Programming language and version used by the job execution. It gives the scientist transparency about the execution environment.
4) **Dependencies of the programming language**
   Dependencies of the programming language used in a job execution (e.g. Python modules), which allow us to track the impact of the software environment on job results.
5) **Result checksum**
   Instead of storing the whole result data, a result *checksum* (e.g. hash) is introduced. It describes the job results, so that a different result can be identified by comparing

the checksum to results of other job executions.

The *Result Handler* component has the responsibility for creating hashes of computation results which are later used to validate whether two executions had the same result. The data created by an EO backend might be too big to store completely.

The *Context Model* is used to persist static and dynamic information collected on the environment and the job execution. The *Context Model* has to be integrated into the database of the backend. Therefore, we recommend to use data formats that are already used by the backend. The elements of the context model are static in size (see Section VII-D), hence a relational database is sufficient.

IV. Implementation

In this section, we describe the prototype implementation at the EODC backend. The backend is implemented in Python. The service is hosted using OpenShift[4] and Docker technology. We give an overview of the components as well as the resulting context model. Fig. 2 shows the general workflow of a job execution at an EO backend.

A. Query Handler - Component

We implemented the *Query Handler* as an additional component that gets executed after the input data query execution. Fig. 5 gives an overview of the structure of the *Query Handler* implementation. The internal query language of the backend is CSW[5], which uses XML queries. The centerpiece of the RDA recommendations is the implementation of a query store, which is responsible for storing the data needed to make queries comparable and to re-execute them in the same manner as the original execution. Queries in the query store have to be comparable, identifiable and persistent. The execution data at EODC is persisted in a relational database (PostgreSQL). To create the query store, we added a query table and a junk table to enable a many-to-many connection between the query table and the job table.

The following list summarizes the entries of the *Query Table*:

1) **Query PID**
   The recommendation R8 of the RDA requires to create a unique persistent identifier for each query record. We generate the persistent identifier using the Python library uuid[6]. EODC is using it to generate unique job identifiers, so we use it also for the query PID.
2) **Dataset PID**
   The dataset PID is the identifier of the satellite product at EODC used in the process graph and according to R3 of the RDA recommendations added to the query record.
3) **Original Query**
   The original query [recommendation R3] is the XML query following the CSW standard, executed by the EODC backend.

[4]https://www.openshift.com/

[5]http://cite.opengeospatial.org/pub/cite/files/edu/cat/text/main.html

[6]https://docs.python.org/3/library/uuid.html

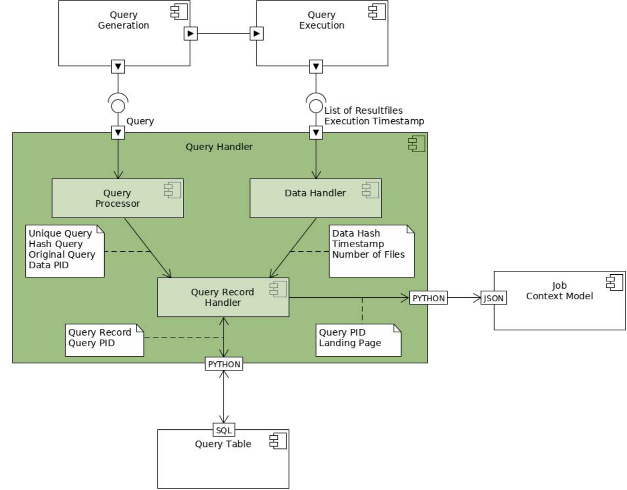

Fig. 5: Overview of the *Query Handler* implementation at the backend.

4) **Unique Query**
   The unique query [recommendation R4] is the restructured query that is comparable to other unique queries. The EODC backend parses the filter arguments into a JSON object. Since the order of the filters makes no difference in the outcome of the query execution, we alphabetically sort the elements of the original query by the JSON keys to generate the unique query.
5) **Unique Query Hash**
   We create the unique query hash [recommendation R3], by applying the SHA-256 hash function (using the Python module ”hashlib”) on the unique query as input, after removing characters that are irrelevant.
6) **Result Hash**
   The result of the query is a list of alphabetically sorted files. We create the result hash [recommendation R6] by the calculation of the SHA-256 hash function of the result file list, after irrelevant characters are removed.
7) **Execution Timestamp**
   The execution timestamp [recommendation R7] is taken from the *Query Execution* component of the EODC backend.
8) **Additional Data**
   There is an additional data column in the *Query Table* to give EODC the opportunity to store additional data about a query. The column is defined as a JSON object. We added the number of result files to the ”meta_data” column.
   Example: ”{ ”number_of_files”: 10}”

The *Query Handler* takes the input data of the EODC query execution and adds the input data PID to the job context model. For every executed job the *Query Handler* has to check if there is already a query with the same combination of unique query hash and result hash. If the combination exists, it adds the

existing query PID (aka data PID) to the context model, else it creates a new query entry and adds the new query PID to the context model.

B. Result Handler - Component

We implemented the *Result Handler* as an additional Python component of the backend, which is executed after the execution of a job finished. In the version 0.3.1 of the openEO API the output of the job execution is limited to a single output image. We implemented the result *checksum* by using the SHA-256 hash function on the resulting output file.

C. Job Capturing - Component

Fig. 6 gives an overview of the added components to capture the environment of an executed job. The job capturing is separated in static and dynamic data. Therefore, the static data of the backend versioning has to be introduced to the backend, and is described in the following list:

1) **Backend version and code identification**
   We use the GitHub repository of the backend[7] as versioning tool for the backend. The latest commit identifier of the master branch is the current version of the backend. Past versions are identifyable, either by the commit identifier or by timestamp. Other backends might use different version-control systems for this purpose.
2) **API version**
   EODC manually updates the core API version in the GitHub repository.
3) **Publication timestamp**
   The publication timestamp of a backend version is defined by the GitHub commit timestamp.

The dynamic job capturing during the job execution is done by adding logging messages to transfer the needed data from the *Process Execution* component to the *Job Capturing* component. The captured elements are described in the next section.

D. Context Model - Data Record

Each executed job creates a context model. We store the context model in an additional column of the job table in the format of a JSON object. The following list shows the data stored in the context model:

1) **Input data persistent identifier**
   The source input data identifier is the PID of the query provided by the EODC query store described in Section IV-A.
2) **Backend version**
   To identify the version of the backend during the execution of the job, the commit identifier of the backend during the execution is persisted in the context model.
3) **Programming language and dependencies of the programming language**
   The *Process Execution* module uses the installed Python module *pip* to list all installed packages with their versions. The *Job Capturing* component uses the tool to capture the installed Python modules and the Python version.
4) **Result checksum**
   The result *checksum* is the output of the *Result Handler* component described in Section IV-B.

[7]https://github.com/Open-EO/openeo-openshift-driver

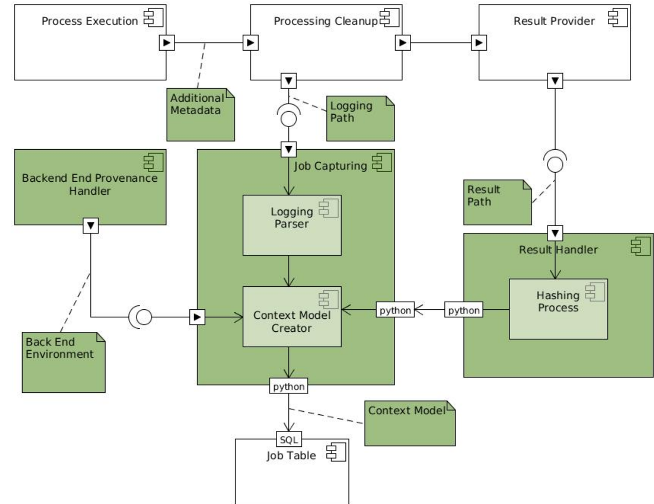

Fig. 6: Overview of the implementation components used by the *Job Capturing* component.

V. User Services

We extended the core API (version 0.3.1) of openEO, so that we were able to implement our concept at the backend. To make the internal extensions of the backend available for the users, we defined user services into the core API. The new endpoints are added to the openEO Python client as well as to the backend driver. These extensions and their implementation are described as below:

1) **Backend Version**
   We added a new end point to the core API to give the user the opportunity to retrieve the current and past backend versions. The new endpoint is a request called GET /version/<timestamp> and no authentication is needed to access it. The response of the version endpoint is the whole job independent provenance information of the backend. It gives the scientist the opportunity to get environment information of the current backend version and from past versions.
2) **Detailed Job Information**
   In the openEO core API there is an endpoint to get detailed information about the job status. The endpoint path is GET /jobs/<job_id>, which only contains the execution state of the job and the job id. We add the whole context model of the job to this endpoint in the implementation to provide scientists with the environment information and the input data PID of the job execution.

3) **Data Identifier Landing Page**
We implemented the landing page of the data PID into an additional endpoint of the openEO core API. We introduce the "GET /data/<data-pid>" endpoint, which returns a description of the data type and the resolvable data PID. The landing page contains a link to another page with the file list after a query re-execution ("GET /data/<data-pid>/result" endpoint). If the result file list differs from the original execution, there is a list of the files that differ, otherwise it states that the file list is equal to the original execution. This helps scientists to reconstruct the input data used by a cited data PID.
4) **Re-use of Input Data**
We extended the process graph definition of the openEO core API, by adding an additional filter argument ("data_pid") to the "get_collection" process. If a process graph uses the input data PID, the backend automatically applies the queries like in the original execution. As a result, the user is able to make additional experiments on the same data without additional effort and is informed if the data changed at the backend.
5) **Comparing two Jobs**
We have implemented a function at the Python client to compare two jobs, by comparing the context models of the jobs. For every item in the context models, the term "EQUAL", if the items are the same in both context models, the difference if the items are not the same in both context models, or "MISSING" if the item is missing in one of the context models. Researchers can compare experiments to see how they were executed and to be able to explain different outcomes.

VI. Use cases

This section provides an example on how a typical EO workflow is executed using our implementation.

1) **Researcher A runs job A at the backend.**
Listing 1 shows the code to run a job at the backend. First we establish the connection to the backend, then the process graph gets created, by choosing the processes and arguments. Furthermore the job gets created and executed.

```
con = openeo.connect(backend_url)
# Choose dataset and filter operations
processes = con.get_processes()
pgA = processes.get_collection(
                    name="s2a_prd_msil1c")
pgA = processes.filter_daterange(pgA,
              extent=["2017-05-01",
                      "2017-05-31"])
pgA = processes.filter_bbox(pgA,
              west=10.288696, south=45.935871,
              east=12.189331, north=46.905246,
              crs="EPSG:4326")
# Choose processes
pgA = processes.ndvi(pgA, nir="B08", red="B04")
pgA = processes.min_time(pgA)
# Create and start job A out of the
# process graph A (pgA)
jobA = con.create_job(pgA.graph)
jobA.start_job()
```

Listing 1: Researcher initializes and runs *jobA*.

2) **Researcher A retrieves the used input data PID of job A.**
Researcher A wants to receive the used data PID, so that he can cite the input data of the experiment in the resulting paper. Listing 2 shows the code to retrieve the query *pidA* of *jobA*.

```
# Get data PID of jobA
# e.g. qu-a3bbe4a0-a875-4687-bb78-9457f33134a9
pidA = jobA.get_data_pid()
```

Listing 2: Researcher A retrieves the persistent input data identifier to cite.

3) **Researcher A retrieves the experiment environment identifier.**
Researcher A wants to get the environment identifier, so that he can cite the used environment of the experiment. Listing 3 shows the code to retrieve the backend version used when *jobA* was executed.

```
# Get backend version of jobA
# e.g. 1a0cefd25c2a0fbb64a78cd9445c3c9314eaeb5b
versionA = jobA.get_backend_version()
```

Listing 3: Researcher A retrieves the backend version used by the execution of jobA.

4) **Researcher B uses the same input data, by applying the data PID of job A for job B.**
Researcher B is a reviewer and wants to check the experiment of researcher A by running the same process graph on the same input data of job A. Listing 4 shows how researcher B can use the same input data PID as researcher A with *jobA*. The backend re-executes the query of *pidA* with the timestamp stored in the query store of the solution backend. The updated file has a creation timestamp after the query timestamp and is not in the query result of *pidA*. Therefore, the input data of job B is the same as the input data of job A.

```
# Take input data of job A by using
# the input data PID A
# to execute jobB.
pgB = processes.get_data_by_pid(data_pid=pidA)
# Choose processes
pgB = processes.ndvi(pgB, nir="B08", red="B04")
pgB = processes.min_time(pgB)
# Create and start Job B
jobB = con.create_job(pgB.graph)
jobB.start_job()
# get pidB of job B and compare it with pidA
pidB = jobB.get_data_pid()
(pidA == pidB) # Returns True
diffAB = jobA.diff(jobB)
```

Listing 4: Create *jobB*, which uses the input data identified by *pidA*.

Researcher B wants to be sure that the environment of his replication is the same as the environment of the analysis of researcher A. The last line of Listing 4 stores the difference between job A and job B into *diffAB*. The content of the dictionary is a comparison of every key of the job context models with each other and can be viewed in Listing 5. It shows that two independent researchers ran the same experiment using

TABLE I: *Query Table* in the beginning of the test case execution.

Query pidA	
Column	**Value**
query_pid	qu-a3bbe4a0-a875-4687-bb78-9457f33134a9
dataset_pid	s2a_prd_msil1c
original	<csw:Query…
normalized	{'extend': {'crs':…
norm_hash	0917c7a21cec960b8a66…
result_hash	abf43f519007050cbaeb59a067a2226d6…
updated_at	2019-03-31 17:36:44.613893
meta_data	{'result_files': 51}
created_at	2019-03-31 17:36:43.064445

identical environments. Hence, we can conclude that experiment was replicable.

```
{'input_data': 'EQUAL','output_data': 'EQUAL',
'process_graph': 'EQUAL','openeo_api': 'EQUAL',
'interpreter': 'EQUAL', 'code_env': 'EQUAL',
'different':
{'back_end_timestamp':'20190417194702.496810',
'job_id': 'jb-b92c688c-7fdc-4126...',
'start_time': '2019-04-17 19:47:02.496810',
'end_time': '2019-04-17 19:47:03.258261'}}
```

Listing 5: Content of the job comparison *diffAB*.

VII. Evaluation

The evaluation is two part: first we simulate different scenarios to test if the proposed solution detects typical changes in data and the software environment. Second, we evaluate the performance and storage impact of the solution on the EODC backend. The Python client, the solution backend and the code for the evaluation is available and further described on GitHub[8].

A. Evaluation Setup

In the evaluation we used a test environment that ran on a local machine which is identical to production environment of the EODC. The difference is in the performance of the hardware. Since the data querying service of EODC is publicly available, we used the actual data that EODC provides for openEO users to test the data identification part of the solution. Changes on the data (like removed files or updated files) are simulated by changing the output of the query directly after execution in the test environment.

B. Data Updates

In this section we evaluate the behavior of the solution on data updates at EODC. In the beginning of each test case there is one job entry and one query entry in the database. Table I shows the content of the *Query Table* of the database.

1) Test Case 1: Is it possible to re-execute a query after a file is updated?: If a file is updated at the EODC backend, the new file gets a different file name than the original one and the creation timestamp is updated. If a new job is executed using the same process graph as the first job, the resulting files are different than in the original execution. Table II shows the *Query Table* after the second job execution. In Table II

[8]https://github.com/bgoesswein/dataid_openeo

TABLE II: *Query Table* after the execution of the second job. Important elements are highlighted blue if they are the same and red if they are different.

Query pidA (full entry in Table I)	
Column	**Value**
query_pid	qu-a3bbe4a0-a875-4687-bb78-9457f33134a9
norm_hash	0917c7a21cec960b8a66…
result_hash	abf43f519007050cbaeb59a067a2226d6…
meta_data	{'result_files': 51}
Query pidB	
Column	**Value**
query_pid	qu-23f5a313-e804-4faa-aa33-60ed1ac69e2d
dataset_pid	s2a_prd_msil1c
norm_hash	0917c7a21cec960b8a66…
result_hash	28088d113de19ce037e9651…
updated_at	2019-03-31 18:01:49.214956
meta_data	{'result_files': 51}
created_at	2019-03-31 18:01:47.695042

the important differences are marked red. There is a different result hash, since the new job uses the updated file instead of the original one. The normalized query is still the same, but since the result of the query changed a new data PID is generated. Next, we executed a third job, which is the same as the second, but uses the input data PID of the first job (pidA) (see Listing 4). The execution timestamp is part of the query in *pidA* and the third job executes the original query of the first job. Therefore, it uses the same data PID as the first job.

2) Test Case 2: Is it possible to re-execute a query after a file is updated with the original one deleted?: This test case shows how the solution behaves if a file is updated, but the original file is deleted at the same time. The re-execution of the query *pidA* results in a file list without the deleted file. Since files are filtered out by the query using the execution time-stamp, the new file does not appear in the result file list. If the re-execution results not in the same file list, the response of the backend contains a "state" attribute, which contains the replaced files. The backend returns the most recent file version, even if there are versions between the original and the most recent file available. Users can see the alternatives for the original file, but not the missing original file. This is because the full file list is not persisted in the query store, but the number of result files. We made the decision of not storing the complete file list, because of the deletion policy of the backend provider, which do not delete files without replacing it with an updated version.
If the researcher runs a second job with *pidA* as input data, he gets a warning message that the query result is different from the original execution.

3) Test Case 3: Is it possible to re-execute a query after a data file is deleted?: The deletion of a file without a new file replacing it, is not within the policies of EODC, since they would restrict their range on available data. If this happens nevertheless, there is when using the solution proposed by this paper no possibility to get the exact files that were removed.

4) Test Case 4: Is it possible to recreate an older version of the backend?: The aim of the implementation is to capture enough data to make it possible to re-run the same job. The backend is created directly by its GitHub repository. To recreate an old version of the backend, the GitHub repository URL and the commit identifier of the original set up is needed. The timestamp of the job execution is persisted in the context model of the original job execution. The original GitHub repository and commit can be resolved by the backend provenance. Unfortunately, in the current solution there is no automatic way of setting a version of the EODC backend. If an older version of the backend has to be activated, the EODC needs to be contacted and asked to load a new instance of the backend with the old version in place.

C. Performance and Storage impact

This section evaluates the performance and storage impact of the implementation on the backend. To achieve this, we define 18 test cases with input process graphs derived from 9 publications ([16][17][18][19][20][21][22][23][24]) that used data provided by EODC from the last two years. Every input process graph is executed 50 times and after each iteration the duration and the used storage size is captured. After each execution the backend gets cleaned up, so that every iteration happens in the same backend conditions. The elements of the *Query Handler* component, the *Job Capturing* component and the *Result Handler* component are measured.
The *Query Handler* component is responsible of creating a new query entry in the query table. Therefore, it generates the hash of the query result, normalized query and the hash value of it, the number of input files, the dataset PID and the original query. The hash over the query result has a complexity of $O(n)$, where n is the size of the query result. Hence, the performance of the query result hash calculation is dependent on the size of the query result and for the test cases it is visualized in Fig. 8. The duration of the other elements of the *Query Handler* are independent of the amount of processes defined in the job or the size of the input data. Therefore, the calculations of these elements are constant in duration throughout the test case executions and are visualized in Fig. 7. The test cases are sorted in ascending order by the size of the query result. The standard deviation of the elements (normalized: 19.35μs, norm_hash: 5.89μs, file_number: 3.98μs, dataset_pid: 2.50μs and orig_query: 5.34μs) over all test cases is minimal. It shows that the duration of the calculation for these elements are not dependent on the size of the query result.

The performance of the *Job Capturing* component is independent on the job complexity, except for the result hash provided by the *Result Handler* component, which is depending on the size of the result image of the job. Fig. 9 shows the duration of the result hash calculation. The duration of the result hash calculation is direct proportional to the result size of the job execution. This is due to the complexity of the hash function of $O(n)$, where n is the size of the result file.

We summarize the performance of the implementation by the mean value of all constant elements of the test case executions. The result is an average of about 20 ms plus the additional time of the result hash of both the *Query Handler* and *Result Handler*. This results in an execution time range between 20ms and 170ms per job execution. Compared to the estimated computation time of the test cases between 10 seconds and 20 minutes at the production version of the EODC backend, we conclude that the impact of the *Query Handler* is negligible.

The storage impact of the solution is constant per job execution. The solution added a query table and a column at the job table to the EODC backend. Both have only elements of fixed size. If a job execution creates a new query entry, the additional storage needed is 2.677 kB. If the job execution uses an already existent input data PID, only the context model needs an additional storage of 1.043 kB. We conclude that the additional needed storage is insignificant due to the size of the storage infrastructure of EODC.

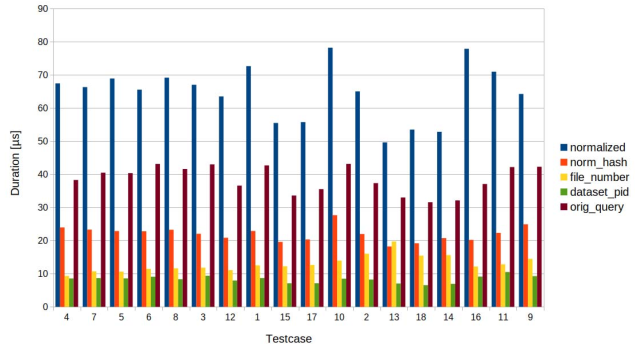

Fig. 7: Duration of the constant *Query Handler* elements of the test cases executions sorted by query result size. Time of the normalized query calculations are blue, time of hashing the normalized queries are red. Yellow bars show the time of calculating the number of result files, whereas green bars the time of fetching dataset identifiers. Storing and fetching the original queries are brown bars.

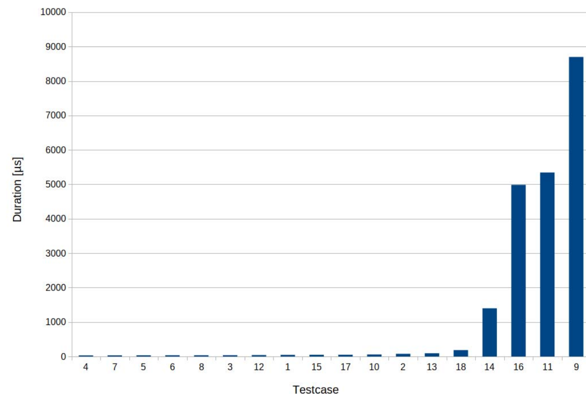

Fig. 8: Execution duration of the result hash by running the test cases, sorted by query result size in ascending order.

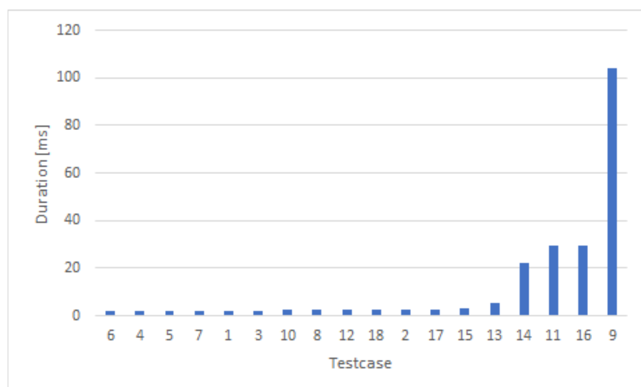

Fig. 9: Execution duration of the *Result Handler* component by running the test cases, sorted by job result size in ascending order.

D. Evaluation Summary

The evaluation showed how the implementation on the EODC backend tackles the issues of reproducibility in the earth observation sciences. The data identification implementation is tested against special test cases regarding data updates and data deletions. The evaluation shows that the solution can re-execute queries properly by returning old versions of updated files. The test cases underline that the usage of the data PID as input data of a new job is superior to the current way of re-executing a job with the same process graph. That's because the process graph does not have the original execution timestamp and therefore does not use the same input data after an update occurs. In the evaluation of deleted data at the backend, the solution happens to be not capable of showing the exact missing files, since not the whole file list result of the query is persisted. We decided to tolerate this drawback, because there is no precedent at EODC that a deletion occurred at the backend before. The test case on job capturing conveyed that the solution is capable of identifying the backend version and therefore, the environment of the job execution. Still, to run a new job on the same environment, EODC has to manually provide it. We evaluated performance and storage impact on the backend by running 18 test cases derived from past publications that used data from EODC. The results show that, except for the result hashes, the calculation of the data identification and the context model are independent from the complexity of the job. The time of the result hashing used for the data identification is dependent on the size of the query result and the result hash of the context model is dependent on the size of the output file of the job execution. Impact of our solution on the performance of the test case executions is between 20ms and 170ms. Compared to the estimated computation time of the test cases between 10 seconds and 20 minutes at the production version of the EODC backend, we conclude that the impact of our solution is negligible. The space needed in addition per job is constant and also minimal compared to the size of data kept at the backend.

VIII. Conclusion and Future Work

In this paper we dealt with the problem of lack of reproducibility of many EO experiments executed using specialised computing backends. Considered scenarios and problems can be observed in other e-Science disciplines as well.

We presented how infrastructure of existing backends can be modified to support precise data identification by following the Research Data Alliance recommendations. We described how jobs can be captured and compared using the VFramework to identify whether any differences in computational environments exist. The proposed solution was implemented at Earth Observation Data Centre in Vienna which is a member of the openEO consortium that develops a common interface for interoperability of EO backends. The implementation involved extending the backend to add reproducibility supporting functionality, as well as, extending client side interfaces to provide additional functionality. Thus, we did not change the way researchers work, but introduced changes to existing environments to improve reproducibility of experiments executed using them.

To evaluate our solution, we simulated use cases representing updates of data and changes in the backend environment. We also measured the performance and storage impact on the backend. We conclude that the solution is capable of making the input data, code and the environment identifiable and reproducible. Impact on backend's performance is minimal.

Future work will focus on implementing the solution on further backend types, e.g. backends with non-file-based result sets. The openEO project is an ongoing project, hence the common API may evolve and our work will have to be adapted to new releases of the API.

Acknowledgment

openEO - This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 776242.

The authors wish to thank the team of EODC, especially Luca Foresta and Thomas Mistelbauer, for the support on implementing the concept into the EODC backend.

References

[1] M. Ramamurthy, "Geoscience cyberinfrastructure in the cloud: Data-proximate computing to address big data and open science challenges," in *2017 IEEE 13th International Conference on e-Science (e-Science)*, Oct 2017, pp. 444–445.

[2] E. H. B. M. Gronenschild, P. Habets, H. I. L. Jacobs, R. Mengelers, N. Rozendaal, J. van Os, and M. Marcelis, "The effects of freesurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements," *PLOS ONE*, vol. 7, no. 6, pp. 1–13, 06 2012. [Online]. Available: https://doi.org/10.1371/journal.pone.0038234

[3] M. Konkol, C. Kray, and M. Pfeiffer, "Computational reproducibility in geoscientific papers: Insights from a series of studies with geoscientists and a reproduction study," *International Journal of Geographical Information Science*, vol. 33, no. 2, pp. 408–429, 2019. [Online]. Available: https://doi.org/10.1080/13658816.2018.1508687

[4] T. Miksa, A. Rauber, and E. Mina, "Identifying impact of software dependencies on replicability of biomedical workflows," *Journal of Biomedical Informatics*, vol. 64, pp. 232 – 254, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1532046416301484

[5] A. Rauber, A. Asmi, D. V. Uytvanck, and S. Pröll, "Identification of reproducible subsets for data citation, sharing and re-use," *IEEE TCDL*, vol. 12, 2016.
[6] R. Mayer, T. Miksa, and A. Rauber, "Ontologies for describing the context of scientific experiment processes," in *2014 IEEE 10th International Conference on e-Science*, vol. 1, Oct 2014, pp. 153–160.
[7] E. Pebesma, W. Wagner, M. Schramm, (...), and P. Soille, "OpenEO - a Common, Open Source Interface Between Earth Observation Data Infrastructures and Front- End Applications," 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1065474
[8] F. O. Ostermann and C. Granell, "Advancing science with vgi: Reproducibility and replicability of recent studies using vgi," *Transactions in GIS*, vol. 21, pp. 224–237, 2017.
[9] Y. Gil, C. H. David, I. Demir, B. T. Essawy, R. W. Fulweiler, J. L. Goodall, L. Karlstrom, H. Lee, H. J. Mills, J.-H. Oh, S. A. Pierce, A. Pope, M. W. Tzeng, S. R. Villamizar, and X. Yu, "Toward the geoscience paper of the future: Best practices for documenting and sharing research from data to software to provenance," *Earth and Space Science*, vol. 3, no. 10, pp. 388–415, 2016. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2015EA000136
[10] C. M. Zwlf, N. Moreau, and M.-L. Dubernet, "New model for datasets citation and extraction reproducibility in vamdc," *Journal of Molecular Spectroscopy*, vol. 327, pp. 122 – 137, 2016, new Visions of Spectroscopic Databases, Volume II. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022285216300613
[11] B. H. Schubert C., "Handling continuous streams for meteorological mapping," in *Lecture Notes in Geoinformation and Cartography*, vol. 8628. Springer Verlag, 2019.
[12] S. Prell, R. Mayer, and A. Rauber, "Data access and reproducibility in privacy sensitive escience domains," in *2015 IEEE 11th International Conference on e-Science*, Aug 2015, pp. 255–258.
[13] T. Miksa and A. Rauber, "Using ontologies for verification and validation of workflow-based experiments," *Journal of Web Semantics*, vol. 43, pp. 25 – 45, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570826817300112
[14] P. Ivie and D. Thain, "Prune: A preserving run environment for reproducible scientific computing," in *2016 IEEE 12th International Conference on e-Science (e-Science)*, Oct 2016, pp. 61–70.
[15] M. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. O. Bonino da Silva Santos, P. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. Evelo, R. Finkers, and B. Mons, "The fair guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, 03 2016.
[16] M. Callegari, L. Carturan, C. Marin, C. Notarnicola, P. Rastner, R. Seppi, and F. Zucca, "A pol-sar analysis for alpine glacier classification and snowline altitude retrieval," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 7, pp. 3106–3121, 2016.
[17] S. Schlaffer, M. Chini, D. Dettmering, and W. Wagner, "Mapping wetlands in zambia using seasonal backscatter signatures derived from envisat asar time series," *Remote Sensing*, vol. 8, no. 5, 2016. [Online]. Available: http://www.mdpi.com/2072-4292/8/5/402
[18] S. Schlaffer, P. Matgen, M. Hollaus, and W. Wagner, "Flood detection from multi-temporal sar data using harmonic analysis and change detection," *International Journal of Applied Earth Observation and Geoinformation*, vol. 38, pp. 15 – 24, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0303243414002645
[19] S. Schlaffer, M. Chini, L. Giustarini, and P. Matgen, "Probabilistic mapping of flood-induced backscatter changes in sar time series," *International Journal of Applied Earth Observation and Geoinformation*, vol. 56, pp. 77 – 87, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0303243416301994
[20] F. Vuolo, M. tak, C. Pipitone, L. Zappa, H. Wenng, M. Immitzer, M. Weiss, F. Baret, and C. Atzberger, "Data service platform for sentinel-2 surface reflectance and value-added products: System use and examples," *Remote Sensing*, vol. 8, no. 11, 2016. [Online]. Available: http://www.mdpi.com/2072-4292/8/11/938
[21] D. B. Nguyen, A. Gruber, and W. Wagner, "Mapping rice extent and cropping scheme in the mekong delta using sentinel-1a data," *Remote Sensing Letters*, vol. 7, no. 12, pp. 1209–1218, 2016. [Online]. Available: https://doi.org/10.1080/2150704X.2016.1225172
[22] B. Bauer-Marschallinger, V. Freeman, S. Cao, C. Paulik, S. Schaufler, T. Stachl, S. Modanesi, C. Massari, L. Ciabatta, L. Brocca, and W. Wagner, "Toward global soil moisture monitoring with sentinel-1: Harnessing assets and overcoming obstacles," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 57, no. 1, pp. 520–539, 2019.
[23] A. Dostlov, W. Wagner, M. Milenkovi, and M. Hollaus, "Annual seasonality in sentinel-1 signal for forest mapping and forest type classification," *International Journal of Remote Sensing*, vol. 39, no. 21, pp. 7738–7760, 2018. [Online]. Available: https://doi.org/10.1080/01431161.2018.1479788
[24] B. Bauer-Marschallinger, C. Paulik, S. Hochstger, T. Mistelbauer, S. Modanesi, L. Ciabatta, C. Massari, L. Brocca, and W. Wagner, "Soil moisture from fusion of scatterometer and sar: Closing the scale gap with temporal filtering," *Remote Sensing*, vol. 10, no. 7, 2018. [Online]. Available: http://www.mdpi.com/2072-4292/10/7/1030