# Entwurf eines Frameworks zur Unterstützung von Reproduzierbarkeit für die openEO Plattform

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering and Internet Computing**

eingereicht von

**Bernhard Gößwein**
Matrikelnummer 01026884

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Projektass. Dr.techn. Mag. Tomasz Miksa

Wien, 23. August 2019

Bernhard Gößwein    Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

# Designing a Framework Gaining Repeatability for the openEO Platform

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**Bernhard Gößwein**
Registration Number 01026884

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Projektass. Dr.techn. Mag. Tomasz Miksa

Vienna, 23rd August, 2019

Bernhard Gößwein     Andreas Rauber

# Erklärung zur Verfassung der Arbeit

Bernhard Gößwein
Vorderer Ödhof 1, 3062 Kirchstetten

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. August 2019

Bernhard Gößwein

# Acknowledgements

To my fiancee Viola: Thank you very much for helping me through the stressful days of working on the thesis and for providing breaks and diversions when I needed them.

To my family: Thank you for supporting me through my whole studying time and for motivating me to go on with the thesis.

To my colleagues of the Remote sensing research group: Thank you for patiently waiting for me to finish my studies and letting me work on a thesis within the openEO project.

To my colleagues at EODC: Thank you for providing me all resources I requested and letting me implement the solution on your system.

Last but not least to my supervisors Andreas Rauber and Tomasz Miksa for always quickly replying to my questions and providing me with constructive feedback.

# Kurzfassung

Wissenschaftler im Bereich der Erdbeobachtung verwenden spezielle rechnergestützte Services um Satellitenbilder bei externen Datenanbietern zu verarbeiten. Die zugrundeliegende Quelle der Daten ist meist ähnlich, beispielsweise werden Sentinel Satellitendaten ausschließlich von Copernicus in Zusammenarbeit der European Space Agency betrieben. Die Art der Aufbereitung, Aktualisierung, Korrektur und anschließenden Analyse kann von Anbieter zu Anbieter unterschiedlich sein. Die Anbieter unterstützen meist keine Datenversionierung, beispielsweise wenn Daten korrigiert werden wird dies nicht dokumentiert. Außerdem werden Änderungen in der verwendeten Software nicht kommuniziert und stellt daher eine Black Box für die Wissenschaftler dar. Daher haben Wissenschaftler die diese Systeme nutzen keine Möglichkeit herauszufinden warum die Durchführung des gleichen Programmcodes unterschiedliche Ergebnisse liefert. Dieser Umstand behindert die Reproduzierbarkeit der Experimente im Bereich der Erdbeobachtung. In dieser Arbeit wird gezeigt, wie existierende Datenanbieter modifiziert werden können um Reproduzierbarkeit zu ermöglichen. Die präsentierten Erweiterungen basieren auf den Empfehlungen der Reseach Data Alliance bezüglich Datenidentifizierung und auf das VFramework bezüglich automatisierte Dokumentation und Identifikation der durchgeführten Prozesse. Zusätzlich werden Vorschläge dafür präsentiert wie Anbieter die gesammelten Informationen für die Wissenschaftler zur Verfügung stellt. Die Implementierung der vorgestellten Erweiterungen werden am Earth Observation Data Centre, einem Partner des openEO Projektes durchgeführt. Die Evaluierung des fertigen Systems erfolgt durch die Durchführung von typischen Szenarien der Erdwissenschaften und durch zusätzliche Tests bezüglich der Effekte der Performance- und Speicherbedarfs für das System des Anbieters. Das Ergebnis der Evaluation lässt darauf schließen, dass Reproduzierbarkeit mit nur minimalen zusätzlichen Performance- und Speicherplatzbedarf möglich ist.

# Abstract

Earth observation researchers use specialised computing services for satellite image processing offered by various data backends. The source of data is similar, for example Sentinel satellites operated by Copernicus and the European Space Agency. The way it is pre-processed, updated, corrected 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 thesis, we present how existing earth observation 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. Additionally, we provide suggestions on how backends make the captured information accessible to scientists. 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.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation

Earth Observation (EO) sciences gathers images of the Earth's surface and atmosphere using instruments and cameras on aircrafts or satellites. Advancing technologies in space agencies lead to the usage of mostly satellite images. The 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. The vast majority of backends are available via Service Oriented Architecture (SOA) interfaces. Data providers like Google Earth Engine (GEE)[1] and Earth Observation Data Centre (EODC)[2] host a Web Application Programming Interface (API).Scientists create a local description of the workflow and satellite data to describe experiments. They send the description to the backend and are notified when the processing has finished [32].

Such an approach addresses the performance issues, but it does not allow researchers to take full control of the environment in which their experiments are executed. The backends present themselves as black boxes to the researchers with no possibility of obtaining information on environment configuration, e.g. software libraries used in processing and their versions. Studies in different domains show that the computational environment can have an impact on the reproducibility of scientific experiments and must be documented in order to ensure reproducibility [10] [15] [23]. Still the vast majority of backend providers do not share such environment information. Another problem constitutes the precise identification of data used for processing. EO backends in Europe usually obtain data from the same source, for example from the Sentinel-2 satellites operated by the European Space Agency (ESA). The data provider releases updates and corrections to the data in the case that one of the instruments used for observation was

[1]https://earthengine.google.com
[2]https://www.eodc.eu

wrongly calibrated or broken and raw data had to be processed again. An example for this is the format update of European Space Agency (ESA) in 2017 of the Sentinel 1 dataset, which affected data records of the backends, see [3]. Updated data is released 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 modification was made. 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 the results. For a better understanding of the problem we present the conclusion of two studies regarding reproducibility in EO science after defining used terms of them.
We define the term "reproducibility" by running a second similar experiment, which arrives at the same conclusion as the original experiment. It is necessary to produce evidence for the outcome of an experiment. We define the term "replicability" as the re-run of the same methods of an original experiment to show that the described methods lead to the claimed results [6].
The first study examines existing publications in the scientific EO community. It does not actively try to reproduce the experiments, but looks at the description of the methods. The aim of the paper is to give an overview of the reproducibility and replicability of the publications. The results show that only half of the publications were replicable and none of them reproducible [26].
The second study intends to replicate the work of EO scientists and executes a survey of geoscientific readers and authors. Its aim is to find reasons for the lack of reproducibility in earth observation sciences. One result is that even though 49% of the participants responded that their publications are reproducible, only 12% of them have linked the used code. This leads to the conclusion that the understanding of open, reproducible research is different among the participating scientists. The interpretations are more in favor of own publications regarding replicability. The main findings for the lack of reproducibility in computational geoscience are listed below [15]:

1. Insufficiently described methods
2. No persistent data identifier
3. Legal concerns
4. The impression that it is not necessary
5. Too time consuming

The reproduction of geoscientific papers fails due to the different individual interpretations of the described approach. When the code was published with the publication, alternative software environments produced unequal results. For example different versions of the

[3]https://sentinel.esa.int/web/sentinel/missions/sentinel-1/news/-/article/sentinel-1-update-of-product-format

CRAN library in R created unequal results. The majority of the replication required changes of code and a deeper understanding of the procedures, for example caused by deprecated functions. System dependent issues occurred, which are related to the usage of random access memory and installation libraries of the operating system. The study shows an example of a problematic replication using a resulting image of a publication. It re-executes the experiment to receive this image and compares it with the original one. Figure 1.1 shows a comparison of the recreated image and the original image. The map shows spatially gridded biomass burning and was published initially in [18]. Even though the resulting numbers of the reproduction remain the same, the different aspect ratio changes the appearance of the resulting image and might lead to different interpretations [15].

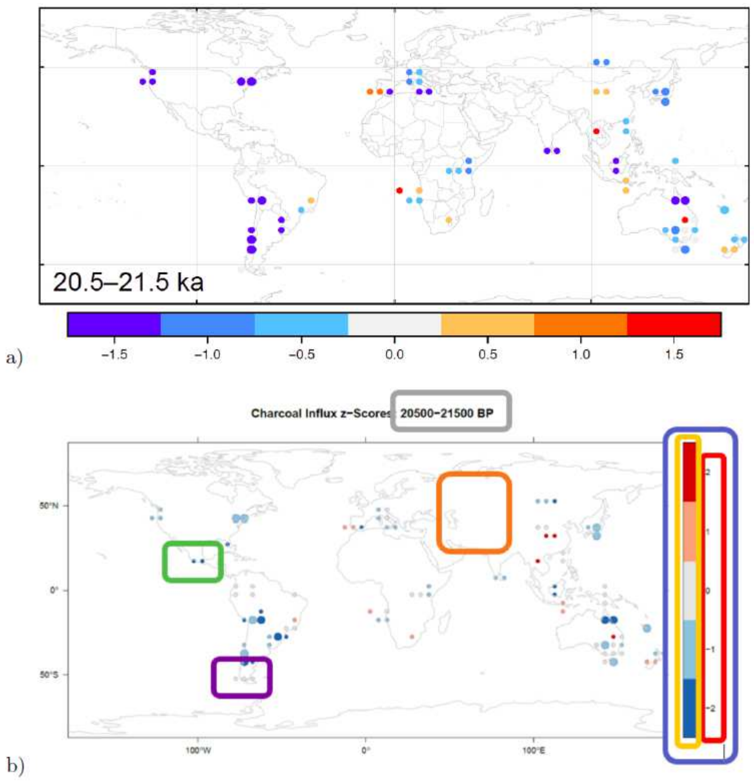


Figure 1.1: Example of a comparison of the original result (a) and a replicated result (b). The boxes are highlighting the differences of the map. The blue box indicates the misplacement of the legend, the purple box shows different color of results, the red box shows a different data type of the legend numbers, the grey box shows a different labeling, the orange box highlights differences in the background map, the yellow box shows a different number of classes and the green box shows results that were not in the original figure [15].

The problem description leads us to create the following three use cases to further specify the aim of the thesis. They describe scenarios focused on scientific experiments that are currently not achievable, but shall be made possible to accomplish with the solution of this thesis. This thesis uses the term job for the definition of a description of a workflow executed at a backend since the term is common in EO sciences. The following sections describe these use cases as well as the example experiment that will be used to guide the reader through the thesis.

### 1.1.1 Example Experiment

This section describes an example of an experiment that a remote sensing scientist wants to execute. The thesis uses the example throughout the thesis for a better illustration of the concepts. They are used in the use cases as well.
The input data of the experiment is Sentinel 2 data developed by the ESA. The area of interest is the province of South Tyrol defined by the bounding area in the "EPSG:4326" projection with the coordinates of a north-west corner (10.288696, 46.905246) and a south-east corner (12.189331, 45.935871). The time of interest is the month of May in 2017. The scientist works on vegetation dynamics and wants to know what the state of the vegetation of South Tyrol was in May 2017. Therefore the minimum of the Normalized Difference Vegetation Index (NDVI)[4] is calculated on the data selection. It derives from the difference between near-infrared (which reflects vegetation strongly) and red light (which vegetation absorbs). So for every pixel of the satellite image, the NDVI value is calculated for every day of May 2017. Then the 31 images are reduced to one by taking the minimum NDVI value of each pixel. Figure 1.2 shows the results of the running example execution.

The execution of this experiment consists of the following workflow:

1. Selecting the Sentinel 2 data records
2. Filtering the Sentinel 2 records by the extent of south tyrol.
   (10.288696, 46.905246) - (12.189331, 45.935871) on "EPSG:4326" projection.
3. Filtering the Sentinel 2 records by May 2017.
4. Calculate NDVI for all days of May 2017.
5. Reduce by the minimum value of May 2017.
6. Create and start a job at the backend.
7. Interpret the resulting image.

[4]https://earthobservatory.nasa.gov/features/MeasuringVegetation/measuring_vegetation_2.php

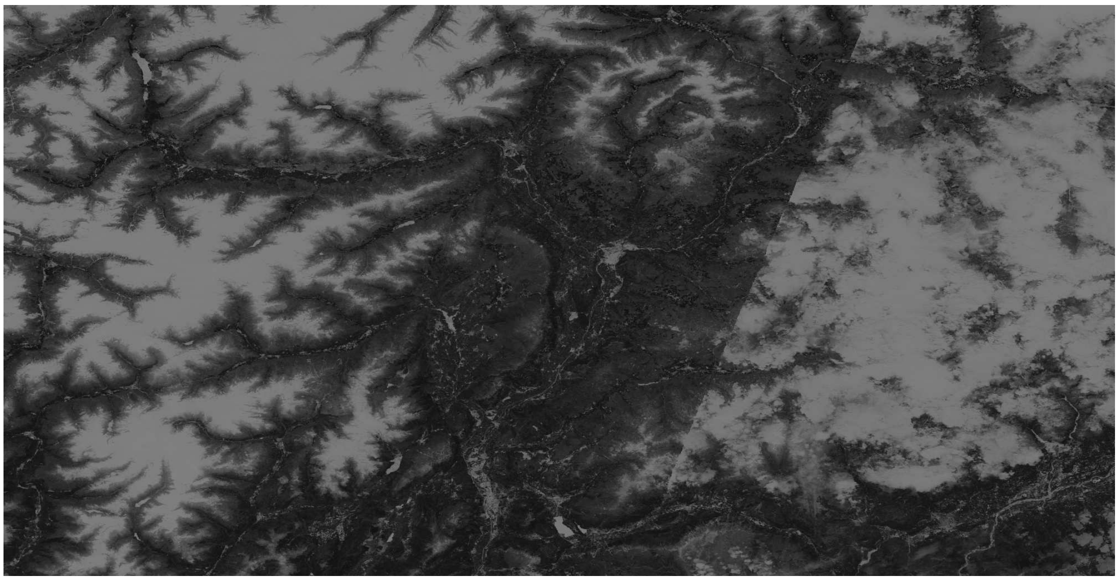


Figure 1.2: Resulting image of the running example.

### 1.1.2 Use Case 1 – Re-use of input data

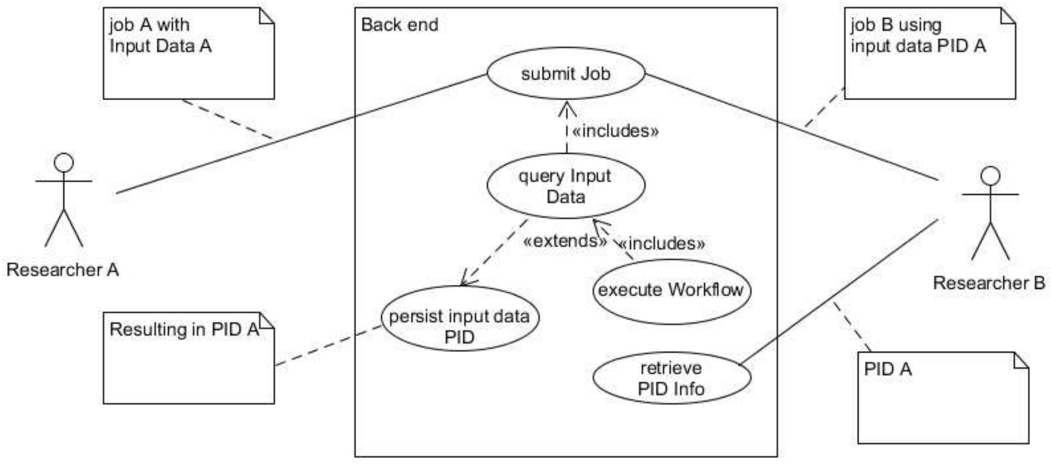


Figure 1.3: Overview of the first use case: Re-use of input data

The first use case is concerned with the re-use of input data between job executions. Reproducible methods are important for the scientific community. Scientists are likely to build on results and methods of publications and this scenario makes the re-use easier. In this use case, a scientist wants to create a publication by running the example experiment, described in Section 1.1.1 using an earth observation backend. By creating and starting the job, the backend generates a Persistent Identifier (PID) for the input data of the experiment. After that, the scientist publishes the results and cites the input data with the resulting PID. It redirects to a human-readable landing page that provides meta information about the dataset. Another scientist, also interested in the vegetation of South Tyrol, wants to use the same input data but chooses a different approach of processing it (for example the maximum instead of minimum reduction function). Hence, the input data PID can be used to re-use the same data. The backend has to be capable of resolving the PID automatically to enable users to work with the same input data for a new job. The data provider needs to persist the data defined by a PID even if updates on the data take place.

- **Input Data A**: Sentinel 2 data of the area of South Tyrol in May 2017.
- **Job A**: Taking the **minimum** NDVI of the area of South Tyrol in May 2017.
- **Job B**: Taking the **maximum** NDVI of the area of South Tyrol in May 2017.

Figure 1.3 gives an overview of the first use case.

The scenario sequence of actions is summarized in the following steps:

1. Researcher A runs job A at the backend.
2. Researcher A retrieves the used input data PID of job A.
3. Researcher A cites the input data with the PID in a publication.
4. Researcher B uses the same input data, by applying the data PID of job A for job B.

### 1.1.3 Use Case 2 – Providing job execution information

The second use case is similar to the first one but it is exclusively concerned with job dependent environment information. The scientist automatically gets environment data about the job execution e.g. used software packages and their versions. The motivation for this is to add transparency of the job execution for users, so that researchers can describe their processes in more detail. It helps geoscientists to understand why results differ from executions in the past. Figure 1.4 gives an overview of the use case. The following steps summarize the scenario sequence of actions:

1. Researcher runs a job (job A) at a backend.
2. Researcher wants to describe the job environment.

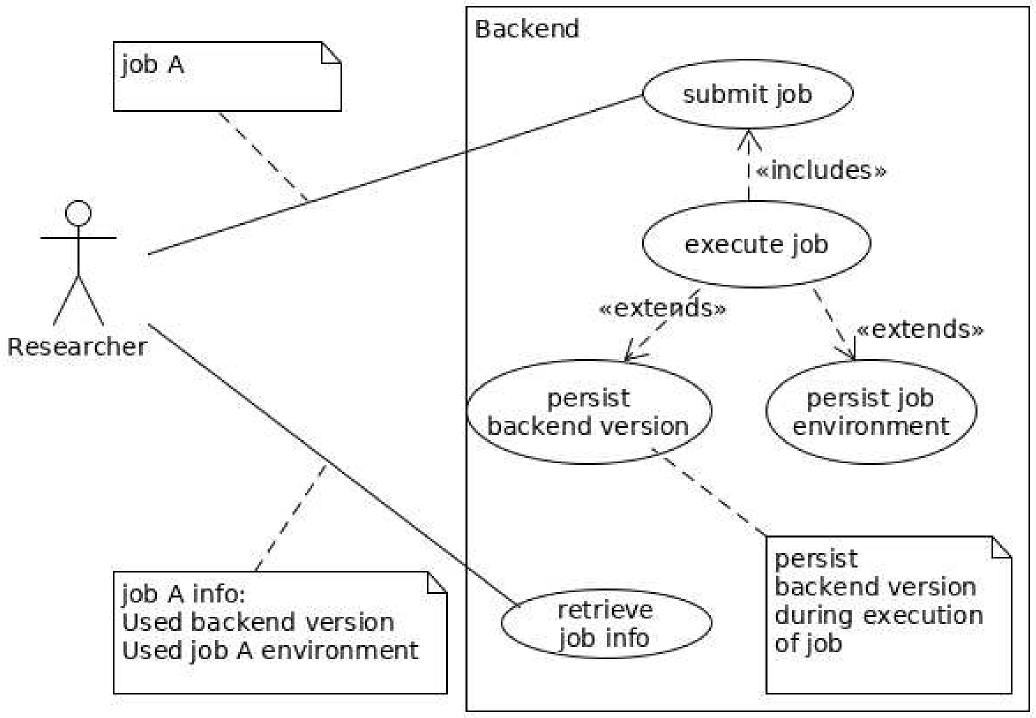


Figure 1.4: Overview of the second use case: Providing job execution information

### 1.1.4 Use Case 3 – Compare different job executions

The third use case is dedicated to the comparison of job executions. The goal is for geoscientists to be able to compare different jobs not only by their results, but by the way they were executed. The same backend applies the comparison between a job execution and another job execution. Therefore, the processing implementation and the input data must be identifiable. To make the comparison more transparent to the users, additional data is added to the job environment data e.g. an output checksum. In addition to the previous conditions, a visualization of the differences for the users lowers the access barrier for them to use the feature. Figure 1.5 gives an overview of this use case. The following steps summarize the scenario sequence of actions:

- **Job A**: Taking the minimum NDVI of the area of South Tyrol in May 2017.
- **Job B**: Taking the minimum NDVI of the area of South Tyrol in May 2017.
- **Job C**: Taking the maximum NDVI of the area of South Tyrol in May 2017.

1. Researcher runs a job (job A) at the backend.

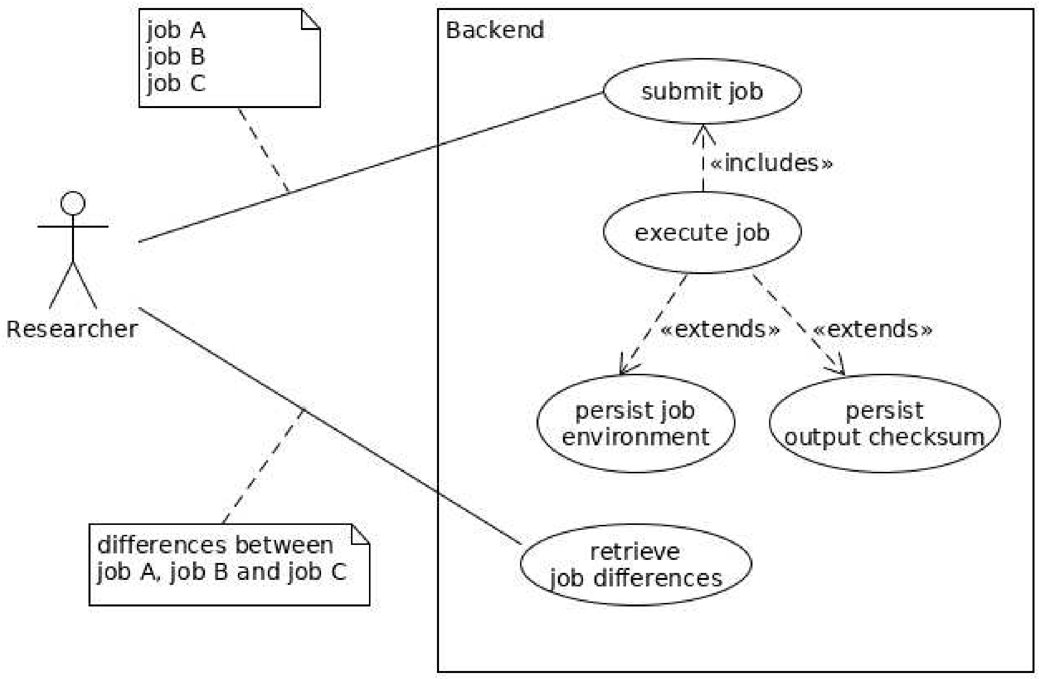


Figure 1.5: Overview of the third use case: Compare different job executions

2. Researcher re-runs the same workflow used for job A at the same backend resulting in a new job (job B).

3. Researcher runs a different job (job C).

4. Researcher receives a comparison of the jobs (A, B, C) by their environment and outcome.

## 1.2 Research Questions

The aim of the thesis is to propose a framework for making reproducibility conceivable in the earth observation community. The solution enables users to re-execute jobs and validate the results. It provides the scientists accessible differences in the execution of the job and the data, without changing their approach of doing research. To achieve this, a model for capturing the environment of the backends has to be discovered. Considering the problem description and the scenarios of the previous sections, the following research questions can be formulated:

- **What information must be captured from an earth observation backend, so that a job execution can be repeated like the original execution?**
  - How can the data of the original execution be identified?
  - How can the environment of the original execution be reproduced?
  - Which parts of the backend need to be extended?
  - How can the result of a re-execution in future environments be verified?
- **What information must be captured to enable validation of a job re-execution on an earth observation backend?**
  - What are the validation requirements?
  - How can differences in the environment between the executions be discovered?

The solution prototype is within Open Source Earth Observation Project (openEO) (details see Section 2.5). It concludes with recommendations for the openEO specification on how to improve re-execution validation for the users. Using the standard of openEO enables to restrict the target of the thesis. It also facilitates all backends compliant to openEO an entrance to the proposed features of this thesis.

## 1.3 Methodological Approach

There are already technologies available to solve parts of the issues described in the determined problem. This section provides the three key parts of the methodology used for the proposed solution of the thesis.

1. **Data Identification** The input data has to be identifiable, to accomplish the capturing of jobs described in the use cases of Section 1.1. The Research Data Alliance (RDA) has identified 14 general rules [34] for 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.

2. **Job Environment** The VFramework [23] and context model [19] were proposed to automatically document environments in which computational workflows execute and to enable their comparison.

3. **Backend Standardization** The openEO project [27] works on creating a common EO interface to enable interoperability of EO backends by allowing researchers to run their experiments on different backends without reimplementing their code. We contribute to the openEO standard to provide compliant backends with reproducibility concepts.

By combining these three elements, 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. We present this solution improving reproducibility of earth observation experiments executed at the 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 executed at backends can be captured and compared using the VFramework to identify whether any differences in software dependencies among two executions exist.

## 1.4 Structure of Work

This thesis is structured as follows. After this introduction, Chapter 2 gives an overview of related scientific activities in the area of reproducibility in the earth observation sciences and reproducibility in other areas with similar objectives. Chapter 3 provides the concept to address the research questions defined in Section 1.2. It is the design used for the prototype implementation on the openEO compliant EODC backend described in the following chapter. Chapter 4 gives detailed insight into the modifications needed at the backend to achieve the features described by the use cases. The next chapter, Chapter 5 is concerned with the evaluation of the implementation of Chapter 4. For the evaluation we simulated typical 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. Chapter 6 summarizes the outcome of the implementation and evaluation. It contains a discussion on results achieved and future work.

CHAPTER 2

# Related Work

This chapter describes the related work that influenced this thesis. It informs the reader about concepts related to the proposed solution. The information is structured in subsections, each representing important technologies or concepts in the context of this thesis.
The first section presents the concepts behind reproducibility in computer science.
The second section describes the state of reproducibility in earth observation science.
The third section presents concepts related to data identification.
The fourth section consists of other existing implementations to achieve reproducibility.
The last section of this chapter describes the openEO project and the EODC backend, which we use for the proof of concept implementation in Chapter 4.

## 2.1 Reproducibility

The term of reproducibility is defined as a new experiment based on an original experiment by an independent researcher in the manner of the original experiment. Reproducibility aims to gain additional evidence on the result of the original result by creating an independent experiment that shows similar results. Repetition defines a re-run of the same experiment with the same method, same environment, and a very similar result. The repetition aims to check if the methods described in a publication are resulting in the purposed outcome [46]. Achieving reproducibility is a common problem in all scientific areas. Therefore there are ten rules defined to gain a common sense about reproducibility. They are motivated by the basic idea that every result of interest has to be associated with a used process and data. The researcher has to provide external programs as well as custom script versions. Using version control software is recommended. Besides, one of the steps defines a rule to make scripts and their results publicly available [36]. Reproducibility is the crucial topic of The Fourth Paradigm. It leads to the term eScience, which has the aim of bringing science and computer technologies closer together. The general concept is to enable scientific procedures with new information technologies used by data-intensive sciences. The expected result of eScience is to get all scientific papers publicly available, including the necessary data and workflows, so that scientists can interact more efficiently [11]. eScience has the potential to enable a boost in scientific discovery. It provides approaches to make digital data and workflows citable. The publication [35] discusses a general way of reaching this. It describes an approach to look at whole research processes by introducing Process Management Plans, other than limiting it to data citation. It demonstrates the capturing, verification, and validation of the input data for a computational process. Computer sciences have the issue of high amounts of published papers that do not provide enough information to make them reproducible. This is not solved by the scientists that need to make additional effort, but by providing new tools for scientists that allow it automatically [22]. There are some additional issues on reproducibility in computer science e.g. in the case that used software technologies are deprecated and not available anymore. Therefore, persisting the execution context is needed to achieve a re-execution of the experiment. One proposed solution is the VFramework described in more detail in Section 2.1.2. The PRIMAD model defines a set of variables that define an experiment. The relationship of the original execution to a re-execution is visualized by noticing changes in the variables. The following variables of an experiment are used to describe the relationship [8]:

- **P** Platform / Execution Environment / Context (e.g. Python 2.7, Windows 10,. . . )
- **R** Research Objectives / Goals (e.g. sorting the input)
- **I** Implementation / Code / Source-Code (e.g. script in Python)
- **M** Methods / Algorithms (e.g. quick sort)
- **A** Actors / Persons (e.g. researcher that is executing the experiment)
- **D** Data (input data and parameter values) (e.g. input data that is to be sorted)

If there is a re-execution that is different on every variable to the original one, except for the same method (M) and goal (R), it is considered a reproduction [8].
Data citation is a vital issue of reproducing results of past experiments. Preserving the exact workflow without persisting the original data has no positive effect on the scientific community. If the data used in an experiment is not available anymore, or not specified explicit enough, then there is no chance of reproducing it no matter how much information about the execution is known. In earth observation persisting the data for reproducibility in the future is an issue discussed in the literature [33]. Gaining data

identification in digital sciences has an official working group named Working Group on Data Citation (WGDC), which created 14 recommendations on data citation further explained in Section 2.3.

### 2.1.1 PROV-O

In 2003 the World Wide Web Consortium published the PROV model as a standard concerning provenance definitions. It is defined in twelve documents. In the context of this thesis the PROV Ontology (PROV-O) is the most relevant [45]. PROV-O is a standard language using OWL2 Web Ontology. It is a lightweight concept capable of a broad spectrum of applications.

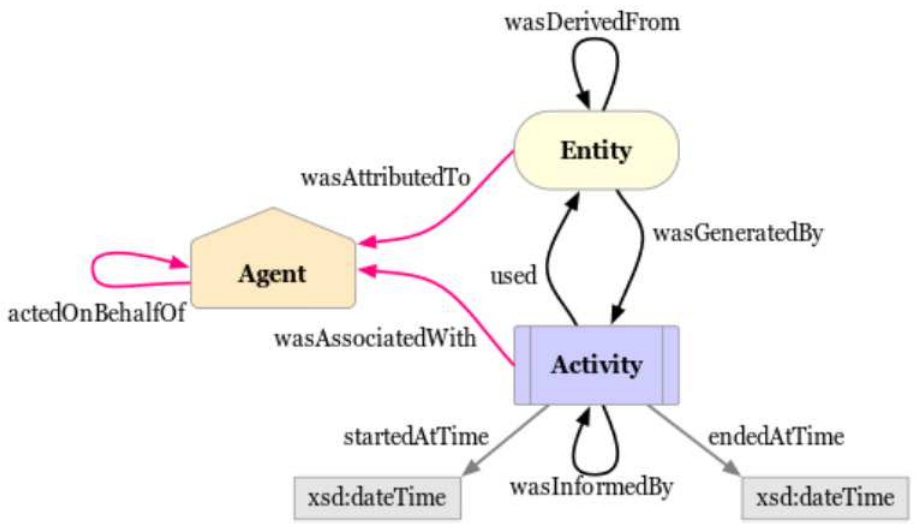


Figure 2.1: Overview of the main components of PROV-O [45]

Figure 2.1 shows the basic setup of the PROV-O concept. It consists of three main elements. The *Entity* is any physical, digital or conceptual thing. Provenance records describe *Entities* that can consist of references to other *Entities*. Another element is the *Agent*, which is responsible for *Activities* and that they are taking place, e.g. software, persons, or organizations. The association of an *Agent* to an *Activity* defines the responsibility of the *Agent* for the *Activity*. An *Activity* describes what happened that the *Entity* has come to existence and how attributes of an *Entity* changed [45]. The design chapter 3 does not specify the representation of the context information. Therefore the PROV ontology can be used to represent the information. This thesis implements the PROV-O standard as one of the available provenance representations for the user. Section 4.4.1 shows the implementation.

### 2.1.2 VFramework

The VFramework defines parallel capturing of provenance data during the workflow execution. During the original execution, evidence gets collected into a repository e.g. logging. The context model of the execution persists the needed data e.g. in a database record. Re-execution is verified and validated using the provided provenance data in the context model of the original execution and the context model of the re-execution. The provenance data divides into static and dynamic data. Static data defines data that is not dependent on the execution of the experiment e.g. the operating system and installed packages. The static environment information is, therefore, independent of the configuration of the workflow. Dynamic data is captured during the execution of the original experiment e.g. Python version of the execution or used input files. It describes data dependent to a workflow execution [21]. Figure 2.2 shows an overview of the VFramework concept described above.

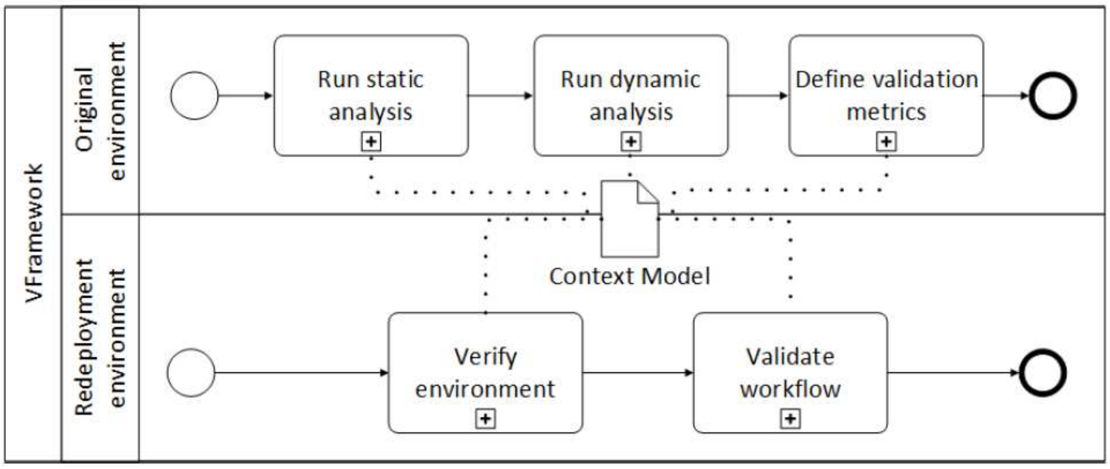


Figure 2.2: Overview of the Concept of the VFramework [21]

## 2.2 Earth Observation Science

This section describes Reproducibility in the context of the computational geoscience. A study tests the reproducibility and replicability of scientific papers in geoscience by obtaining more than 400 papers [26]. In [26] a reproduction is defined by an exact duplicate of an experiment, whereas it defines replication as a resemblance of the original execution, but allowing variation e.g. different scales. Table 2.1 describes the difference between reproduction and replicability in PRIMAD terms. Since the definition of the paper differs from the definition of this thesis the terms are marked as Reproducibility' and Replicability'. Only half of the test group publications are replicable, and none of them reproducible. There are publications to address the lack of reproducibility in the earth observation science. The following sections summarize these concepts.

Table 2.1: PRIMAD description of reproduction and replication according to [26]

| | **Reproduction'** | **Replication'** |
|---|---|---|
| **P**latform | same | different |
| **R**esearch Objectives | same | same |
| **I**mplementation | same | different |
| **M**ethods | same | different |
| **A**ctors | different | different |
| **D**ata | same | different, but similiar |

### 2.2.1 Vadose Zone Journal (VZJ)

In order to face the issue of reproducibility in geoscience the Vadose Zone Journal (VZJ) started a Reproducible Research (RR) program in 2015 [43]. The earth observation science is a big part of VZJ publications, and most of them are not applying the open computational science guidelines. The main reasons are behaviors of scientists that do not see the overall benefit of putting effort into documentation. Therefore, the VZJ started an RR program to publish the code and data alongside scientific papers. The aim of the strategy is to lower the access barrier for scientists to publish their research work entirely. Goal of the project is to create a community of researchers with a shared sense of reproducibility and data citation on the platform. The community then animates other scientists to join the approach. On time this thesis is written, the service is still available[1], but there were no results available on how much it is used [43].

### 2.2.2 The Geoscience Paper of the Future (GPF)

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 and with unique identifiers and pointers to public code repositories;
3. documentation of the provenance and workflow for each figure or result.

Figure 2.3 visualizes the differences with a reproducible paper. In addition to the characteristics of the reproducible paper, the GPF focuses on publishing the data publicly with open licenses with citable persistent identifiers. The GPF proposes a set of 20

[1]https://dl.sciencesocieties.org/publications/vzj/author-instructions-reproducible-research

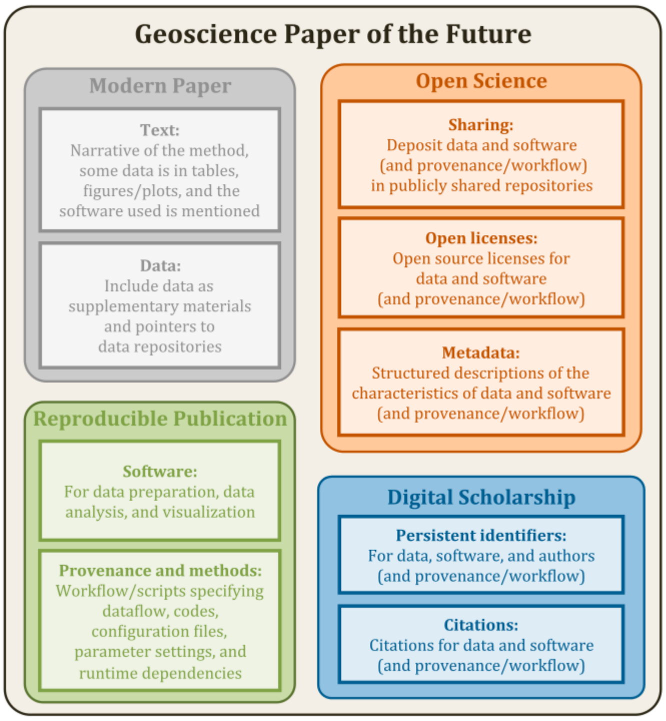

Geoscience Paper of the Future
Modern Paper
Text:
Narrative of the method, some data is in tables, figures/plots, and the software used is mentioned
Data:
Include data as supplementary materials and pointers to data repositories
Open Science
Sharing:
Deposit data and software (and provenance/workflow) in publicly shared repositories
Open licenses:
Open source licenses for data and software (and provenance/workflow)
Metadata:
Structured descriptions of the characteristics of data and software (and provenance/workflow)
Reproducible Publication
Software:
For data preparation, data analysis, and visualization
Provenance and methods:
Workflow/scripts specifying dataflow, codes, configuration files, parameter settings, and runtime dependencies
Digital Scholarship
Persistent identifiers:
For data, software, and authors (and provenance/workflow)
Citations:
Citations for data and software (and provenance/workflow)

Figure 2.3: Relationship between reproducible publications and geoscientific papers of the future [9]

recommendations for geoscientists regarding data accessibility, software accessibility, and provenance information. GPF authors may have reasons to be not able to follow all rules and therefore, have to find workarounds and propose areas for future improvements. The strategy of the GPF community is to educate the scientist to make reproducible publications, by making training sessions[2], instead than providing tools. The solution of this thesis 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 implementation of their experiments.

### 2.2.3 Climate Change Centre Austria (CCCA)

The Climate Change Centre Austria (CCCA) is a research network for Austrian climate research available since 2016. Its aim is the provision of climate-relevant information, the inter-operable interfaces, and long term archiving of scientific data. In 2017, data citation for the Network Common Data Form (NetCDF) format was added to the project. The architecture of the implementation and the technology used in the background is similar to the set up of the EODC backend. Therefore, the approach of enabling data

[2]http://scientificpaperofthefuture.org/gpf/events.html

identification was similar to this thesis. CCCA is open source and available at GitHub[3]. Similar to the approach of this thesis, the concept for data identification are the RDA recommendations. Figure 2.4 shows a technical overview of the CCCA implementation. It uses a ckan[4] web server to handle the request and responses of the user, which are then passed to a Python application. The Python application is responsible for the query store functionality. The core element is the Thredds Data Server (TDS)[5], which is responsible for the data archiving. The Python application of the CCCA overview is similar to the implementation of this thesis. The main difference is the objectives of the CCCA service compared to the EODC backend. On the CCCA platform, any climate-relevant information (e.g. air temperature or frost days) can be uploaded and persisted. On the EODC backend, preprocessed global earth observation data is persisted, which is used as input data for processing chains. Therefore, the data on EODC is more homogeneous than the data on the CCCA platform. Nevertheless, the concept of enabling data identification is similar in both projects. The CCCA implementation inspired the query store implementation of this thesis. The query result differs, because EODC uses a different file format (Georeferenced Tagged Image File Format (GeoTiff) instead of NetCDF). Another difference is that CCCA uses HTTP GET requests as query, whereas the EODC backend uses Extensible Markup Language (XML) based queries, which are restricted by the openEO API specification [40].

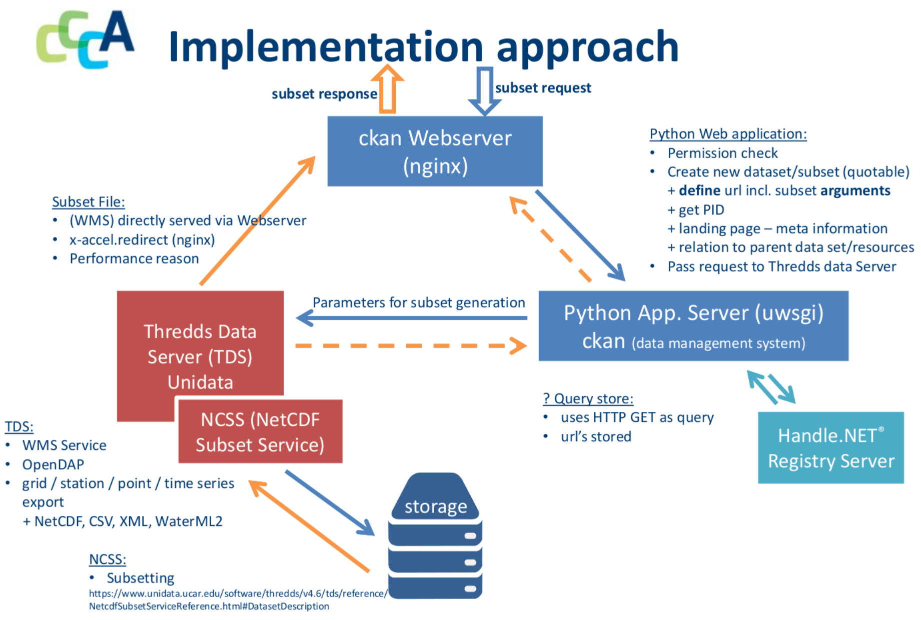

CCCA
Implementation approach
subset response
subset request
ckan Webserver (nginx)
Python Web application:
• Permission check
• Create new dataset/subset (quotable)
+ define url incl. subset arguments
+ get PID
+ landing page – meta information
+ relation to parent data set/resources
• Pass request to Thredds data Server
Subset File:
• (WMS) directly served via Webserver
• x-accel.redirect (nginx)
• Performance reason
Parameters for subset generation
Thredds Data Server (TDS) Unidata
Python App. Server (uwsgi)
ckan (data management system)
NCSS (NetCDF Subset Service)
? Query store:
• uses HTTP GET as query
• url's stored
Handle.NET® Registry Server
TDS:
• WMS Service
• OpenDAP
• grid / station / point / time series export
+ NetCDF, CSV, XML, WaterML2
storage
NCSS:
• Subsetting
https://www.unidata.ucar.edu/software/thredds/v4.6/tds/reference/NetcdfSubsetServiceReference.html#DatasetDescription

Figure 2.4: Technical overview of the CCCA NetCDF data citation implementation

[3]https://github.com/ccca-dc
[4]https://ckan.org
[5]https://www.unidata.ucar.edu/software/thredds/current/tds/

## 2.3 Data Identification

For the aim of this work, the input data is a key element of the captured data. If the input data can not be identified correctly, the capturing of the processing on it does not gain useful information. Therefore the identity of the data has to be guaranteed. The Research Data Alliance (RDA)[6] is an international body issuing recommendations helping to remove barriers in data sharing. Recommendations are based on a community consensus worked out within working groups. The Data Citation working group[7] has identified 14 rules for identification of data used in computation. It allows to identify and cite arbitrary views of data, from a single record to an entire data set in a precise, machine-actionable manner. Further it enables to cite and retrieve that data as it existed at a certain point in time, whether the database is static or dynamic. In the following the recommendations are summarized [34]:

- **R1: Data Versioning**
  Changes on a data record must result in a new version of the data record and the persistence of the deprecated data records. All data record versions have to be identifiable and accessible.

- **R2: Timestamping**
  All changes to the database have to be comprehensible via timestamps. Every time changes are applied to the data, there has to be a timestamp persisted to describe when it happened.

- **R3: Query Store Facilities**
  There has to be a query store implemented at the data provider. The applications uses it to store queries and additional information. The database has to store, according to [34], the following things:

  – The original query as applied to the database
  – A potentially re-written unique query created by the system (R4, R5)
  – Hash of the (unique) query to detect duplicate queries (R4)
  – Hash of the result set (R6)
  – Query execution timestamp (R7)
  – Persistent identifier of the data source
  – Persistent identifier for the query (R8)
  – Additional information (e.g. author or creator information) required by the landing page (R11)

[6]https://rd-alliance.org
[7]https://rd-alliance.org/groups/data-citation-wg.html

- **R4: Query Uniqueness**
  Since it is not desirable to have equal queries with the same result stored at the query store, there needs to be a unique query that can be directly compared to other queries. Hence, there needs to be an algorithm to normalize the queries and to guarantee their uniqueness.
- **R5: Stable Sorting**
  The sorting of the resulting data has to be unambiguous, if the sequence of data item presentation is essential for the reproduction.
- **R6: Result Set Verification**
  To ensure that the resulting data of the query is comparable there have to be a checksum or hash key of it.
- **R7: Query Timestamping**
  There has to be a timestamp assigned to every query in the query store, which can be set to the latest update of the entire database, or the query dependent data of the database, or simply the time of query execution.
- **R8: Query PID**
  Every query record in the query store must have a Persistent Identifier (PID). There must not be a query with the same normalized query and query result checksum tuple.
- **R9: Store the Query**
  The data described in previous recommendations have to be persisted in the query store.
- **R10: Automated Citation Texts**
  To make the citation of the data more convenient for researchers, there shall be a automatic generation of the citation text snipped containing the data PID.
- **R11: Landing Page**
  The PID shall be resolvable in a human readable landing page, where data mentioned in the previous recommendations is provided to the scientist.
- **R12: Machine Actionability**
  Providing an API landing page so that not only humans, but machines can access the data by resolving the PID.
- **R13: Technology Migration**
  If the query store needs to be migrated to a new system, the queries have to be transferred too. In addition the queries have to be updated according to the new setup, so that they still work exactly like in the old system.
- **R14: Migration Verification**
  There shall be a service to verify a data and query migration (see R13) automatically, to prove that the queries in the query store are still correct.

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 [48], climate change [41] to health policy planning [31]. 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.

## 2.4 Tools for Reproducibility

The section describes tools that are designed to solve similar problems or subproblems addressed by this thesis. There is an explanation of why the specific tool was not used for the prototype of this thesis or how it was used in parts of the solution.

### 2.4.1 noWorkflow

noWorkflow is introduced in 2015 as a provenance capturing tool with the aim of not influencing the way researchers work. As proof of concept, noWorkflow uses Python as programming language. A SQLite database stores the provenance information categorized in trials. A trial represents the environment information of one execution. The main benefit of noWorkflow is that it does not instrument the code, and it automatically captures the definition, deployment, and execution environment in a local SQLite database. The command line interface of noWorkflow is capable of providing access to the stored data. In addition to just retrieving the information about the execution environment, analyses features are added Figure 2.5 gives an overview of the noWorkflow architecture. It shows the three modules of capturing, storing and analyzing of the provenance data [24].

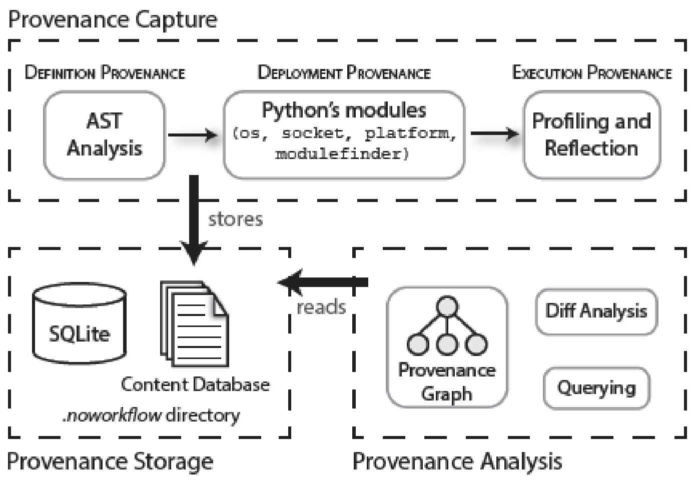


Figure 2.5: Architecture of noWorkflow [24]

The noWorkflow framework improved regularly after the first announcement. noWorkflow was extended by an feature of tracking the evolution of the trials [30]. It improves the possibility to compare trials and to visualize the history of past trials. Next, the fine-grained provenance tracking extension of noWorkflow is introduced [28]. It enables to split the trial into single execution lines. It adds a visualization of all called functions in a graph. It has the limitation of multiple function calls in one line on complex data structures such as dictionaries, lists, and objects. Next, noWorkflow combines with yesWorkflow, the concept of gathering information about the provenance using comments and annotations [20]. The combination enabled more detailed environment information, querying, and visualizations [29]. In this thesis, noWorkflow was used in an early attempt for the implementation but is not part of the final solution due to the high amount of the captured data by noWorkflow and the additional requirements needed by the EODC backend for using it.

### 2.4.2 ReproZip

ReproZip is a packaging tool to enable the reproducibility of computational executions of any kind. It automatically tracks the dependencies of an experiment and stores it into a package. ReproZip can execute the package on another machine. Additionally it is capable of letting the re-executor modify the original experiment. It was developed for the SIGMOD Reproducibility Review[8]. Figure 2.6 shows the architecture of ReproZip. ReproZip traces the system calls to create a package configured by the configuration file. Thus, it produces a single file with the extension ".rpz". ReproZip opens these files, unpack them and re-executes it on a different machine. ReproZip aims to make reproducible science easy to apply for single experiments [4]. The reason why it is not used in the solution of this thesis is that the capturing is very fine granulated. This takes too much performance from the backends, which is a key selling point for backend providers. Depending on the backend, the payment for users may be dependent on the duration time of the processing. Another issue with ReproZip in the context of this thesis is that it is not capable of capturing the big data of the backends within the package, because it would take too much space and performance.

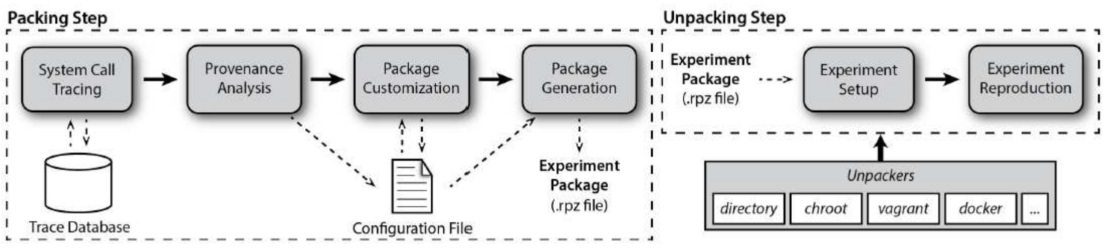


Figure 2.6: Overview of the ReproZip concept from [4]

[8]http://db-reproducibility.seas.harvard.edu/

### 2.4.3 Docker / Smartcontainer

Docker containers are ubiquitous in geoscience executions. The advantages of reproducible research and cost savings by using Docker containers are discussed in the community for the Geographic Object-Based Image Analysis (GEOBIA) [14]. The implementation of the image analysis is implemented with a docker image with a user interface that can also be used by non-experts. Studies for the more general Object-Based Image Analysis (OBIA) using this Docker containers were carried out [13]. The conclusion is definite, with only little shortcomings in the usability. The aim was to use docker images to make it easier for scientists to re-run an experiment on the OBIA system. The remaining question is how the docker configuration is preserved in a manner that it can be reproduced within different environments. Therefore, a "workflow record" is introduced by storing the environment and entities involved in addition to the Docker description file. SPARQL query is introduced to create the possibility to use the container as a repository of data [7].
Smart container is another approach of preserving a docker container. The aim of a smart container is an ontology and software to preserve docker data. It uses the PROV-O standard to define the provenance [12]. Docker containers are used at the EODC backend for running all services. Therefore they are part of the solution. The description files of the used docker containers are persisted in the GitHub repository of the EODC backend. Hence they are identifiable by the backend version defined in Section 3.

### 2.4.4 Version Control Systems

Version Control Systems (VCS) became an essential part of all computational sciences. It enables to persist versions of code and the possibility to head back to a particular version. Before that, programmers tend to have multiple directories to make versions of the code. The basic idea of VCS is a command line interface that makes it possible to set a version of the current state of the code. These versions can be accessed in the future, without changing other versions of the code and without manually added folder structures [16]. In this thesis, we use Gitorious (Git) as VCS of the solution. Versions in Git are defined as commits and are stored locally and can be published to an external server. There, other users can access the commits if they have enough rights set. The commits are stored both, locally and remotely [17]. openEO uses Git as default code versioning tool. GitHub is used as the publicly available server. Since openEO is an open source project, the code of every backend, core API and the client software is available at GitHub without restrictions.

### 2.4.5 Hash

Hash functions are used to validate data without having to save the whole data. They have two important properties to work correctly. First, the probability that two different inputs have the same hash outcome has to be low. Second, it must to be hard to find a message with the same hash value as an already known message. These properties makes the hash functionality a standard tool to identify data without having to save the original [44]. In this thesis, the Secure Hash Algorithm (SHA)-256 is used for the entries of the context model, mostly to compare differences in data outcomes.

## 2.5 openEO

The openEO project consists of three modules. The client module written in the programming language of the users and entry point of the users. Second, the backend drivers that enables for every backend to understand the calls from the clients, so the interface at the backends to support openEO. Third, the core API, which specifies the communication. The core API is the key element of the openEO project and a standard that the backend providers accepted to implement on their systems to be openEO compliant. The backend drivers are the translation of the client calls to the backends specific APIs. This architecture decouples the clients from the backends so that every openEO client can connect to every openEO compliant backend. An example of a workflow is the example defined in Section 1.1.1 using the Python client to access the openEO interface of the EODC backend [27].

The communication is specified as an OpenAPI description, which is a way of defining Representational State Transfer (REST)ful communication in a standardized way. The definition consists of the endpoints at the backend and the requests and the responses. The whole communication protocol is specified with OpenAPI [42]. In the following, the relevant RESTful request types in openEO and the policy of choosing between them are introduced:

- **GET Request**
  GET requests are used to retrieve data from the backends. The functionality is limited to read operations.
  (e.g. GET /collections returns a list of available collections at the backend.)

- **POST Request**
  POST requests are used to create new data records at the backend. It is also used to send information in the body of the request.
  (e.g. POST /jobs creates a new processing job at the backend, which is defined in the body of the request. It creates a new job identifier.)

- **PATCH Request**
  PATCH requests are used to update existing records at the backend.
  (e.g. /PATCH /job/job_id modifies an existing job at the backend but maintains the job identifier.)

- **DELETE Request**
  DELETE requests are used to remove existing records at the backend.
  (e.g. /DELETE /job/job_id removes an existing job from the backend.)

### 2.5.1 Job Execution

Figure 2.7 shows a sequence diagram of a job execution using the client application and an openEO compliant backend. Users define the workflow in a client-specific programming language using the openEO client. The main parts of the job execution definition is the description of the input data, the filter operations and the processes that should be executed on the filtered data. Therefore, openEO introduces the process graph. openEO defines it as a tree structure of processes with their input data and successor process. The openEO coreAPI specifies every operation as process, even filter and data retrieval operations. The process graph is in JavaScript Object Notation (JSON) format and gets generated by the clients in the background without users having to deal with it directly. Figure 2.8 shows the running example process graph defined in Section 1.1.1.

The backends interpret the process graph from inside out. Figure 2.8 displays the process graph of the running example of Section 1.1.1. It describes the calculation of the minimum NDVI image of the Sentinel 2 satellite over South Tyrol in May 2017. The element in the center of the process graph defines the input data identifier in the "imagery" block, with

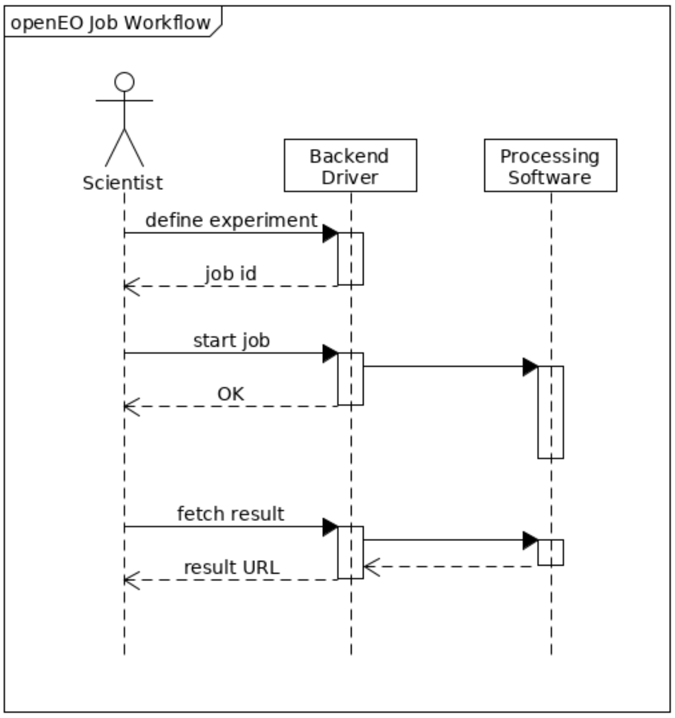
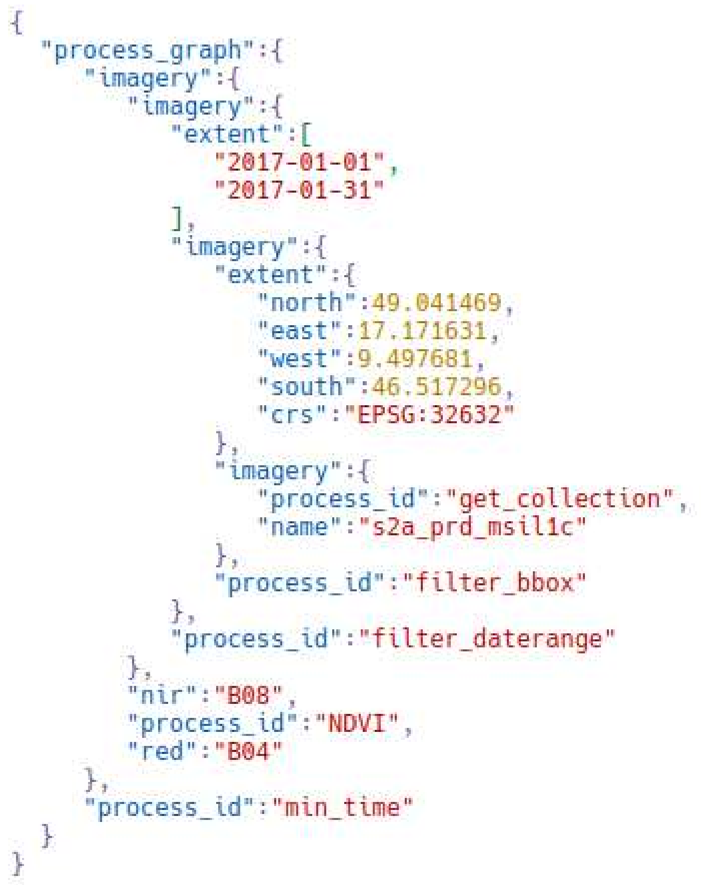


Figure 2.7: Overview of the openEO procedure to execute a job and retrieve the results

the "get_collection" process id. In this case, "s2a_prd_msil1c" is chosen as input data identifier, since it defines Sentinel 2 data at the EODC backend. After reading the input data id, the backend iterates one step up in the hierarchy of the process graph and calls the process "filter_bbox" with the parameters "west", "east" etc.. It filters the satellite data spatially, by only considering data within the bounding box (e.g. the area over South Tyrol). Next, the "filter_daterange" process is used to filter the imageries temporally, by only using data from May 2017. Every process beginning with "filter_" is defined as filter process, so an operation that restricts the input data. How the filter operations are implemented depends on the infrastructure of the backend provider. Section 2.5.3 shows how EODC implements it. The output data of the previous process is the input data of the next process. After the last filtering process, the NDVI gets called by the process "NDVI" with the parameters "red" and "nir". They are setting the identifier of the bands of near-infrared and red light used by the backend. The two bands are used in the formula of the NDVI calculation (see Section 1.1.1). Next, the minimum value is taken from all images using the "min_time" process. It is a reducer process, which transforms the image stack of May 2017 into a single result image. Figure 2.9 shows the same process graph from the backend point of view. It visualizes the order of how the processes are executed. To transmit the process graph of Figure 2.8 at a backend, the openEO client puts it into the body of the POST /jobs endpoint request.



```
{
  "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"
  }
}
```

Figure 2.8: Process graph of the running example defined in Section 1.1.1

There are two different kind of process executions depending on the capabilities of the backend named synchronous and asynchronous calls. Synchronous calls are directly executed after the backend receives them, and the script of the user has to wait until the job is finished for proceeding. For example, the program waits after sending the process graph to the backend until the backend returns the result. An asynchronous call does not get executed until the user starts the execution on the backend through an additional endpoint call. When the processing is finished, the user can download the result at another endpoint of the backend. For asynchronous calls, there is the possibility to subscribe to a notification system on the backend, so that the user gets notified when the job execution finished. The processes are defined at the openEO core API and independent of the backend they get called at, other than the data identifier, which may differ at each backend.
The previous example uses a process graph that only consists of the available processes and data of the backend. Within the openEO[9] project, there is the possibility to define

[9]https://github.com/Open-EO/openeo-openshift-driver/tree/release-0.0.2

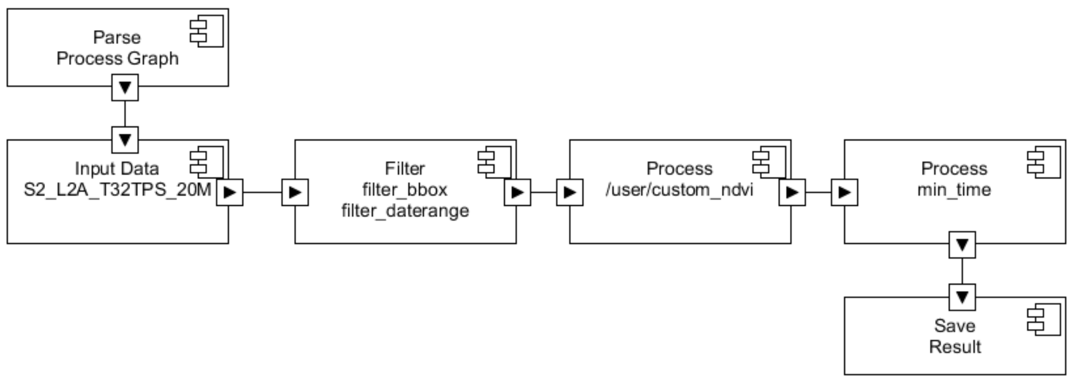


Figure 2.9: Action chain of the backend after receiving the process graph of Figure 2.8

Table 2.2: List of all backend providers of the openEO project

| **Organisation** | **GitHub** |
|---|---|
| EODC | `https://github.com/Open-EO/openeo-openshift-driver` |
| VITO | `https://github.com/Open-EO/openeo-geopyspark-driver` |
| Google | `https://github.com/Open-EO/openeo-earthengine-driver` |
| Mundialis | `https://github.com/Open-EO/openeo-grassgis-driver` |
| JRC | `https://github.com/Open-EO/openeo-jeodpp-driver` |
| WWU | `https://github.com/Open-EO/openeo-r-backend` |
| Sinergise | `https://github.com/Open-EO/openeo-sentinelhub-driver` |
| EURAC | `https://github.com/Open-EO/openeo-wcps-driver` |

individual processes and execute them on the backend. In the project, they are called "user defined functions" and are at the writing of this thesis still not well-defined. The plan is to send code written by the openEO user to the backend and execute it there in a secure virtual environment. The user can define processes and can run them with the data provided at the backend, using the infrastructure of the backend. Every backend has to define what the restrictions on user defined functions are.

### 2.5.2 Backends Overview

Even though the backends implement the openEO core API standard, they are still diverse behind this abstraction layer. Most backends have already an API, where the openEO calls are translated to. There are 7 partners within the openEO project that are implementing a backend driver. The backends have to implement a translation of the process graph to the internal system. The billing of the users can be completely different on every backend. Table 2.2 gives an overview of all contributing openEO backends and their related GitHub repository.

### 2.5.3 EODC Backend

The EODC backend is one of the contributing backend providers of the openEO project. The backend is implemented in Python and operates job executions with docker. EODC decided to use OpenShift (using Kubernetes) [10] to balance the workload of the docker container. It is capable of scaling docker containers and provides version control on them. The docker container execute the Python code for executing the processes. The docker description files and the Python code are available on GitHub. In this thesis, the latest version of the EODC backend provided in GitHub using openEO coreAPI version 0.3.1 is used. Every process of the openEO process graph is represented by an own Docker container. The Python library flask accomplishes the service layer for the RESTful API of the EODC backend driver. EODC provides only data from Sentinel 2 and Sentinel 1 within the openEO project. They are satellite images from Copernicus in cooperation of the European Space Agency (ESA), which receives the raw data directly from the Sentinel satellites.
The data management of EODC is file-based, so every image data is stored in a different directory and filename combination. The path of the file is the identifier of a data record. Data is provided via PostgreSQL database including the PostGIS[11] plug-in. In addition to the basic PostgreSQL functionality the plugin enables filtering by spatial and temporal extent. The datasets contain the bounding box coordinates and the timestamp of the capturing as well as a creation timestamp. It enables to query all datasets inside of a given area and timerange within SQL syntax. The plugin is capable of returning the newest datasets in terms of creation timestamps, so that only one version of a dataset is returned. We use an additional "WHERE" clause to filter datasets that have a creation timestamp before a certain time. So that we can get the versions of datasets available at a given time.The provided query tool for EODC users is the Open Geospatial Consortium (OGC) standard interface Catalogue Service for the Web (CSW)[12].It is used for the communication to the web interface of the EODC database. Figure 2.10 gives an overview of the EODC database structure. It is retrieved from the GitHub repository of the EODC backend[13], where every database entity is defined. A process entity can have parameters described by the parameter entity. The process node is representing one node in a process graph and is therefore related to exactly one process graph entity. Every job is related to a process graph. There may be jobs that use the same process graphs, but in the current set up they are persisted both separately in the database.

[10]https://www.openshift.com/learn/what-is-openshift/
[11]https://postgis.net
[12]http://cite.opengeospatial.org/pub/cite/files/edu/cat/text/main.html
[13]https://github.com/Open-EO/openeo-openshift-driver

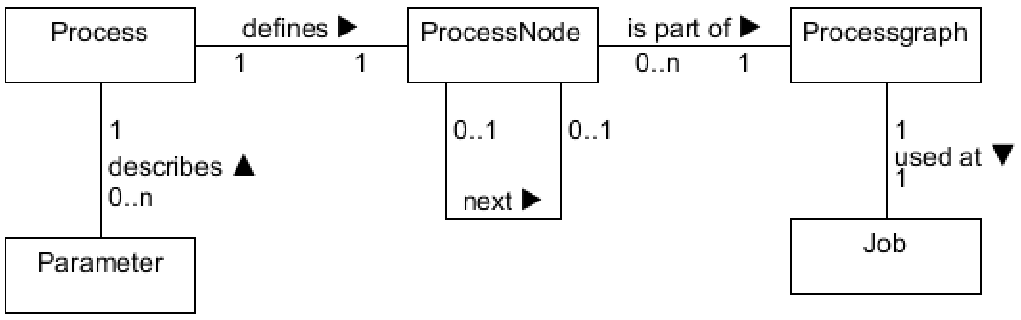


Figure 2.10: Overview of the EODC database structure.

## 2.6 Summary

The research areas related to the solution of this thesis has a vast amount of available literature. The problem description of the thesis has many related existing solutions described in this chapter. The outcome of this thesis has the aim of making it easy for the scientists to reproduce experiments and therefore enable job re-execution at the backend. Other than presented solutions for earth observation science like the approach of VZJ and the GPF, which only provide informal support for scientists. The implementation of Section 4 builds on existing systems like the implementation of the CCCA query store, the RDA recommendations and the VFramework. The following chapter describes the design of the solution.

# CHAPTER 3

# Design

The core of this thesis is an extension design for EO backends providing users reproducibility. This chapter describes the proposed generic design that can also be applied to other similar domains. This chapter aims to present the design and is structured in six parts. The first part presents an overview of the design. The next three sections describe the main components in more detail. First, the data identification component, then the component of capturing the provenance of the job execution. Third, the component for capturing and validation the result. The next section defines the resulting context model and its elements. The last two sections define the user services, suggestions on backend API extensions and User Defined Functions (UDF). The next Chapter 4, shows an implementation of the outcome of this chapter at the EODC backend.

## 3.1 Overview

This section gives an overview of the design. Figure 3.1 shows an overview of the design. White boxes represent the components that every backend driver has already in place. The green elements in Figure 3.1 are the proposed extensions to the backend. The following steps describe a typical job execution workflow for a better understanding of the backend functionality:

1. **EO Client**
   The user defines input data, filter operations, and processes, which define the workflow of an experiment, via an EO client. After that, the user orders with the client the creation of a new job at the backend. Therefore, the client creates a process graph description and sends it to the backend. The client application is used to start the execution of the job and to receive the results. The following steps describe the backend internal procedures after starting the execution of a job.

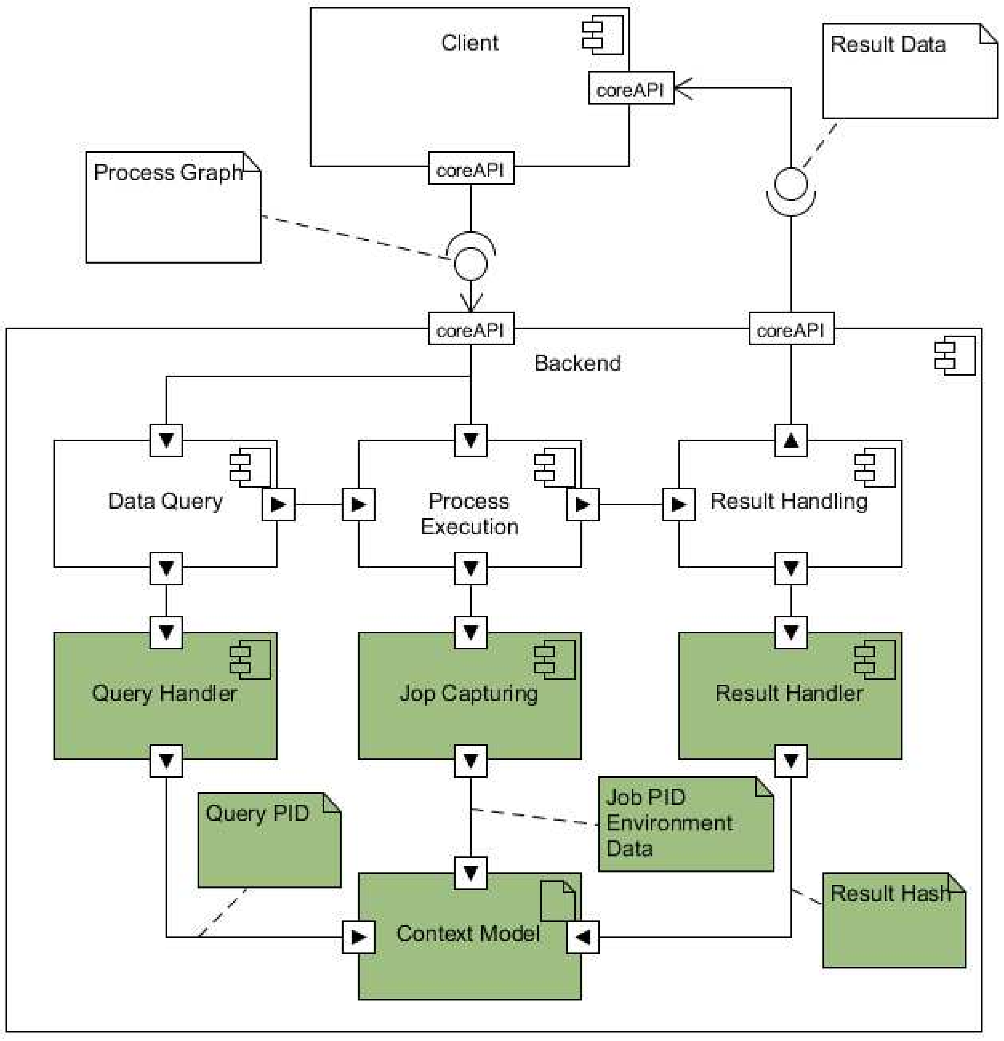


Figure 3.1: Overview of the design

2. **Data Query**
The *Data Query* component receives the process graph and parses the data identifier and the filter operations. They are used to build a internal query to select the input data required by the job execution. *Data Query* forwards the resulting data of the query execution to the *Process Execution* component.

3. **Process Execution**
The *Process Execution* component receives the process graph and the input data from the *Data Query* component. It parses the processes from the process graph and executes them in the order of appearance. After every process executed, the

*Process Execution* component forwards the resulting data to the *Result Handling* component.

4. **Result Handling**
   The *Result Handling* component receives the results from the *Process Execution* component and stores all job related meta data and the result. If ordered by the client application, the *Result Handling* sends a link to the output files back to the client. In addition does the *Result Handling* component provide meta data of the job execution to the user via the client.

The design of this thesis aims to make all three backend internal components identifiable. This enables to make the whole job execution reproducible. Hence, we introduce the following elements as additional components to the backend.

- **Query Handler**
  The *Query Handler* component applies data identification to the backend. It does so by applying the RDA recommendations described in Section 2.3. The component takes the query and a checksum of the resulting data of the query execution from the *Data Query* component. Every query and query result combination is identifiable by a data PID. Section 3.2 describes the functionality of the *Query Handler* in more detail.

- **Job Capturing**
  The *Job Capturing* component enables code identification as well as execution environment information. Therefore, it introduces a PID of the job execution code. In addition, it captures data of the execution environment to gain additional information for users. Section 3.3 describes the *Job Capturing* component in more detail.

- **Result Handler**
  The *Result Handler* component creates a comparable result checksum or hash. The data created by an earth observation backend might be too big to be persisted entirely. Hence, the component introduces a checksum or hash to be capable of confirming equality of job results. Section 3.4 describes the *Result Handler* component in more detail.

- **Context Model**
  The context model is not a component, but a data record containing the job related data produced by the previous components. Every job execution is related to one context model. Section 3.5 provides more detailed information about the context model and its elements.

## 3.2 Query Handler

The input data of the processing is crucial for the outcome of the job execution. Even though the process graph contains an identifier of the input dataset (e.g. Sentinel 2), internal changes to the data set might not result in a new dataset identifier. Later execution of the job might use another version of the input data. The input data has to be stored in a query store according to the 14 recommendations of data provenance defined by the RDA [34]. By implementing these at the *Query Handler* component, input data becomes identifiable. The *Query Handler* is the module where the data persistence is implemented and depends highly on the structure and architecture of the backend. Therefore, there is no general design to achieve it in this section. Chapter 4 shows an implementation of the RDA recommendations at the EODC backend. The context model contains the input data with the following element:

(a) **Input data persistent identifier**
The output of the *Query Handler* component is the input data PID used by a job execution. Every job execution accesses input data with a PID. Therefore, the input data PID is added to the job dependent context model.

## 3.3 Job Capturing

This section describes the *Job Capturing* component. It handles the job execution environment capturing and the code identification. The data provided by *Job Capturing* is categorized in static and dynamic data. We define static data as not dependent on the configuration of the job. We define dynamic data as job dependent information. The following two sections describe the static data (backend provenance) and dynamic data (job dependent environment).

### 3.3.1 Backend provenance

The scope of the backend provenance is to get the static environment of the job execution. It contains the provenance data independent of the job configuration. The elements of the context model not changing regardless of how many jobs are executed. Only the maintainer of the backend are capable of changing this data. The following data is suggested for the backend provenance:

(A) **Code identification**
Earth observation backends have to provide an identifier for the code. Therefore, the backends must apply a version control system. If the backend is open source, it can use a public repository using e.g. GitHub. The advantage of this is to get other backends with similar settings to reuse the already existing solutions. By using Git, the backend achieves code identification by adding repository information to the backend provenance. It stores the Git repository URL and the commit identifier of

the job execution. If the backend is not open source, a local or secure version control repository is implemented to keep track of the different code versions. The code identifier is necessary to identify differences in the backend code, and the version control system enables to jump back to versions of the past.

(B) **API version**
The backend must stores the version of the currently used interface API. The API defines the syntax and semantics of communication. The same API calls can resolve in unequal results if the API version differs. Hence, the backend stores the API version of the job execution in the context model.

(C) **Backend version**
The backend version is an identifier of the backend state. The backend updates its version on every internal change (e.g. update of dependencies). It updates the version on changes in the hardware as well as the software. It is essential that the backend provider implement a tool to automatically capture the environment data and persist it in a separate backend version store. Every change on any of the described backend data has to be detected automatically and have to result in a new backend version.

(D) **Publication timestamp**
The publication timestamp describes the time when the version was accessible at the backend. The timestamp stands for the beginning of a new backend version. The newest timestamp means that the version is the current one. The timestamp enables to find a version of the backend used in a job execution of the past, just by knowing the time it was executed.

### 3.3.2 Job dependent environment

This section describes the job dependent provenance of the context model. The data captured is tied to specific job executions. The process graph is a description of processes related to a job execution. Sending two equal process graphs to the same backend should result in the same outcome. To assure that the process graph can be re-executed in the same way as the original execution, data about the first execution gets captured. The way of executing a specific process graph is not only related to the code running it, but also by the dependencies of the code. Therefore, the programming language version and the additional used libraries are persisted in the context model. To gain meta-information about the processing the start time and end time can be added to the context model via timestamps. These suggestions lead to the following capturing elements in the context model:

(b) **Backend provenance / code identifier**
The backend provenance represents the version of the backend used for the job execution. Since for every change on the backend, a new version is applied, the version of the backend is used as a code identifier of the execution.

(c) **Programming language**
The programming language of the code used for the job execution. Besides, the version of the programming language is included to this information.

(d) **Dependencies of the programming language**
The dependencies of the programming language must be captured and added to the context model to describe the environment of the job execution. The version of the packages have a high influence on the outcome, hence are included in this element. For example, in Python, the installed modules are added with their versions to the context model. The information stored in this element is part of the job dependent environment, because it depends on the configuration of the job if the backend has a dynamically generated container for every job.

(e) **Start and end time of process execution**
The start and end time of the job execution is persisted in the context model.

## 3.4 Result Handler

The resulting data of a job execution must be captured to enable the oucome of jobs. In earth observation jobs, the results can be rather big. Therefore, the *Result Handler* generates a hash value over the resulted files. The result data does not have to be identifiable within the scope of this thesis, but checkable of equality with others. Therefore, a hash value of the output data is sufficient. The aim of the output data capturing is not to find differences between results, but to show that results are different or equal. Even though the input data of typical earth observation experiments are significant, the output of such experiments are images of various sizes.

(f) **Result hash**
To make the result of a job execution verifiable it gets persisted. One way to achieve this, without too much impact on the backend storage, is to take a hash value over the resulting files sorted by the alphabetical ascending filenames.

## 3.5 Context Model

The context model is the data record containing the provenance of a job execution. The backend provider infrastructure defines the type of storage it is saved. In example it can be stored in a relational database or a file-based system as a file. It must be integrated into the database structure of the backend. Figure 3.2 shows an overview of the elements of the backend used to run a job. Besides, it shows how the context model elements are used to identify the components of the backend.

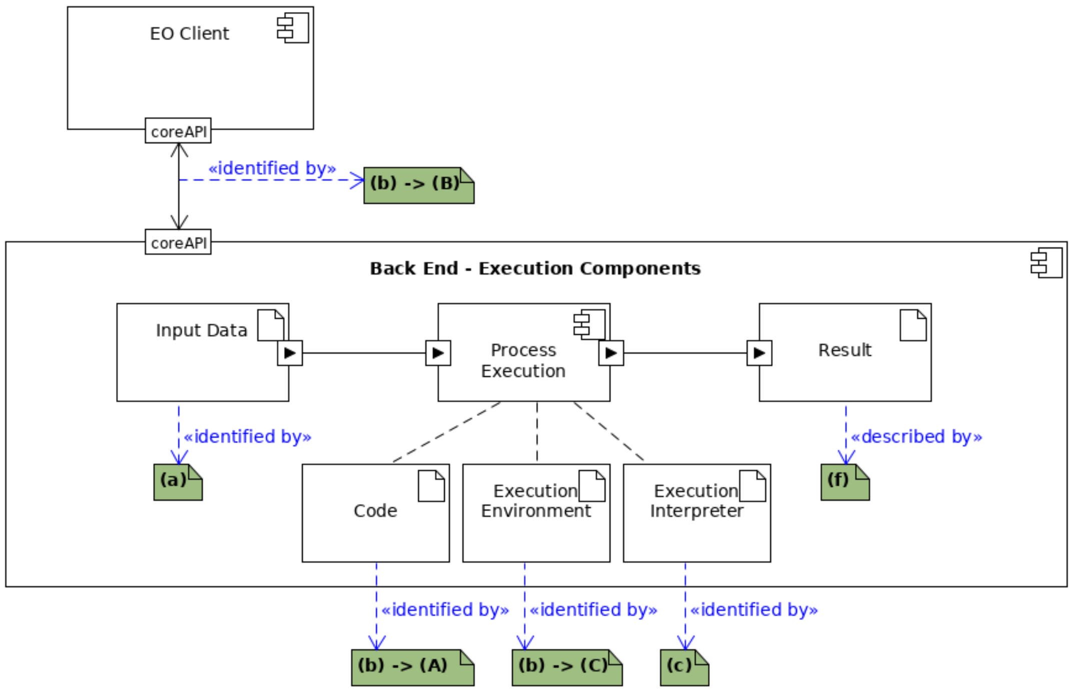


Figure 3.2: Overview of the backend execution components and the context model elements that identifies them.

The backend provenance dataset contains job independent provenance and is identified by a backend version. Therefore, the necessary element of the backend provenance is the backend version that defines the state of the backend in a time period. The backend version must be resolvable by a code version present during a past execution. Besides, meta data about the backend can be added to the elements of the backend provenance. The elements ((A) - (D)) defined in Section 3.3.1 are the suggestions of elements in the backend environment record. The following list recaps all of the backend environment elements:

(A) **Code identification**
Enables the identification of the code of a backend version.

(B) **API version**
Enables the identification of the API version of a backend version.

(C) **Backend version**
Identifies the whole backend.

(D) **Publication timestamp**
Enables the identification of a backend version at a specific time.

The job execution context model, which includes all information related to a job execution, is stored in the context model. There are three necessary elements in the job context model. The input data identifier must be part of the context model, so that the input data of the job is identifiable. The backend version of the job execution must be included in the context model, to be able to re-execute it with the same backend state in the future. The job dependent context model must contain the output hash, so that differences in the job results are detectable. The following list recaps the elements of the job context model described in Section 3.3.2:

(a) **Input data persistent identifier**
Identifies the input data of the job.

(b) **Backend provenance / code identifier**
Identifies the backend version (C) during the execution of the job. Connects the job context model with the backend environment dataset.

(c) **Programming language**
Identifies the programming language and version used by the job execution.

(d) **Dependencies of the programming language**
Identifies the dependencies of the programming language used by the job execution.

(e) **Start and end time of process execution**
Provides the start and end time of the job execution.

(f) **Result hash**
Describes the results of the job execution.

## 3.6 User Services

This section describes the provided information for users and how users access it. The capturing described in the previous sections consist of information about the backends that backend provider do not want to pass entirely to users. For example, it can be risk to provide information on specific programming language packages if they have vulnerabilities. There is captured information that might not be necessarily useful for users. Therefore, there must be a filter on the shown information at the user services. The earth observation community has a diverse set of backends with unique company security guidelines.
Provenance information should be available to users to be useful. Therefore, we suggest additions to the backend API specification. It consists of additional endpoints for the users to get information about the backend and the client application. The following recommendations are for backend and client developer to make context model information accessible for users.

I. **Backend version**
We suggest an endpoint to retrieve the backend specific information, especially the backend version. Additionally there need to be an endpoint to retrieve the backend version of a specific time. The aim of it is to present the users with information about the current state of the backend and to help users decide, which backend they want to use.

II. **Detailed Job Information**
We recommend an endpoint to retrieve detailed information about an already executed job. The endpoint includes the provenance of the job, therefore the resolvable persisted identifier of the input data, the backend version, and the result set hash. Additionally, the whole data of the context model can be made accessible, depending on the security decisions of the backend. We recommend to provide this information in PROV-O. In the current implementation the detailed job information can be accessed in PROV-JSON, PROV-XML or visualized exported as PNG (see Section 4.4.1).

III. **Comparing Two Jobs**
We suggest an additional endpoint either on the backend or at the client application to compare two jobs. Every item of both of their context models is compared on equality. If they are not equal, the differences should be transparent to the user. The response of the comparison consists of the differences in the context model between two jobs. In this thesis this is the only comparison representation implemented. Future work will lead to a representation of the comparison according to PROV-O.

IV. **Data Identifier Landing Page**
After the job is executed and the input data got an PID. The PID must be resolved by a machine-actionable landing page. The landing page provides the user with information about the input dataset. We suggest an additional endpoint to re-execute the query showing the input data files if given sufficient permissions.

V. **Re-use of Input Data**
We recommend to add the functionality to a backend to re-use input PIDs in a new job. The API allows to include the use of an input data PID in the process graph, so that users can include cited data directly in a newly created job description.

## 3.7 User Defined Functions

UDFs are customized code written by the user that gets executed in a virtualized environment at the backend provider. In theory, the user defines a docker base image and the code running in it. Therefore, it is a black box for the backend provider. They cannot know how the code looks like in the docker container. Therefore, the capturing concept needs to be different than the typical way of executing jobs at the backend. The code and the docker base image has to be persisted by the backend provider. Additionally, the timestamp of the execution, to be able to identify the backend version at the execution. This is a rather new concept in earth observation science and not implemented on the EODC backend, hence the implementation of the capturing is not part of this thesis.

## 3.8 Summary

This chapter presented the design of the proposed solution. It shows how a backend can be extended to enable reproducible job executions. The data needed to be able to reproduce a job execution are presented and described. Data identification must be implemented according to the RDA recommendations. The elements of the backend environment are defined and are identifiable by the backend version. We defined the job dependent context by the job execution environment and the result hash. The data needed for a reproduction of a job execution is stored in the context model and it is the data needed to answer the research questions of Chapter 1. In addition to the data description, recommendations on user services are listed. They provide the functionality for the users. The next chapter presents the implementation of the proposed design at the openEO compliant EODC backend.

CHAPTER 4

# Implementation

In this chapter we present the implementation of the thesis following the design described in Chapter 3. It is implemented at the Python based EODC backend, a consortium member of the openEO project. The implementation consists of the suggested extensions to the backend and client application. It contains suggestions to the specification of openEO. Thus, we modify all three parts of the openEO project architecture in the presented solution. We modify the Python client[1] for the purpose of this thesis. Python is the most common programming language at the contributing backends of the openEO project. The implementation is open source and other backend providers with similar setup can use it.

Similar to the design of this thesis, we structure the implementation in four parts presented in the first four sections. The first section describes the data identification implementation following the RDA recommendations at the backend. Section 4.2 presents the implementation of the backend provenance capturing at the backend. Next, Section 4.3 presents the implementation of the job dependent provenance at the backend. The last section describes the implementation of the proposed user services in the client application and the backend defined in Section 3.6.

## 4.1 Data Identification

In Section 3.2 we suggested that every backend provider has to implement data identification following the RDA recommendations. This section presents the data identification solution for the EODC backend. Other openEO backend providers can use the presented approach as well. This section focuses on the implementation of the core components on the backend. The mechanics of users getting and using the data identifier including the

[1] https://github.com/Open-EO/openeo-python-client

landing page is part of Section 4.4.

### 4.1.1 Database and Query Overview

This section describes the services used by the backend to query datasets and how the underlying database is structured. Figure 4.1 gives an overview of the communication layers of the query execution. The backend uses a PostgreSQL database with the PostGIS plugin for querying the datasets. It is not only used by the openEO compliant backend, but also by other applications from EODC and partners. The PostgreSQL Server is an internal service inside the infrastructure of EODC. To make the data discovery accessible from the web, EODC uses a web service layer compliant with the CSW standard. The backend sends XML formatted CSW queries, which the *CSW Webservice Layer* translates into SQL queries for the internal *PostgreSQL Server*. The result goes through the inverse communication layer, so that the backend receives a CSW compliant response.

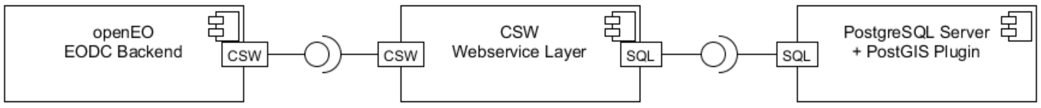


Figure 4.1: Overview of the query services at EODC. The backend sends CSW compliant queries to the *CSW Webservice Layer*, which translates it to SQL. It enables a decoupling of the database and the services as well as publicly available data discovery via the CSW endpoint of EODC.

The backend uses a filebased data storage, where every file is identified by its path. Nevertheless, there is additional metadata stored in the PostgreSQL database. The following list describes the metadata relevant for the thesis:

- **path**
  The path is the identifier of the data record. The backend uses it to access the actual data file inside of its infrastructure.
- **creation timestamp**
  The creation timestamp is the time, since the data record is available at the backend. We use it in our implementation to query data versions from the past after updates occurred.
- **publisher**
  Contains the name of the publisher providing the source data e.g. ESA for Sentinel data. We use it for the description of datasets on the landing page.
- **description**
  Describes the source data. We use it to provide a description of the source data on the landing page.

- **satellite identifier**
  Identifies the satellite used to retrieve the data record.
- **tile number & orbit number & baseline number**
  These three numbers identify the geographical location of the data record. The PostGIS plugin is capable of filtering tiles by bounding box coordinates. This information is used by the backend to apply the filter arguments.
- **sensing timestamp**
  The sensing timestamp describes when the data record was recorded by the satellite. This information is applied by the backend for the filter arguments.

The backend creates a new version of a data record by appending a new data record with a new path and a new creation timestamp to the database table.
The database structure is hidden for the backend, because the mapping is configured by the *CSW Webservice Layer*. The CSW standard specifies properties and operators that the backend uses to filter the data without knowing the concrete structure. Listing 4.1 shows the available properties. The backend uses these properties to access data of the metadatabase e.g. *apiso:ParentIdentifier* for the satellite identifier.

```
<ogc:PropertyName>apiso:ParentIdentifier</ogc:PropertyName>
<ogc:PropertyName>apiso:TempExtent_begin</ogc:PropertyName>
<ogc:PropertyName>apiso:TempExtent_end</ogc:PropertyName>
<ogc:PropertyName>ows:BoundingBox</ogc:PropertyName>
<ogc:PropertyName>apiso:Modified</ogc:PropertyName>
```

Listing 4.1: CSW properties, used by the backend and our implementation.

Listing 4.2 shows the available operators. They are used by the backend to specify how the parameters are filtered.

```
<ogc:PropertyIsLessThan></ogc:PropertyIsLessThan>
<ogc:PropertyIsLessThanOrEqualTo></ogc:PropertyIsLessThanOrEqualTo>
<ogc:PropertyIsEqualTo></ogc:PropertyIsEqualTo>
<ogc:PropertyIsGreaterThanOrEqualTo></ogc:PropertyIsGreaterThanOrEqualTo>
<ogc:PropertyIsGreaterThan></ogc:PropertyIsGreaterThan>
```

Listing 4.2: CSW operations.

Listing 4.2 shows the available operators. They are used by the backend to specify how the parameters are filtered. All operators and properties compatible with the EODC CSW server are listed at the official CSW endpoint[2].

[2]https://csw.eodc.eu

```
<ogc:PropertyIsEqualTo>
  <ogc:PropertyName>apiso:ParentIdentifier</ogc:PropertyName>
  <ogc:Literal>s2a_prd_msil1c</ogc:Literal>
</ogc:PropertyIsEqualTo>
  <ogc:PropertyIsGreaterThanOrEqualTo>
  <ogc:PropertyName>apiso:TempExtent_begin</ogc:PropertyName>
  <ogc:Literal>2017-05-01T00:00:00Z</ogc:Literal>
</ogc:PropertyIsGreaterThanOrEqualTo>
<ogc:PropertyIsLessThanOrEqualTo>
  <ogc:PropertyName>apiso:TempExtent_end</ogc:PropertyName>
  <ogc:Literal>2017-05-31T23:59:59Z</ogc:Literal>
</ogc:PropertyIsLessThanOrEqualTo>
<ogc:BBOX><ogc:PropertyName>ows:BoundingBox</ogc:PropertyName>
  <gml:Envelope>
    <gml:lowerCorner>46.905246 10.288696</gml:lowerCorner>
    <gml:upperCorner>45.935871 12.189331</gml:upperCorner>
  </gml:Envelope>
</ogc:BBOX>
```

Listing 4.3: Example CSW query of the backend.

Listing 4.3 shows an example CSW query of the backend if it runs the running example described in Section 1.1.1. The query uses the *ogc:Literal* annotations to set the values of the properties specified by the *ogc:PropertyName* annotations. The operations define the relation between the value and the property e.g. the query only returns data records with satellite identifiers that are equal to *s2a_prd_msil1c*. For our solution we need to be able to filter by the creation timestamp of the data record. It enables us to re-execute a query in the manner of the original execution. Listing 4.4 shows the additional query part. *apiso:Modified* is specified by the CSW standard as the timestamp the dataset got modified. EODC does not modify existing data records. An update appends an additional data record. That is why EODC maps the *apiso:Modified* property to the creation timestamp of the data record.

```
<ogc:PropertyIsLessThanOrEqualTo>
<ogc:PropertyName>apiso:Modified</ogc:PropertyName>
<ogc:Literal>2019-03-31T17:36:43.06Z</ogc:Literal>
</ogc:PropertyIsLessThanOrEqualTo>
```

Listing 4.4: CSW query addition by our implementation to filter by creation timestamp.

### 4.1.2 Query Store

The centerpiece of the RDA recommendations is the implementation of a Query Store. Queries in the Query Store must be comparable, identifiable, and persistent. The query data has to be stored at the backend infrastructure. The backend has a PostgreSQL database to store the executed jobs (see Figure 2.10 in Section 2.5.3). We realized the Query Store at the backend with two additional tables to this database. Figure 4.2 visualizes the proposed additional tables. The new *Query* table consists of the query datarecord specified by the RDA recommendations. The *QueryJob* table maps the relation between Job and Query. In the current version of the openEO core API, it is only possible to have one input data query used by a job. In the future, there may be the possibility to have more than one input data query related to one job execution. Hence we introduce the *QueryJob* table.

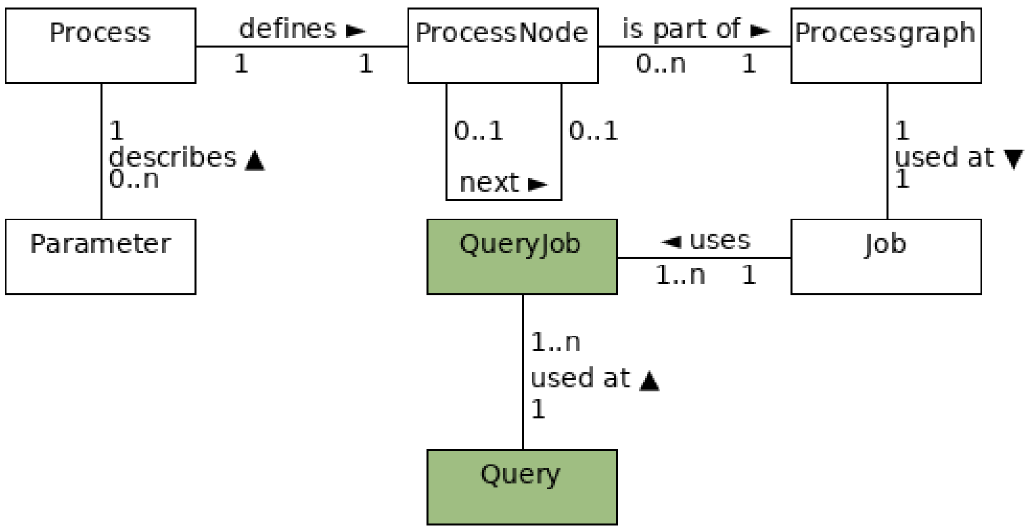


Figure 4.2: Overview of the database of the backend with the proposed additional tables (green).

### 4.1.3 Query Handler

Figure 4.3 shows an overview of the data identification solution at the backend. The green parts of the diagram represent proposed extensions. We implemented the *Query Handler* component as an additional Python module in the backend called after the job execution.

The *Query Generation* and *Query Execution* components provide the input data of the *Query Handler*. The process graph is the raw input process graph received by the backend driver from the client application. The list of result files is the result of the query execution and therefore, the input data of the job. The *Query Execution* component provides the execution timestamp. The *Query Handler* contains the following components:

- **Query Processor**
  The *Query Processor* takes the executed query, parses and generates the necessary

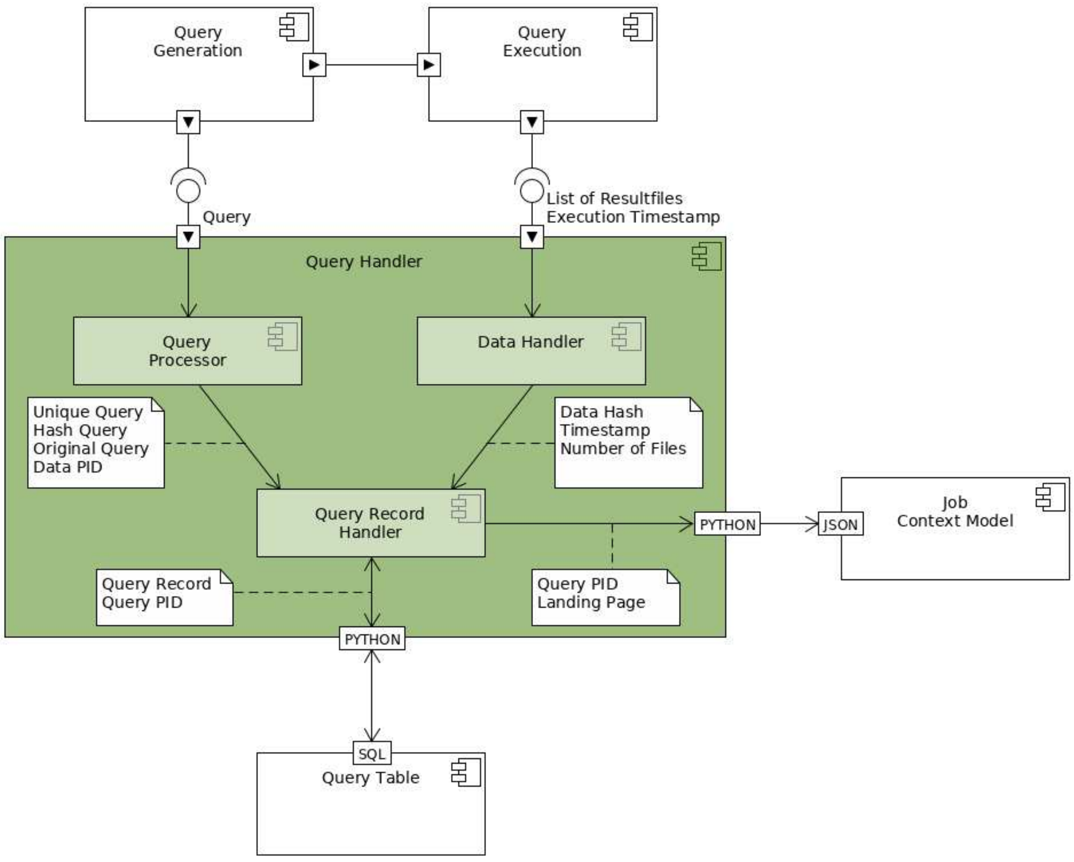


Figure 4.3: Overview of the proposed data identification component at the backend

query data and forwards the information to the *Query Record Handler*. This step is necessary, since the process graph at this point also includes the execution processes and not only the filter processes. The output is composed of the original query, the unique query, the hash of the unique query and the persistent data identifier (PID) of the query. This information is forwarded to the *Query Record Handler*. The mentioned resulting items are described in more detail in the next section.

Table 4.1: Structure of the Query Table in the PostGres database.

| **Query PID** | **Dataset PID** | **Original Query** | **Unique Query** |
|---|---|---|---|
| VARCHAR(100) | VARCHAR(100) | TEXT | VARCHAR(300) |
| **Query Hash** | **Result Hash** | **Execution Timestamp** | **Metadata** |
| VARCHAR(65) | VARCHAR(65) | TIMESTAMP | TEXT |

- **Data Handler**
  The *Data Handler* is responsible for creating the elements of a query record related to the query result. The output contains the hash over the file list, the execution timestamp and the number of resulting files. Since the backend uses the OGC standard CSW[3] to query the data, the sorting of the resulting files is predefined. The order of the files has no impact on the processing, and openEO users are not able to choose a sorting type. Therefore, the predefined CSW sorting is used for the hash production. The results of the *Data Handler* component are forwarded to the *Query Record Handler*.

- **Query Record Handler**
  The *Query Record Handler* communicates with the database of the backend to find existing identical query records. Figure 4.4 gives an overview of the activities of the *Query Record Handler*. If the query record already exists, the *Query Record Handler* returns the existing PID, otherwise a new PID is created and the query is stored in a new query record. The combination of the result-file hash and the unique query hash must not have duplicates in the query table. On saving the query PID in the job context model, a queryjob record gets created. It sets the relation between the job execution and the input data query PID.

### 4.1.4 Query Table Structure

The query table stores the query data with additional result and data information. Table 4.1 visualizes the structure of the query table. We added example values to explain how the parts of the query record are generated. Figure 4.5 shows the example input process graph.

The following list describes how the elements of the query data set record are created in more detail:

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

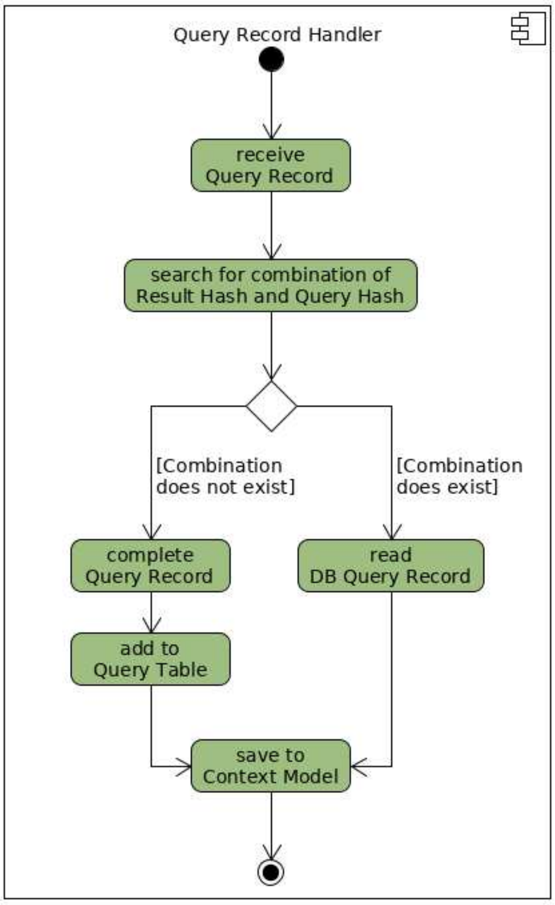
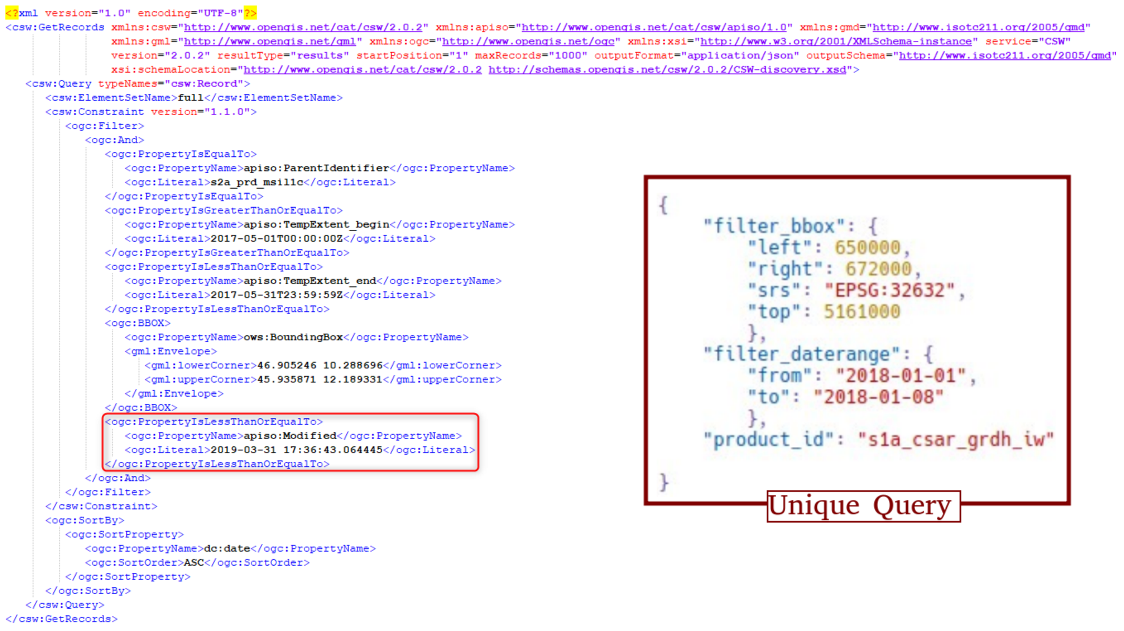


Figure 4.4: Activity diagram of the Query Record Handler component.

1. **Query PID**
   We generate the query PID with the Python library uuid[4]. The library generates unique identifiers and the backend uses it for creating the job identifier. The identifier of the backend have codes related to the database table in the beginning (e.g. "jb-UUID" for job entities). That is why the id for the newly introduced query table is structured like "qu-UUID".
   *Example: "qu-16fd2706-8baf-433b-82eb-8c7fada847da"*

2. **Dataset PID**
   The dataset PID is the identifier of the satellite in the process graph (e.g. the identifier in the "load_collection" process).
   *Example: "s1a_csar_grdh_iw"*

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



```
<?xml version="1.0" encoding="UTF-8"?>
<csw:GetRecords xmlns:csw="http://www.opengis.net/cat/csw/2.0.2" xmlns:apiso="http://www.opengis.net/cat/csw/apiso/1.0" xmlns:gmd="http://www.isotc211.org/2005/gmd"
                xmlns:gml="http://www.opengis.net/gml" xmlns:ogc="http://www.opengis.net/ogc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" service="CSW"
                version="2.0.2" resultType="results" startPosition="1" maxRecords="1000" outputFormat="application/json" outputSchema="http://www.isotc211.org/2005/gmd"
                xsi:schemaLocation="http://www.opengis.net/cat/csw/2.0.2 http://schemas.opengis.net/csw/2.0.2/CSW-discovery.xsd">
  <csw:Query typeNames="csw:Record">
    <csw:ElementSetName>full</csw:ElementSetName>
    <csw:Constraint version="1.1.0">
      <ogc:Filter>
        <ogc:And>
          <ogc:PropertyIsEqualTo>
            <ogc:PropertyName>apiso:ParentIdentifier</ogc:PropertyName>
            <ogc:Literal>s2a_prd_msil1c</ogc:Literal>
          </ogc:PropertyIsEqualTo>
          <ogc:PropertyIsGreaterThanOrEqualTo>
            <ogc:PropertyName>apiso:TempExtent_begin</ogc:PropertyName>
            <ogc:Literal>2017-05-01T00:00:00Z</ogc:Literal>
          </ogc:PropertyIsGreaterThanOrEqualTo>
          <ogc:PropertyIsLessThanOrEqualTo>
            <ogc:PropertyName>apiso:TempExtent_end</ogc:PropertyName>
            <ogc:Literal>2017-05-31T23:59:59Z</ogc:Literal>
          </ogc:PropertyIsLessThanOrEqualTo>
          <ogc:BBOX>
            <ogc:PropertyName>ows:BoundingBox</ogc:PropertyName>
            <gml:Envelope>
              <gml:lowerCorner>46.905246 10.288696</gml:lowerCorner>
              <gml:upperCorner>45.935871 12.189331</gml:upperCorner>
            </gml:Envelope>
          </ogc:BBOX>
          <ogc:PropertyIsLessThanOrEqualTo>
            <ogc:PropertyName>apiso:Modified</ogc:PropertyName>
            <ogc:Literal>2019-03-31 17:36:43.064445</ogc:Literal>
          </ogc:PropertyIsLessThanOrEqualTo>
        </ogc:And>
      </ogc:Filter>
    </csw:Constraint>
    <ogc:SortBy>
      <ogc:SortProperty>
        <ogc:PropertyName>dc:date</ogc:PropertyName>
        <ogc:SortOrder>ASC</ogc:SortOrder>
      </ogc:SortProperty>
    </ogc:SortBy>
  </csw:Query>
</csw:GetRecords>
```

```
{
    "filter_bbox": {
        "left": 650000,
        "right": 672000,
        "srs": "EPSG:32632",
        "top": 5161000
        },
    "filter_daterange": {
        "from": "2018-01-01",
        "to": "2018-01-08"
        },
    "product_id": "s1a_csar_grdh_iw"
}
```

Unique Query

Figure 4.5: Example for the original query and the unique query of the example process graph.

3. **Original Query**
   The initially executed CSW query of the query execution component.
   *Example: See the original query in Figure 4.5*

4. **Unique Query**
   The unique query is the restructured query that is comparable to other unique queries. The order of the filters make no difference in the outcome of the query execution. Therefore, the filter arguments of the original query are alphabetically sorted by the JSON keys to generate the unique query.
   *Example: See the unique query in Figure 4.5*

5. **Unique Query Hash**
   We remove newline and space characters from the unique query string. After that, we create the unique query hash by running the SHA-256 over the resulting unique query using the "hashlib" Python module.
   *Example: "AE7EF888CDEDF8A9A371. . . "*

6. **Result Hash**
   The result hash is the output of the SHA-256 hash function using the Python module "hashlib" over the result file list. The sorting of the files is fixed by the CSW query standard used by the backend. We clean up the string of the file list by removing the newline and white space characters, before it is applied to the hash function.
   *Example: "565D229FCE4772869343. . . "*

7. **Execution Timestamp**
   The execution timestamp is the input parameter of the *Query Handler* transformed to the data type needed by the database. We take the timestamp from the *Query Execution Handler* and is part of the result.
   *Example: "2018-10-17 18:03:20,609"*

8. **Additional Data**
   The additional data column of the query table can be used by the backend to store additional information about the query execution. In the implementation of this thesis, we only store the number of output files. The column is defined as a JSON object and can be extended with additional data without changing the structure of the table.
   *Example: "{ "number_of_files": 10}"*

## 4.2 Backend Provenance

The backend provenance is idenfifyable by the backend version and contains the job independent provenance information. The following subsections explain how the (in Section 3.3.1 defined) provenance data elements are implemented.

(A) **GitHub Repository**
The backend services deploy automatically from the GitHub repository. Therefore, we read the used GitHub repository information via the Command Line Interface (CLI) of Git. We access the checked out commit and branch of the backend directly via the Git CLI, initiated with Python. Listing 4.5 shows the Git CLI calls used to retrieve the GitHub repository information needed by the backend provenance.

```
# Receiving the Git Repository URL
git config --get remote.origin.url
# Receiving Branch
git branch
# Receiving the commit messages with the timestamps
git log
```

Listing 4.5: Git CLI calls to get access the backend provenance.

(B) **Core API Version**
The backend provider update the openEO core API version of the backend manually. The currently used API is in the GitHub repository of the backend.

(C) **Back End Version**
The backend fetches the code for the services directly from GitHub and the Git commit identifies a certain state of the code. Therefore, we use the commit identifier as the version of the backend.

(D) **Publication Timestamp**
The publication timestamp of the version of the backend is defined by the timestamp when the Git commit happened. GitHub stores it and it can be retrieved via the Git CLI.

## 4.3 Job Dependent Provenance

The backend transforms the process graph into separate docker containers. For every process in the process graph, there is a docker container running the Python implementation of the process. The input of the current process is the output of the previous process. The first process docker container has the input data defined in the process graph as input data, which is the result of the query execution. Every process saves the results in a temporary folder dedicated to the specific process execution. Every process has its temporary output directory until the whole process chain is finished. After that, the backend deletes the temporary folders, and it only keeps the result in the job directory. The job must identify the input data, the output data and the execution environment to achieve the job environment data capturing described in Chapter 3.3.2.
The implementation adds the captured information to the logging of the job execution. After the job execution it reads the logging files to generate the context model. This solution extends the code of the backend with minor logging calls. Figure 4.7 gives an overview of the implemented job capturing procedure. Section 4.3.2 describes every part of the overview in more detail.

### 4.3.1 Context Model Repository

Each executed job generates a context model related to the job execution. If the job gets re-executed, the context model gets replaced by a new context model related to the later execution. The backend handles job re-executions as new jobs with the same process graph and assigns a new job identifier. Letting the same job be re-executed without creating a new job id is dropped from the agenda of the openEO project since version 0.3.1 (see the GitHub repository[5]). The aim of this design decision is to turn the job id into an job execution identifier and the process graph to the identifier of a job. Therefore, different job ids with the same process graph represent different executions

[5] https://open-eo.github.io/openeo-api/v/0.3.1/apireference/

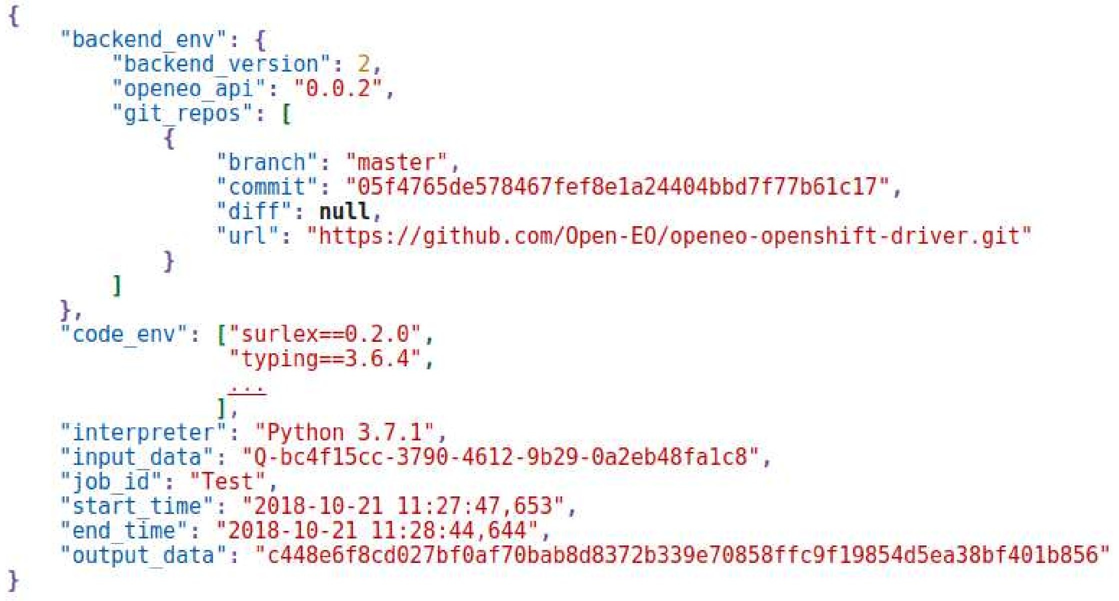
Table 4.2: Relation of context model elements and the implementation JSON context model.

| Context Model Definition | JSON Key |
|---|---|
| **(a): Input data persistent identifier** | input_data |
| **(b): Backend provenance / code identifier** | backend_env |
| **(c): Programming language** | interpreter |
| **(d): Dependencies of the programming language** | code_env |
| **(e): Start and end time of the process execution** | start_time, end_time |
| **(f): Result hash** | output_data |

of the same job. We store the context model formatted as a JSON object in the job execution database. After the job is carried out in the backend, the results are saved in a folder named after the job identifier and the meta data information is stored in the PostgreSQL database (see Section 2.5.3). Jobs are stored in the *Job* table of the database. We added an additional column to this table to store the context model in this solution. The creation of the context model is described with the implementation below.
Table 4.2 provides the mapping between the context model elements from Section 3 and the keys of the JSON context model object of the prototype. The elements have a one to one mapping of the context model and the JSON key except for the timestamps of the execution. The execution timestamps are part of the *Job* table in the EODC database. Figure 4.6 shows an example context model. It consists of all elements described by the context model in the Design chapter. Information on the backend environment during the execution of the job is stored in the context model. We store the backend version and the execution timestamp in the context model to be able to identify backend provenance of the execution The code environment is a list of Python dependencies of the job execution with their versions. Besides, the Python interpreter version is added to the context model. How the data is captured in detail is described in the sections below.

### 4.3.2 Python Implementation

The implementation of this thesis is an example for other backends with similar setups. Therefore, we implemented it in the Python version of the backend without any additional requirements on Python modules. The Python solution uses logging messages to transfer the needed data from the process execution to the capturing modules. The cleanup service of the EODC backend driver triggers the modules of the solution. It is an additional Python module and parses the log files after the job is finished. This solution generates, except for the additional logging calls, little impact on the existing backend implementation. The EODC backend driver has already a logging system installed. Hence the modifications are added to the existing logging policy.
Figure 4.7 visualizes the additional Python modules *Job Capturing* and *Result Handler* in the context of the backend environment.



```
{
    "backend_env": {
        "backend_version": 2,
        "openeo_api": "0.0.2",
        "git_repos": [
            {
                "branch": "master",
                "commit": "05f4765de578467fef8e1a24404bbd7f77b61c17",
                "diff": null,
                "url": "https://github.com/Open-EO/openeo-openshift-driver.git"
            }
        ]
    },
    "code_env": ["surlex==0.2.0",
                 "typing==3.6.4",
                 ...
                ],
    "interpreter": "Python 3.7.1",
    "input_data": "Q-bc4f15cc-3790-4612-9b29-0a2eb48fa1c8",
    "job_id": "Test",
    "start_time": "2018-10-21 11:27:47,653",
    "end_time": "2018-10-21 11:28:44,644",
    "output_data": "c448e6f8cd027bf0af70bab8d8372b339e70858ffc9f19854d5ea38bf401b856"
}
```

Figure 4.6: Example context model of a job execution at the backend implementation.

The following list describes the modules of the backend and the additional modules of the solution. The order of the list shows the order of the module execution.

1. **Process Execution**
   The *Process Execution* module at the backend is responsible for the execution of the job. It creates the additional logging information (see (a), (c), (d) and (e) of the context model) and stores it in a logging file.

2. **Processing Cleanup**
   After finishing the job execution, the backend deletes all temporary folders from the file system and copies the results to the newly created job folder. Then the solution module *Job Capturing* starts given the path of the logging file.

3. **Result Provider**
   The *Result Provider* is responsible of making the result available for the user. It provides the user with information and feedback about the job execution. Additionally it invokes the *Result Handler* module with the path of the resulting file.

4. **Result Handler**
   The *Result Handler* reads the result file and calculates an SHA-256 hash over it. After completion, it sends it to the *Context Model Creator* module.

5. **Job Capturing**
   The *Job Capturing* module parses the logging file to extract the information needed for the context model ((a), (c), (d) and (e)). It passes the rightful formatted

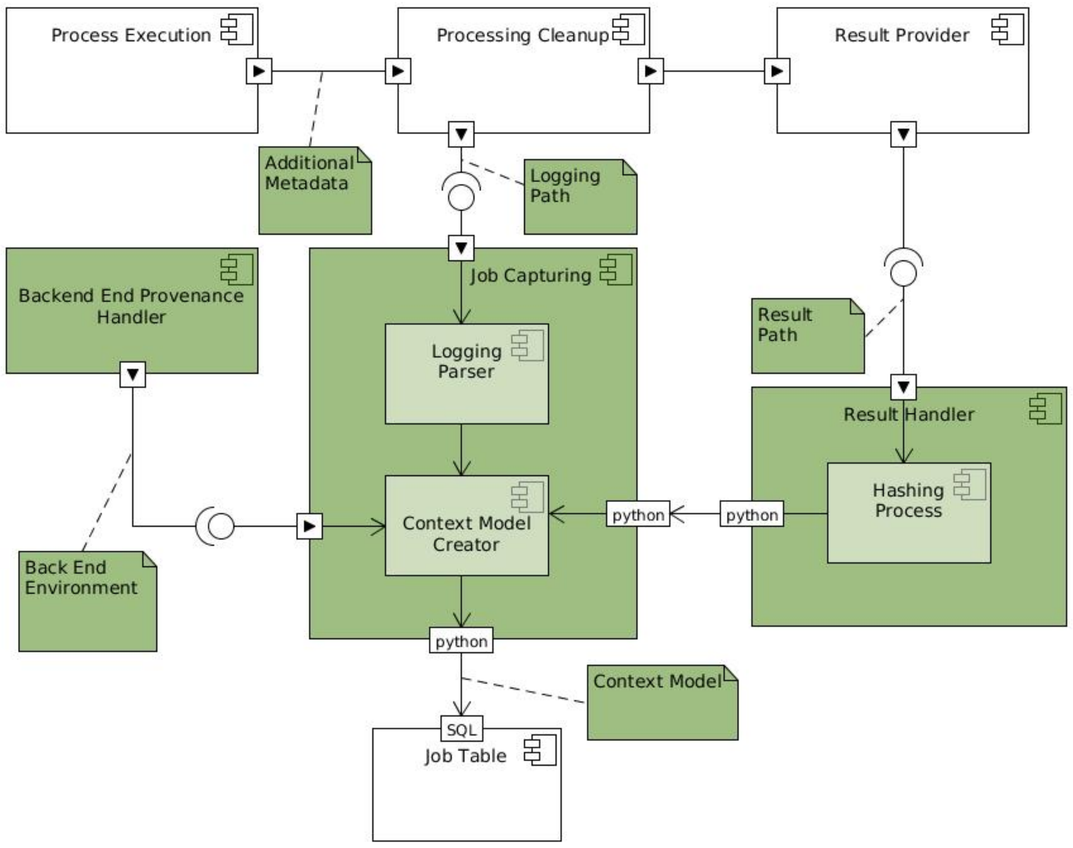


Figure 4.7: Overview of the Job capturing architecture at the backend. Green blocks are additional modules

information to the *Context Model Creator*. The *Context Model Creator* needs the *Result Handler* (f) and *Back End Provenance Handler* (b) for the remaining parts of the context model. After receiving all necessary information, the *Context Model Creator* creates the JSON context model and saves it into the *Job* table of the database.

The following sections describe the capturing of each data element of the job dependent context model in more detail.

(a) **Source Input Data Identifier**
The source input data identifier is the PID of the input data provided by the query store described in Section 4.1. It is forwarded to the *Job Capturing* module by the *Processing Cleanup* module.

(b) **Backend provenance / Code Identifier**
The *Job Capturing* module reads the backend provenance described in Section 4.2. The backend version active at the beginning of the execution is copied to the context model.

(c)(d) **Programming language and used libraries**
The *Process Execution* module uses the installed Python module *pip* to list all installed packages with their versions. The module is at the moment used to manage the Python packages of the backend. The GitHub repository of the backend includes a Python environment file to install all needed dependencies of Python via *pip* automatically. A feature of that tool is the *pip freeze* call, which returns all installed Python packages with their versions. It is then transformed into a JSON object and saved to the context model. In addition to this the *Process Execution* captures the Python version by using the *sys.version* function of the *sys* module. All of this executions are done in the *Process Execution* module in the actual processing environment and stored in the output log file of the job execution.

(e) **Start and end time of process execution**
The start and end time of the process execution is already done by the backend in the *Process Execution* module. The resulting timestamps are persisted in the *Job* table of the database.

(f) **Result Identifier**
The result identifier consists of the resulting data of the whole job execution. It is an SHA-256 hash (using the *hashlib* Python library) of the resulting alphabetical sorted output files, which are placed in the resulting folder of the job execution directory. In the current version of the backend, there is only one result file created, due to the limitations of the available processes.

## 4.4 User Services

The previous sections describe the technical insight of the backend. This section describes the implementation of the interfaces used by the users. Therefore, endpoints are added to the current existing openEO core API. Besides, the proposed endpoints are applied to the Python client and the backend. The implementation of the recommended additions of Section 3.6 are described in the following:

I. **Backend version**
We added a new endpoint for retrieving additional information about the backend. The new endpoint is a GET request with the path "/version" and no authentication is needed to access it. The response of the version endpoint is the job independent provenance information of the backend ((A)-(D), see Section 4.2). In a production version, the data may be filtered for information marked as a security risk. This endpoint is added to the backend to return the latest backend version. The endpoint

takes a timestamp as parameter to get the backend version that was active at a specific time. We added a method to the Python client, so that the user can retrieve this information by calling "version()". The result is a JSON object consisting of the backend provenance data. Listing 4.6 shows a result example.

```
{'branch': 'master',
'commit': '1a0cefd25c2a0fbb64a78cd9445c3c9314eaeb5b',
'url': 'https://github.com/bgoesswein/implementation_backend.git'}
```

Listing 4.6: Backend version example.

II. **Detailed Job Information**
In the openEO coreAPI, there is an endpoint for getting detailed information about a job execution. The endpoint path is "GET /jobs/<job_id>" , which by the current release version (0.3.1) only contains the execution state of the job and the job id. In addition to this, we add the job dependent provenance to the endpoint (e.g. see Figure 4.6). Since the Python client returns the resulting JSON response from the backend as a Python dictionary, there is no modification of the client needed. There is an option to retrieve the detailed job information in PROV-JSON format.

III. **Comparing two Jobs**
There does not exist any user interface to compare jobs in the coreAPI. The solution API defines a new endpoint in the manner of existing endpoint definitions. For this thesis, we introduce the endpoint "POST /jobs/<job_id>/diff". In the URL of the request, the user defines the base job id, which context model is compared with other jobs. In the body of the request, the target job ids are defined in a JSON object. After getting the request from the user, the backend compares the context models of the base job with every target job occurring in the request body. The result from the backend consists of a term for every item in the base job context model. 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 "REMOVED" if the item is in the base job context model, but missing in the target job context model, or "ADDED" if it is the other way around. If an element is different, the elements that differ are visible. The latest mentioned outcome can occur if the context model definition is modified in future job executions and there are e.g. additional fields. The response contains the context model of all jobs with one of the previously described three states inside of the value of every item. In the Python client, this feature is added with an additional function of the Job class called "diff(target_job)". Listing 4.7 provides an example of the dictionary output of this function.

```
{
"process_graph":"EQUAL",
"input_data":"EQUAL",
"code_env":"EQUAL",
"output_data":"EQUAL",
"openeo_api":"EQUAL",
"back_end_timestamp": "REMOVED",
"end_time": "ADDED",
"different":{
   "interpreter": {"source": "Python 3.4", "target": "Python 3.5"},
   "job_id": {"source": "jb-47e062e4-d39c-4f7f-bc5e-aa877f039a84",
              "target": "jb-b5e000f9-f586-40d1-b0b8-c813e5d93b4b"},
   "start_time": {"source": "2019-04-05 12:16:38.286217",
                  "target": "2019-04-05 13:14:22.369015"}}
}
```

Listing 4.7: Example of a job comparison regarding the context model.

IV. **Data Identifier Landing Page**
Depending on the backend, the input data may be restricted to the openEO interface. Therefore, the resolver of the input data PID is set within the coreAPI specification. There exists an endpoint to retrieve detailed information about a data set. We introduce the additional "GET /data/<data-pid>" endpoint. If the user calls the endpoint with a data PID, the result are the details of the underlying dataset and besides, the result of the query execution and the original query parameters. It has to be machine-actionable in JSON format providing the attributes of the query record. 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 first execution. The landing page contains the citation text for the data. In EODC only Sentinel data is available for the users. EODC already stores the official citation text from ESAfor Sentinel. In addition to this text we add the resolvable data PID (see Listing 4.10).

V. **Re-use of Input Data**
To re-use the input data in different job execution, the client user can use the data PID in the process graph directly as source data, instead of just the unfiltered dataset identifier. If a process graph uses the input data PID, the backend automatically applies the queries in the way of the original execution. It is inserted in the process "get_collection" as an additional filter argument named "data_pid", to pass the data PID to the process graph. Section 4.5.1 provides an example of such a process graph. If the query result data changed from the original execution of the PID, the job shall be processed nonetheless, but a warning message appears to notify the

user. The notification appears in the Python client, when the user fetches the result either automatically with synchronous jobs, or by a separate call in asynchronous jobs.

### 4.4.1 PROV Modeling

This section describes the mapping of the PROV elements with the provenance information of the job execution. As mentioned in Section 2.1.1 there are three core elements of the PROV-O annotation. The following enumeration shows how they are mapped with the job provenance information:

1. **Entity element**
   We use entities to represent the data of job executions. Therefore, we model the input data as well as the result data as entities. The PROV-DM definition specifies that an entity needs an identifier and can have additional attributes. We identify the input data with the query PID from the query store and set the entries of the query table as its attributes. The resulting entity is identified by the result hash and has no additional attributes. We set the type of both elements to "dataset".

2. **Activity element**
   An activity has an identifier, an optional start time, an optional end time and optional additional attributes. It takes entities and modifies them to generate new entities. We use the activity element to represent jobs. They are identified by the job id and have the additional information captured in the context model. The connection from the job to the entities is "used" for the input data and "wasGeneratedBy" for the result.

3. **Agent element**
   Agents are capable of starting activities. In our solution we have two instances of agents. First, the user that starts the job execution. Second, the backend in which environment the job is executed. The user id identifies the user and the backend version identifies the backend. Since the backend does not start jobs on his own we add an "actedOnBehalfOf" relation between the backend and the user. We assign the type "Person" to the user and "SoftwareAgent" to the backend.

We implement the PROV-O representation with the "prov"[6] module for Python. It is capable of exporting the provenance in XML, JSON and RDF format. We also use the module to the provenance in images of PNG format. Figure 4.8 shows the PROV-O visualization of the example experiment.

[6]https://github.com/trungdong/prov

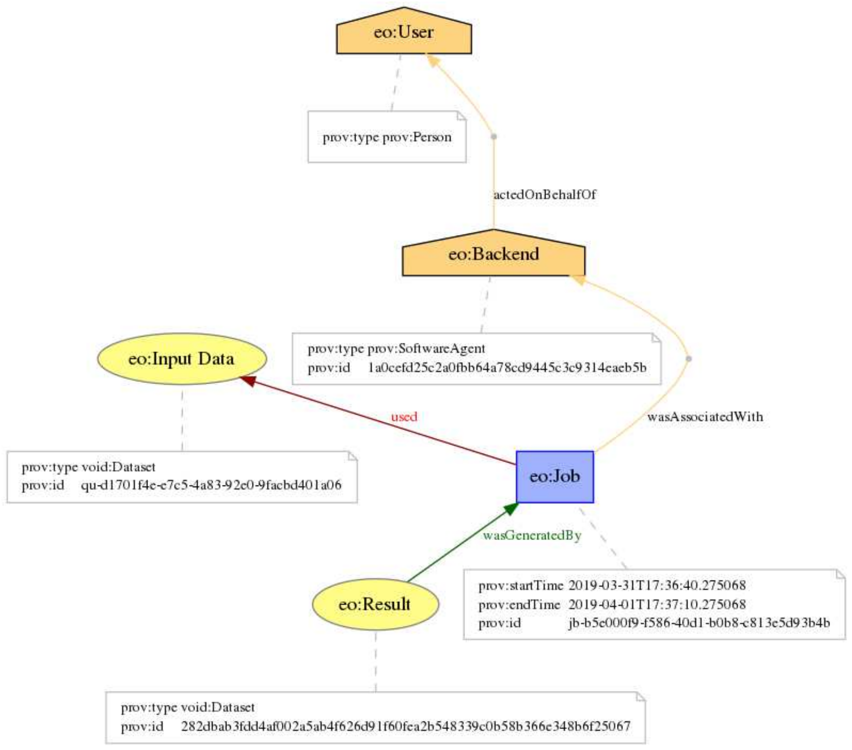

eo:User
prov:type prov:Person
actedOnBehalfOf
eo:Backend
prov:type prov:SoftwareAgent
prov:id 1a0cefd25c2a0fbb64a78cd9445c3c9314eaeb5b
eo:Input Data
used
wasAssociatedWith
prov:type void:Dataset
prov:id qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06
eo:Job
wasGeneratedBy
eo:Result
prov:startTime 2019-03-31T17:36:40.275068
prov:endTime 2019-04-01T17:37:10.275068
prov:id jb-b5e000f9-f586-40d1-b0b8-c813e5d93b4b
prov:type void:Dataset
prov:id 282dbab3fdd4af002a5ab4f626d91f60fea2b548339c0b58b366e348b6f25067

Figure 4.8: PROV-O representation of the example job execution.

## 4.5 Use Cases

This section shows how the use cases of Section 1.1 are addressed using the implementation described in the sections before.

### 4.5.1 Use Case 1 – Re-Use of Input Data

The first use case describes how a researcher uses openEO as a processing environment. In this use case, the focus is on data identification and data citation. Researchers that use openEO may want to cite the data that is used in the applied job. Other scientists then want to use this data in their related research experiment. Section 1.1.2 shows a step-by-step description of the use case. In this section, we execute the steps of the use case with the solution. The implementation of the use cases is available on GitHub[7].

1. **Researcher A runs an experiment (job A) at the EODC backend.**
   This step is basic openEO functionality and is not influenced by the solution from the user point of view. The researcher chooses Sentinel-2 data by loading the Sentinel-2 collection of the backend with the dataset identifier "s2a_prd_msil1c".

[7]https://github.com/bgoesswein/dataid_openeo/tree/master/openeo-python-client/examples

In the next step, the researcher filters the collection data temporally (May of 2017) and spatially (bounding box of the South Tyrol area). In the next step the researcher applies the NDVI[8] on the filtered data as well as the minimum value of each pixel during the time range with the "min_time" process. The NDVI calculation needs the measurements of the satellite in near-infrared (parameter "nir") and the measurements of red light (parameter "red"). The backend represents the band identifiers of the two measurements by "B08" for near-infrared and "B04" for the visible red light. The last two lines create a job at the backend with the process graph and starts the execution of it at the backend. At the backend the *Query Handler* generates a new data PID or returns an already existing one, if the same query was executed in the past. Listing 4.8 represents the experiment Python client code of Researcher A.

```python
import openeo
EODC_DRIVER_URL = "http://openeo.local.127.0.0.1.nip.io"

con = openeo.connect(EODC_DRIVER_URL)

# Choose dataset
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 job A out of the process graph A (pgA)

jobA = con.create_job(pgA.graph)
jobA.start_job()
```

Listing 4.8: Researcher A runs job A with the Python client.

2. **Researcher A retrieves the used input data of job A.**
   Researcher A wants to receive the input data identifier by calling the job description endpoint. Listing 4.9 shows the calls used to do that with the Python client.

```python
pidA_url = jobA.get_data_pid_url()
print(pidA_url)
```

[8]https://earthobservatory.nasa.gov/features/MeasuringVegetation/measuring_vegetation_2.php

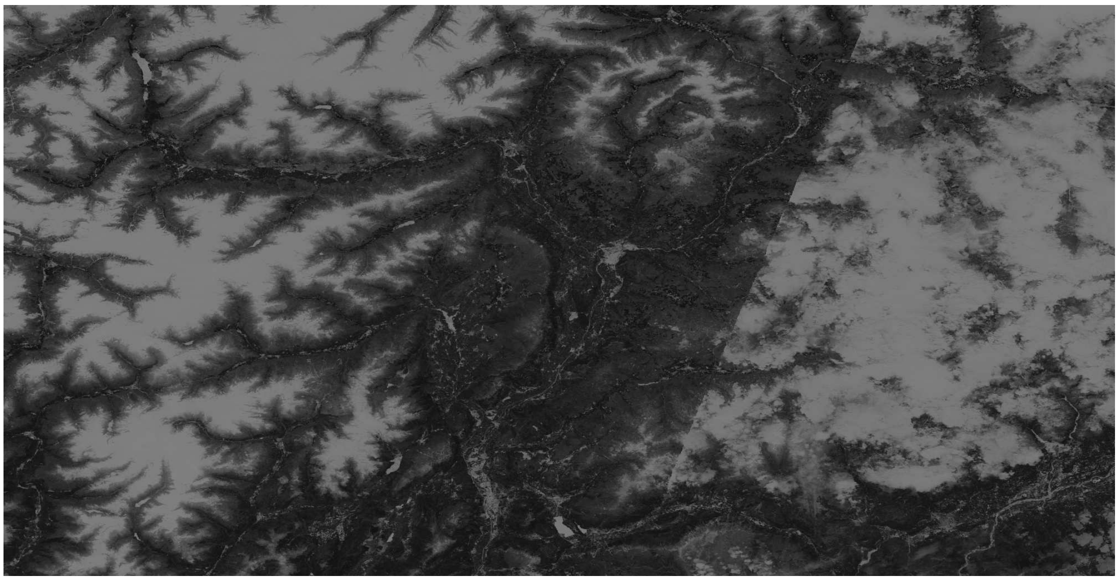


Figure 4.9: Resulting image of the first step of Use Case 1.

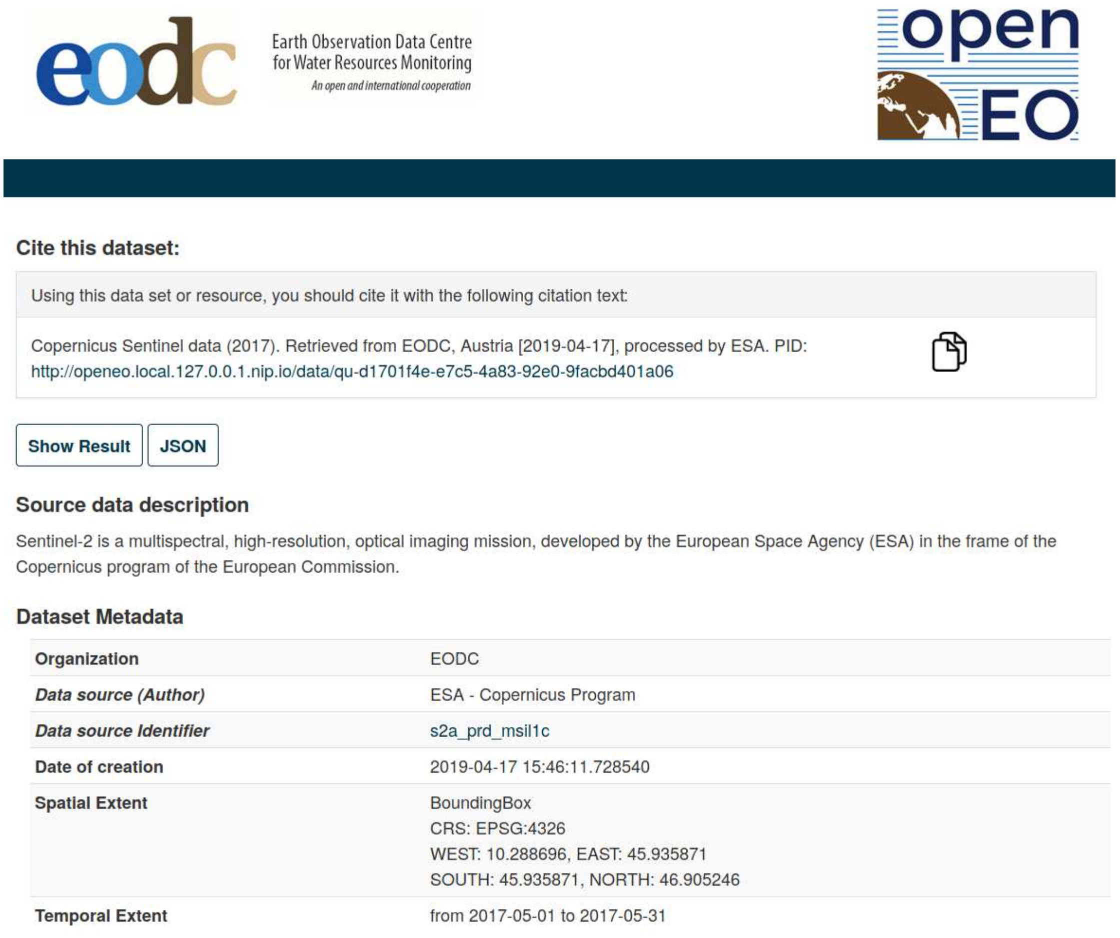
```
# Output: EODC_DRIVER_URL/data
#         /qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06
pidA = jobA.get_data_pid()

# retrieve information about the pidA
# e.g. executed query and description about the dataset.
desc = con.describe_collection(pidA)

query = desc["query"]
# re-execute query and get the resulting
# file list from the backend
file_list = con.get_filelist(pidA)
```

Listing 4.9: Researcher A retrieves the used input data PID.

The user can get the resolvable data PID of the used input data by calling the "get_data_pid_url" method. It contains the input data PID as a resolvable web address. Figure 4.10 shows the response data of the data PID information. After calling the page, the website provides the researcher with the filter parameters, the dataset identifier and a description of the dataset. Besides, the link to get the results of a query re-execution (see "Show Result" in Figure 4.10) and a link to the machine actionable landing page in JSON format are added to the page (see "JSON" button in Figure 4.10). The machine actionable JSON pages for the result and the landing page are extensions to the core API of openEO. The resulting file list of the re-execution is accessible with the "BACKEND_URL/data/data-PID/result" endpoint. The


eodc
Earth Observation Data Centre for Water Resources Monitoring
An open and international cooperation
open EO
Show Result
JSON

**Cite this dataset:**

Using this data set or resource, you should cite it with the following citation text:

Copernicus Sentinel data (2017). Retrieved from EODC, Austria [2019-04-17], processed by ESA. PID: http://openeo.local.127.0.0.1.nip.io/data/qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06

**Source data description**

Sentinel-2 is a multispectral, high-resolution, optical imaging mission, developed by the European Space Agency (ESA) in the frame of the Copernicus program of the European Commission.

**Dataset Metadata**

| | |
|---|---|
| **Organization** | EODC |
| ***Data source (Author)*** | ESA - Copernicus Program |
| ***Data source Identifier*** | s2a_prd_msil1c |
| **Date of creation** | 2019-04-17 15:46:11.728540 |
| **Spatial Extent** | BoundingBox<br>CRS: EPSG:4326<br>WEST: 10.288696, EAST: 45.935871<br>SOUTH: 45.935871, NORTH: 46.905246 |
| **Temporal Extent** | from 2017-05-01 to 2017-05-31 |

Figure 4.10: Screenshot of the pid A landing page. The button "JSON" redirects to a landing page with the same information in JSON format. The button "Show Result", redirects to a new page, which re-executes the query and shows the resulting file list in JSON format.

JSON format landing page is accessible at "BACKEND_URL/data/data-PID/json". The user can retrieve the information of both endpoints via the Python client. The last three calls of the code block above show how the user can gather information about the input data directly in the Python client code.

3. **Researcher A cites the input data in a publication.**
This step is independent from the implementation and therefore not explained in detail. For the further steps it is assumed that Researcher A used the resolvable PID from step 2 (*"BACKEND_URL/data/qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06"*) for the citation. Listing 4.10 shows the whole generated citation text.

```
Copernicus Sentinel data (2017). Retrieved from EODC, Austria
[2019-04-17], processed by ESA.
PID: BACKEND_URL/data/qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06
```

Listing 4.10: Generated citation text of pidA.

4. **Researcher B uses the same input data of job A for job B.**
   To use the same input data as Researcher A, Researcher B uses the data PID from the publication and puts it into the input data element of the process graph of job B. Listing 4.11 gives an example of the code needed to use the same input with a different process constellation (max_time instead of min_time process).

```python
import openeo

# Take input data of job A by using the input data PID A
# pidA = qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06
pgB = processes.get_data_by_pid(
data_pid="qu-d1701f4e-e7c5-4a83-92e0-9facbd401a06")
# Alternative: data_pid="EODC_DRIVER_URL/data/pidA"

# Choose processes
pgB = processes.ndvi(pgB, nir="B08", red="B04")
pgB = processes.max_time(pgB)

# Create job B out of the process graph B (pgB)

jobB = con.create_job(pgB.graph)
jobB.start_job()
```

Listing 4.11: Researcher B uses PID A for different job.

### 4.5.2 Use Case 2 – Capturing job dependent environments

This use case focuses on the execution environment. It handles the need of researchers to describe the execution environments of his experiments. Section 1.1.3 describes the second use case in detail. The implementation at the backend gives the users the option to gain additional data about the execution. In the following, we show the implementation steps to realize the use case with our solution.

1. **Researcher runs an experiment (job A) at a backend.**
   The researcher A runs a job at the backend with the Python client code of Listing 4.12. It is the usual code of executing a job using the Python client.

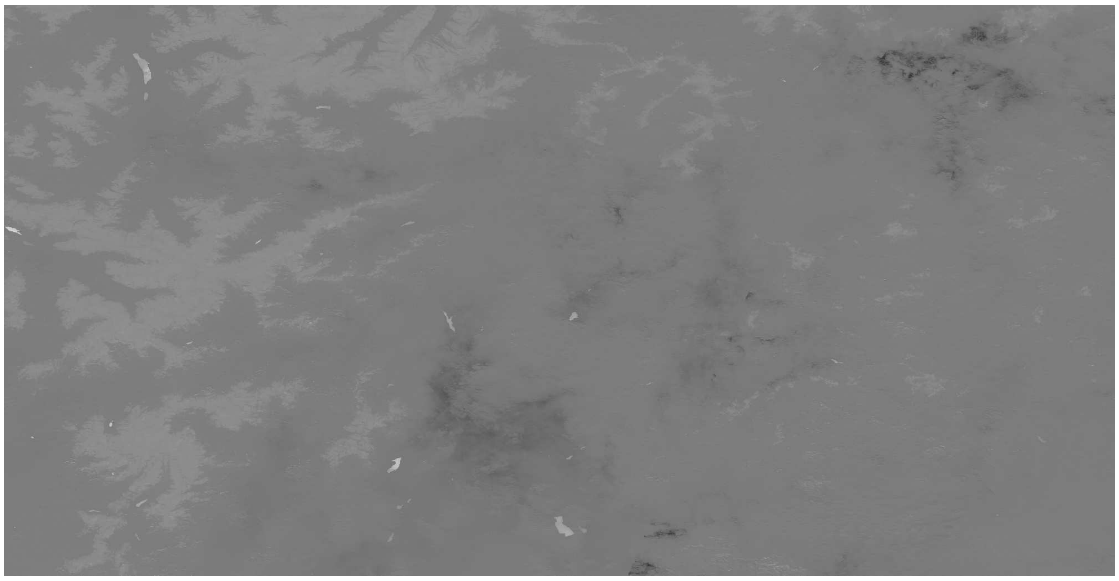


Figure 4.11: Resulting image of the last step of Use Case 1.

```
import openeo
EODC_DRIVER_URL = "http://openeo.local.127.0.0.1.nip.io"

con = openeo.connect(EODC_DRIVER_URL)

# Choose dataset
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 job A out of the process graph A (pgA)
jobA = con.create_job(pgA.graph)
jobA.start_job()
```

Listing 4.12: Researcher A runs job A at the backend using the Python client.

2. **Researcher wants to describe the experiment environment.**
   The researcher wants to publish the result of the experiment and therefore, needs to describe the environment in detail. Listing 4.13 provides the code to give the user detailed information about the job A execution:

   ```
   # Get context model of job A
   context_model = jobA.describe_job["context_model"]
   ```

   Listing 4.13: Describe *jobA* execution environment.

   After the execution of the line above, the researcher can retrieve the used programming language and the installed packages with their versions from the context model. Besides, the backend version identifies the used code for the job execution. Listing 4.14 shows the context model value.

```
{ "backend_version": "16c3b32b5cb2d92d1c32d8c1f929065ee6bf2831",
  "code_env": ['alembic==0.9.9', 'amqp==1.4.9',
      ..., 'GitPython==2.1.11', 'numpy==1.16.2', 'GDAL==2.4.0'],
  "end_time": "2019-04-01 18:13:06.221436"
  "input_data": "qu-3f0b66b8-3cb0-414d-bff4-95e08eb99393",
  "interpreter": "Python 3.7.1",
  "job_id": "jb-f022b568-b674-48d7-9f98-d9555e5eb6f3",
  "openeo_api": "0.3.1",
  "output_data": "5090732d9fb0620773edfcdfc4aad1e9ac771c6eb17b418e5ce5e224aa2bb2a0",
  "start_time": "2019-04-01 18:12:36.221436"
}
```

Listing 4.14: Value of context_model from the second Use Case (see Listing 4.13).

### 4.5.3 Use Case 3 – Getting differences of job executions

The last use case focuses on the users of openEO. It describes the need for transparency of job executions for the users. If results differ with the same job later in time, the user can access data to find reasons why it happened. Therefore, we use the new endpoint to compare job execution environments. Section 1.1.4 describes the third use case in more detail.

1. **Researcher runs an experiment (job A) at a backend.**
   Listing 4.15 shows the Python code used by researcher A to run a new job at the backend.

```
import openeo
EODC_DRIVER_URL = "http://openeo.local.127.0.0.1.nip.io"

con = openeo.connect(EODC_DRIVER_URL)

# Choose dataset
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 job A out of the process graph A (pgA)

jobA = con.create_job(pgA.graph)
jobA.start_job()
```

Listing 4.15: Researcher A runs job A using the Python client.

2. **Researcher re-runs the same experiment (job B).**
   Listing 4.16 shows the re-execution of the same job e.g. by using the same process graph. The new execution gets a new job id and is called "job B" in the following.

```
jobB = con.create_job(pgA.graph)
jobB.start_job()
```

Listing 4.16: Researcher re-reruns job A resulting in job B.

3. **Researcher runs a different experiment (job C).**
   The researcher runs a third job (job C) with a different process graph and input data query. Listing 4.17 shows the code to create job C.

```
# Choose dataset
processes = con.get_processes()
pgC = processes.get_collection(name="s2a_prd_msil1c")
pgC = processes.filter_daterange(pgC, extent=["2017-05-01",
                                              "2017-05-31"])
pgC = processes.filter_bbox(pgC, west=10.288696, south=45.935871,
                                 east=12.189331, north=46.905246,
                                 crs="EPSG:4326")


# Choose processes
pgC = processes.ndvi(pgC, nir="B08", red="B04")
pgC = processes.max_time(pgC) # differs from job A

# Create job C out of the process graph C (pgC)

jobC = con.create_job(pgC.graph)
jobC.start_job()
```

Listing 4.17: Researcher runs experiment different from job A.

4. **Researcher wants to compare the jobs by their environment and outcome.**
   Now the researcher wants to compare job B and job C with the original job A. Therefore he executes the code presented in Listing 4.18.

```
diffAB = jobA.diff(jobB)
diffAC = jobA.diff(jobC)
logging.info("diffAB: \n {}".format(diffAB))
logging.info("diffAC: \n {}".format(diffAC))
```

Listing 4.18: Researcher compares the different jobs.

The researcher gets two dictionaries for the comparisons between job A with job B (diffAB) and job A with job C (diffAC). The content of the dictionary is a comparison of every key of the jobs context model. Listing 4.19 shows the logging output of Listing 4.18 (using logging.info).

```
INFO:root:diffAB:
 {'input_data': 'EQUAL', 'output_data': 'EQUAL',
 'process_graph': 'EQUAL', 'openeo_api': 'EQUAL',
 'interpreter': 'EQUAL', 'code_env': 'EQUAL',
 'different':
 {'back_end_timestamp': {'target': '20190417194702.496810',
                         'source': '20190417154611.728540'},
 'job_id': {'target': 'jb-b92c688c-7fdc-4126-bcdf-85bc07030237',
            'source': 'jb-b5e000f9-f586-40d1-b0b8-c813e5d93b4b'},
 'start_time': {'target': '2019-04-17 19:47:02.496810',
                'source': '2019-04-17 15:46:11.728540'},
 'end_time': {'target': '2019-04-17 19:47:03.258261',
              'source': '2019-04-17 15:46:13.015937'}}


INFO:root:diffAC:
 {'input_data': 'EQUAL', 'output_data': 'EQUAL',
 'openeo_api': 'EQUAL', 'interpreter': 'EQUAL', 'code_env': 'EQUAL',
 'different': {'process_graph': { 'target': {'imagery': {'imagery': {'extent':
 {'crs': 'EPSG:4326', 'east': 12.189331, 'north': 46.905246,
 'south': 45.935871, 'west': 10.288696}, 'imagery':
 {'extent': ['2017-05-01', '2017-05-31'],
 'imagery': {'name': 's2a_prd_msil1c', 'process_id': 'get_collection'},
 'process_id': 'filter_daterange'}, 'process_id': 'filter_bbox'},
 'nir': 'B08', 'process_id': 'NDVI', 'red': 'B04'}, 'process_id':'max_time'},
 'source': {'imagery': {'imagery': {'extent':
 {'crs': 'EPSG:4326', 'east': 12.189331, 'north': 46.905246,
 'south': 45.935871, 'west': 10.288696}, 'imagery':
 {'extent': ['2017-05-01', '2017-05-31'],
 'imagery': {'name': 's2a_prd_msil1c', 'process_id': 'get_collection'},
 'process_id': 'filter_daterange'}, 'process_id': 'filter_bbox'},
 'nir': 'B08', 'process_id': 'NDVI', 'red': 'B04'}, 'process_id':
 'min_time'}},
 'start_time': {'target': '2019-04-17 19:47:09.089075',
                'source': '2019-04-17 15:46:11.728540'},
 'end_time': {'target': '2019-04-17 19:47:11.786845',
              'source': '2019-04-17 15:46:13.015937'},
 'back_end_timestamp': {'target': '20190417194709.089075',
                        'source': '20190417154611.728540'},
 'job_id': {'target': 'jb-ecdd5768-3c22-4c73-85b8-ac6f4bdc138f',
            'source': 'jb-b5e000f9-f586-40d1-b0b8-c813e5d93b4b'}}
```

Listing 4.19: Logging output of the job comparisons diffAB and diffAC.

## 4.6 Summary

This chapter presented an implementation of the design of Chapter 3 at the EODC backend. It contains the implementation of the RDA recommendations. We implemented the query store with two additional tables at the database of the backend. We store the queries of the backend that follow the CSW[9] standard formatted in XML. The additional modules needed to achieve data identification are presented in Section 4.1. Our implementation of the backend provenance uses GitHub as CVS to make the backend environment as well as the code for the job executions identifiable. We capture the job dependent environment by retrieving the Python environment using PIP. Our implementation uses the logging system of the backend to transfer the data to the additional modules. They create the context model by reading the logging files. The data of the query and the context model are stored in additions to the existing PostgreSQL database. Besides, the endpoints needed to access the provenance information lead to extensions of the openEO API. We implemented these extensions into the backend and the Python client. In the last section of the chapter, we present the code to run the use cases of Section 1.1. The following chapter evaluates the implementation by the impact on the EODC backend and by testing the implementation against exceptional cases.

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

CHAPTER 5

# Evaluation

This chapter evaluates the concept of the prototype implementation described in Chapter 4. We evaluate the solution with test cases, which simulate updates on data as well as on the backend environment. After that, we evaluate the performance and storage impact of the implementation by applying 18 test cases derived from 9 publications that used EODC data in the past. The chapter is structured as follows:
First, we describe the setup of the evaluation environment in Section 5.1. Next, Section 5.2 evaluates how the solution behaves on data updates at the backend. The following Section 5.3 evaluates the recreation of an older back end version using the solution. Section 5.4 provides measurements on the performance and storage impact of the prototype implementation.

## 5.1 Evaluation Setup

EODC uses an OpenShift[1] service to manage the backends functionality. For this evaluation we installed OpenShift locally to run the backend with our extensions (furthermore called *Solution Backend*). The data querying of EODC is publicly available and we use it in the evaluation. The *Query Handler* component of the solution operates with the actual data of the backend provided for openEO users.
Figure 5.1 gives an overview of the evaluation setup. The job execution service of the backend cannot be executed locally, because data files are only inside the EODC infrastructure available. Therefore, the processing mechanism is mocked up by the *Processing Mockup* component (see yellow box). It creates an array with mockup values in the size of the query results uses it for the process graph execution. The *Solution Extensions* component (see green box) contains the backend extensions described in Chapter 4. The *Solution Python Client* is the openEO Python client with the extensions of Section 4.4,

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

Table 5.1: Evaluation system specifications.

| | **Hardware** |
|---|---|
| **CPU** | Intel(R) Core(TM) i7-3770T CPU @ 2.50GHz |
| **GPU** | Radeon HD 7750/8740 / R7 250E |
| **RAM** | 16 GB |
| | **Software** |
| **OS** | Kubuntu 18.04.1 LTS |
| **OpenShift** | 3.9.0 |
| **Python** | 3.7.1 |

which give the user the possibility to operate with the proposed features via the Python client. The test cases are Python programs using the *Solution Python Client* to create and execute jobs at the *Solution Backend*. We added a "reset" endpoint to the backend, to be able to run test cases independently. If called, it deletes all existing job and query records from the database.
Table 5.1 specifies the local machine used for the evaluation. To get a minimum performance bias, irrelevant background programs are disabled during the evaluation execution. The *Solution Python Client*, the *Solution Backend*, and the code for the evaluation is available and further described on GitHub[2].

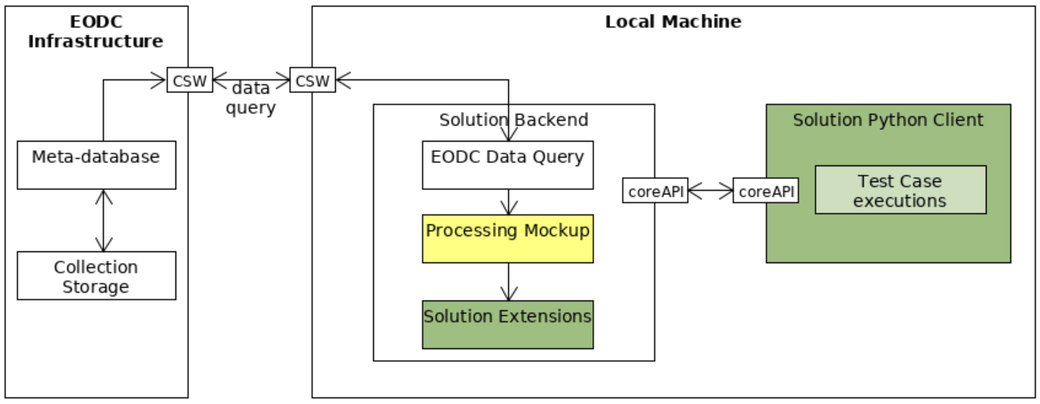


Figure 5.1: Overview of the evaluation setup. Green boxes are components of the solution, white boxes are unmodified existing components and yellow boxes are components added only for the evaluation.

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

## 5.2 Data Identification

This section evaluates the data identification mechanism of the *Solution Backend*. We first show how it fulfills the RDA recommendations. Next, we show how the solution behaves if data updates or deletions occur at the backend.
The policy of the EODC regarding updates on datasets is dependent on the type of data:

1. **Sentinel Data**
   The Sentinel data, which is used by EODC for the openEO project, has never been updated before. The data comes directly from ESA, and updated datasets from ESA were not applied to the EODC data yet. However, the occurrence of updates is, in general, not impossible in future of the backend. If a dataset was updated, it would follow the update policy of ESA, therefore having a new file name for the updated file. An update adds a new dataset record to the PostGIS database used for the data querying. It has the same attribute values as the original one, except for the new generated unique filename and the creation timestamp. In this solution we do not modify the PostGIS server of EODC. We simulate updates by modifying the query execution result. Hence, we simulate updates on the data by renaming input files and their creation timestamp. Other than updating the existing datasets, there are regular updates on the extent of the Sentinel datasets.

2. **Processed Data**
   Data that is processed by partners of EODC is maintained and updated by the partners. EODC archives or deletes old versions depending on the decision of each partner. This type of data is not available using openEO, since it is the property of the partners and not EODC. Besides, to be able to use the same processes on different openEO backends, the data layer needs to be the same. Therefore, all openEO compliant backends provide unprocessed input data. Since the datasets are not available for openEO, the data identification of them is not in the scope of the thesis. By the time this thesis is written, EODC plans to use the openEO framework for their whole infrastructure, which brings the partners to use openEO for their processing. Therefore they can use the data identification implementation for their tasks.

### 5.2.1 RDA Recommendations

The Design (see Section 3.2) proposes the implementation of the RDA data identification recommendations at the backend. Therefore, the following enumeration shows how we tackle the recommendations.

- **R1: Data Versioning**
  The backend does versioning of data already, by following the versioning policies of ESA. The path to a file represents the version identifier. Modified files must have a different file path. Every file is represented as a data record in the PostGIS database,

which stores the metadata e.g. timestamp of capturing and geographical extent. Every data record has an creation timestamp. It represents the time since the data record version is available at the backend. If a file is updated, the filename has to change as well as the creation timestamp. A query execution without filtering by the creation timestamp results in the file with the most recent creation timestamp. We filter by the creation timestamp to get the the file version of a specific time.

- **R2: Timestamping**
  The creation timestamp of the files are stored by the file system of the backend and the PostGIS database. Since the files have to change the path after an update, the creation timestamp of the updated file is the update timestamp of the original file.

- **R3: Query Store Facilities**
  We introduce a new table in the database (PostGreSQL) of the backend for the query store. The query table stores the following data about each query:
  - The original CSW query in XML format
  - The unique query: The backend generates the unique query using the filter arguments of the original query. It is then stored as a alphabetically sorted JSON object.
  - The hash of the unique query after removing characters with no semantic meaning.
  - The hash of the alphabetically sorted resulting file list, after removing characters with no semantic meaning.
  - The timestamp of the original query execution.
  - The dataset identifier is set to the used collection identifier of the query.
  - The persistent identifier of the query generated using the UUID[3] library of Python.
  - A JSON object with the number of result files.

- **R4: Query Uniqueness**
  The alphabetically sorted filter criteria generated by the backend (See Figure 4.5).

- **R5: Stable Sorting**
  The original CSW query assures stable sorting. It alphabetically sorts the resulting file list in ascending order.

- **R6: Result Set Verification**
  The resulting file list is alphabetically sorted in ascending order. It transfers to a string object that cleaned up by not relevant characters, which we hash by the SHA-256 hash function.

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

- **R7: Query Timestamping**
  The implementation stores the timestamp of the original query execution.

- **R8: Query PID**
  The query PID is created using the Python UUID library if the same unique query and resulting hash combination is not in the database yet.

- **R9: Store the Query**
  We implemented the query store as an additional table of an existent relational database of the backend (PostgreSQL).

- **R10: Automated Citation Texts**
  The citation text for the dataset is already available at EODC. We added the generated data PID and filter arguments (See Listing 4.10).

- **R11 & R12: Landing Page & Machine Actionability**
  We define the landing page as an additional openEO endpoint. It is publicly accessible as a human readable website and additionally in JSON format, which makes it machine actionable (see Figure 4.10). The landing page includes a link to re-execute the query and lists the result files also in JSON format. Since it is part of the openEO API, it can also be accessed from the openEO clients and be used in future jobs as input data.

- **R13 & R14: Technology Migration & Migration Verification**
  These recommendations are not implemented in this thesis, since there are no EODC migrations planned. Nevertheless, we estimate potential risks of a migration process. Using the Git commit as identifier of code might cause issues, when migrating to a different VCS. If the repository just moves to a different Git server, the whole history, including commit identifiers, can be migrated. The creation timestamps of the datasets have to be migrated properly, since they are an important information needed by our solution.

The original data of EODC can't be changed for this evaluation since the evaluation backend is using the actual data source (https://csw.eodc.eu). To simulate changes on the data, we modify the query result received by the backend. Figure 5.2 gives an overview of where the data update simulator is located in the data identification implementation overview of Figure 4.3. The test cases describe what data is modified.
The test cases of the following sections focus on the Sentinel Data of EODC, and how the solution behaves on data updates. We use the running job (referenced as *jobA*) defined in Section 1.1.1 for the test case execution. Every test case starts with one executed job at an empty database. It stores the first query table entry. The next section describes how we achieve the initial test case state.

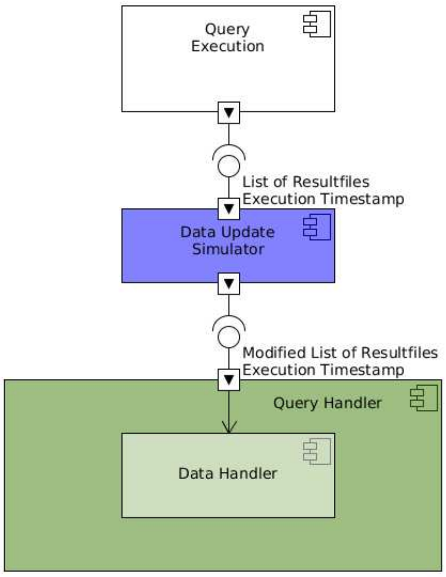


Figure 5.2: Overview of the data update simulation. It shows the extension on the data identification implementation of Figure 4.3.

### 5.2.2 Test Case Preparation

This section describes the test case preparation, so the step executed before every test case execution. It creates an initial job and input query entry, which is needed for the test cases. In the beginning the database is empty, we achieve this by calling the introduced reset endpoint.

1. **Run *jobA*, which creates query *pidA*. Get result files of *pidA***
   Listing 5.1 shows the code to run *jobA* using the Python client. The researcher defines the spatial and temporal filter arguments and applies the NDVI and the "minimum time" process on it. This is the running example described in Section 1.1.1. The "con.create_job()" call sends the process graph to the backend and retrieves a job object containing the newly generated job id. After executing *jobA* (calling start_job()), a job entity and a query entity are created and stored in the database. The query entity with a newly generated data PID is created by the *Query Handler*. Table 5.2 shows the state of the query table after the execution of Listing 5.1.

```
con = openeo.connect("http://openeo.local.127.0.0.1.nip.io")
# Choose dataset
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()
# Get data PID of jobA
pidA = jobA.get_data_pid()
# Re-execute the query to print the used files.
file_listA = con.get_filelist(pidA)
# Get state of the resultfiles, so if they changed since
# the original execution
file_listA["input_files"]["state"] # Returns "EQUAL"
```

Listing 5.1: Researcher runs *jobA* and retrieves the result files status.

Figure 5.3 shows the original query. It is has an XML format and contains the filter values defined in the process graph (pgA). Figure 5.3 highlights the timestamp of the first query execution. The content of the database tables as well as query re-execution results after each execution step are available in the result folder[4] of the GitHub repository.

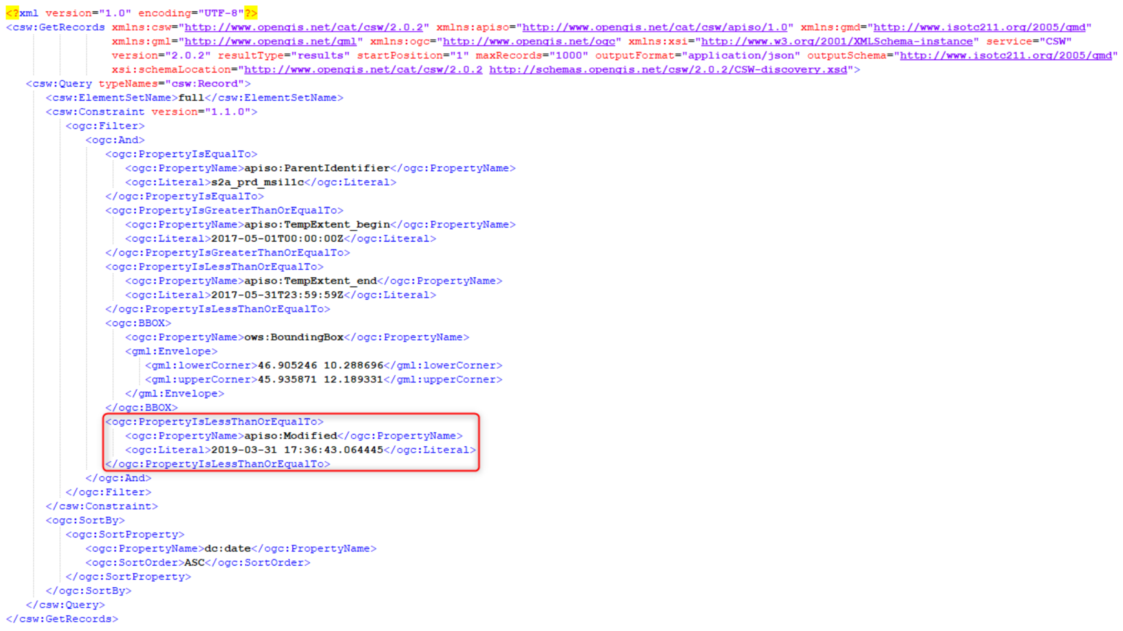
Listing 5.2 shows the normalized query created by the *Query Handler*. It contains all filter arguments and their values in a JSON object alphabetically sorted. Note that filter options that the user does not use have a "None" value.

```
{'bands': None,
'data_id': None,
'derived_from': None,
'extent': {'extent': {'crs': 'EPSG:4326', 'east': 12.189331,
'north': 46.905246, 'south': 45.935871, 'west': 10.288696}},
'license': None,
'name': 's2a_prd_msil1c',
'time': {'extent': ['2017-05-01', '2017-05-31']}}
```

Listing 5.2: Normalized query of the initial query entry.

Listing 5.3 shows the first four file paths and timestamps of the resulting file list, after executing the query of Figure 5.3. The query in Figure 5.3 in XML format

[4]https://github.com/bgoesswein/dataid_openeo/tree/master/openeo-python-client/examples/results



```xml
<?xml version="1.0" encoding="UTF-8"?>
<csw:GetRecords xmlns:csw="http://www.opengis.net/cat/csw/2.0.2" xmlns:apiso="http://www.opengis.net/cat/csw/apiso/1.0" xmlns:gmd="http://www.isotc211.org/2005/gmd"
                xmlns:gml="http://www.opengis.net/gml" xmlns:ogc="http://www.opengis.net/ogc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" service="CSW"
                version="2.0.2" resultType="results" startPosition="1" maxRecords="1000" outputFormat="application/json" outputSchema="http://www.isotc211.org/2005/gmd"
                xsi:schemaLocation="http://www.opengis.net/cat/csw/2.0.2 http://schemas.opengis.net/csw/2.0.2/CSW-discovery.xsd">
   <csw:Query typeNames="csw:Record">
      <csw:ElementSetName>full</csw:ElementSetName>
      <csw:Constraint version="1.1.0">
         <ogc:Filter>
            <ogc:And>
               <ogc:PropertyIsEqualTo>
                  <ogc:PropertyName>apiso:ParentIdentifier</ogc:PropertyName>
                  <ogc:Literal>s2a_prd_msil1c</ogc:Literal>
               </ogc:PropertyIsEqualTo>
               <ogc:PropertyIsGreaterThanOrEqualTo>
                  <ogc:PropertyName>apiso:TempExtent_begin</ogc:PropertyName>
                  <ogc:Literal>2017-05-01T00:00:00Z</ogc:Literal>
               </ogc:PropertyIsGreaterThanOrEqualTo>
               <ogc:PropertyIsLessThanOrEqualTo>
                  <ogc:PropertyName>apiso:TempExtent_end</ogc:PropertyName>
                  <ogc:Literal>2017-05-31T23:59:59Z</ogc:Literal>
               </ogc:PropertyIsLessThanOrEqualTo>
               <ogc:BBOX>
                  <ogc:PropertyName>ows:BoundingBox</ogc:PropertyName>
                  <gml:Envelope>
                     <gml:lowerCorner>46.905246 10.288696</gml:lowerCorner>
                     <gml:upperCorner>45.935871 12.189331</gml:upperCorner>
                  </gml:Envelope>
               </ogc:BBOX>
               <ogc:PropertyIsLessThanOrEqualTo>
                  <ogc:PropertyName>apiso:Modified</ogc:PropertyName>
                  <ogc:Literal>2019-03-31 17:36:43.064445</ogc:Literal>
               </ogc:PropertyIsLessThanOrEqualTo>
            </ogc:And>
         </ogc:Filter>
      </csw:Constraint>
      <ogc:SortBy>
         <ogc:SortProperty>
            <ogc:PropertyName>dc:date</ogc:PropertyName>
            <ogc:SortOrder>ASC</ogc:SortOrder>
         </ogc:SortProperty>
      </ogc:SortBy>
   </csw:Query>
</csw:GetRecords>
```

Figure 5.3: Original query of *jobA* after the test case preparation step.

follows the CSW standard for EO web service catalogs. The backend sends it to the PostGIS server of EODC, which translates it to the SQL based internal query and executes it. The resulting file list (file_listA["input_files"]) are 51 files and the re-execution of the query results in the same 51 files. The path of each file provides information about the data record. The term "s2a_prd_msil1c" defines the dataset identifier of Sentinel 2 at the backend. The following folders define the date of the data record. The filename follows the ESA naming convention[5]. The Nxxyy value defines the baseline number, Rxxx defines the orbit number of the satellite and Txxxxx defines the tile number. The first date is the sensing time and the second at the end of the file name is the product disclaimer, therefore distinguishes products that came from the data of the same sensing time. The timestamp shows when the data was first available at EODC.

[5]https://sentinel.esa.int/web/sentinel/user-guides/sentinel-2-msi/naming-convention

Table 5.2: Query table after the execution of Listing 5.1

| Query *pidA* | |
|---|---|
| **Column** | **Value** |
| Query PID | qu-a3bbe4a0-a875-4687-bb78-9457f33134a9 |
| Dataset PID | s2a_prd_msil1c |
| Original Query | *XML CSW query (see Figure 5.3)* |
| Unique Query | *Sorted filter criteria (see Listing 5.2)* |
| Query Hash | 0917c7a21cec960b8a6617b22ad26578c2c67f0b0501ba1a359b078c6c51d77d |
| Result Hash | abf43f519007050cbaeb59a067a2226d64b041c6d6ec323b2401109176e66455 |
| Execution Timestamp | 2019-03-31 17:36:43.064445 |
| Metadata | {'result_files': 51} |

```
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQS_20170504T101349.zip',
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPT_20170504T101349.zip'},
...
```

Listing 5.3: First four resulting files with creation timestamp.

Table 5.2 shows the content of the query table after the execution of the test case preparation step. It has one entry with a new generated data PID, since there was no query entry before.

### 5.2.3 Test Case 1: Is it possible to re-execute a query after a file is updated?

2. **Update one of the resulting files of the *pidA* query**
   We added the "update_file()" method to the client and the backend to let the backend activate the *Data Update Simulator* component shown in Figure 5.2. If it is activated, it simulates the update of the first file in the query result. It sets the creation timestamp of the data record to the execution time of the "update_file()" method and appends "_new" to the file path. The update does not replace the old file but adds a new file to the result. Therefore, we simulate a new additional dataset record in the PostGIS database.

```
# Update the first file of the pidA query resulting files.
con.update_file()
```

Listing 5.4: Update one of the *pidA* resulting files, but keep the original file.

Listing 5.5 shows the result file list with the activated *Data Update Simulator*. There is now an additional file with "_new" at the end of the path, but with the same information in the file path and name as the first file. Besides, the creation timestamp is set to the time of the execution of Listing 5.4.

```
{'timestamp': '2019-03-31 17:44:43',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPR_20170504T101349_new.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQS_20170504T101349.zip',
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPT_20170504T101349.zip'},
...
```

Listing 5.5: Modified file list output after the *Data Update Simulator* component.

3. **Re-execution of *pidA* query**
   Listing 5.6 shows the re-execution of the query with the identifier *pidA*, after the activation of the *Data Update Simulator* component in the previous step. The second query re-execution results in the same file list as the first one (see Listing 5.7). The reason for this is that the old file is still available and the updated file is added after the execution timestamp of the original query execution (see timestamp element in the query in Figure 5.3). The query from the query store contains the execution timestamp. This shows that earlier states of datasets can be retrieved with our solution.

```
# Get state of the resultfiles, so if they changed since
# the original execution.
file_listA = con.get_filelist(pidA)
file_listA["input_files"]["state"] # Returns "EQUAL"
```

Listing 5.6: Re-execute *pidA* query after one file got updated.

```
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQS_20170504T101349.zip',
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPT_20170504T101349.zip'},
...
```

Listing 5.7: First four resulting files of the file list.

4. **Run duplicate of *jobA* named *jobB***
   Listing 5.8 shows the execution of a second job using the same process graph as *jobA*. Such scenarios happen within the openEO project when users want to re-run their old experiments using new data. In the first line of Listing 5.8 the process graph of *jobA* (defined in Listing 5.1) is used to create a new job instance named *jobB*. Therefore, *jobB* uses the same process graph as *jobA* but gets executed after the update of the file in Listing 5.4. It results in a different file list with the new file is in the query result file list (see Listing 5.3).

```
# Reuse the defined process Graph (pgA) from jobA at Step 1 to create jobB
jobB = con.create_job(pgA.graph)
jobB.start_job()
# re-execute query and get the resulting file list from the backend
pidB = jobB.get_data_pid()
file_listB = con.get_filelist(pidB)
# comparing the resultfiles of jobA with the resultfiles of jobB
(file_listA == file_listB) # Returns False
```

Listing 5.8: Step 4: Create *jobB*, which uses the same process graph as *jobA*.

Table 5.3: Query table after the execution of Listing 5.8. Important elements are highlighted blue if they are the same and red if they are different.

| **Query *pidA* (full entry in Table 5.2)** | |
|---|---|
| **Column** | **Value** |
| Query PID | qu-a3bbe4a0-a875-4687-bb78-9457f33134a9 |
| Query Hash | 0917c7a21cec960b8a6617... |
| Result Hash | abf43f519007050cbaeb59... |
| Metadata | {'result_files': 51} |
| **Query *pidB*** | |
| **Column** | **Value** |
| Query PID | qu-23f5a313-e804-4faa-aa33-60ed1ac69e2d |
| Dataset PID | s2a_prd_msil1c |
| Original Query | *see Figure 7.1 in the appendix* |
| Unique Query | *see Listing 5.2* |
| Query Hash | 0917c7a21cec960b8a6617... |
| Result Hash | 28088d113de19ce037e965... |
| Metadata | {'result_files': 51} |
| Execution Timestamp | 2019-03-31 18:01:47.695042 |

Table 5.3 shows the query table after the execution of Listing 5.8. Now there is an additional query entry. Table 5.3 marks the important differences red. There is a different result hash, because the resulted file name of the updated file is set to the most recent one and not the one from the original query execution.. The normalized query is still the same, but the result of the query changed therefore, a new data PID is generated.

Table 5.4: Resulting mapping of the jobs and the used data PID of the first test case.

| Job | Query PID |
|---|---|
| *jobA* | qu-a3bbe4a0-a875-4687-bb78-9457f33134a9 |
| *jobB* | qu-23f5a313-e804-4faa-aa33-60ed1ac69e2d |
| *jobC* | qu-a3bbe4a0-a875-4687-bb78-9457f33134a9 |

5. **Run duplicate of *jobA* named *jobC*, by using the data PID of *jobA***
   This step shows how a researcher can run a new job (*jobC*) with the original data version of *jobA*. Other than *jobB*, it uses the original data version even if there are new versions available. The persistent input data identifier of job A (*pidA*) is used as the input data for *jobC*. The execution timestamp is part of the query entry behind *pidA*, and the backend executes the original query for *jobC*. *jobC* uses the same data PID as *jobA* therefore, the updated file is not in the result file list and there is no new data PID generated. The query table is still in the state of Table 5.3.

```
# Take input data of job A by using the input data PID A of job A
pgC = processes.get_data_by_pid(data_pid=pidA)
# Choose processes
pgC = processes.ndvi(pgC, nir="B08", red="B04")
pgC = processes.min_time(pgC)
# Create and start Job C
jobC = con.create_job(pgC.graph)
jobC.start_job()
# re-execute query and get the resulting file list from the backend
pidC = jobC.get_data_pid()
file_listC = con.get_filelist(pidC)
# Compare resulting files with the original execution of jobA
(file_listA == file_listC) # Returns True
```

Listing 5.9: Create *jobC*, which uses the input data identified by *pidA*.

Table 5.4 presents the mapping between the executed jobs and the input data PIDs of the first test case. The results are consistent if files are updated at the backend. Jobs using the original data PID (*jobC*), also use the data defined by the PID, even after the update. The reason for this is the original file was still available. The current way of reproducing a job in openEO, by applying the same process graph (*jobB*), fails since the resulting files differ from the first execution (*jobA*). The query execution timestamp information is missing. Hence, the input data of *jobB* result in a different data PID.

### 5.2.4 Test Case 2: Is it possible to re-execute a query after a file is updated with the original one deleted?

2. **Update one of the resulting files of the *pidA* query and remove the original one.**
The method in Listing 5.10 updates the first file of the query result as described in Section 5.2.3 and removes the original file. Listing 5.11 shows the updated file list, where the first entry replaced the original first file (see Listing 5.5).

```
con.update_file(deleted=True)
```

Listing 5.10: Update one of the *pidA* resulting files and delete the original file.

```
{'timestamp': '2019-03-31 17:44:43',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPR_20170504T101349_new.zip'}
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQS_20170504T101349.zip',
{'timestamp': '2017-05-08 00:00:00',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TQR_20170504T101349.zip'},
{'timestamp': '2017-05-08 00:00:00',
'path':'/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/04/
S2A_MSIL1C_20170504T101031_N0205_R022_T32TPT_20170504T101349.zip'},
...
```

Listing 5.11: Modified file list output of the *Data Update Simulator* component, by removing the original file from the list.

3. **Get File-list of *pidA***
The re-execution of the query *pidA* results in a file list without the deleted file. Since files are filtered by the query execution timestamp. The new file does not appear in the result file list. If the re-execution results not in the same file list, the "state" attribute contains "Incomplete Result" and a list of files that replaced no longer available files are stored in the "diff" attribute. Listing 5.13 shows the content of the "diff" attribute of Listing 5.12. The backend returns the most recent file version, even if there are versions between the original and the most recent file available. In the result of Listing 5.12 the "diff" contains only the one file that replaces the original file. Users can see the alternatives for the original file, but not the original file itself. This is because the full file list is not persisted in the query store, but the number of result files. The decision for not storing the complete file list was made by EODC. The reason is because of the additional needed storage size it would cause and the rare occasion of a data update. If the number of resulting files is different at a re-execution, we execute the query without the creation timestamp filter, and the result is compared to the re-execution with the original execution timestamp. We compare the attributes of the original files (except for the creation

timestamp) with the most recent files. If there is no matching file in the original file list, the most recent file is added to the "diff" attribute. The researcher can use the resulting "diff" file list to contact EODC or ESA to order older version.

```
# re-execute query and get the resulting file list from the backend
file_listA = con.get_filelist(pidA)
# Stdout: Warning: The resulting file list changed from the original query
# execution! Look into the "diff" attribute to see the list of files that
# have changed. The original data might be still obtainable by the original
# data provider. Please contact the backend provider for further information.
file_listA["input_files"]["state"] # Returns "Incomplete Result"
file_listA["input_files"]["diff"] # Returns one file entry
```

Listing 5.12: Re-execute *pidA* query after one file is updated and the old version is erased.

```
[{'timestamp': '2019-03-31 17:44:43',
'path': '/eodc/products/copernicus.eu/s2a_prd_msil1c/2017/05/24/
S2A_MSIL1C_20170524T101031_N0205_R022_T32TQR_20170524T101353_new.zip'}]
```

Listing 5.13: List of files that replaced original files of the query result.

4. **Run duplicate of *jobA*, by using the data PID of *jobA* named *jobD***
Listing 5.14 shows the code of running a new *jobD* with the data of *pidA*. The code is executed after the deletion of one file from the resulting file list of *pidA*. After the creation of the job, the backend notices that the number of result files is different than the first execution of the input data PID. Therefore, a warning message is displayed at the client of the researcher (see Listing 5.14). If the user starts the job nevertheless, the backend looks for updated files like described in the previous step and adds the most recent version of the missing file to the query result. EODC chose this strategy, since updates on datasets usually fix errors, users expect the most recent ones. Most backends e.g. Google Earth Engine give the users the most recent version automatically. The backend creates a new data PID for *jobD*, since the query result changed. Table 5.5 shows the query table status after the execution. The second query entry has a different result hash, since one file changed.

Table 5.5: Query table after the execution of Listing 5.8. Important elements are highlighted blue if they are the same and red if they are different.

| **Query *pidA* (full entry in Table 5.2)** | |
|---|---|
| **Column** | **Value** |
| Query PID | qu-a3bbe4a0-a875-4687-bb78-9457f33134a9 |
| Unique Hash | 0917c7a21cec960b8a6617... |
| Result Hash | abf43f519007050cbaeb59... |
| Metadata | {'result_files': 51} |
| **Query *pidD*** | |
| **Column** | **Value** |
| Query PID | qu-3544aeae-cd24-4b6d-ad34-0d674c2a400f |
| Dataset PID | s2a_prd_msil1c |
| Original Query | *see Figure 7.2 in the appendix* |
| Unique Query | *see Listing 5.2* |
| Query Hash | 0917c7a21cec960b8a6617... |
| Result Hash | 28088d113de19ce037e965... |
| Metadata | {'result_files': 51} |
| Execution Timestamp | 2019-03-31 18:10:25.46402 |

```
# Take input data of job A by using the input data PID A of job A
pgD = processes.get_data_by_pid(data_pid=pidA)
# Choose processes
pgD = processes.ndvi(pgC, nir="B08", red="B04")
pgD = processes.min_time(pgD)
# Create and start Job D
jobD = con.create_job(pgD.graph)
# Stdout: Warning: The resulting file list changed from the original query
# execution! Look into the "diff" attribute to see the list of files that
# have changed. The original data might be still obtainable by the original
# data provider. Please contact the backend provider for further information.
jobD.start_job()
pidD = jobD.get_data_pid() # pidD != pidA
```

Listing 5.14: Run duplicate of *jobA*, by using the data PID of *jobA* named *jobD*.

The result of the second test case shows how the backend behaves on data updates replacing the original data. If the file is deleted, the researcher gets a warning message and a list of files that are replacing the original files. Since the path of the file identifies the date and tile that was used, the researcher can notify EODC about the concrete changed file. On creation of the new *jobD* that uses the input data identifier *pidA*, the client notifies the user that the result files changed, before the execution happens. Then the user can decide to start the job nevertheless or contact EODC about the modified files.

### 5.2.5 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 it would restrict their range on available data. If it happens nevertheless, the data records are still persisted in the database and can therefore be used to identify the missing files. If they are also deleted in the database it can only occur by mistake, because EODC does not delete data records from the database. If it happens nevertheless, there is in the current solution no possibility to get the exact removed files. The number of result files in the "meta_data" column shows how many files changed. The whole result files need to be added to the query table to achieve the knowledge of missing files. In our solution, the user has to notify EODC about the missing files. How EODC gets the files from ESA in such a case is not in scope of this thesis.

## 5.3 Job Capturing

This section reviews the capabilities of the solution regarding job capturing. The focus is on the context model of the job execution. The following question is used to discuss the impact of the captured data regarding job execution.

**Is it possible to recreate an older version of the backend?**
The solution aims to capture enough data to make it possible to re-run the same job at the same backend. The backend is created directly by its GitHub repository. The GitHub repository URL and the commit identifier of the original setup are needed to recreate an old version of the backend. The timestamp of the job execution is stored in the context model of the original job execution. The backend provenance can resolve the version of the backend and therefore the GitHub repository and commit via execution timestamp. Therefore, the information on re-creating a backend version from an older version is captured. The process graph of the job is persisted as well. The re-execution of the job is the same, assuming the input data has not been deleted in the meantime, which can be checked by re-executing the data query of the job. The resulting hash makes it possible to validate if the job re-execution us done in the same way. The following code presents how this can be achieved in our solution.

1. **Run *jobA*, which creates query *pidA* and *job_idA*.**
   The first step shows the definition, creation, and execution of *jobA*. The last two lines show the retrieval of the backend version. The "con.version()" method returns the current version of the backend, but can also take a timestamp as parameter to get the version of a particular time. The "jobA.get_backend_version()" method returns the version of the backend during the execution of *jobA*. Both versions are at the end of Step 1 identical.

```
import openeo
# Connect with GEE backend
con = openeo.connect("http://openeo.local.127.0.0.1.nip.io")
# Choose dataset
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
jobA = con.create_job(pgA.graph)
jobA.start_job()
# Get current backend version
version_old = con.version()
# Get backend version of jobA
versionA = jobA.get_backend_version()
```

Listing 5.15: Step 1: Researcher runs *jobA* and gets the used backend version.

```
{'branch': 'master',
'commit': '1a0cefd25c2a0fbb64a78cd9445c3c9314eaeb5b',
'url': 'https://github.com/bgoesswein/implementation_backend.git'}
```

Listing 5.16: Step 1: Version of the *jobA* execution *version_old*.

2. **Publish *job_idA* and *pidA*.**
   The researcher retrieves the persistent input data identifier *pidA* and the job identifier *jobA_id* to cite the provenance of the execution. The timestamp of the execution in the context model is used to identify the backend version of the job execution.

```
# Get input data PID of jobA
pidA = jobA.get_data_pid()
jobA_id = jobA.job_id
```

Listing 5.17: Researcher gets the input data PID of *jobA* and the *job_id* of *jobA*.

3. **Update backend version**
   In this step, we modify the backend by updating one Python package in the requirements file used for the job execution. We edit the file by replacing the line "urllib3==1.23" with "urllib3==1.24.1". After editing we call the "git commit" command to get a new version of the backend.

4. **Get original context of *jobA***
   Listing 5.18 shows the code needed by the researcher to get the original version of the backend during the *jobA* execution. The variable *versionA* contains the version displayed in Listing 5.16.

5. **Re-run *jobA* with the original version.**
   To get the original version of the backend, EODC has to create a second instance of the backend. Then they have to check out the commit of the job execution, by running "git checkout commit_id" in the console, where "commit_id" is the value of *versionA["commit"]*.

```python
import openeo
# Connect with GEE backend
con = openeo.connect("http://openeo.local.127.0.0.1.nip.io")
# Get jobA using the jobA_id.
jobA = con.get_job(jobA_id)
# Get the version of the backend that was active during the job A execution
versionA = jobA.get_backend_version()
```

Listing 5.18: Researcher retrieves the original backend version of the *jobA* execution.

## 5.4 Performance and Storage Impact

This section evaluates the performance and storage impact of the solution on the EODC backend. We define 18 input process graphs from 9 publications using data provided by EODC from the last two years. The data used in the papers provide spatial and temporal extents. These 18 input process graphs define the 18 test cases. Table 5.6 shows the papers used for the test cases. The evaluation code (see evaluation_impact.py on GitHub) contains the values of the spatial and temporal extent. The performance of the *Solution Backend* is compared to a local EODC backend, without the extensions of this thesis (from now on referred to as *Reference Backend*). Note that in the *Reference Backend* the processing is mocked up in the same way as in the *Solution Backend*. Figure 5.4 gives an overview of the performance and storage evaluation setup.

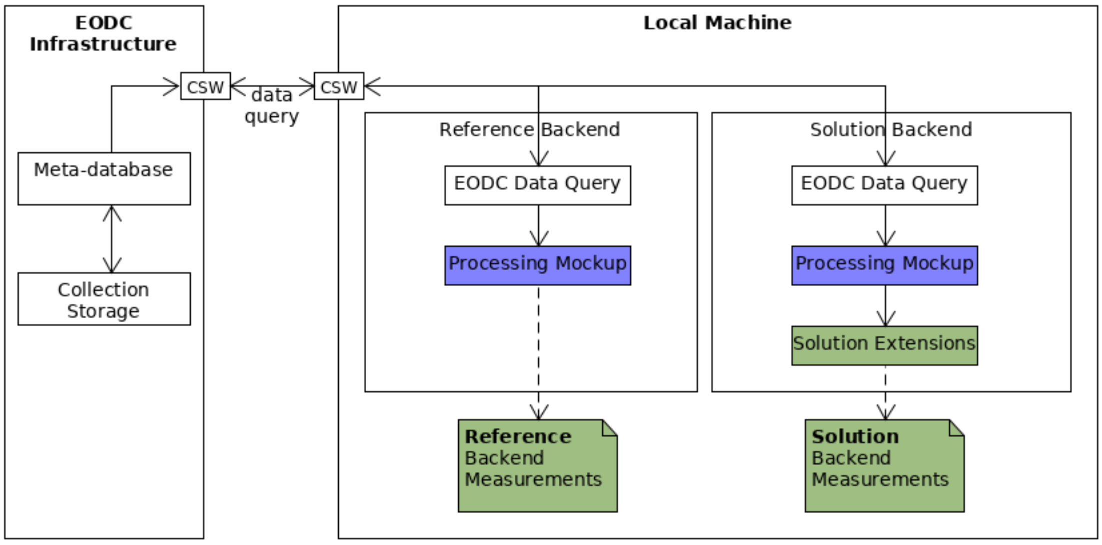


Figure 5.4: Overview of the storage and performance evaluation setup.

### 5.4.1 Performance

In this section, we evaluate the performance impact on the backend by measuring the difference of job execution durations between the *Reference Backend* and the *Solution Backend*. We measure the execution time by writing timestamps into a log file and calculating the duration afterward. We measure the duration of the *Query Handler* (see Section 4.1) execution, as well as the duration of the context model creation process (described in Section 4.3). Each test case is executed 50 times at each backend. Before the test case execution, we cleanup the backend, so that every iteration has the same backend condition.

Table 5.6: List of geoscientific papers used for the input data of the impact evaluation in Section 5.1

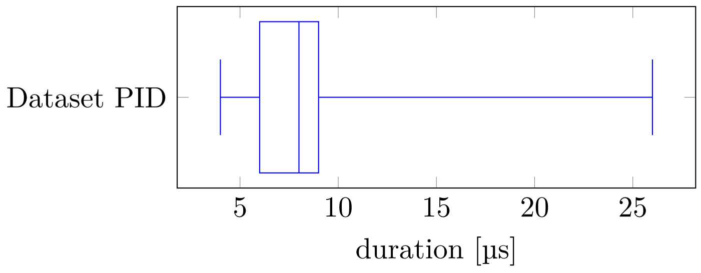
| Test Cases | DOI | Citation |
|---|---|---|
| 1 | 10.1080/01431160902887339 | [3] |
| 2-4 | 10.3390/rs8050402 | [37] |
| 5,6 | 10.1016/j.jag.2014.12.001 | [39] |
| 7 | 10.1016/j.jag.2016.12.003 | [38] |
| 8 | 10.3390/rs8110938 | [47] |
| 9 | 10.1080/2150704X.2016.1225172 | [25] |
| 10-13 | 10.1109/TGRS.2018.2858004 | [1] |
| 14 | 10.1080/01431161.2018.1479788 | [5] |
| 15-18 | 10.3390/rs10071030 | [2] |

### Performance of Query Handler

This section provides a description regarding performance constraints for each element of the query table.

- **Dataset PID**
  We take the *Dataset PID* directly from the parsed filter arguments of the backend. Therefore, the duration has a complexity of $O(1)$. Figure 5.5 shows the performance results, which are between 4 µs and 26 µs with a median of 8 µs. the standard deviation over all test cases is 2.5 µs. The results show that the duration time of retrieving the dataset PID is, except for a few aberrations, similar between all test case executions and independent of the job configuration.

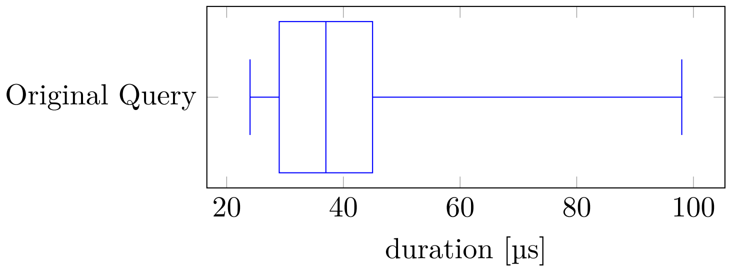
Figure 5.5: Boxplot of the duration needed in the test cases to handle the *Dataset PID* entry.



- **Original Query**
  The *Query Execution* component passes the *Original Query*. The *Query Handler* transforms the query into a string. The query string has the same size for each query, and just the argument values are exchanged. Therefore the duration has a complexity of $O(1)$. The execution time of the test cases is between 24 µs and 98 µs,

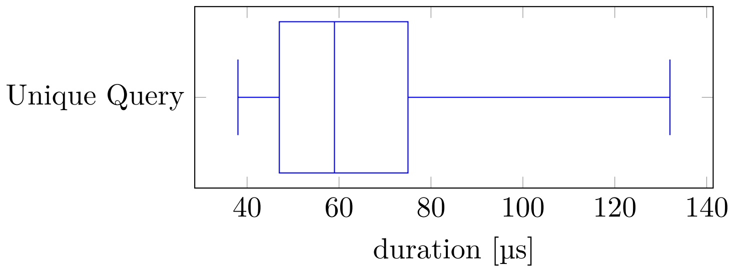
with a median of 37 µs. The standard deviation for all test cases is 5.34 µs. Figure 5.6 shows the boxplot of all test case executions and therefore the distribution of duration. The measurements indicate that most of the *Original Query* retrieval time is in a small range. The duration of retrieving the original query is constant in time and independent on the job configuration.

Figure 5.6: Duration boxplot of the test cases to handle the *Original Query* entry.



- **Unique Query**
  The *Unique Query* is the result of an alphabetical sorting of the parsed filter arguments of the backend. It takes nearly constant duration time, since there are only 4 filter arguments allowed in the openEO API version 0.3.1, but they may not all be used. The sorting algorithm has a complexity of $O(n \log n)$, where "n" is the number of crucial elements in the dictionary (amount of filter operations). In this evaluation, four filter arguments were used for each test case. Therefore it has a constant complexity in every test case execution. It takes between 38 µs and 132 µs with a median of 59 µs of duration time with a standard deviation of 19.35 µs. Figure 5.7 shows the boxplot of all test case executions to visualize that the duration is not widely distributed.

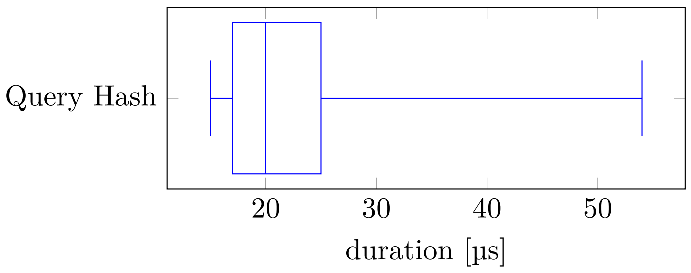
Figure 5.7: Boxplot of the duration time of the test cases to handle the *Unique Query* entry.



- **Query Hash**
  The performance of the unique *Query Hash* is dependent on the size of the unique query ($O(n)$, where "n" is the size of the unique query string). In this evaluation, it

is constant because every test case uses four filter arguments. The circumstances makes the unique query have a constant length. In the test cases, the duration is between 15 µs and 54 µs with a median of 20 µs and a standard deviation of 5.89 µs. Figure 5.8 shows the boxplot of all test case executions. The duration has a small range of distribution, except for a few executions. The measurements show that the duration is independent of the job configuration.

Figure 5.8: Boxplot of the duration time of the test cases to handle the *Query Hash* entry.



- **Result Hash**
  Duration of the *Result Hash* creation is dependent on the length of the result file list. The SHA-256 operation has a complexity of $O(n)$, where "n" is the length of the resulting file list string. In the test cases, the duration time of the *Result Hash* calculation is between 28 µs and 9167 µs with a median of 51 µs. Figure 5.9 shows the duration of the *Result Hash* creation of the test cases, sorted by the size of the result file set in an ascending way. Table 5.7 shows the values of the chart. They indicate the relationship between the number of files and the duration.

- **Metadata**
  In the solution the *Metadata* contains only the number of result files. It measures it by the built-in len() operator of Python, which has a complexity of $O(1)$ according to the official Python description[6]. In the test cases, the data calculation takes a duration between 6 µs and 359 µs with a median of 12 µs and a standard deviation of 3.98 µs. Figure 5.10 shows the boxplot of the test case execution. The measurements indicate no correlation of the test case configuration and the duration.

- **Database operations**
  To ensure that there are no duplicate query entries in the database, the *Query Handler* executes a SQL SELECT statement to check if there is already an entry with the same unique query hash and the same result hash. In the evaluation, the query table is empty before the SELECT statement. Hence, the complexity is $O(1)$. There is currently no statistic on query executions at EODC, but an estimation of a few hundred per month. The INSERT statement to store the

[6]https://wiki.python.org/moin/TimeComplexity

Table 5.7: *Result Hash* performance of the test cases depending on the number of result files.

| Test Case | Number of files | *Result Hash* duration [µs] |
|---|---|---|
| 1 | 14 | 47.2 |
| 2 | 24 | 78.0 |
| 3 | 11 | 37.2 |
| 4 | 9 | 32.4 |
| 5 | 10 | 35.0 |
| 6 | 10 | 35.9 |
| 7 | 10 | 33.8 |
| 8 | 12 | 36.9 |
| 9 | 2255 | 8698.7 |
| 10 | 17 | 59.4 |
| 11 | 1551 | 5343.9 |
| 12 | 12 | 41.5 |
| 13 | 28 | 95.2 |
| 14 | 420 | 1400.7 |
| 15 | 15 | 50.1 |
| 16 | 1356 | 4985.0 |
| 17 | 15 | 51.6 |
| 18 | 54 | 187.2 |

query has a complexity of $O(1)$. The database operations in the test cases have a duration between 10.501 ms and 49.283 ms with a median of 13.377 ms. Figure 5.11 shows the boxplot of the test case executions indicating that the performance is independent of the job configuration.

Figure 5.12 shows the duration of the constant query elements of the test cases. The test cases are sorted by the result size in ascending order. It shows that even though we configured the test cases differently, the duration time of the constant query elements are similar between them.

### Performance of Context Model Creation

1. **Constant context model elements**
   The creation of the following elements of the context model have a constant duration:

   - The programming language
   - The input data identifier
   - The backend version
   - The start and end timestamp

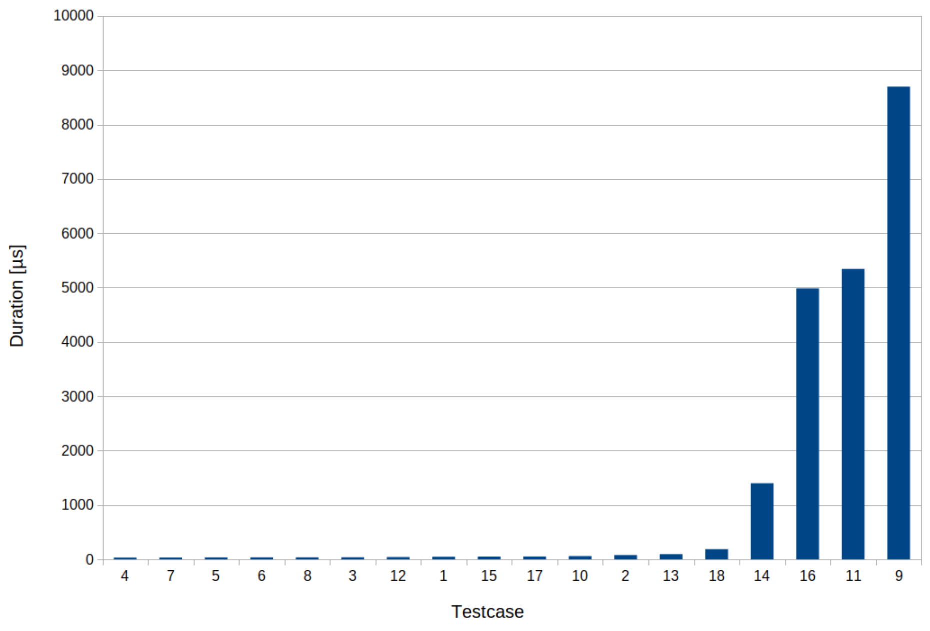


Figure 5.9: *Result Hash* duration of the test cases sorted by result size.

Figure 5.10: Boxplot of the duration time of the test cases to handle the *Metadata* entry.

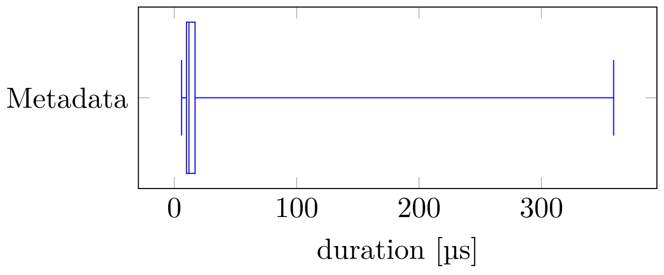
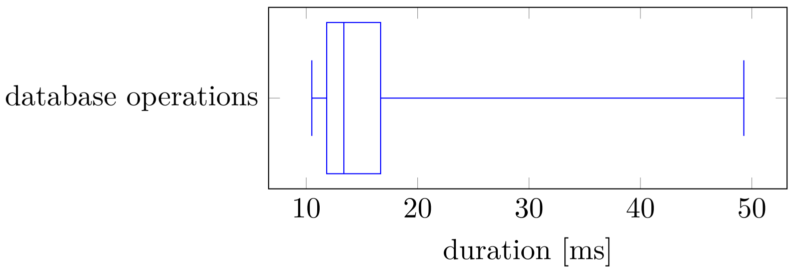


Figure 5.11: Boxplot of the duration time of the test cases to make the needed database operations.



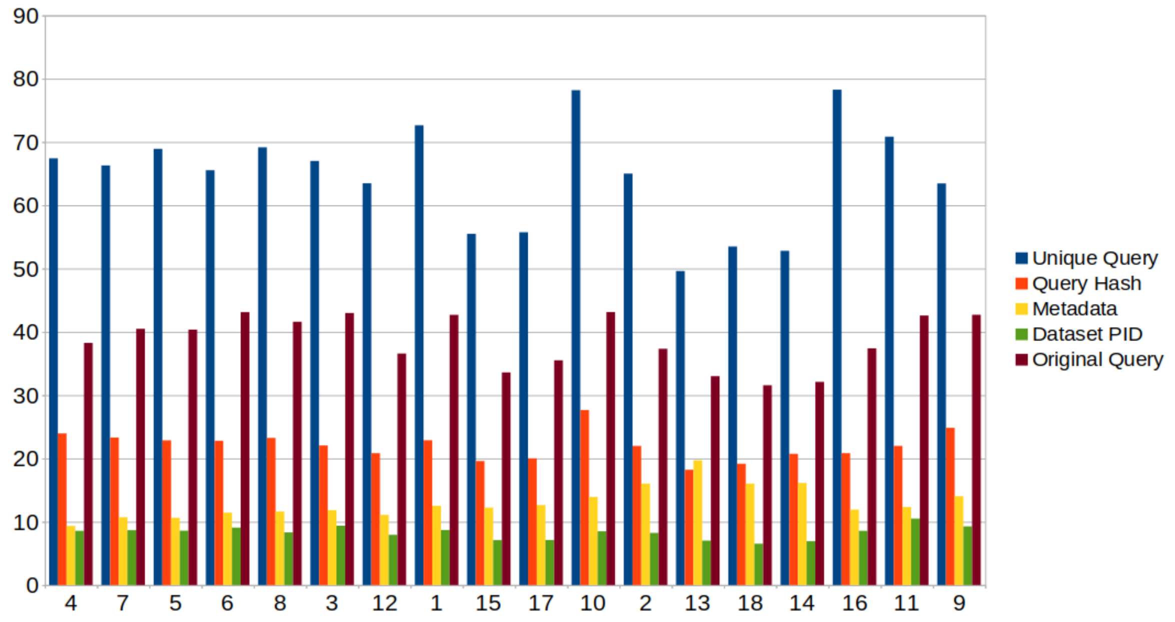
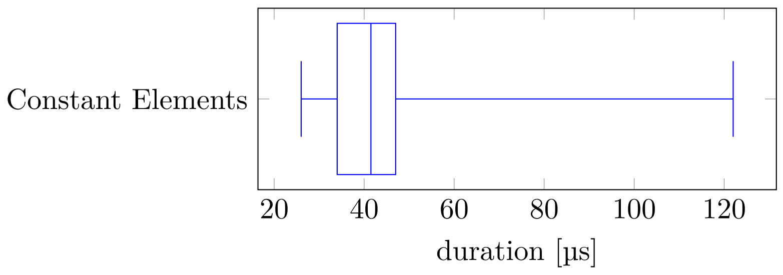


Figure 5.12: Constant elements of the *Query Handler* output in relation to the test cases sorted by result size.

These are independent of the job configuration and are read operations with a complexity of $O(1)$. The constant context model elements in the test cases have a duration between 26 µs and 122 µs with a median of 41.5 µs. Figure 5.13 shows the boxplot of the test case executions to visualize the distribution.

Figure 5.13: Boxplot of the duration time of the test cases to handle the constant context model elements.



2. **Dependencies of the programming language**
   A "pip freeze" execution retrieves the dependencies of the backend. It is independent on the complexity of the job, hence has a constant duration time. The duration in the test cases executions is between 92 µs and 289 µs with a median of 133 µs. Figure 5.14 shows the boxplot of the test case execution to visualize the distribution.

Figure 5.14: Boxplot of the duration time of the test cases to retrieve the modules of Python.

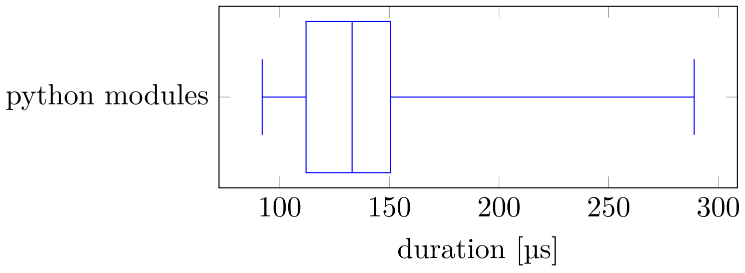


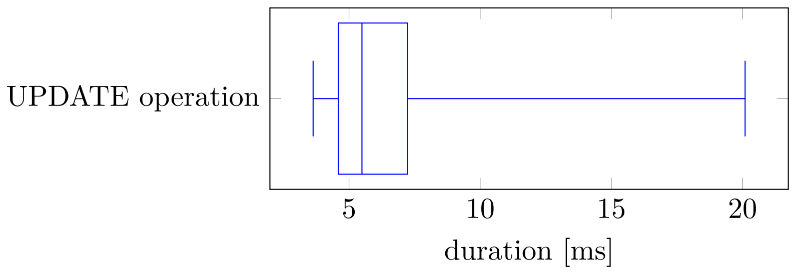
Table 5.8: Result hash performance of the test cases in relation to the result size.

| Test Case | Result size [kByte] | Result hash duration [ms] |
|---|---|---|
| 1 | 587 | 2.2 |
| 2 | 721 | 2.7 |
| 3 | 600 | 2.2 |
| 4 | 556 | 2.1 |
| 5 | 570 | 2.1 |
| 6 | 553 | 2.1 |
| 7 | 578 | 2.1 |
| 8 | 638 | 2.4 |
| 9 | 27 999 | 104.2 |
| 10 | 621 | 2.3 |
| 11 | 7 969 | 29.7 |
| 12 | 649 | 2.4 |
| 13 | 1 376 | 5.1 |
| 14 | 6 026 | 22.4 |
| 15 | 777 | 2.9 |
| 16 | 7 978 | 29.7 |
| 17 | 733 | 2.7 |
| 18 | 706 | 2.6 |

3. **Result hash**
The duration of the resulting hash calculation is dependent on the length of the resulting image. The SHA-256 operation has a complexity of $O(n)$, where "n" is the size of the output image. In the test cases, the duration is between 1.924 ms and 106.270 ms with a median of 2.521 ms. Table 5.8 shows the test cases and their result size concerning the average duration time of the result hash calculation. In the experiment setup, every 100 kByte of output data resulted in average in an additional duration of 371 µs. The test cases 9, 11, 14, and 16 have the most prominent result file and therefore needed the longest for the result hash calculation.

4. **Database UPDATE operation**
   The job entry gets updated to store the context model in the database by a SQL UPDATE statement. It is independent of the job configuration. The duration of the test case execution is between 3.631 ms and 20.099 ms, with a median of 5.493 ms. Figure 5.15 shows the boxplot of the test case execution to visualize the distribution.

Figure 5.15: Boxplot of the duration time of the test cases to retrieve the modules of Python.



The context model creation performance is not affected by the complexity of the test cases. The captured data for the context model is not related to the complexity of the job execution. The "Increase" column of Table 5.9 shows that simple test cases are affected the most in terms of relative performance loss because the *Query Handler* and the context model have elements that need the same execution duration. In conclusion, it can be argued that the execution of the *Query Handler* and the context model creation is less affecting complex test cases, because of the constant overhead, which adds a fixed amount of time. Besides, it should be mentioned that the *Query Handler* and the context model creation happens after the processing. So that the result files of the job execution are available for the user even if both modules have not finished yet.
Table 5.9 summarizes the result of the mean duration time over the 50 runs of each the test case. The second column presents the duration of the *Reference Backend*. The other columns are measurements of the *Solution Backend*. It includes the total execution duration of the *Solution Backend*, the duration of the *Query Handler* and the duration of the context model creation. The last column shows the additional time the *Solution Backend* needs in comparison to the *Reference Backend*. The last row of Table 5.9 shows the mean duration over all test cases. It indicates that the solution adds between 20 ms and 175 ms to the duration of the *Reference Backend*. It depends on the result sizes and not on the execution duration of the 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.

Table 5.9: Mean duration time over 50 runs of the *Solution Backend* and the *Reference Backend* by executing the test cases

| | **Reference Backend** | **Solution Backend** | | | |
|---|---|---|---|---|---|
| **Test Case** | **Total** | **Comparison Total** | **Query Handler** | **Context Model** | **Solution Addition** |
| 1 | 322.946 ms | 345.127 ms | 14.187 ms | 7.994 ms | 22.181 ms (6.8 %) |
| 2 | 369.066 ms | 393.505 ms | 16.342 ms | 8.097 ms | 24.439 ms (6.6 %) |
| 3 | 281.657 ms | 305.407 ms | 15.669 ms | 8.081 ms | 23.750 ms (8.4 %) |
| 4 | 276.324 ms | 298.954 ms | 14.015 ms | 8.615 ms | 22.630 ms (8.2 %) |
| 5 | 312.150 ms | 334.802 ms | 13.925 ms | 8.727 ms | 22.652 ms (7.3 %) |
| 6 | 314.571 ms | 337.290 ms | 13.985 ms | 8.734 ms | 22.719 ms (7.2 %) |
| 7 | 320.081 ms | 343.552 ms | 14.555 ms | 8.916 ms | 23.471 ms (7.3 %) |
| 8 | 304.998 ms | 328.633 ms | 14.742 ms | 8.893 ms | 23.635 ms (7.7 %) |
| 9 | 565.289 ms | 740.751 ms | 48.766 ms | 126.696 ms | 175.462 ms (31.0 %) |
| 10 | 401.922 ms | 425.026 ms | 14.874 ms | 8.230 ms | 23.104 ms (5.7 %) |
| 11 | 521.022 ms | 605.185 ms | 34.660 ms | 49.503 ms | 84.163 ms (16.2 %) |
| 12 | 387.536 ms | 412.079 ms | 15.711 ms | 8.832 ms | 24.543 ms (6.3 %) |
| 13 | 510.517 ms | 538.784 ms | 17.070 ms | 11.197 ms | 28.267 ms (5.5 %) |
| 14 | 657.989 ms | 706.329 ms | 19.010 ms | 29.330 ms | 48.340 ms (7.3 %) |
| 15 | 345.806 ms | 371.984 ms | 17.027 ms | 9.151 ms | 26.178 ms (7.6 %) |
| 16 | 585.730 ms | 658.493 ms | 23.956 ms | 48.807 ms | 72.763 ms (12.4 %) |
| 17 | 563.755 ms | 589.776 ms | 16.778 ms | 9.243 ms | 26.021 ms (4.6 %) |
| 18 | 836.377 ms | 862.271 ms | 16.801 ms | 9.093 ms | 25.894 ms (3.1 %) |
| **Avg.** | **437.652 ms** | **477.664 ms** | **19.004 ms** | **21.008 ms** | **40.012 ms (8.9 %)** |

### Storage

This section describes the storage needed for the captured data. The *Solution Backend* stores all of the captured data in a PostgreSQL database. Therefore, we estimate the needed storage using the PostgreSQL command line interface. Listing 5.19 presents the commands to retrieve the size of the database entries. The id of the data record is inserted into "”" (e.g. job id in the first one).

The storage need for the evaluation is constant. The mean storage of the 50 runs per test case and the average storage size of the test cases are measured. Three parts of the implementation are storing data in the database. First, we store the context model in an additional column named "context_model" in the job table. There is no element in the context model that can vary in size, except for the list of packages of Python, which didn't change during the evaluation. It takes an additional 1.043 kByte of size for every job entry. The same occurs at the queryjob table, which maps the query table and the job table and therefore contains the needed identifiers and timestamps for creation and modification. Every record of the queryjob table needs 0.113 kBytes of storage space. The query records have a varying size because of the string length of the parameters

```
-- Context Model
SELECT sum(pg_column_size(context_model)) as filesize, count(*) as filerow
FROM jobs as t WHERE id = '{}';
-- Query Table
SELECT sum(pg_column_size(t)) as filesize, count(*) as filerow
FROM query as t WHERE query_pid = '{}';
-- QueryJob Table
SELECT sum(pg_column_size(t)) as filesize, count(*) as filerow
FROM queryjob as t WHERE job_id = '{}';
```

Listing 5.19: PostgreSQL commands to retrieve the size of one data record in the job table, the query table and the queryjob table.

in the original query. The remaining parts of the query table are constant in storage usage. Each query record of the test cases needs between 1.520 kByte and 1.533 kByte of space. The original query makes up the biggest part in the query table (e.g. 959 Bytes of 1521.432 Bytes in test case 18). The reason for this is the high amount of XML annotations. In summary, the average additional storage needed by the *Solution Backend* per job with a new query entry is 2.677 kBytes. If the used query is already in the query table, only additional 1.043 kBytes are needed by the *Solution Backend*. For EODC this are unproblematic numbers with the estimate of under five hundred queries and job executions per month (maximum additional 1.338 MBytes per month).

## 5.5 Summary

The evaluation in this chapter showed how the solution tackles the goals of the research questions. First, it summarizes how the RDA recommendations are implemented in the solution, except for the migration recommendations (R14 and R15). The data identification implementation is then tested against exceptional test cases regarding data updates and data deletions at the backend. The evaluation shows that the solution can re-execute queries properly by returning old versions of updated files. The test cases show 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 at EODC. The reason for this is that 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. The test case on job capturing showed that the solution is capable of identifying the backend version and therefore, the environment of the job execution. Still, to run a new job in the same environment, EODC has to provide it manually. The evaluation also contains a section about the performance and storage impact on the EODC 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 of 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 resulting hash of the context model is dependent on the size of the output file of the job execution. The evaluation of storage impact results in the conclusion that the additional needed space per job is unrelated to the job configuration. It depends on if there is a new query entry added to the database or not. The performance impact of the test case execution 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 the Query Handler is negligible. The additional needed space per job is constant and also minimal if compared to size of data kept at the backend.

CHAPTER 6

# Conclusion and Future Work

## 6.1 Conclusion

In this thesis, the challenges of providing reproducibility in earth observation science have been explored. We dealt with the problematic state of reproducibility in EO experiments, which use extern computational backends. We suggested how existing infrastructure of EO backends can be adapted to enable data identification using the Research Data Alliance recommendations. We presented how job environments can be captured and compared, by applying the VFramework to make differences in the computational environment measurable. The presented solution was implemented at the backend of the Earth Observation Data Centre which is compliant with the openEO specification that provides a common interface for earth observation backends. The implementation involved extending the backend to add reproducibility supporting functionality, as well as, extending client applications to provide additional functionality. This allows scientists not to change the way of work but, due to the introduced changes into the existing environment, they can improve the reproducibility of experiments. The evaluation of our solution consists of 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 concluded that the solution is capable of making the input data, code and the environment identifiable and reproducible with minimal impact on the backend's performance.

### 6.1.1 Research questions revisited

This section revisits the research questions defined in Section 1.2 to discuss how the solution suits them.

- **What information must be captured from an earth observation backend, so that a job execution can be repeated like the original execution?**
  - **How can the data of the original execution be identified?**
    The data of an EO job execution is defined by a list of satellite images needed to execute the job. It it is specified by the satellite identifier and the filter operations of the job, which result in a backend specific query (e.g. CSW). We made it identifiable by storing the original query following the RDA recommendations. Therefore, we assigned a persistent identifier to every unique query and result combination. We defined the result as the hash value of the resulting file list of the query execution. The landing page of the resolvable input data PID is a human-readable and machine-readable website. We extended the openEO API specification to access the information of the landing page using an openEO client application.
  - **How can the environment of the original execution be reproduced?**
    The original environment consists of the executed code, the version of programming language, and the installed libraries. We use Git and GitHub in our implementation to identify a version of the backend. Each commit represents a version of code. The version of the programming language as well as the installed libraries are captured during the execution of the job delusing. In our implementation we used the Python tool pip. The context model stores the three elements of the environment during the job execution. The backend has to jump back to the version of the backend used during the job execution to reproduce the original execution environment.
  - **Which parts of the backend need to be extended?**
    The RDA recommendations have to be implemented at the query execution component of the backend. To realize the backend version, we recommend to use a version control system. The job execution environment has to be modified to capture the job dependent environment. After the execution of the job finished, we added the creation of a result hash. We suggest an additional database table to store the context model for every job.
  - **How can the result of a re-execution in future environments be verified?**
    The solution contains a hash of the resulting output file. The user compares the result hash value of the re-execution with the original execution output hash to see if the result differs. If the result hash differs in a re-execution, the user is able to investigate which parts of the job environment may caused the inequality.

- **What information must be persisted to enable validation of a job re-execution on an earth observation backend?**
  - **What are the validation requirements?**
    We defined the validation requirements as the equality of the captured data in the context model. We proposed the equality of the input data identifier, the execution environment and the hash of the output for validation,because they define the job execution comprehensively. If any of these parts are different in a re-execution, we conclude that the way the execution happened is different from the original one.
  - **How can differences in the environment between the executions be discovered?**
    The solution contains a user service to compare two job executions. It compares the context models of the two jobs and returns the differences. If a backend environment changes between two executions, the backend version is different.

## 6.2 Future Work

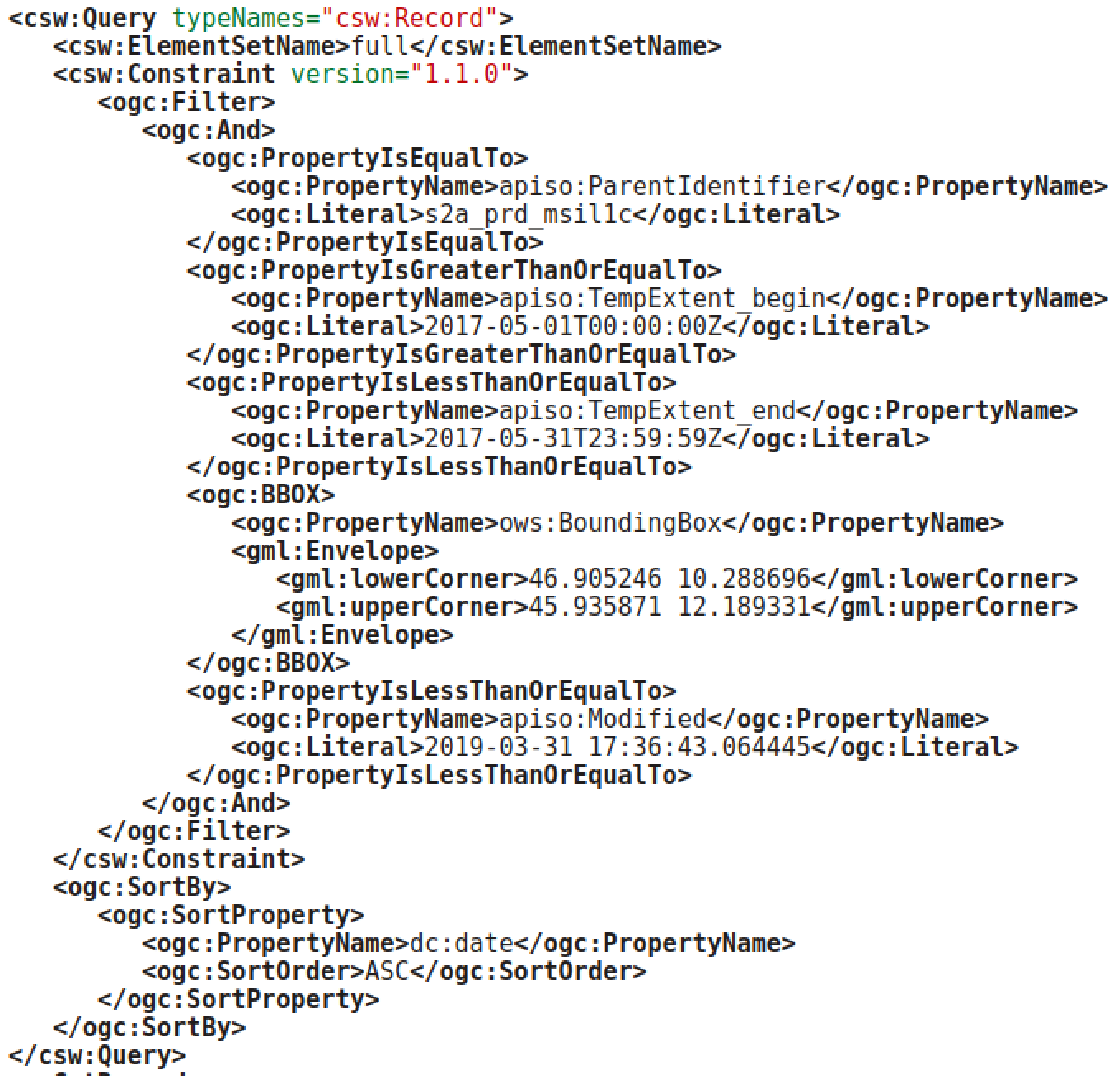
The prototype of this thesis applies reproducibility using a file-based openEO compliant backend. The openEO project is an ongoing project, hence the common API may evolve and our work will have to be adapted to new releases of the API. User defined functions (UDF) are part of the project, but not well defined yet. The extension of the design to support reproducible UDFs is a useful extension of the solution. Future work will lead to the implementation of the proposed solution at other backends of the openEO consortium. Further, the solution will be applied to backends that are not openEO compliant. Future effort will go into differently structured backends e.g. backends with non-file-based result sets. In the solution of this thesis, we capture environment information to identify how jobs are executed. Jobs are represented by the process graph and therefore, a compilation of single processes. Future work will improve the provenance of process implementations at the backend by introducing process versions. This will improve the transparency for users by enabling the identification of updated process implementations that may cause different results in a re-execution.

CHAPTER 7

# Appendix

```xml
<csw:Query typeNames="csw:Record">
   <csw:ElementSetName>full</csw:ElementSetName>
   <csw:Constraint version="1.1.0">
      <ogc:Filter>
         <ogc:And>
            <ogc:PropertyIsEqualTo>
               <ogc:PropertyName>apiso:ParentIdentifier</ogc:PropertyName>
               <ogc:Literal>s2a_prd_msil1c</ogc:Literal>
            </ogc:PropertyIsEqualTo>
            <ogc:PropertyIsGreaterThanOrEqualTo>
               <ogc:PropertyName>apiso:TempExtent_begin</ogc:PropertyName>
               <ogc:Literal>2017-05-01T00:00:00Z</ogc:Literal>
            </ogc:PropertyIsGreaterThanOrEqualTo>
            <ogc:PropertyIsLessThanOrEqualTo>
               <ogc:PropertyName>apiso:TempExtent_end</ogc:PropertyName>
               <ogc:Literal>2017-05-31T23:59:59Z</ogc:Literal>
            </ogc:PropertyIsLessThanOrEqualTo>
            <ogc:BBOX>
               <ogc:PropertyName>ows:BoundingBox</ogc:PropertyName>
               <gml:Envelope>
                  <gml:lowerCorner>46.905246 10.288696</gml:lowerCorner>
                  <gml:upperCorner>45.935871 12.189331</gml:upperCorner>
               </gml:Envelope>
            </ogc:BBOX>
            <ogc:PropertyIsLessThanOrEqualTo>
               <ogc:PropertyName>apiso:Modified</ogc:PropertyName>
               <ogc:Literal>2019-03-31 18:01:47.695042</ogc:Literal>
            </ogc:PropertyIsLessThanOrEqualTo>
         </ogc:And>
      </ogc:Filter>
   </csw:Constraint>
   <ogc:SortBy>
      <ogc:SortProperty>
         <ogc:PropertyName>dc:date</ogc:PropertyName>
         <ogc:SortOrder>ASC</ogc:SortOrder>
      </ogc:SortProperty>
   </ogc:SortBy>
</csw:Query>
```

Figure 7.1: Original query of *jobB* from the evaluation in Section 5.2.



```
<csw:Query typeNames="csw:Record">
   <csw:ElementSetName>full</csw:ElementSetName>
   <csw:Constraint version="1.1.0">
      <ogc:Filter>
         <ogc:And>
            <ogc:PropertyIsEqualTo>
               <ogc:PropertyName>apiso:ParentIdentifier</ogc:PropertyName>
               <ogc:Literal>s2a_prd_msil1c</ogc:Literal>
            </ogc:PropertyIsEqualTo>
            <ogc:PropertyIsGreaterThanOrEqualTo>
               <ogc:PropertyName>apiso:TempExtent_begin</ogc:PropertyName>
               <ogc:Literal>2017-05-01T00:00:00Z</ogc:Literal>
            </ogc:PropertyIsGreaterThanOrEqualTo>
            <ogc:PropertyIsLessThanOrEqualTo>
               <ogc:PropertyName>apiso:TempExtent_end</ogc:PropertyName>
               <ogc:Literal>2017-05-31T23:59:59Z</ogc:Literal>
            </ogc:PropertyIsLessThanOrEqualTo>
            <ogc:BBOX>
               <ogc:PropertyName>ows:BoundingBox</ogc:PropertyName>
               <gml:Envelope>
                  <gml:lowerCorner>46.905246 10.288696</gml:lowerCorner>
                  <gml:upperCorner>45.935871 12.189331</gml:upperCorner>
               </gml:Envelope>
            </ogc:BBOX>
            <ogc:PropertyIsLessThanOrEqualTo>
               <ogc:PropertyName>apiso:Modified</ogc:PropertyName>
               <ogc:Literal>2019-03-31 17:36:43.064445</ogc:Literal>
            </ogc:PropertyIsLessThanOrEqualTo>
         </ogc:And>
      </ogc:Filter>
   </csw:Constraint>
   <ogc:SortBy>
      <ogc:SortProperty>
         <ogc:PropertyName>dc:date</ogc:PropertyName>
         <ogc:SortOrder>ASC</ogc:SortOrder>
      </ogc:SortProperty>
   </ogc:SortBy>
</csw:Query>
```

Figure 7.2: Original query of *jobD* from the evaluation in Section 5.2.

# List of Figures

# List of Tables

# List of Listings

## Glossary

|  |  |
|---|---|
| **backend** | A web service provider, capable of processing and providing geoscientific data. |
| **experiment** | Practical scientific earth observation research using at least one backend with at least one job. |
| **job** | Definition of a workflow execution on a backend, including the input data. |
| **process** | Algorithm that gets executed over earth observation data. May use the output of another process as input data. |
| **process graph** | openEO definition of a job in a JSON format. |
| **workflow** | Step by step description of an experiment. |

## Acronyms

|  |  |
|---|---|
| **API** | Application Programming Interface. |
| **CCCA** | Climate Change Centre Austria. |
| **CLI** | Command Line Interface. |
| **CPU** | Central Processing Unit. |
| **CSW** | Catalogue Service for the Web. |
| **EO** | Earth Observation. |
| **EODC** | Earth Observation Data Centre. |
| **ESA** | European Space Agency. |
| **GEE** | Google Earth Engine. |
| **GEOBIA** | Geographic Object-Based Image Analysis. |
| **GeoTiff** | Georeferenced Tagged Image File Format. |
| **GPU** | Graphics Processing Unit. |
| **JSON** | JavaScript Object Notation. |
| **NDVI** | Normalized Difference Vegetation Index. |

|  |  |
|---|---|
| **NetCDF** | Network Common Data Form. |
| **OBIA** | Object-Based Image Analysis. |
| **OGC** | Open Geospatial Consortium. |
| **openEO** | Open Source Earth Observation Project. |
| **OS** | Operating System. |
| **PID** | Persistent Identifier. |
| **RAM** | Random Access Memory. |
| **RDA** | Research Data Alliance. |
| **REST** | Representational State Transfer. |
| **RR** | Reproducible Research. |
| **SHA** | Secure Hash Algorithm. |
| **SOA** | Service Oriented Architecture. |
| **TDS** | Thredds Data Server. |
| **UDF** | User Defined Functions. |
| **VCS** | Version Control Systems. |
| **VZJ** | Vadose Zone Journal. |
| **WGDC** | Working Group on Data Citation. |
| **XML** | Extensible Markup Language. |

# Bibliography

[1] 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*, 57(1):520–539, 2019.

[2] B. Bauer-Marschallinger, C. Paulik, S. Hochstöger, 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*, 10(7), 2018.

[3] 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*, 9(7):3106–3121, 2016.

[4] F. Chirigati, R. Rampin, D. Shasha, and J. Freire. Reprozip: Computational reproducibility with ease. In *SIGMOD 2016 - Proceedings of the 2016 International Conference on Management of Data*, volume 26-June-2016, pages 2085–2088. Association for Computing Machinery, 2016.

[5] A. Dostálová, W. Wagner, M. Milenković, and M. Hollaus. Annual seasonality in sentinel-1 signal for forest mapping and forest type classification. *International Journal of Remote Sensing*, 39(21):7738–7760, 2018.

[6] C. Drummond. Replicability is not reproducibility: Nor is it good science. *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, 2009.

[7] I. Emsley and D. De Roure. A framework for the preservation of a docker container. *12th International Digital Curation Conference (IDCC17)*, 2017.

[8] N. Fuhr. The primad model of reproducibility. Dagstuhl Seminar 16111 - Rethinking Experimental Methods in Computing, 2016.

[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*, 3(10):388–415, 2016.

[10] 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*, 7(6):1–13, 06 2012.

[11] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[12] D. Huo, J. Nabrzyski, and C. Vardeman. Smart Container: An Ontology Towards Conceptualizing Docker. In *Process of International Semantic Web Conference*, 2015.

[13] C. Knoth and D. Nust. Enabling reproducible obia with open-source software in docker containers. *GEOBIA 2016 : Solutions and Synergies, At University of Twente Faculty of Geo-Information and Earth Observation (ITC)*, 2016.

[14] C. Knoth and D. Nüst. Reproducibility and practical adoption of geobia with open-source software in docker containers. *Remote Sensing*, 9(3), 2017.

[15] 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*, 33(2):408–429, 2019.

[16] G. T. Konrad Hinsen, Konstantin Laeufer. Essential tools: Version control systems. *Computing in Science and Engineering*, 11(06):84–91, 2009.

[17] H. P. Langtangen. Quick introduction to git and github. `http://hplgit.github.io/teamods/bitgit/Langtangen_bitgit-solarized.html`. Accessed: 2018-08-10.

[18] J. R. Marlon, R. Kelly, A.-L. Daniau, B. Vannière, M. J. Power, P. Bartlein, P. Higuera, O. Blarquez, S. Brewer, T. Brücher, A. Feurdean, G. G. Romera, V. Iglesias, S. Y. Maezumi, B. Magi, C. J. Courtney Mustaphi, and T. Zhihai. Reconstructions of biomass burning from sediment-charcoal records to improve data–model comparisons. *Biogeosciences*, 13(11):3225–3244, 2016.

[19] 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*, volume 1, pages 153–160, Oct 2014.

[20] T. McPhillips, S. Bowers, K. Belhajjame, and B. Ludäscher. Retrospective provenance without a runtime provenance recorder. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, Edinburgh, Scotland, 2015. USENIX Association.

[21] T. Miksa, S. Pröll, R. Mayer, S. Strodl, R. Vieira, J. Barateiro, and A. Rauber. Framework for verification of preserved and redeployed processes. In *Proceedings of the 10th International Conference on Digital Preservation, iPRES 2013, Lisbon, Portugal, September 2 - 6, 2013*, 2013.

[22] T. Miksa and A. Rauber. Using ontologies for verification and validation of workflow-based experiments. *Web Semantics: Science, Services and Agents on the World Wide Web*, 43:25 – 45, 2017.

[23] T. Miksa, A. Rauber, and E. Mina. Identifying impact of software dependencies on replicability of biomedical workflows. *Journal of Biomedical Informatics*, 64:232 – 254, 2016.

[24] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. No workflow: Capturing and analyzing provenance of scripts. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8628, pages 71–83. Springer Verlag, 2015.

[25] 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*, 7(12):1209–1218, 2016.

[26] F. O. Ostermann and C. Granell. Advancing science with vgi: Reproducibility and replicability of recent studies using vgi. *Transactions in GIS*, 21:224–237, 2017.

[27] E. Pebesma, W. Wagner, M. Schramm, A. Von Beringe, C. Paulik, M. Neteler, J. Reiche, J. Verbesselt, J. Dries, E. Goor, T. Mistelbauer, C. Briese, C. Notarnicola, R. Monsorno, C. Marin, A. Jacob, P. Kempeneers, and P. Soille. OpenEO - a Common, Open Source Interface Between Earth Observation Data Infrastructures and Front- End Applications, 2017.

[28] J. a. F. Pimentel, J. Freire, L. Murta, and V. Braganholo. Fine-grained provenance collection over scripts through program slicing. In *Proceedings of the 6th International Workshop on Provenance and Annotation of Data and Processes - Volume 9672*, pages 199–203, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[29] J. F. Pimentel, S. Dey, T. McPhillips, K. Belhajjame, D. Koop, L. Murta, V. Braganholo, and B. Ludäscher. Yin and yang: Demonstrating complementary provenance from noworkflow & yesworkflow. In M. Mattoso and B. Glavic, editors, *Provenance and Annotation of Data and Processes*, pages 161–165, Cham, 2016. Springer International Publishing.

[30] J. F. Pimentel, J. Freire, V. Braganholo, and L. G. P. Murta. Tracking and analyzing the evolution of provenance from scripts. In *IPAW*, 2016.

[31] S. Pröell, R. Mayer, and A. Rauber. Data access and reproducibility in privacy sensitive escience domains. In *2015 IEEE 11th International Conference on e-Science*, pages 255–258, Aug 2015.

[32] 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)*, pages 444–445, Oct 2017.

[33] H. Ramapriyan, J. Moses, and R. Duerr. Preservation of data for earth system science - towards a content standard. In *Process of 2012 IEEE International Geoscience and Remote Sensing Symposium*, pages 5304–5307, 2012.

[34] A. Rauber, A. Asmi, D. V. Uytvanck, and S. Pröll. Identification of reproducible subsets for data citation, sharing and re-use. *TCDL Bulletin*, 12, 2016.

[35] A. Rauber, T. Miksa, R. Mayer, and S. Pröll. Repeatability and re-usability in scientific processes: Process context, data identification and verification. In *Data Analytics and Management in Data Intensive Domains» conference/RCDL*, 2015.

[36] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig. Ten simple rules for reproducible computational research. *PLOS Computational Biology*, 9(10):1–4, 2013.

[37] 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*, 8(5), 2016.

[38] 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*, 56:77 – 87, 2017.

[39] 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*, 38:15 – 24, 2015.

[40] C. Schubert. CCCA Data Centre: RDA Pilot Dynamic Data Citation for NetCDF files. `https://www.rd-alliance.org/system/files/documents/CCCA_DC_RDA_DynamicDCite_v3_inkl_manual.pdf`. Accessed: 2018-11-25.

[41] B. H. Schubert C. Handling continuous streams for meteorological mapping. In *Lecture Notes in Geoinformation and Cartography*, volume 8628. Springer Verlag, 2019.

[42] S. Schwichtenberg, C. Gerth, and G. Engels. From open api to semantic specifications and code adapters. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 484–491, 2017.

[43] T. Skaggs, M. Young, and J. Vrugt. Reproducible research in vadose zone sciences. *Vadose Zone Journal*, 14(10):vzj2015.06.0088, 2015.

[44] S. Thomsen. *Cryptographic Hash Functions*. PhD thesis, DTU Orbit, 2009.

[45] D. M. Timothy Lebo and S. Sahoo. *PROV-O: The PROV Ontology*. W3C Recommendation. World Wide Web Consortium, United States, 2013.

[46] J. Vitek and T. Kalibera. Repeatability, reproducibility and rigor in systems research. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 33–38, 2011.

[47] F. Vuolo, M. Żółtak, 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*, 8(11), 2016.

[48] C. M. Zwölf, N. Moreau, and M.-L. Dubernet. New model for datasets citation and extraction reproducibility in vamdc. *Journal of Molecular Spectroscopy*, 327:122 – 137, 2016. New Visions of Spectroscopic Databases, Volume II.