FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

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: Dr. Tomasz Miksa

Wien, 21. Dezember 2018

Bernhard Gößwein	Andreas Rauber

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

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: Dr. Tomasz Miksa

Vienna, 21st December, 2018

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, 21. Dezember 2018

Bernhard Gößwein

Acknowledgements

To my girlfriend 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 process 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 Tomas Miksa for always quickly replying on questions and providing me with constructive feedback.

Kurzfassung

Publikationen in den Bereichen der Geodäsie und Geoinformation sind meist, aus diversen Gründen, nicht reproduzierbar. Daher gab es in den letzten Jahren ein Umdenken in Bezug auf reproduzierbare Publikationen in den computergestützten Geowissenschaften. Eine Problematik dieser Forschungsbereiche ist die große Anzahl an unterschiedlichen Anbietern im Bereich der Datenbeschaffung und Datenverarbeitung. Daher startete die Europäische Union das Horizon 2020 Projekt OpenEO, welches eine standardisierte Abstraktionsschicht über den diversen Anbietern darstellt. Mit diesem Projekt bekommen Geowissenschaften die Möglichkeit einer Einheitlichen Strategie der Reproduzierbarkeit. Diese Arbeit beschäftigt sich mit dem Design eines Konzeptes zur Erlangung der Reproduzierbarkeit innerhalb des OpenEO Projektes. Herausforderungen liegen hierbei an den großen Daten der Anbieter, als auch der komplexen Prozessierungen. Dabei wird eine einheitliche Lösung für alle OpenEO Anbieter erstellt indem aktuelle Datenidentifikationsstrategien und Konzepte der Reproduzierbarkeit angewandt werden. Dabei soll der Aufwand sowohl der Anbieter als auch der Benutzer von OpenEO in Grenzen gehalten werden. Zusätzlich wird in dieser Arbeit ein vollwertiger Prototyp beim EODC Anbieter implementiert. Hinzu werden die notwendigen Erweiterungen in Absprache mit dem OpenEO Team gemacht um die Änderungen in eine zukünftige Version des OpenEO Standards zu bekommen. Schlussendlich wird die Implementierung bei EODC durch Szenarios, welche die Benutzung von OpenEO durch Wissenschaftler darstellen, evaluiert.

Abstract

Many areas of earth observation sciences lack on creating reproducible research papers. In the last years there was an extensive movement towards policies defining reproducibility for geoscientists. The diverse set of data providers and processing back ends are one reason for scientists not being able to reproduce experiments. The European Union has launched a Horizon 2020 project called OpenEO to create a unified abstraction layer above the processing back ends. This leads to the opportunity of a unified concept of reproducibility. The thesis provides a design to achieve this within the OpenEO project, by facing the challenges of big data and complex work-flows. In doing so a general concept for all back ends contributing to OpenEO is created. The design solution combines existing data identification strategies and work-flow capturing concepts. Part of the solution is limiting the effort for researchers using OpenEO . Additionally, we implement a complete system solution at the EODC back end and design suggestions to the OpenEO project to add the changes to a productive release of OpenEO. Finally, the implementation is evaluated by predefined scenarios of scientists using a back end that supports OpenEO for their experiments.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Over the last decades remote sensing agencies have increased the variations of data processing and therefore the amount of resulting data. To preserve the data for further usage in the future it is necessary to have citable data and processes on the data to ensure repeatability in a long-term.[31] Reproducibility is discussed in many science areas such as computer science, biology and also in computational geoscience. According to [27] where the reproducibility and replicability of scientific papers in geoscience got tested, only half of the publications were replicable and none of them reproducible. In [19] a survey of geoscientific readers and authors of papers got executed, with the goal of finding reasons for the lack of reproducibility in earth observation sciences. One of the results is that even though 49% of the participants responded that their publications are reproducible, only 12% of them have linked the used code. The understanding of open reproducible research is different among the participating scientists in a way that the interpretation is more related to repeatability and replicability. The main reasons for the lack of reproducibility in computational geoscience are, according to [19], insufficiently method descriptions, persistence of data identifier, legal concerns, the general impression that it is not necessary and too time consuming. Reproducing geoscientific papers failed by the different individual interpretation of the approach and alternative software versions that produce unequal results. The vast majority of issues on reproducing results are required changes of code and a deeper understanding of the procedures. In some cases there occur system dependent issues related to the usage of random access memory and installation libraries of the operating system. [19]

1.2 Problem Description

The vast majority of data used in earth observation sciences are retrieved and provided via Service Oriented Architecture (SOA) interfaces. Data Provider like Google Earth Engine[1] and EODC[2] host an Web Application Programming Interface (API) for data download and processing data. Due to a different range of functionality and a difference between the endpoints of the providers it is great afford to create a workflow for more than one provider. The OpenEO project has the goal to be an abstraction layer above different EO data providers. The underlying structure of OpenEO consists of three parts:

- **Client Module**: Is written in the program language of the user and transfers the users commands to the backends.
- **Core Module**: A standard on how the communication should take place between client and backend.
- **Backend Module**: The provider of the data and the services, which gets the instructions from the clients and returns the results.

Further information on the software architecture of the project is defined in the project proposal ([28]) and in section 3. On the creation time of this thesis there is no consideration of repeatability in the OpenEO architecture. Verification of work-flows for users of OpenEO is not in the agenda of the OpenEO project. Generalized layers have the opportunity to be implemented in a way that makes processes and data scientifically verifiable and reproducible, because it handles data and processes on the data in a standardized way for different providers. Even though the range of functionality and the API endpoints are well-defined in the OpenEO core-API, the contributing content providers (OpenEO back ends) will have different underlying software execution environments. The used technology of an OpenEO back end will evolve in the future, hence it can lead to different results on the same workflow execution. Considering the following: A scientist runs an experiment using OpenEO as research tool and gets an arbitrary result. The same scientist runs the same experiment with the same input data some months later and gets a slightly different result. The question occurs, why are the results distinct? Has the used data changed, has the user accidentally submitted different code or has some underlying software inside the back end provider changed. Adding a possibility for the users of OpenEO to gain this information is an important feature for the scientific community. The aim of this thesis is to design an extension to OpenEO, so that users are able to retrieve provenance data about a job re-execution. [28]

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

1.3 Aim of the Work

The expected outcome of this thesis is to discover and develop a possible framework for making repeatability conceivable in the OpenEO project. This enables users to re-execute work-flows and validate the result, so that differences on the process or data are visible for the users. To achieve this goal a model for capturing the provenance of the back ends within the project has to be discovered and implemented to evaluate the ability of the model. The solution shall then conclude recommendations for the OpenEO project on how to improve re-execution validation for the users and how it can be achieved. Considering the problem description and the recommendations above, the following research questions can be formulated:

- **How can an OpenEO job re-executed be applied like the initial execution?**
  - How can the used data be identified after the initial execution?
  - How can the used software of the initial execution be reproduced?
  - What data has to be captured when?
  - How can the result of a re-execution in future software versions be verified?
- **How can the equality of the OpenEO job re-execution results be validated?**
  - What are the validation requirements?
  - How can the data be compared?
  - How can changes of the OpenEO backend environment be recognised?
  - How can differences in the environment between the executions be discovered?

1.4 Use Cases

To validate the design proposed in this thesis, use cases are defined in the following sections. They describe scenarios about stakeholders of the OpenEO project that are currently not possible, but shall be accomplish-able with the solution of this thesis.

1.4.1 Re-Use of Input Data

The first use case is about the re-use of input data between job executions. Reproducible methods are especially important for the scientific community. Since scientists are likely to build on results or methods of publications, it is an opportunity for OpenEO to make the re-use simple. In this case a scientist creates a publication by using OpenEO and citing the method and data accordingly. Another scientist in a similar area of research wants to use the same input data but use a different approach of processing it. Hence

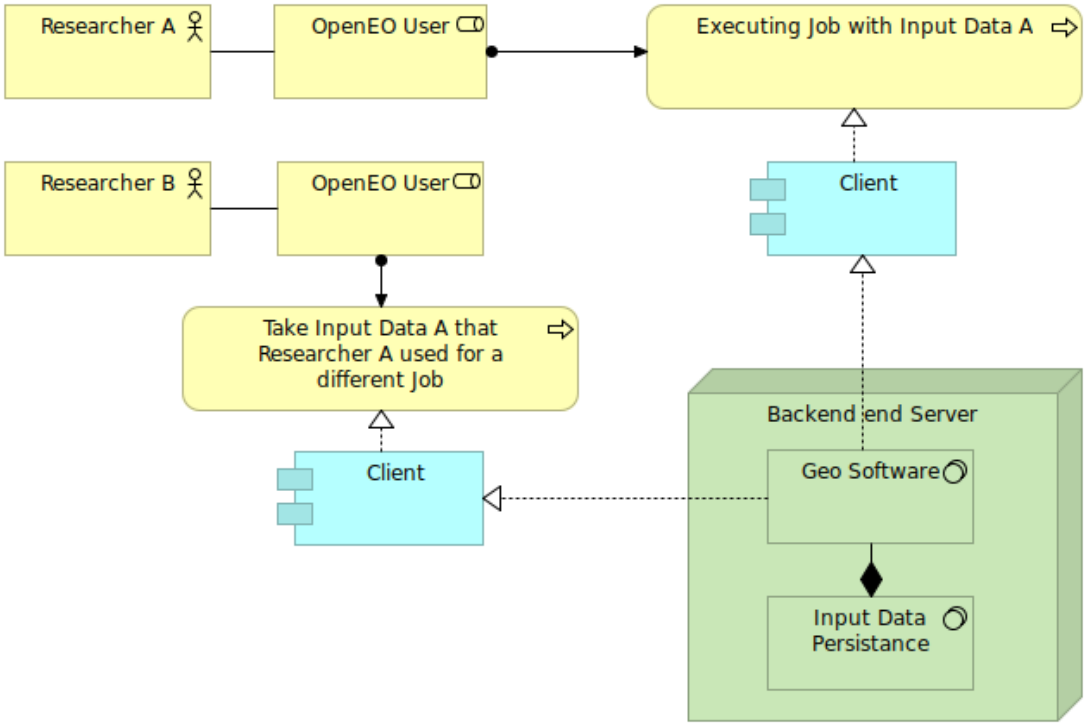

Figure 1.1: Overview of the first Use Case.

there needs to be a possibility to cite the input data in a way, that another user can re-use it in a different processing chain. An overview of the architecture of the use case can be viewed in figure 1.1.
The scenario can be summarized in the following steps:

1. Researcher A runs an experiment (job A) at a back end.
2. Researcher A retrieves the used input data of job A.
3. Researcher A cites the input data in a publication.
4. Researcher B uses the same input data of job A for job B.

1.4.2 Providing Job Execution Information

This use case is like the first one, but is only about job dependent environment information. The back end provider can automatically get provenance data about the job execution, like used software packages and their versions. The user gets additional information on a job execution. Motivation for this is to add transparency of the job processing for the

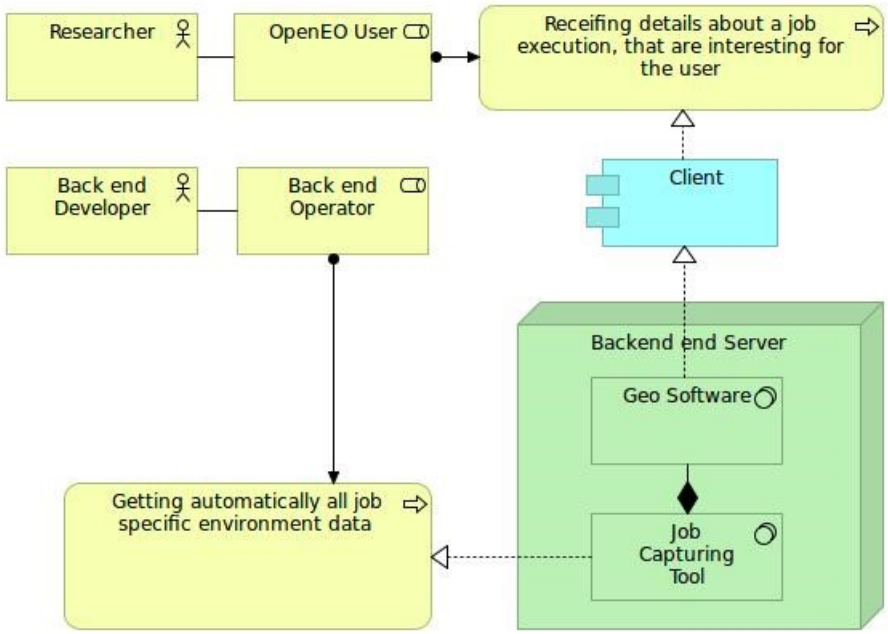

Figure 1.2: Overview of the second Use Case

users, so that researchers can describe their processes in more detail. It can also help OpenEO users to understand why results differ from executions in the past. An overview of the architecture of the use case can be viewed in figure 1.2. The following steps are part of this use case:

1. Researcher runs an experiment (job A) at a back end.
2. Researcher wants to describe the experiment environment.
3. Back End Developer releases a new version.
4. Back End Developer runs test jobs to find differences.

1.4.3 Compare different Job executions

The third use case is dedicated to the OpenEO users. The goal is to add a possibility for users of OpenEO to compare different jobs not only by their results, but also on the way they were executed. The comparison is between a job execution and another one on the same back end. Therefore, the processing and the input data has to be identifiable. To make the comparison interesting to the users there also has to be additional information. For usability the comparison needs to be easy to apply and easy to understand, therefore accessible in the environment of the user at the OpenEO client. In addition to the previous conditions a visualization of the differences for the users can lower the access

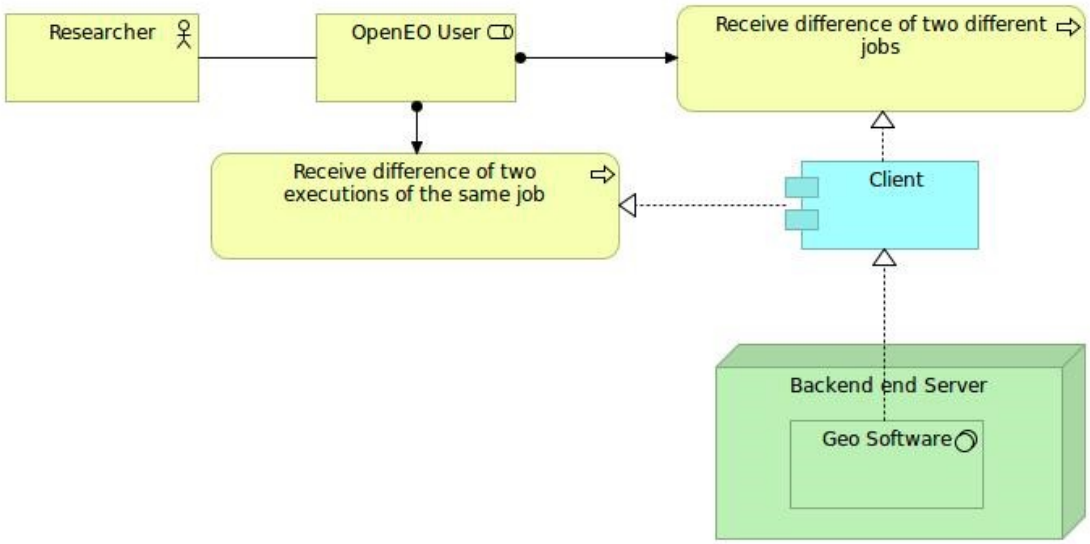

Figure 1.3: Overview of the third Use Case

barrier for users to use the feature. An overview of the architecture of this use case can be viewed in figure 1.3. The following steps describe the scenario:

1. Researcher runs an experiment (job A) at a back end.
2. Researcher re-runs the same experiment (job B).
3. Researcher runs a different experiment (job C).
4. Researcher wants to compare the jobs by their environment and outcome.

1.5 Methodological Approach

The use cases defined in the previous section require specific changes to the components of OpenEO. In the following list parts of the suggested solution are briefly described.

1. **Data Identification** In order to accomplish the capturing of processing workflows described in the use cases in section 1.4, the input data has to be identifiable. In order to achieve this the recommendations of the RDA (see [32]) have to be implemented by additional versioning and query databases.
2. **Process Versioning** To accomplish the verification of different process executions, the process has to be identifiable. Therefore, a versioning of the process code can be used to persist different states of the code. The thesis is for an OpenSource project and the code of the back ends are published at Github, so Git is used as the tool to capture the code state.

3. **Additional Information** In order to gain an description of interest for users of OpenEO some additional environment information about the data and the code gets persisted. This information is forwarded to the user and visualized.

4. **User Endpoints** The interface for the user has to be implemented so that the use cases can be executed by the users. Therefore, the OpenEO client and core API has to be modified. The adapted client in this thesis is the python client.

1.6 Structure of Work

The following sections are structured as follows:
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 describes the technologies and concepts that are used in the implementation of this thesis. This chapter also provides an overview of the EODC back end used for the implementation.
Chapter 4 provides the concept to address the research questions defined in section 1.3. This is the theoretical definition of what has to be implemented in the OpenEO project.
Chapter 5 has a detailed explanation to the proof of concept prototype implementation and how the concept of chapter 4 was realized for the EODC back end.
Chapter 6 dives into the evaluation of the implementation by applying the use cases to the implementation of chapter 5.
Chapter 7 summarizes the outcome of the implementation and evaluation. It contains a discussion about possible flaws and future work.

CHAPTER 2

Related Work

2.1 Reproducibility

According to [41] reproducibility is defined as the re-run of an experiment by a different researcher in the manner of the original experiment. Repetition on the other hand is the re-run of the exact same experiment with the same method, same environment and very similar result. Reproducibility is a common problem in all scientific areas. It is an issue of the whole scientific community to produce results that are reproducible. Therefore in [35] there are ten rules defined to gain a common sense about reproducibility in all science areas. They consist of the basic idea that every result of interest has to be associated with a process and data used. External programs have to be persisted as well as custom script versions. The usage of version control is recommended. In addition there is a rule defined to make the scripts and their results publicly available. [35] Reproducibility is also the key element of The Fourth Paradigm published in [13]. 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 in the 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 processes so that scientists are able to have a fast interaction with each other. [13] eScience has the potential to enable a boost in scientific discovery by providing approaches to make digital data and work-flows citable. In [34] is a common way of reaching the goal formulated above. It describes an approach to look at whole research processes, other than only data citation by introducing Process Management Plans. The capturing, verification and validation of the input data for a computational process is also demonstrated within the paper.[34] Computer sciences have the issue of a high amount of published papers that are not provided by enough information to make them reproducible. The problem will not be solved by the scientists that need to do additional afford to make reproducibility possible, but by providing new tools for scientists that allow it automatically as stated in [23].

There are some additional issues on reproducibility in computer science like that the used software technologies are deprecated and not available any more. Therefore persisting the execution context is critical to make a re-execution of the experiment possible. One proposed concept of doing so is the VFramwork described in in more detail in the method section 3.
Data citation is a key issue of reproducing results of past experiments. Preserving the exact process without persisting the original data is no real gain for 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 was persisted. In earth observation persisting the data for reproducibility in the future is an issue discussed in [31].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 3.4.

2.1.1 PROV-O

In 2003 the World Wide Web Consortium published the PROV model as a standard for provenance definitions. It is defined by twelve different documents. In the context of this thesis the PROV Ontology (PROV-O) is the most relevant. [2] PROV-O is a standard language used to describe OWL2 Web Ontology. It is designed as a lightweight concept to be used in a broad spectrum of applications.

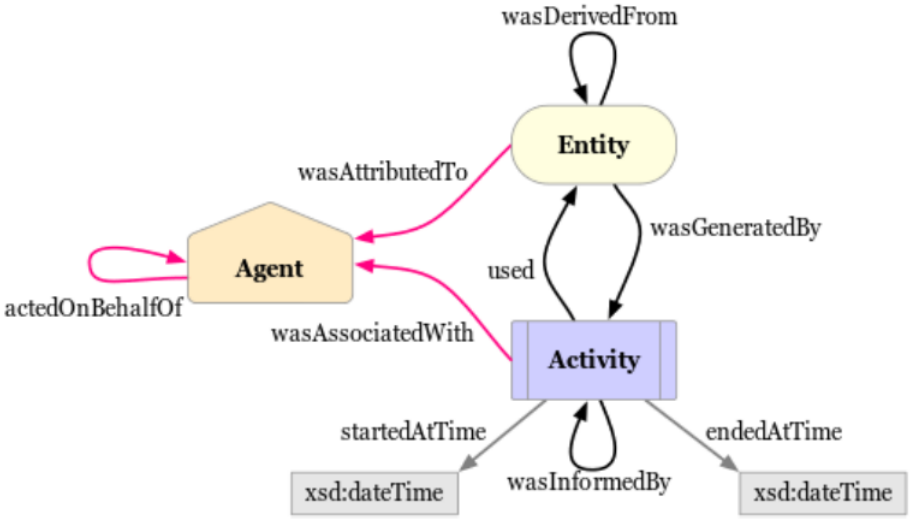

Figure 2.1: Overview of the main components of PROV-O [3]

In figure 2.1 there is the basic setup of the PROV concept. It consists of three main elements. The Entity is any physical, digital, 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. Examples for Agents are software, persons or organizations. The association of an Agent to an

Activity defines the responsibility of the Agent to the Activity. An Activity describes what happened that the Entity has come to existence and how attributes of an Entity changed.[2] In this thesis the PROV-O standard is not implemented, but since it is a reasonable extension it is mentioned in the future work section 7.2.

2.2 Earth Observation Science

Reproducibility is a well discussed topic of the computational geoscience. The high amount of earth observation data leads to complex research programs and therefore high complexity in the papers. According to [27] where the reproducibility and replicability of scientific papers in geoscience are tested, only half of the test group publications are replicable and none of them reproducible. There are publications to address this lack of reproducibility in the earth observation science. The sections below summarize some examples of earth observation communities progresses in terms of the lack of reproducibility.

2.2.1 Vadose Zone Journal (VZJ)

In order to face the issue of lacking reproducibility in geoscience the Vadose Zone Journal (VZJ) started a Reproducibility Research (RR) program in 2015. [39] The earth observation science is a big part of VZJ publications and most of them are not conform with 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 alongside the scientific paper also the code and data used by the scientists for evaluating the publication. This shall lower the access barrier for scientists to publish their research work entirely. Aim of the project is to create a community of researchers with a common sense of reproducibility and data citation on the platform and to animate other scientists to join the approach.[39]

2.2.2 The Geoscience Paper of the Future (GPF)

The geoscience paper of the future (GPF) is according to [11] a proposed standard to help geographical scientists by making reproducible publications. It defines recommendations on data management and software management, by introducing the reader to available repositories. The gain of it is applying concepts of open science and reproducibility to earth observation papers. A GPF needs to apply the following requirements:

- **Reusable data** in a public repositories and persistent identifiers.
- **Reusable software** (including software for preparing and post editing of the data) in a public repositories and persistent identifiers.
- **Documenting the computational provenance of results** in a public repositories and persistent identifiers.

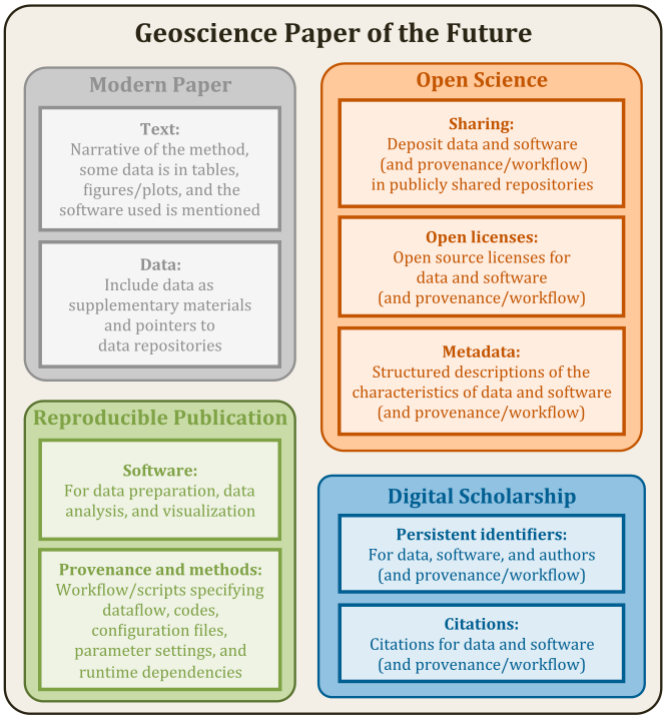

Figure 2.2: Comparison between reproducible publications and geoscientific papers of the future [11]

In figure 2.2 the differences with a reproducible paper gets visualized. 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 consists of a set of 20 recommendations for geoscientists regarding data accessibility, software accessibility and also provenance information.

2.2.3 Climate Change Centre Austria (CCCA)

The Climate Change Centre Austria (CCCA) is a research network for Austrian climate research online available since 2016 and has 28 members. The main tasks are the provision of climate relevant information, the inter-operable interfaces and long term archiving of scientific data. In 2017 the NetCDF data citation was added to the project. The set up of the data service and the technology used in the background is similar to the set up of the EODC back end. Therefore the process of enabling data identification was similar to this thesis. The concept used for gaining data citation was the RDA recommendations defined in [33] and can be viewed in figure 2.3. The main difference is the more complex architecture of the CCCA service compared to the EODC back end used for the OpenEO project. File formats also differ, because EODC uses GeoTiff instead of NetCDF at their back end. Nevertheless the concept of enabling data identification is similar in both projects.[37] CCCA is Open-source and available at GitHub, see [36].

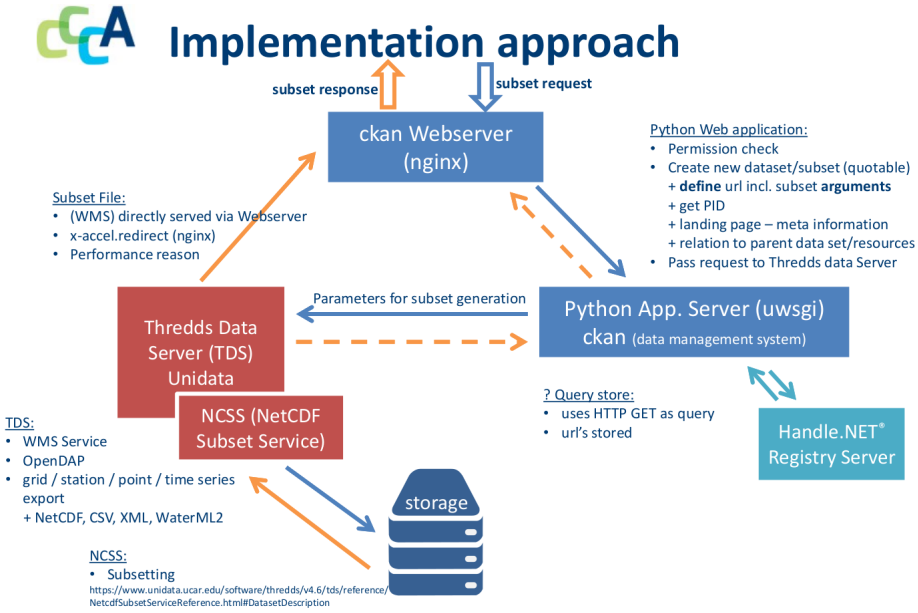

Figure 2.3: Concept of the CCCA NetCDF Cata Citation [37]

2.3 Existing Tools

In this section tools that are designed to solve similar problems like this thesis get summarized and described further. There is also an explanation why the specific tool was not used for the solution of this thesis.

2.3.1 Noworkflow

Noworkflow is introduced in [24] as a script provenance capturing tool with the aim to not influence the way researchers implement experiments. As proof of concept the noWorkflow command line tool got implemented for python. The provenance is captured in a SQLite database by different trials. A trial represents the environment information about one execution of the experiment.The main benefits of noWorkflow is that it does not instrument the code and it automatically captures the definition, deployment and execution provenance in a local SQLite database. The stored meta-data can be accessed via the command line interface of noWorkflow. In addition to just retrieving the information about the execution environment there is also analyses features added.[24]

The noWorkflow framework got improved regularly after the first announcement. In [30] an additional feature of tracking the evolution of the experiment execution is added. It improves the possibility to compare different trials of an experiment and to visualize the history of past executions. In [29] the fine-graned provenance tracking extension of noWorkflow is implemented. With it the execution of the python script can be viewed as a set of execution lines. It also adds a visualization of all called functions

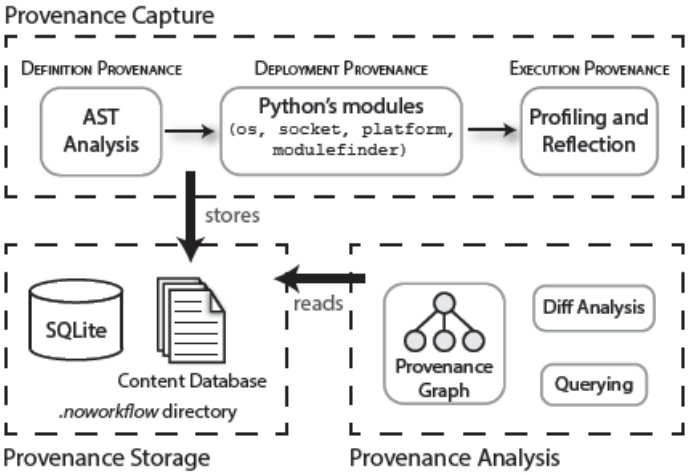

Figure 2.4: Architecture of noWorkflow [24]

with some limitations on multiple function calls in one line, dictionaries and lists. In [4] the provenance capturing of noWorkflow in combination of yesWorkflow, which gathers information about the provenance using comments and annotations (see [21]), got combined. The combination enabled a broader provenance information, querying and visualizations. In this thesis noworkflow is used in an alternative attempt of the implementation described in section 5.3.3, but is not part of the final solution due to reasons provided in 6.5.

2.3.2 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 saves it to a package that can be executed on another machine by ReproZip. It is even capable of letting the re-executor modify the original experiment. It was developed for the SIGMOD Reproducibility Review. In figure 2.5 the architecture of ReproZip is shown in detail. It traces the system calls to create a package defined in the configuration file. So that a single file with the extension ".rpz" gets produced. These type of files can then be unpacked on a different machine and re-executed. The aim of ReproZip is to make reproducible science easy to apply for single experiments. [5] The reason why it is not used in the solution of this thesis, is that the capturing is very fine granulated and takes too much performance from the back ends, which is a key selling point for back end providers. The payment for users of OpenEO, depending on the back end, may have to pay by the duration time of the processing. Another issue with ReproZip in the context of this thesis is that it is not possible to capture the big data of the back ends within the package, because it would take too much space.

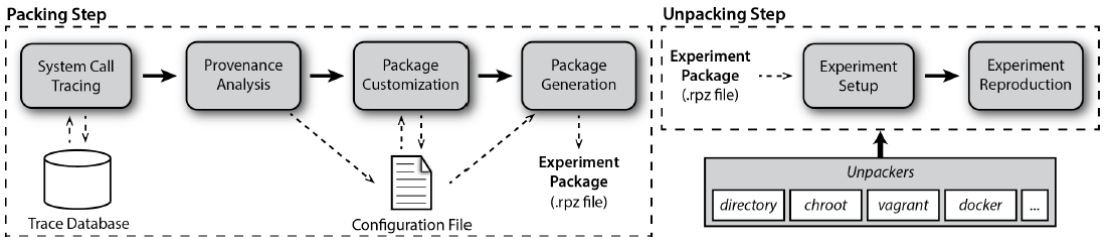

Figure 2.5: Overview of the ReproZip concept. [5]

2.3.3 Docker / Smartcontainer

Docker containers are very common in geoscience executions. The advantages on reproducible research and cost savings by using docker containers is discussed in [18] for the Geographic Object-Based Image Analysis (GEOBIA). The docker implementation of the image analysis was implemented with a docker image including a user interface that can be used by non-experts. There are experiments for the more general Object-Based Image Analysis (OBIA) with docker containers presented in [17]. Conclusion of the previous mentioned paper is very positive with only little shortcomings in the usability. The two papers mentioned above are using the docker images to make it easy to re-run an experiment on the OBIA system. Remaining question is how the docker configuration can be preserved in a manner so that it can be reproduced in different environments. The aim of [9] is to answer this question by introducing in addition to the Docker description file a workflow record saving the environment and entities involved. A SPARQL query got introduced to create the possibility to use the container as a repository of metadata. Another approach of preserving a docker container are smart container introduced in [14]. The aim of smart container is an ontology and software to preserve docker metadata. It uses the PROV-O standard to define the provenance. Docker containers are not part of the solution of this thesis, because the impact of the implementation shall not be too big for the back end providers. The solution has to be simple and general to be applied on all different back ends. Other back end implementations may use docker containers to apply the defined context model in 4.

CHAPTER 3

Methodology

In this chapter the methodology used in the development of the solution of the thesis gets presented. In the following the reader gets informed about basic concepts used in the prototype implementation. The information is structured in subsections, where each is presenting different technologies or concepts.
In the first two sections there are technologies presented that are crucial for the concept of the design in section 4.
After that there are two sections presenting concepts used in the solution of this thesis. The last section of this chapter describes the OpenEO project further and more specifically the EODC back end, which is used for the proof of concept implementation.

3.1 Version Control Systems

Version control systems (VCS) became a very important part in all computational sciences. It enables to persist versions of code and the possibility to head back to a certain version of it. Before that programmers tend to have multiple directories to version the code. The basic idea of VCS is that via a command line interface it is possible to set a version of the current state of the code. These versions can be accessed in future, without changing other versions of the code and without multiple folders. [20] In this thesis Gitorious (Git) is used as version control system. Versions in git are defined as commits and are stored locally, but can be published to an external server, where, depending on the user rights, they are accessible for other users. The commits are stored locally and remotely. [1] In OpenEO Git is defined as the used versioning tool and GitHub is used as the publicly available server. Since OpenEO is an open source project, the code of every back end, core API and client is available at GitHub. In this thesis the Git commit and the GitHub repository is used as the identifier of code versions running at the back end.

3.2 Hash

Hash functions are used to validate the data without having to save the whole data. They have three important properties to work properly. First it has to be difficult that two different inputs have the same hash outcome. Second is has to be difficult to recreate the original input data from only the hash information. Last but not least it needs to be hard to find a message with the same hash value as an already known message. The properties described above makes the hash functionality a common tool to identify data without having to save the original one. [40] There are different hash functions available. In this thesis the SHA-256 is used for the metadata of the context model, mostly to compare differences in data outcomes.

3.3 VFramework

The basic concept of the VFramework is that the execution is done with parallel capturing of provenance data. During the original execution evidence gets collected into a repository. All of which is persisted in the context model of the execution. A re-execution can then be 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 is divided into static and dynamic data. Static data defined in the VFramework is data that is not dependent on the execution of the experiment. The static environment information is therefore independent from the set up and the configuration of the execution. Dynamic data on the other hand has to be captured during the execution of the original experiment. It is the information specific for the workflow. [22] In figure 3.1 there is an overview of the VFramework concept.

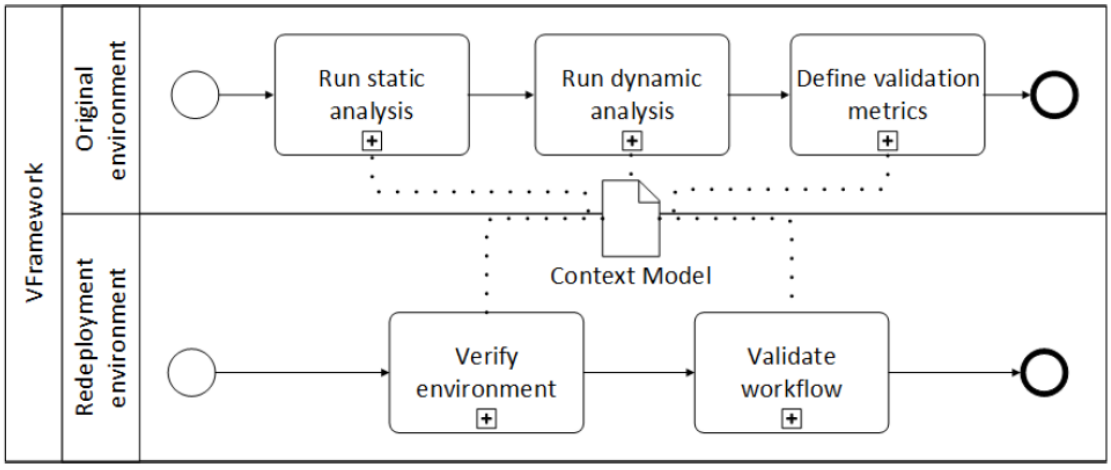

Figure 3.1: Overview of the Concept of the VFramework [22]

3.4 Data Identification

Data identification and citation is a main concern in many computer relying sciences. For the aim of this work, the input data is a key element of the capturing. 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) presents general solutions to achieve data identifications. There are 14 recommendations defined to achieve the identification of an exact version and subset of input data. The recommendations are independent of the type of data and database. In the following the most important recommendations for this thesis get briefly introduced.[33]

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

- **R2: Timestamping**
  All changes to the database have to be comprehensible via timestamps. So every time changes are applied to the data, there needs to be a timestamp when it happened.

- **R3: Query Store Facilities**
  There needs to be a query store implemented at the data provider to store queries including their metadata to be able to re-execute them in the future. The database has to store, according to [33] the following things:

  - The original query as posed to the database
  - A potentially re-written query created by the system (R4, R5)
  - Hash of the 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)
  - Other metadata (e.g. author or creator information) required by the landing page (R11)

- **R4: Query Uniqueness**
  Since there should not be equal queries with the same result stored at the query store, there needs to be a normalized query that can be directly compared to other queries. Hence there needs to be a programming logic to normalize the queries and to enable their uniqueness.

- **R5: Stable Sorting**
  The sorting of the resulting data needs to be unambiguous and therefore reproducible.

- **R6: Result Set Verification**
  To ensure that the resulting data of the query is comparable there needs to be a checksum or hash key of it.

- **R7: Query Timestamping**
  There needs to be a timestamp assigned to every query in the query store, which represents the last update of the entire database or the query dependent data of the database.

- **R8: Query PID**
  Every query record in the query store needs to have a Persistent Identifier (PID). There must not be a query with the same normalized query and the same result checksum.

- **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 a generation of the citation text snipped including the query PID.

- **R11: Landing Page**
  The PID need to be resolvable in a human readable landing page, where all the 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 also machines can access the metadata by query re-execution.

- **R13: Technology Migration**
  If the database where the query store is implemented needs to be mi grated to a new system, the queries need to be transferred too and have to be updated according to the new setup.

- **R14: Migration Verification**
  There shall be a service so automatically verify a data and query migration, to prove that the queries in the query store are still correct.

The implementation of the recommendations for the purpose of this thesis is described in section 5.1.

3.5 OpenEO

The OpenEO project contains of three modules, the client module written in the programming language of the users, the back end drivers that enables for every back end to understand the calls from the clients and the core API that specifies how the communication takes place. The core API is a standard that the back end providers accepted to implement on their systems. The back end drivers are the translation of the client calls to the back end specific API. This architecture decouples the clients from the back ends so that every client can connect to every back end that applies to the OpenEO core API standard see figure 3.2 . An example of a workflow would be that a scientist wants to run an experiment on the EODC back end and defines the processing with the python client in pure python code.[28]

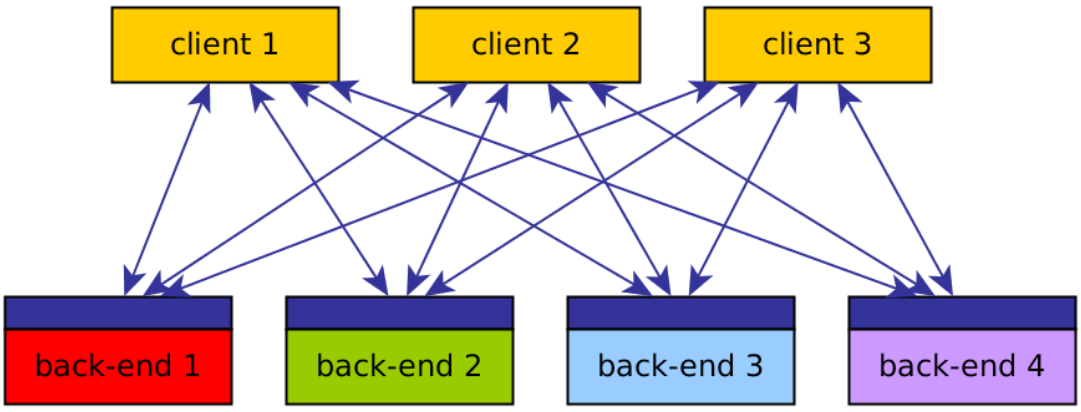

Figure 3.2: Overview of the OpenEO architecture. [25]

The communication is specified as an OpenAPI description, which is a way of defining RESTful communication in a standardized way. The definition consists of the possible end points at the back end and also the requests and the responses. The whole communication protocol is specified with OpenAPI.[38] In the following the relevant RESTful request types in OpenEO and the policy of choosing between them gets introduced.[25]

- **GET Request**
  GET requests are used to retrieve data from the back ends. The functionality is limited to read operations on the data records.
  (e.g. GET /collections returns a list of available collections at the back end.)
- **POST Request**
  POST requests are used to create new data records at the back end.
  (e.g. POST /jobs creates a new processing job at the back end.)
- **PATCH Request**
  PATCH requests are used to update an existing record at the back end.

(e.g. /PATCH /job/job_id modifies an existing job at the back-end but maintains the identifier.)

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

3.5.1 Job Execution

The job execution workflow in OpenEO starts at a client application that let the user define what has to be processed in the client specific programming language. The main part of the job execution definition is based on the description of what input data shall be used, which filters have to be applied and the processes that should be executed on the data. Therefore, OpenEO introduces the process graph, which is defined as a tree structure describing the processes with their data and the input data identifier. The input data id is back end specific. The process graph has a JSON format and gets generated by the clients in the background without users noticing it directly. In figure 3.3 there is an example of a process graph.

```
{
  "process_id":"min_time",
  "args":{
    "imagery":{
      "process_id":"/user/custom_ndvi",
      "args":{
        "imagery":{
          "process_id":"filter_daterange",
          "args":{
            "imagery":{
              "process_id":"filter_bbox",
              "args":{
                "imagery":{
                  "product_id":"S2_L2A_T32TPS_20M"
                },
                "left":652000,
                "right":672000,
                "top":5161000,
                "bottom":5181000,
                "srs":"EPSG:32632"
              }
            },
            "from":"2017-01-01",
            "to":"2017-01-31"
          }
        },
        "red":"B04",
        "nir":"B8A"
      }
    }
  }
}
```

Figure 3.3: Proof of Concept process graph of the EODC back end for the OpenEO core API version 0.0.2 . [26]

The back ends interpret the process graph from inside out. In figure 3.3 the element in the center of the process graph defines the input data identifier in the "imagery" block, with the "product_id". In this case the "s2a_prd_msil1c" is chosen as input data identifier. After reading the input data id, the back end iterates one step up in the hierarchy of the process graph and calls the process "filter_bbox" with the parameters "left", "right" and so on. Every process beginning with "filter_" is a filter operation that specifies the selection of the input data. The output data of the previous process is the input data of the next process. In figure 3.4 the same process graph is visualized from the back end point of view, so the order of how it gets executed.

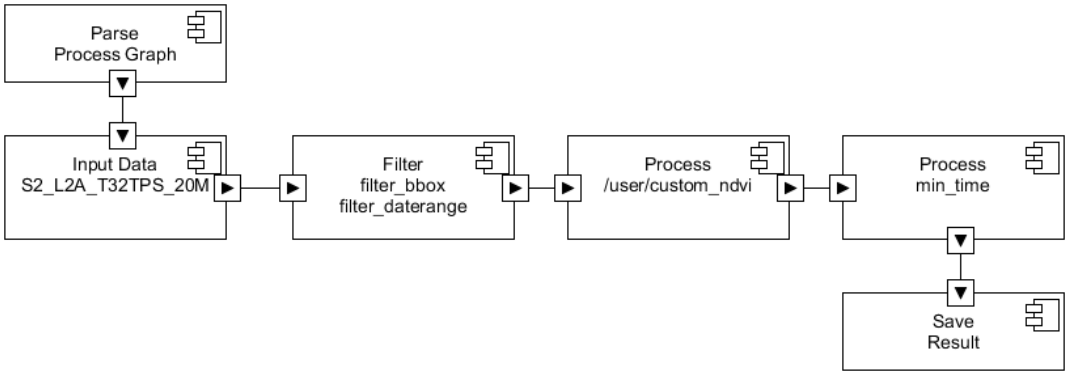

Figure 3.4: Action chain of the back end after receiving the process graph of figure 3.3

There are two different kinds of process executions depending on the back ends capabilities, synchronous and asynchronous calls. Synchronous calls are directly executed after the back end receives them and the user has to wait until the job is finished. For example on the python client the program waits after sending the process graph to the back end until the back end returns the result, which is directly returned to the user. An asynchronous call does not get executed until the user starts the execution on the back end through an additional endpoint call. When the processing is finished the user can download the result at another endpoint of the back end. For asynchronous calls there is also the possibility to subscribe to a notification system on the back end, so that the user gets notified when the job execution finished. The processes are defined at the OpenEO core API and therefore independent of the back end they get called at, other than the data identifier, which is different for every back end.
The previous example shows a process graph that only uses predefined processes and data. Within the OpenEO project there is the possibility to define individual processes and execute them on the back end. In the project they are called “user defined functions” and are at the writing of this thesis still not well-defined, but are basically code written by the OpenEO user that gets sent to the back end and executed there at a secure environment. The user can define processes and can run them with the data provided at the back end, using the infrastructure of the back end. Every back end has to individually define what

Table 3.1: List of all back end 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

the restrictions on user defined functions are. [10]

3.5.2 Back end Overview

Even though the back ends implement the OpenEO core API standard, they are still diverse behind this abstraction layer. Some back ends have already an API, where the OpenEO calls have to be adapted to. There are 7 partners within the OpenEO project that are implementing a back end driver. The back ends have to manage the translation of the process graph to the actual code that executes the defined process chain. The billing of the users can be completely different on every back end. In table 3.1 there is an overview of all contributing OpenEO back ends and the related GitHub repository.

3.5.3 EODC Back End

The EODC back end is one of the contributing back end providers of the OpenEO project. It is in general a python implemented back end that uses virtualisation technologies for the job execution. The overlaying technology is OpenShift (using Kubernetes) [12], which is capable of scaling docker containers and handles the execution of them. In the docker containers the python code for the processing gets executed. The docker description files and the python code is available on GitHub[1]. In this thesis the latest version of the EODC back end provided in GitHub with the version 0.0.2 is used. In this version every process of the OpenEO process graph is represented by an own docker container and python code of the processing. The service layer for accessing the back end is accomplished by the python library Flask. Even though EODC has a lot of already processed data, it only uses data from Sentinel 2 and Sentinel 1 in OpenEO. The provided data are satellite images from the European Space Agency (ESA), which gets the raw data coming directly from the Sentinel satellites.
The data management of EODC is file-based, so every image data is stored in a unique directory and filename combination. Meta-data is provided at via a PostgreSQL database

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

including the PostGIS[2] plug-in. The provided query tool for EODC users is the OGC standard interface CSW[3]. In figure 3.5 an overview of the EODC meta-database is displayed. The structure is retrieved from the github repository of the EODC back end [10], where every database entity is defined. Every Process entity can have parameters desribed by the Parameter entity. The ProcessNode is representing one node in a process graph and is therefore related to exactly one ProcessGraph entity. Every Job has a related process graph, there may be jobs that use the same process graph, but in the current set up they are persisted both in the database.

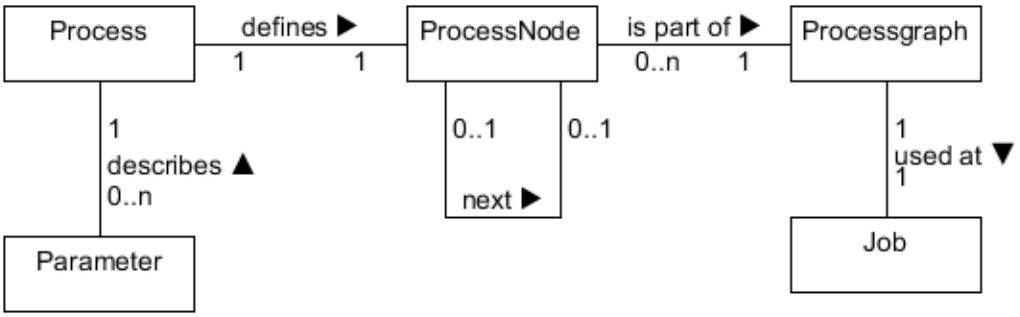

Figure 3.5: Overview of the EODC meta-database structure.

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

CHAPTER 4

Design

In this chapter the design of the provenance capturing in the OpenEO project gets described. The aim of this chapter is to explain the general concept of how to gain reproducibility in the OpenEO project. The chapter is structured in six parts. In the first part an overview of the concept gets presented. In the next three sections the main components presented at the overview are described in detail. These are the Data Handler, the Job Capturing and the Result Handler. The next section defines the resulting context model. In the last section of this chapter defines the user interfaces needed to use the data of the context model from an OpenEO user perspective. The following chapter 5 shows a proof of concept implementation of the design.

4.1 Overview

In this section an overview of the concept is presented and in figure 4.1 visualized. In this figure the workflow that every OpenEO back end driver needs to implement in the white components. The green elements in the diagram are the proposed extensions of this thesis design. The typical job execution work-flow is described in the following steps.

1. **OpenEO Client**
   The user defines the input data, the filter operations and the processes that need to be executed at the back end via the OpenEO client. Then the user orders the job to be executed, therefore the client creates a process graph. The user identification and the process graph is sent via the RESTful endpoint defined in the OpenEO coreAPI to the back end driver.

2. **Data Query**
   The Data Query component receives the process graph and parses the data identifier and the filter operations out of it to query for the input data records needed for the job. These are forwarded to the Process Execution component.

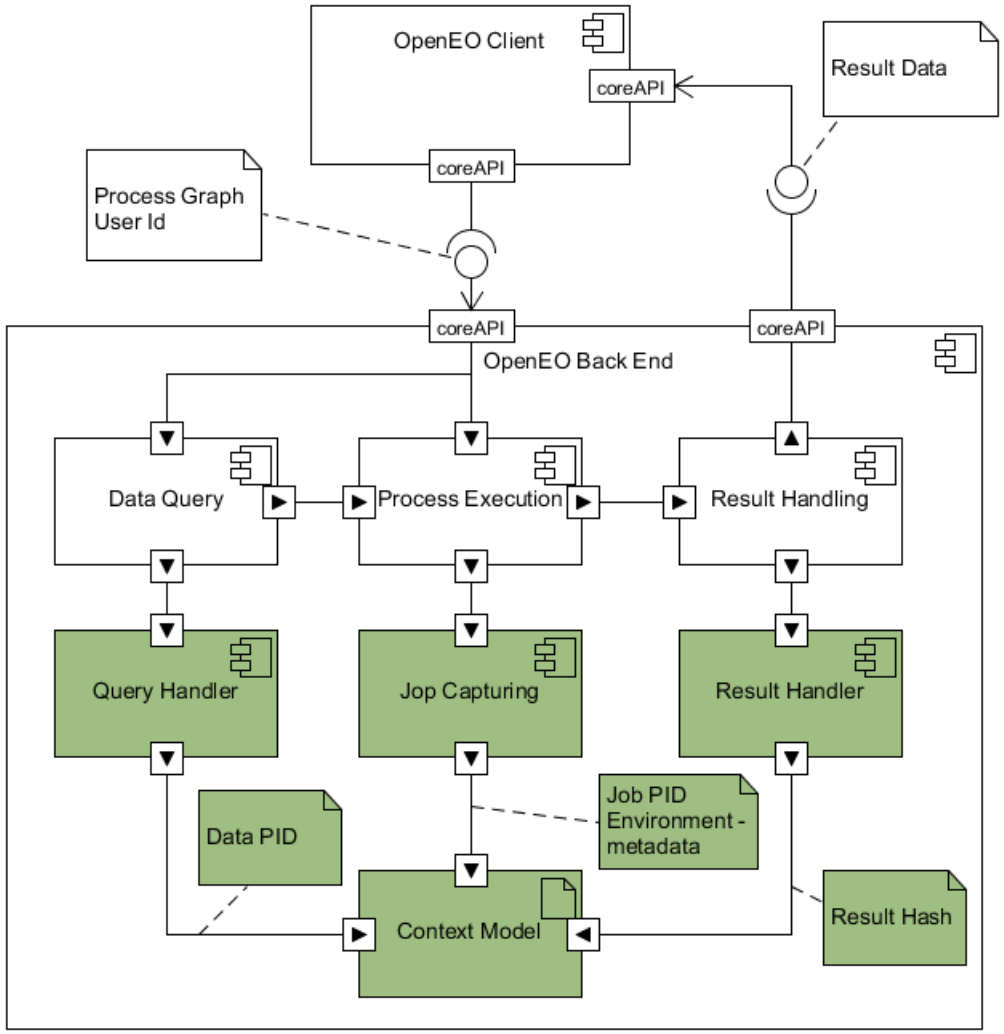

Figure 4.1: Overview of the Design

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 defined in the process graph got executed, the resulting data is forwarded to the Result Handling component.

4. **Result Handling**
   The Result Handling component receives the results from the Process Execution component and persists all meta-data about the job and the result. In the meantime

the resulting data is sent back to the OpenEO client, if the job was not a batch job. If it is a batch job the user has to actively order the results from the back end.

To make the whole workflow reproducible, the single components have to be identifiable. Hence, the following elements get introduced as additions to the back end.

- **Query Handler**
  The Query Handler component is responsible for applying data identification to the back end. The query and the resulting data generated by the Data Query component needs to be persisted and identifiable by a Persistent Identifier (PID). The Query Handler is described in more detail in section 4.2.
- **Job Capturing**
  The Job Capturing component is responsible for enable code identification of the used processing at the back end. Therefore, a PID of the code used for the processing has to be introduced. In addition, meta-data of the job execution environment gets captured, to gain feedback information for the users. The Job Capturing component is described in more detail in section 4.3.
- **Result Handler**
  The Result Handler component is responsible for creating a comparable result checksum or hash. The data created by the OpenEO back end can be too big to be persisted in the future, hence there need to be a checksum or hash to be at least capable of confirming equal results. The Result Handler component is described in more detail in section 4.4.
- **Context Model**
  The Context Model is not a component, but a data set that consists of all the meta-data mention in the previous components. One context model is related to one job execution. The details about the Context Models can be found at section 4.5.

4.2 Query Handler

The input data of the processing is crucial for the outcome of the job execution. Even though the process graph already contains an identifier for the input data that has to be unique for the specific back end, changes to this data might not result in a new identifier. So jobs called later in time might use another version of the input data than the previous jobs. To prevent this the input data has to be persisted according to the 14 steps of data provenance defined by the RDA [33]. Every back end is responsible for applying data identification. In the design chapter it is assumed that the back ends preserve the data as described in 4.2. The Query Handler is the module where the data persistence is implemented and depends highly on the structure and architecture of the back end. Therefore, no common definitions can be made for all back ends in this section. In chapter 5 there is

a proof of concept implementation on the EODC back end. The context model elements of the job executions are specified with a "J" in the beginning for readability reasons. Elements of the back end environment context model are specified with a "B" in the beginning.

J1: Input Data Persisted Identifier
The output of the Query Handler is the PID of the input data of a job execution. Every job execution has input data that has already assigned a PID or where a new PID needs to be created. Therefore the PID is added to the job dependent context model.

4.3 Job Capturing

The aim of this section is to describe the job execution capturing and a description of data that shall be captured. The meta-data provided by the job capturing can be categorized in static and dynamic data. Where static data is defined as not dependent on job properties and dynamic data is defined as job dependent information. In the following two sections the static data (in this thesis called back end provenance) and dynamic data (in this thesis called job dependent provenance) are further described.

4.3.1 Back end provenance

The scope of this part of the context model is to get the static environment of where the job execution at the back end takes place. It contains the provenance data that is independent of a job execution, so does not change regardless of how many jobs were processed. It can only change from inside of the back ends by their maintainers. In the OpenEO project there are a great variety of different back end providers with very distinct setups, so the challenge of this part is to make it as simple and generic as possible. The data captured in this part of the context model is not meant to be shown to the user directly, because of security issues. The following meta-data is suggested for the back end provenance. The context model elements of the back end environment are specified with a "B" in the beginning for readability reasons. Elements of the job dependent context model are specified with a "J" in the beginning.

B1: Code Identification
Since the OpenEO project is open source, every back end has a GitHub repository were at least the basic setup is stored. The aim of this strategy is to get other back ends with similar settings to reuse the already working setups of the running back ends. Therefore every back end provider has a GitHub repository were the code is public-ally available. This information is added to the back end provenance by saving the git repository URL, the used branch, the used commit and local changes to the repository. This information can be used to identify the used code at the back end.

B2: Core API Version
The back ends core API version is the version of the OpenEO core API it currently

is capable of. In the productive usage of the OpenEO project it can happen, that the versions of the clients and the back ends differ, so that the behavior might not be compatible. So the back end server configuration depends on the core API version of OpenEO, hence it shall be added to the context model.

B3: Back End Version
The captured data described in the previous sections will result in a back end version. The back end version shall be an identifier (e.g. a number) that gets updated on every change of the backend considering the provenance data described above.
To provide long term stability of the capturing, it is a good idea to implement a tool to automatically capture the provenance data and persist it in a back end context model. Every change on any of the previously described back end meta-data has to be detected automatically and have to result in a new back end version.

4.3.2 Job dependent provenance

In this section the job dependent provenance of the context model gets described. The data captured is tied to a specific job execution, so for every job execution a new job context model gets created. The process graph is already a description of the processes that run at the back end. So sending the same process graph to the same back end again shall result in the same outcome. To assure that the process graph can be re-executed in the same way as the original execution, meta-data about the original execution gets captured. The single parts of the capturing are described more in detail in the following subsections.

Process Environment

In this section the capturing of the process itself gets described. As mentioned above the process is described with the incoming process graph, but to be certain that the same process id results in the same code, the code version is added to the context model. To be able to identify code from different versions a source control management needs to be installed. An example of doing so is persisting the back end source code on a public platform like GitHub. 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 of it are persisted in the context model. To gain more meta-information about the processing the start time and end time can be added to the context model via timestamps. This leads to the following capturing elements in the job context model.

J3: Programming Language
The programming language of the code that is used for the processing. In addition the version of the programming language has to be included in this information.

J4: Installed packages of the programming language
To identify the environment of the code execution, the installed packages of the program-

ming language can be captured and added to the context model. The version of the packages have a high influence on the outcome, hence can also be included in the context model. For example in python the installed modules can be added with their versions to the context model.

J5: Start and End time of process execution
On the beginning of the process execution a time-stamp can be created and at the end of the process execution. These timestamps can then be persisted in the context model.

Back end provenance

The back end provenance described in 4.3.1 defines the version of the code that is executed and therefore has to be added to the job context model. Over time the back end provenance will get updated so that every job need to persist the original back end configuration at the execution time. There are two possible solutions to achieve this. First the identifier of a specific version of the back end gets added to the context model. In this case the data of the back end provenance versions have to be persisted and accessible. The second solution is to put the whole back end provenance of the current version into the job execution context model.

J2: Back end provenance / Code Identifier
The version of the back end provenance during the job execution. Since for every change on the back end a new version will be applied, the back en version is the code identifier of the execution. If the back end environment is not persisted properly at the back end, the whole back end provenance can be added to the context model.

4.4 Result Handler

The output data of the processing has to be captured as well to be able to compare job execution outcomes. In the OpenEO project the results can be rather big, that is why a simple generated hash value of the output result is enough in the context model defined in this thesis. The output data does not have to be identifiable within the scope of this thesis, but just comparable with other executions. Therefore a hash of the output data is sufficient. The aim of the output data capturing is not to add the possibility to find differences between results, but to be able to see that the results differ.

J6: Result Hash
The output result of the job need to be verifiable and therefore need to be persisted. An easy way to achieve this is to take the hash value over the sorted output files.

4.5 Context Model

The context model is the data record that saves the provenance of a job execution. The type of storage is defined by the infrastructure of the back end provider. It can be stored in a relational database or in a file-based system as a file on the file system. It has to

be added to the meta-data database of the back end. There are two different context models that need to be persisted at the back ends.
First the back end environment context model that saves job independent provenance and results in a back end version. The mandatory element of the back end environment is the back end version that defines the state of the back end in a specific time. The back end version need to be resolvable to the specific code version that was present in a past execution. In addition meta-data about the back end can be added. The elements (B1-B5) defined in section 4.3.1 are suggestions of elements in the back end environment context model.
Second the job execution context model that includes all information that is dedicated to a specific job. There are three mandatory elements in the job context model. The input data identifier has to be persisted in the context model, so that it can be identified and used in future job executions. The used version of the back end for the execution has to be included in the context model, to be able to re-execute it with the same code in the future. The output hash has to be added to the job dependent context model, so that different results can be detected. In section 4 the elements (J1-J6) are recommendations of these data and suggestions on additional job specific meta-data.

4.6 User Interface

In this section the information for the users and how it is provided gets described. The capturing described in the previous sections consists of information about the back ends that shall not be completely passed to the users in detail. For example it can be a security risk to provide specific programming language packages. On the other hand the users need to be able to see the differences of job executions and get environment information about the back ends. Therefore there has to be a filter on which data can be shown to the users and what not. Every captured information is also not necessarily interesting for the users. Data that is not secure to the user has to be defined by the different back ends themselves. The OpenEO project has a very diverse set of back ends and every one has an unique company security guideline.
Provenance information need be provided to the user. Therefore there have to be additions to the core API specification. Additional endpoints for the users to get information about the back end has to be created for the core API, back end and client. The following points summarize the set of needed additions, they are structured with a "U" in the beginning to add readability through the thesis.

U1: Back end version
There has to be an endpoint for users to retrieve back end specific information especially the current version. The aim of it is to present the users with information about the current state of the back end and to help OpenEO users to decide, which back end they want to use. There is already an endpoint defined in the OpenEO core API specification for the capabilities of the back end, where the back end information can be added.

U2: Detailed Job Information

There is already an endpoint to get detailed information about the job status. But the provenance of the job is not in there yet, so it has to be specified in the API what information from the job context model is provided there. At least the resolvable persisted identifier of the input data, the back end version and the result set hash has to be added to the view for comparison reasons. To make it human readable e.g. the resolvable identifier URL of the input data and the GitHub repository and commit identifier can be provided.

U3: Comparing two Jobs
The API has to add an end point for making it possible to compare two different jobs. There does not exist an endpoint for this in the current(version 0.0.2) core API specification. The response of the comparison request consists of the differences in the context model between two jobs.

U4: Data Identifier Landing Page
After the job is executed and the input data has got an PID, the user has to be provided by a landing page. On the website the user is provided with information about the input dataset and the query gets re-executed to view the concrete input data.

U5: Re-use of Input Data
The current core API version (v0.0.2) is not capable of letting the user choose, which version of data the user want to use. Hence, there has to be an addition to the process graph so that product identifier has an additional time-stamp. So every input data identifier has now a time-stamp for when the data was retrieved. The meaning of the time-stamp is to tell the back end that the user wants to use the data how it was accessible at the time-stamp. This gives the back ends the possibility to design their own strategy of applying a version to the data. It has to be persisted, which version of the data was available on what date and time. In addition the API can allow to include the use of an input data PID in the process graph. So that users can include cited data directly in a new created job.

CHAPTER 5

Implementation

In this chapter the EODC prototype of the concept described in 4 gets presented. The implementation consists of all suggested changes that need to be made to the OpenEO workflow including parts of the back end, core API and the clients. Thus all three parts of the OpenEO project architecture get modified in the presented solution. There are several different back ends and clients implemented in parallel, which are all compatible with each other. One of each of the clients and the back ends is used for the proof of concept of this thesis. The python client[1] is modified for the purpose of this thesis. The EODC back end[2] gets used for the back end part. Since python is the most common programming language at the contributing back ends of the OpenEO project, these implementations get used. Both of the chosen implementations are using python as their main programming language. The adaptations described in 4 are implemented for the usage of the python client accessing the EODC back end. Both of the solution implementations are open source and can therefore be used by other back end providers with a similar setup.

The implementation can be divided into four independent parts. In the following four sections these parts are described in detail. First the data identification implementation gets described, so the implementation of the RDA recommendations on the EODC back end. The next section is about the implementation of the automated back end provenance tracking tool that captures data defined in section 4.3.1. After that there is a section about the job execution provenance as defined in 4.3.2. The last section is about the implementation of user interface additions defined in 4.6.

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

5.1 Data Identification

As defined in section 4.2, the RDA recommendations have to be applied to every back end provider individually. In this section the data identification solution on the EODC back end is presented. Other OpenEO back end provider can use the presented approach as well to enable data identification on their back end setup. Some parts of the implementation can be used by all back end providers. The landing page and work-flow of users getting and using the data identifier is described in detail in section 5.4.

5.1.1 Query Store

The main objective of the RDA recommendations is the implementation of a Query Store. Queries in the Query Store must be comparable, identify able and persisted. At the EODC back end the Query Store is realized with two additional tables in the existing PostGreSQL meta-database. In figure 5.1 the EODC database structure from figure 3.5 is visualized with the proposed additional tables. The added Query table consists of the data specified by the RDA recommendations. The QueryJob table defines 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, hence the QueryJob table gets introduced.

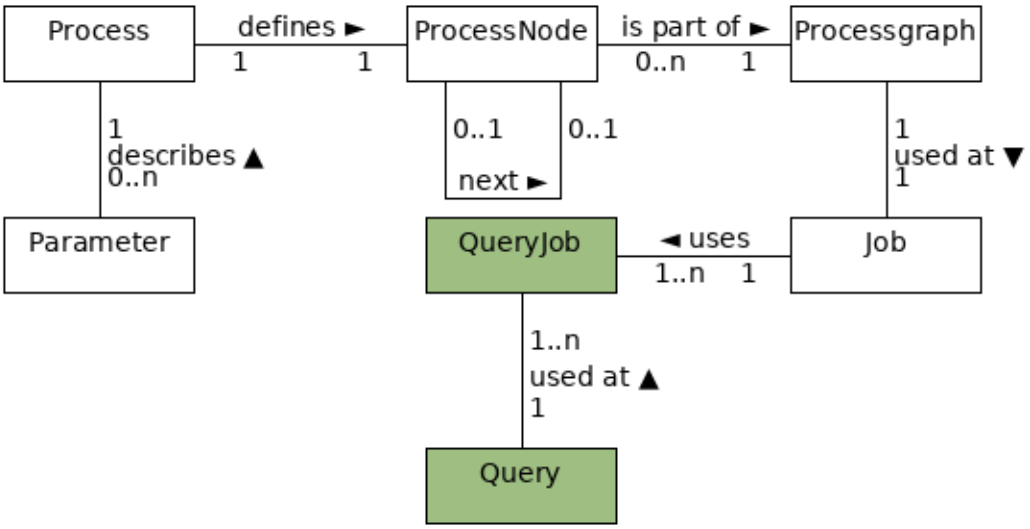

Figure 5.1: Overview of the meta-database of EODC with the new introduced tables (green).

5.1.2 Query Handler Work-flow

In figure 5.2 there is an overview of the data identification solution of the EODC back end. The green parts of the diagram are representing the additions proposed in the thesis solution. The Query Handler itself is an additional python module in the EODC back end that gets called in parallel of the job execution.

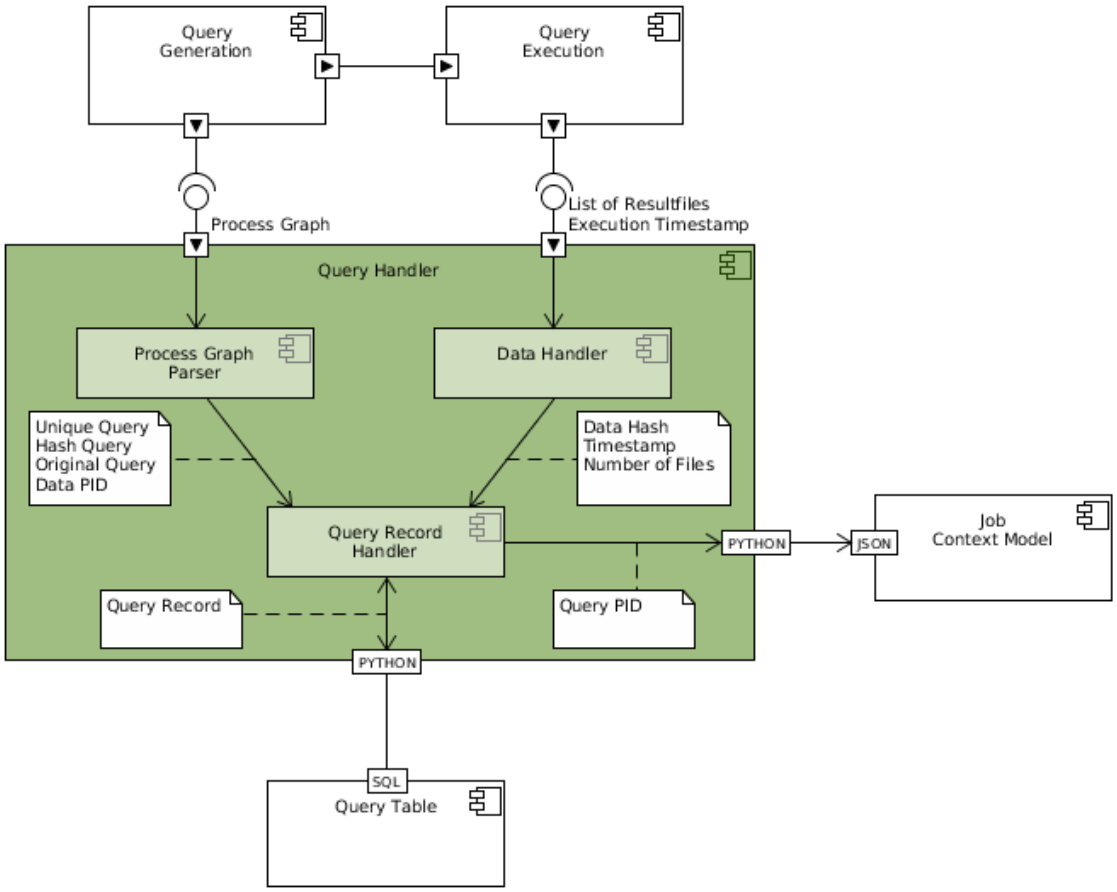

Figure 5.2: Overview of the proposed data identification workflow at the EODC back end

The Query Generation and Query Execution components of the EODC back end provide the input data of the Query Handler addition. The process graph is the input process graph received by the back end driver from the client application. The list of result files is the result of the query execution process and it comes with the execution time stamp. The Query Handler consists of the following components:

- **Process Graph Parser**
  The Process Graph Parser takes the process graph as input and generates the necessary query data out of it. The output consists of the original query, the unique/normalized query, the hash of the normalized query and the persisted data

Table 5.1: Structure of the Query Table in the EODC meta database.

Query PID	Dataset PID	Original Query	Unique Query	Hash Query	Hash Result	Exec. Timest.	Add. Metad.
1	2	3	4	5	6	7	8
...	...	...	...	...	...	...	...

identifier of the query. This information is forwarded to the Query Record Handler. The mentioned data items are described more in detail in the next section.

- **Data Handler**
  The Data Handler is responsible of creating the result data related information. The output is the hash over the file list result, the execution time stamp and additional meta data like the number of files. Since EODC 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 do not have a possibility to choose a specific sorting, therefore the predefined CSW sorting is used for the hash production. The results of the component is forwarded to the Query Record Handler.

- **Query Record Handler**
  The Query Record Handler communicates with the meta database from EODC to see if a Query is already existing. The Work-flow of the Query Record Handler can be viewed in figure 5.3. If the Query Record already exists the Query Record Handler forwards the existing Query PID to the context model, otherwise a new Query PID gets created and persisted in the Query Table. The combination of the result-file hash and the normalized query hash must not have duplicates in the Query table. On saving the Query PID in the job context model, a QueryJob table entry gets created to store the relation between the job id and the Query PID.

5.1.3 Query Table Structure

The Query Table stores the data related to the query instance and additional result and meta data information. In table 5.1 the stored datasets are displayed. To illustrate the Query data record, example input values are used to explain how the single data entries are generated. Therefore, in figure 5.4 there is an example input process graph.

The elements of the query data set records is described in the following list:

1. **Query PID**
   The Query PID gets generated by the python library uuid[4]. The library is used to

[3] http://cite.opengeospatial.org/pub/cite/files/edu/cat/text/main.html
[4] https://docs.python.org/3/library/uuid.html

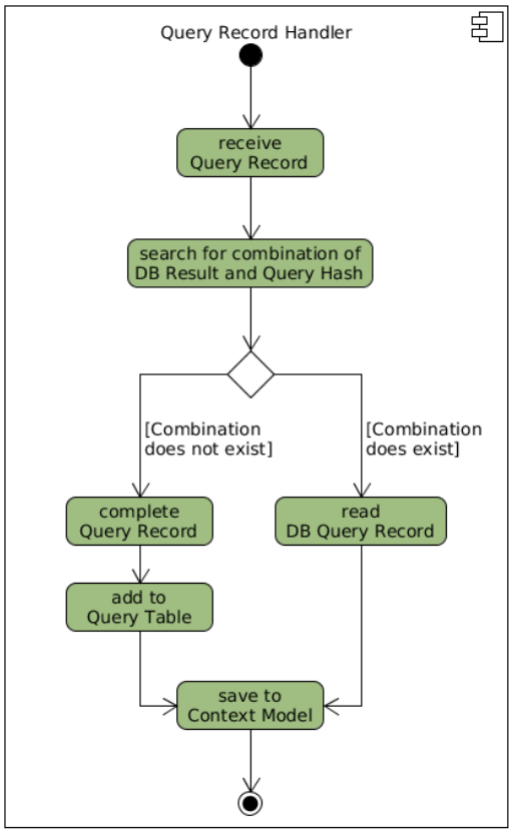

Figure 5.3: Activity diagram of the Query Record Handler component.

generate unique identifiers and also used in the EODC back end for generating the job ids. Since ids in the EODC back end have always in the beginning a code for the specific entity the id is for (e.g. "jb-" for Job entities) a "Q-" is added to the python UUID.
Example: "Q-16fd2706-8baf-433b-82eb-8c7fada847da"

2. **Dataset PID**
   The dataset PID is the identifier of the satellite in the query part of the process graph.
   Example: "s1a_csar_grdh_iw"

3. **Original Query**
   The original query is extracted from the process graph by parsing only the filter elements out of it. Filters in the OpenEO project are, according to the openEO core API (see [26]), defined by the prefix "filter_". In addition to the filter operations the data identifier is stored for the query.
   Example: See the Original Query in figure 5.4

Figure 5.4: Example Input process graph of the Query Handler component.

4. **Unique Query**
   The unique query is the restructured query that is comparable to other unique queries. Since the order of the filters makes no difference in the outcome of the query, the elements of the original query can be alphabetically sorted by the JSON keys to get the unique query.
   Example: See the Unique Query in figure 5.4

5. **Unique Query Hash**
   The unique query hash is the output of the SHA-256 hash function with the unique query as its input.
   Example: "AE7EF888CDEDF8A9A371F3F3A492368E8C14AE8E0BBE92927E09DE0F395D6041"

6. **Result Hash**
   The unique query hash is the output of the SHA-256 hash function with the result file list as its input.
   Example: "565D229FCE4772869343B1F49ABB2F98D4C79345D6A2542D8063BD8203D39434"

7. **Execution Time-stamp**
   The execution time stamp is the input parameter of the Query Handler and is just transformed to the data type needed by the data base.
   Example: "2018-10-17 18:03:20,609"

8. **Additional Meta-data**
   The additional meta-data column of the Query table can be used by the EODC

back end to store additional information about the query execution. In the implementation of this thesis, only the number of output files are persisted in the database. The column is defined as a JSON object of additional information.
Example: "{ "number_of_files": 10}"

5.2 Back end provenance

The aim of the automated back end provenance computation is to implement a simple approach capturing changes on the back end. First only the parts of EODC that have an influence on the back end processing results is defined. In addition to this the implementation generates a back end version automatically, when changes happen at the back end. The proposed tool can be used by every OpenEO back end with a Linux system with some simple configuration changes. It is completely independent of an existing back end driver service, so that it is a separate tool that gets executed at the back end. In this section the tool implemented for the EODC back end is called "*BackendMonit*".

BackendMonit is a standalone tool that gets called periodically via the cron daemon (see [6]) by the back end hosting machine. EODC uses python as default programming language on a Linux machine, therefore the capturing tool is written in python and uses linux tools. It is implemented highly configurable, so that other linux based back ends can use it to capture back end changes too. The back end provenance tool uses a command line interface (CLI) and needs a configuration file as an argument. The workflow of running *BackendMonit* is as follows:

1. **Install Requirements**
   BackendMonit requires for the execution a view linux packages. First the tool runs on Python 3.6, hence it has to be installed on the system. There are no additional python packages necessary for it. The version control system Git is a requirement on the back end, since every back end has its own GitHub repository it may be already installed anyways.

2. **Setup Configuration File**
   To make *BackendMonit* adaptable for other systems a configuration file gets introduced to describe the behavior of it. The tool comes with a default configuration file, which the back end provider adapt to their back end. All paths to the tools described in the previous point is in the config file specified. Additionally, the path to the GitHub Repository used for the local back end repository is set in the configuration file.

3. **Starting Service**
   After the configuration is done, *BackendMonit* gets executed in daemon mode, where it starts inotify (see [16]) to run in the background. To start the tool in daemon mode the "–init" parameter has to be set on execution. Inotify is used to

notify *BackendMonit*, if the repository of the back end has changed. It writes to a log file all writing operations on the repository.

4. **Start Cronjob**
   Now that inotify is configured and started as a service, *BackendMonit* has to check periodically if something changed on back end side. Therefore, the tool needs to be started every minute adding the "–check" parameter. Every minute it compares the current state of the back end with the current context model. If the context models differ it adds a new back end version with all defined meta-data to the context model JSON file.

Provenance Repository
The context model of the back end provenance is a single JSON file that consists of all versions of the back end and their meta-data described below. In figure 5.5 there is an example snipped of the provenance repository.


```
{
      "backend_version": 2,
      "api_version": "0.0.2",
      "git_repos": [
          {
              "branch": "master",
              "commit": "05f4765de578467fef8e1a24404bbd7f77b61c17",
              "diff": null,
              "url": "https://github.com/Open-EO/openeo-openshift-driver.git"
          }
      ]
},
{
      "backend_version": 1,
      "api_version": "0.0.2",
      "git_repos": [
          {
              "branch": "master",
              "commit": "1576122009d3a9cd20015e9673156aebd9466237",
              "diff": null,
              "url": "https://github.com/Open-EO/openeo-openshift-driver.git"
          }
      ]
}
```


Figure 5.5: Example snipped of the provenance repository of the EODC back end.

In the following it is explained how the in section 4.3.1 defined provenance data points are read from the system by *BackendMonit*.

B1: GitHub Repository
To capture the GitHub repository, all folders that consists of GitHub repositories are added to the configuration file. In case of EODC there is only one Git repository involved. The path to the Git installation directory is defined in the configuration file. The

monitoring tool writes the GitHub URL, the branch, the commit identifier and a hash of all local differences to the context model. If there are any changes to the previous model, a new back end version is generated. To automatically keep track on changes *Inotify* (see [15]) is used. It is used to call a command if any changes on the file system in a specific folders occur. The changes can be further specified by 12 different operations on the file system like *IN_DELETE*, *IN_ACCESS*. For the purpose of capturing changes in this implementation only write access on the folders trigger *Inotify*. Below there is the command used to configure *Inotify*.

```
inotifywait -d -e modify -e attrib -e moved_to -e create -e delete DIR
                    -o outfile.log
```

B4: Core API Version
The core API version of the EODC back end is updated manually from the back end developers. Therefore the data file consisting this information is specified in the configuration file of *BackendMonit*.

B5: Back End Version
The back end version begins with 1 and gets incremented on every change of the above described data. On every call of the cron job, *BackendMonit* captures all above points and compares them to the most recent context model. If there are differences the version gets incremented by one and gets persisted in the context model.

5.3 Job dependent provenance

The capturing of the job dependent data is limited to the job execution modules of the EODC back end. The EODC back end of the release version "0.0.2" transforms the process graph into separate docker containers. For every process in the process graph there is a docker container running python code. The input of the process is the output of the previous process. The first process docker container has the input data defined in the process graph as input value. The result gets into a temporary folder separated by the different process results. Every process has its own temporary output directory until the whole process chain is finished. After that the temporary folders get deleted and only the result is persisted in the job directory. To achieve the job provenance data capturing described in chapter 4.3.2, the processes must provide meta-data about the input data, the output data and the processing itself.
Two different implementations were created to achieve the job dependent provenance capturing. The first implementation uses the python wrapper noworkflow [24]. Instead of calling the processing code with the python interpreter, the noworkflow wrapper gets called. It automatically creates a meta-data database regarding the python execution. Since the wrapper is capturing the provenance in a detailed way, which influences the performance (see 6.5), a second more simpler implementation was done. The new implementation adds the specific information of interest to the logging of the processing

Table 5.2: Relation of context model definition and EODC JSON context model.

Context Model Definition	JSON Key
J1: Input Data Persisted Identifier	input_data
J2: Back end provenance / Code Identifier	backend_env
J3: Programming Language	interpreter
J4: Installed packages of the programming language	code_env
J5: Start and End time of process execution	start_time, end_time
J6: Result Hash	output_data

and reads the logging files to generate the context model. The solution uses in plain python code and is used for the evaluation of this thesis. In figure 5.7 an overview of the job capturing work-flow is presented. The single parts of the overview are described in more detail in section 5.3.2.

5.3.1 Provenance Repository

There is a context model created for each executed job. If the job gets re-executed, the context model gets replaced by the new context model. The job re-execution is internally handled as a new job with the same process graph. The functionality of letting the same job be re-executed without creating a new job id got dropped from the agenda of the OpenEO project in version 0.0.2 [26]. The context model is stored as a JSON object in the job execution meta-database. After the job is carried out in the EODC back end the results are saved in a folder named after the unique job identifier and the meta data information is stored in a PostGreSQL database (see section 3.5.3). Jobs are saved in the Job table of the database and in this solution the context model is an additional column of it. The context model JSON object gets created by the implementations described below.
In Table 5.2 the mapping between context model definition from section 4 and key of the JSON context model object is provided. 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 time stamps are already part of the Job table in the EODC meta-database. In the current system they are added to the context model for the reason that it is not guaranteed from EODC that the timestamps of the Job table do not change in the future. In figure 5.6 there is an example context model on how the user sees it. It consists of all points described for the context model in the Design section. The back end environment during the execution of the job is persisted in the context model. In the context model the back end version and the back end provenance is stored to be independent from the back end version persistence. The code environment is a list of all python dependencies with their versions installed in the execution environment. In addition the python interpreter version is added to the JSON object. How the data is captured in detail is described in the sections below.

```
{
    "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 5.6: Example context model of a job execution at the EODC back end.

5.3.2 Python Implementation

The EODC implementation proposed by this thesis is an example for other back ends and hence needs to be easy to implement and maintain. Therefore the implementation is done in python without any additional requirements to the EODC back end provider.The python solution uses logging messages to transfer the needed data from the process execution program to the capturing tool. The capturing tool is started by the cleanup service of the EODC back end driver. It is simply an additional python module and reads the log files after the job is finished. This solution leads, except for the additional logging calls, very little impact on the existing back end implementation. The EODC back end driver has already a logging system installed, hence the modifications can be added in the existing execution policy.
In figure 5.7 the additional python module *Job Capturing* and *Result Handler* are visualized in the context of the back end environment. In the following the work-flow of the solution at the EODC back end driver is described.

1. **Process Execution**
   The *Process Execution* module at the EODC back end is responsible of the actual execution of the job. There the additional logging information is added (J1, J3, J4 and J5) and persisted in the logging file of the job.

2. **Processing Cleanup**
   After the job execution is finished, the temporary folders are deleted from the file

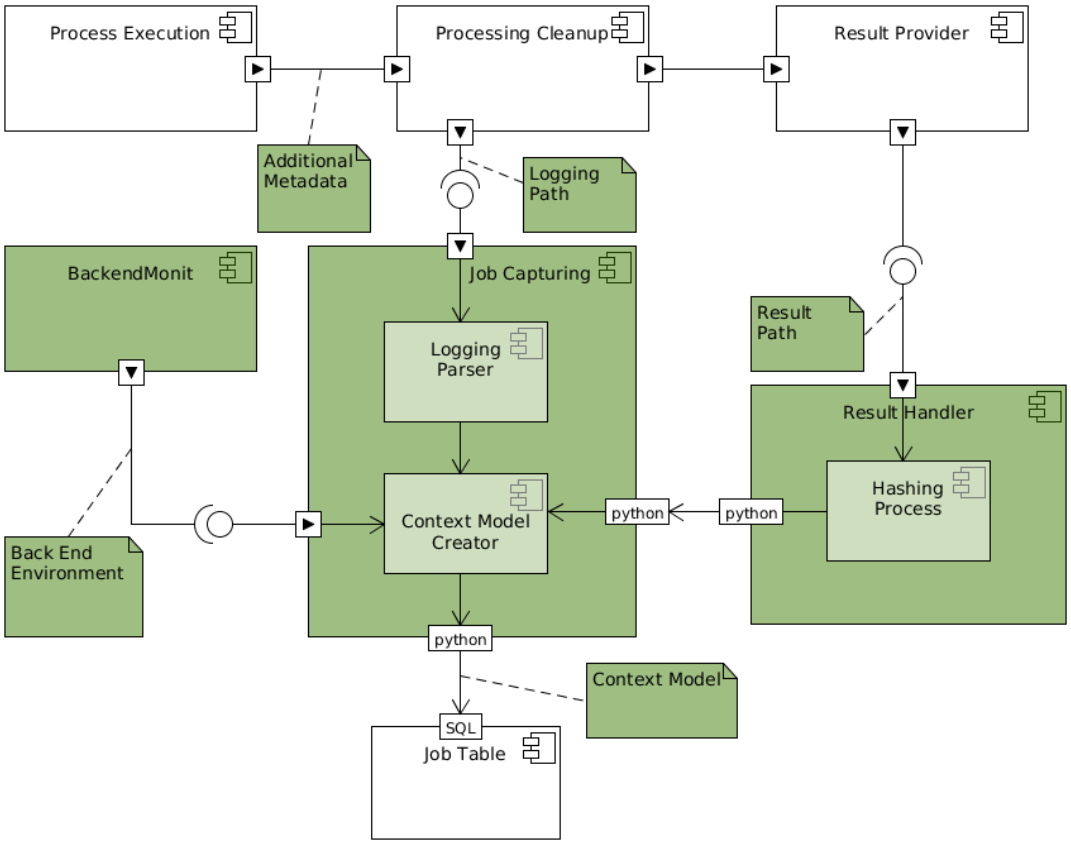

Figure 5.7: Overview of the Job capturing architecture at the EODC back end

system and the result is copied to the persisted job folder. There the additional *Job Capturing* module is started with the path of the logging file.

3. **Result Provider**
   The *Result Provider* is responsible of making the result available for the user and providing the user with the feedback of the finished job. In addition it invokes the added *Result Handler* module with the path of the result file.

4. **Result Handler**
   The *Result Handler* is reading the result file and processes an SHA-256 hash over it. This is then provided to the *Context Model Creator*.

5. **Job Capturing**
   The *Job Capturing* module parses the provided logging files for the information needed for the context model (J1, J3, J4 and J5) and passes the formatted information to the *Context Model Creator*. The *Context Model Creator* demands the *Result Handler* (J6) and *BackEndMonit* (J2) for the remaining parts of the context model.

After receiving all necessary information the JSON context model is created and stored to the Job table in the EODC meta-database.

In the following the capturing of the single data elements of the job dependent context model are described in more detail.

J1: Source Input Data Identifier
The source input data identifier is the id for the input data and query provided by the EODC query store described in section 5.1. It is forwarded to the *Job Capturing* module by the *Job Execution* module.

J2: Back end provenance / Code Identifier
The back end provenance is read from the *BackendMonit* context model JSON file at the back end described in section 5.2. The latest back end version during the execution gets copied to the context model. The whole back end provenance gets included to the context model so that the environment is independent of the persistence of the back end provenance file.

J3 and J4: Programming Language and Installed packages of programming language
The programming language of the EODC back end is python and also the capturing tool is written in python. To list all installed packages in python the already installed *pip module* is used. It is used to manage the python packages of the back end. The EODC back end GitHub repository includes a python environment file to automatically install all needed dependencies of python via *pip*. A feature of that tool is the *pip freeze* call, which returns all installed python packages including their concrete versions. This gets transformed to a JSON list object and saved to the context model. In addition to this the python version gets captured by the *sys.version* call. All of this executions are done int the *Process Execution* module, in the actual processing environment and stored in the output log file of the job execution.

J5: Start and End time of process execution
The start and end time of the process execution is already done at the EODC back end in the *Process Execution* module. The resulting time stamps are persisted in the newly created Job table entry. For the context model the values are added to the output logging file right after they are created. These logging entries are not added by this solution, because they were already implemented by the EODC back end provider.

J6: Result Identifier
The result identifier consists of the resulting data of the whole process graph execution. It is a SHA-256 hash (using the *hashlib* python library) of the resulting sorted output files, which are placed in the resulting folder of the job execution directory. In the current version of the back end there is only one result file created, due to the limitations of the current available processes.

5.3.3 Alternative Implementation

Noworkflow is a python module used to capture the provenance of a python script execution. The main aim of it is that the program itself does not have to be changed, it only has to be executed with the noworkflow interpreter instead of python. It then automatically creates a provenance database on every execution of the script called "trial". Every time the job gets executed a new trial is created. A trail in the past can be re-executed with the same conditions and trails themselves can be easily compared and visualized with noworkflow (see [24]). Noworkflow and its capabilities are described in section 2.3.1. For the purpose of the job dependent provenance capturing described in the Design section, only parts of the provenance capturing functionality that noworkflow offers is suitable. In the following the features of noworkflow used in the alternative implementation that differ from the actual solution at section 5.3.2 are presented.

J2: Back end provenance / Code Identifier
The job identifier is defined by the back end version, which includes the current state of the locally checked out GitHub repository.
To make a more detailed comparison possible, noworkflow captures the hash of the executed code. The hash from the database of noworkflow gets retrieved via the command line interface. It contains the command "show" (see code block below), which returns the data of a specific trial. The code hash can get extracted from the output and added to the context model for easy an comparison.

```
now show --dir=JOB_DIR
```

J3 and J4: Programming Language and Installed packages of programming language
The programming language gets captured with the modules of the python packages in the noworkflow database. To retrieve the modules the command line interface is used with the command "-m" (see code block below) that returns all used python modules including their versions and the version of python. The advantage of using noworkflow instead of the *pip* tool described in the python solution is that only the used packages are listed.

```
now show -m --dir=JOB_DIR
```

5.4 User Interface

In the previous sections only the technical insight of the back end is described. In this section the implementation of the interfaces to the user gets described. Therefore, endpoints are added to the current existing OpenEO core API version 0.0.2. In addition, the proposed endpoints are applied to the python client and the EODC back end. The relevant additions defined in section 4.6 are implemented in the following way:

U1: Back end version
Retrieving additional information about the back end was added into a new end point of

the API. The new endpoint is a GET request called “/version” and no authentication is needed to access it. Response of the version endpoint is the whole job independent provenance information of the back end (B1-B5, see section 5.2). In a productive version the data may be filtered for information marked as a security risk. This endpoint is added to the EODC back end to return the latest back end version including meta-data about it. Further additions are created to the python client, so that the user can call a method in the python client called “version()” to retrieve the information of the back end directly in the python environment. The result is a JSON object consisting of the back end provenance data similar to figure 5.5.

U2: Detailed Job Information
In the OpenEO coreAPI there is already an endpoint to get detailed information about the job status. The endpoint path is “GET /jobs/<job_id>” , which by the current release version of 0.0.2 only contains the execution state of the job and the job id. In addition to this the available information of the whole job dependent provenance gets added to this endpoint in the implementation of this thesis (e.g. see 5.6). Since the python client just returns the resulting JSON response from the back end as a python dictionary, there is no modification needed.

U3: Comparing two Jobs
The comparison of two jobs is not defined in the version 0.0.2 of the core API, hence there does not exist any user interface. A new endpoint is defined in the manner of existing endpoint definitions. For the purpose of this thesis the endpoint “POST /jobs/<job_id>/diff” gets introduced to the core API. In the url of the request the user defines the base job id, which context model will be compared with other jobs. In the body of the request the target job ids are defined in a JSON list. After getting the request from the user, the back end compares the context models of the base job with every target job occurring in the request body list. The result from the back end consists of, for every item in the base job context model, the term “EQUAL”, if the item is the same in both context models, “DIFF” if the items are not the same in both context models, or “MISSING” if the item is in the base job context model, but missing in the target job context model. The latest mentioned outcome can occur if the context model definition is modified in future job executions and there are e.g. additional fields of it. The response contains the context model of all jobs with one of the previously described three states inside of every items value. In the python client this feature is added with an additional function of the Job class called "diff(target_job)". In figure 5.8 an example of the dictionary output of this function is provided.

U4: Data Identifier Landing Page
Depending on the back end, the input data may be restricted to the OpenEO interface. Therefore, the resolver of the input data PID is set within the core-API specification. In the coreAPI release version 0.0.2, there exists an endpoint to retrieve detailed information about a data set. The "GET /data/<dataset-id>" endpoint is extended, by also accepting query PIDs. If the user calls the endpoint with a data PID, the result is the description of the underlying dataset and in addition the result of the query execution and the original

```
{
    "base_job": "TEST",
    "target_jobs": [
        "TEST2": {
                "backend_env": "EQUAL",
                "code_env": "EQUAL",
                "interpreter": "EQUAL",
                "input_data": "DIFF",
                "job_id": "DIFF",
                "start_time": "DIFF",
                "end_time": "DIFF",
                "output_data": "DIFF",
                "process_graph": "DIFF"
            }
    ]
}
```


Figure 5.8: JSON Example of a job comparison result.

query parameters.

U5: Re-use of Input Data
To re-use the input data in a different job execution, the data PID can be used 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 EODC back end automatically applies the queries in a way of the original execution. To pass the data PID to the process graph, it is inserted in the "product_id" entry. Examples of such a process graph are in section 6.2.

CHAPTER 6

Evaluation

In this chapter the prototype implementation described in the previous section is evaluated. The work-flow of the evaluation is applying the three use cases defined in section 1.4 to the solution of section 5.3.2 and evaluate if the outcome reaches the expectations. First the set up of the evaluation environment is getting described in section 6.1.
In the following sections the use cases are tested against the implementation solution.
The last section in this chapter presents a comparison between the used python implementation and the noworkflow implementation described in section 5.3.3 on different aspects.

6.1 Evaluation Setup

The evaluation setup for testing the prototype of this thesis is provided by the EODC back end providers. It is a duplicate of the productive endpoint of the OpenEO back end driver at the time when the thesis is written. EODC uses the Open Shift service to provide the back ends functionality. It uses the version provided at GitHub from EODC to build the local instance of the EODC back end. In this setup a GitHub fork of the EODC back end is used to deplay the adapted back end described in section 5. Since the used back end instance is a duplicate of the actual productive service, the data used in the back end is the actual data that EODC provides for the OpenEO interface. This allows for a unrestricted evaluation of the use cases, because if the changes suit EODC, it can switch easily the productive instance with this test instance.
The Evaluation is done from a user point of view, hence an OpenEO client needs to be used for the evaluation. The python client is chosen for this purpose with the additional functionality described in section 5.4.

6.2 Re-Use of Input Data

The first scenario describes the process of a researcher using OpenEO as processing environment. In this use case the focus is on the data identification and data citation part of the solution. Researchers that use OpenEO may want to cite the data that is used in the applied process chain. Other scientist then may want to use this data in their related research experiment. The step-by-step description of the scenario can be viewed in the section 1.4.1. In this section the steps of the use case are executed at the evaluation environment.

1. **Researcher A runs an experiment (job A) at the EODC back end.**
   This step is basic OpenEO functionality and is not influenced by the solution from the user point of view. At the back end the Query Handler is generating a new data PID or returns an already existing one if the same query got executed in the past. The following python client code represents the experiment of Researcher A. The last two function calls are sending the process graph to the back end and start the job execution. After the execution the user retrieves a message that the job finished successfully.

```
import openeo
session = openeo.session(EODC_USER_A, endpoint=EODC_DRIVER_URL)
# Authenticate with bearer token
token = session.auth(EODC_USER_A, EODC_PWD_A, BearerAuth)

s2a_prd_msil1c = session.image("s2a_prd_msil1c")
timeseries = s2a_prd_msil1c.bbox_filter(left=652000,
                        right=672000, top=5161000,
                        bottom=5181000, srs="EPSG:32632")
timeseries = timeseries.date_range_filter("2017-01-01",
                                          "2017-01-08")
timeseries = timeseries.ndvi("B04", "B08")
composite = timeseries.min_time()
jobA = timeseries.send_job()
status = jobA.queue()
```

2. **Researcher A retrieves the used input data of job A.**
   If Researcher A wants to receive the input data identifier, the job description endpoint has to be called. The following code snipped shows the call using the python client.

```
job_details = jobA.describe_job()
input_data = job_details["input_data"]
print(input_data)
# Output: EODC_DRIVER_URL/data
#                         /Q-e5a3409e-fa48-41b2-8043-5b9c2b5a0b23
desc = session.describe_collection(input_data)
query = desc["query"]
files = desc["input_files"]
```

The job_details variable is an unfiltered python dictionary object that contains the response of the EODC back end. It has a data element at the key of "input_data" that consists of the input data PID as a resolvable web address. After calling the page, the user is provided by the information of the data set (e.g. "s2a_prd_msil1c"). In addition the resulting files of the query and the query (filter arguments) are also added to the website.
To gather the information about the input data directly in the python client code, the last three calls of the code block above can be used.

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 PID from step 2 for the citation.

4. **Researcher B uses the same input data of job A for job B.**
   In order to use the same input data as Researcher A, Researcher B just copies the data PID from the publication and puts it into the process chain of job B. In the block below an example of code needed to use the same input with a different process chain (different band configuration) is printed.

```
import openeo
session = openeo.session(EODC_USER_B, endpoint=EODC_DRIVER_URL)
# Authenticate with bearer token
token = session.auth(EODC_USER_B, EODC_PWD_B, BearerAuth)

s2a_prd_msil1c = session
                .image("Q-e5a3409e-fa48-41b2-8043-5b9c2b5a0b23")
timeseries = s2a_prd_msil1c.ndvi("B02", "B04")
composite = timeseries.min_time()
jobB = timeseries.send_job()
status = jobB.queue()
```

6.3 Capturing job dependent environments

In this scenario the scope is on the execution environment. It handles the need of researchers to describe the execution process. The implementation at the EODC back end gives the users the option to gain additional meta-data about the execution. In addition the advantages for the back end provider gain additional information about job executions are part of the use case. In the following the steps of the scenario are run with the implemented instance of the EODC back end.

1. **Researcher runs an experiment (job A) at a back end.**
   The researcher A runs a job at the EODC back end with the following python client code. This is the default workflow of executing a job using the python client.

```
import openeo
```

```
session = openeo.session(EODC_USER, endpoint=EODC_DRIVER_URL)
# Authenticate with bearer token
token = session.auth(EODC_USER, EODC_PWD, BearerAuth)

s2a_prd_msil1c = session.image("s2a_prd_msil1c")
timeseries = s2a_prd_msil1c.bbox_filter(left=652000,
right=672000, top=5161000,
bottom=5181000, srs="EPSG:32632")
timeseries = timeseries.date_range_filter("2017-01-01",
"2017-01-08")
timeseries = timeseries.ndvi("B04", "B08")
composite = timeseries.min_time()
job = timeseries.send_job()
status = job.queue()
```

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. The following code-lines provide the user with detailed information about the job execution:

```
job_details = jobA.describe_job()
context_model = job_details["context_model"]
interpreter = context_model["interpreter"]
code_env = context_model["code_env"]
backend_version = context_model["backend_env"]["backend_version"]
```

   After the execution of the lines above, the researcher is able to get the used interpreter including the version and installed packages. In addition the back end version describes the processing code. For more transparency the researcher can link to the used GitHub repository and the commit used for the experiment.

3. **Back End Developer releases a new version.**
   This step is independent from the implementation and therefore not explained further.

4. **Back End Developer runs test jobs to find differences.**
   With the new provided information about the job execution environment, the back end provider can compare the environment to previous versions of the back end. Using the output hash of the job execution, the back end provider is able to find different outcomes compared to previous versions.

```
jobA = testA.send_job()
status = jobA.queue()
jobA_details = jobA.describe_job()
jobB = testB.send_job()
status = jobB.queue()
jobB_details = jobB.describe_job()
#...
```

6.4 Getting differences of job executions

The last use case is focused on the users of OpenEO. It describes the need of transparency of job executions for the users. If results differ with the same job later in time, the user can access meta-data to find reasons why it happened. Other users might compare different job execution environments.

1. **Researcher runs an experiment (job A) at a back end.**
   The researcher A runs a job at the EODC back end with the following python client code.

```
import openeo
session = openeo.session(EODC_USER_A, endpoint=EODC_DRIVER_URL)
# Authenticate with bearer token
token = session.auth(EODC_USER_A, EODC_PWD_A, BearerAuth)

s2a_prd_msil1c = session.image("s2a_prd_msil1c")
timeseries = s2a_prd_msil1c.bbox_filter(left=652000,
right=672000, top=5161000,
bottom=5181000, srs="EPSG:32632")
timeseries = timeseries.date_range_filter("2017-01-01",
"2017-01-08")
timeseries = timeseries.ndvi("B04", "B08")
composite = timeseries.min_time()
jobA = timeseries.send_job()
status = jobA.queue()
```

2. **Researcher re-runs the same experiment (job B).**
   Since re-execution of the same job id is handled internally with the re-execution of the same job configuration with a new job id, the the following code is used to create the same job again (job B):

```
jobB = timeseries.send_job()
status = jobB.queue()
```

3. **Researcher runs a different experiment (job C).**
   The third job (job C) is created with a different process configuration and input data query.

```
jobC = timeseries2.send_job()
status = jobC.queue()
```

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 first job A. Therefore the following code has to be executed:

```
diffAB = jobA.diff(jobB)
diffAC = jobA.diff(jobC)
```

With the lines above the researcher gets two dictionaries of 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 with each other. An example of the result can be viewed in figure 5.8.

6.5 NoWorkflow vs Python Implementation

The two implementation approaches are implementing the same features, even though there are major differences between them. In this sections the different implementations get compared and the advantages and disadvantages of them get discussed. Therefore three categories get confronted by both implementations. First the effort of applying the implementation gets compared, then the performance of the python and the noworkflow implementation is compared. The last comparison is about the maintenance and the extensibility of the solutions.

6.5.1 Implementation Effort

The implementation of this thesis applies the purposed reproducibility context model creation only for the EODC back end. There are six other back end providers in the project were there needs to be a solution too. Therefore the effort of the implementation gets graded so that other back end providers can decide, which solution they want to implement.

noWorkflow
The implementation of noworkflow is by the definition of noworkflow itself quiet easy. The EODC back end is a python implemented back end and therefore just the call of the python scripts has to be changed to the noworkflow command line interface, which is a wrapper around python. That was everything that has to be done on the code provenance capturing part. The created database has to be moved to the job output folder. To get the provenance data the command line tool has to be called, so there is some effort to find the needed data in the documentation of the tool. After finding the information it has to get parsed for the information needed for the context model, which takes some effort, but can be done in a view hours.

Python
The python implementation does not affect the processing code of the EODC back end, but adds some additional logging calls to it. So that the actual code has to be modified, which takes longer than the noworkflow implementation. After the logging calls are added to the code the capturing part is done. For retrieving the information of the logging file it has to get parsed to transfer it to the context model of the job. This is easily done, because the added logging messages are only the needed information and it is self formatted so that the parsing is easy to implement.

noWorkflow vs Python
The implementation effort of both solutions are very similar, the noworkflow solution has

Table 6.1: Performance comparison between noworkflow and python implementation

original execution	**noworkflow solution**	**python solution**
4069 ms	11900 ms	4074 ms

the advantage, that the capturing does not require changes in the code. The downside of noworkflow is that it captures much more provenance information than needed for the context model and the needed information has to be extracted from the database in case to get it.

6.5.2 Performance

Most of the OpenEO back end provider have customers that pay by the length of the execution time and therefore the provenance capturing addition has to affect performance as little as possible. Therefore the performance of the two implementations get compared.

noWorkflow
Due to the wrapper around the python interpreter, the performance after applying the solution was very low. The execution time a bit more than three times the original execution without capturing. The fact about the bad performance of noworkflow is the reason for the implementation of the python solution.

Python
The python implementation adds almost no overhead to the execution. The parsing of the log file is the only part of the python code that has an real effect on the performance. Hence, the execution duration using the python solution is only a view milliseconds longer compared to the unmodified execution.

noWorkflow vs Python
The python implementation is by far much more efficient when it comes to performance. The main reason is the wrapper around python and the high amount of captured data in the noworkflow solution of data that is not needed in the context of this thesis. In the table below the timings of the proof of concept execution is recorded. The timing in the table is the time between sending the request and getting response from the server about the finished processing.

6.5.3 Maintenance and Extensibility

OpenEO is still involving and will be in the future due to the rapidly changing earth observation market. So in this section the extensibility and the maintenance of the solutions gets compared.

noWorkflow
NoWorkflow is already capturing much more information about the execution. It also comes with a high amount of features to query the provenance data and to visualize it. NoWorkflow is also still evolving and new features may be added to it in the future. On

the other hand the noworkflow solution is depending on the external maintenance of noworkflow, which is not guaranteed. So it can happen, that noworkflow does not work with a new python version, or if the back end provider switches to another programming language noworkflow can not be used any more. In addition to this disadvantages it is hard to add additional provenance data that is not already captured by noworkflow, because then noworkflow has to be adapted.

Python
The python solution is only capturing exactly the information that the back end provider wants it to capture. On fine granulated capturing and parsing the additional effort can be quite high. The advantage on this implementation is that the tool is easy extensible by just adding more logging information. Since logging is a simple feature of python it can be assumed that it will not change much in the future. If the back end provider decides to switch to a different programming language it can easily switch by adapting the code parsing to the new language and add the logging messages to the new back end implementation.

noWorkflow vs Python
Noworkflow comes with more features, especially for the back end provider to query and visualize the provenance data, but it is much harder to add information to the provenance capturing mechanism. So the python solution is much more flexible on changes and more independent from the environment than the noworkflow implementation.

CHAPTER 7

Conclusion and future Work

7.1 Conclusion

In this thesis, we explored the challenges of providing reproducibility in earth observation science. Solutions for data identification and code identification in the restrictions of the project framework had to be defined. Since the service is open source and an important part of the project is to motivate users to use the framework, one of the challenges was to make the solution as simple as possible. Therefore the solution is a concept for the OpenEO project on gaining reproducibility to the work-flow and input data. The suggested solution consists of light-weight recommendations to the back end design, which can be adapted to the setup of the contributing back ends of the OpenEO project. The implementation at EODC proofs that the concept can be applied. The core advantage for researchers using the features of this thesis is that it happens automatically. Scientists can run experiments on the OpenEO back end and generate automatically a data identifier that other scientist can use directly. In addition the scientists can cite the version of the back end and describe the execution environment, so that researchers wanting to reproduce the result in different environments can at least try to do so. In addition it is possible to compare different job executions to bring more transparency to the users of OpenEO. Without the additional information about the execution the back ends are just black boxes and users may be confused if results are not as expected. Applying the suggested additions is an advantage for the OpenEO project itself since back end providers get additional execution information. In addition the concept of easy to apply data citation and re-use may attract more users to use OpenEO. Still the OpenEO project will involve and the concept has to be adapted to the future changes. Hence the outcome of this thesis can be seen as a starting point of reproducibility in the OpenEO project rather than a final solution.

7.2 Future Work

In this thesis a concept for applying reproducibility in the OpenEO project is purposed. Still there are major issues to be addressed in the future of the OpenEO project. There are several ways to improve the concept that need to be addressed in the future.
OpenEO is still evolving until the official project end in 2021 and afterwards it will be continued. There are a lot of features that have to be defined and may interfere with the concept proposed in this thesis. User defined functions are part of the project, but not well defined yet. To be capable of reproduce them may lead to adaptations to the concept. Another main issue of the OpenEO project is the billing concept that has to be apply-able to a diverse set of back ends. Since the capturing process and the generation of the context model needs resources, it has to be considered in the billing plan of the back ends. The growing diversity of the back ends applying the OpenEO standard is a huge challenge to remain the design of the context model suitable in the future.
The storage of the context models in JSON objects was used in this thesis , but has limitations that have to be addressed in the future e.g. querying over data sets. In the solution of this work the data is captured for identifying how jobs are executed and to be able to identify how the job was executed. An addition to this can be a way to automatically apply the reproduction of an already executed job. To achieve this a method of fully validating the output of a job execution needs to be introduced. The possibility to get automated data and process citation texts are not part of the solution and are therefore a task for future work. Special users (e.g. back end providers) may need a finer granularity of the process capturing, leading to a definition of scalability of the processing capturing. A possible concept achieving this is presented in the section 7.2.1. Within this finer granularity is the limitations of the python environment to the actually used packages rather then the whole environment, like it is in the current implementation.

7.2.1 Benchmarking Mode

So far only the capturing of the whole process chain, including input and output data, are described. The general idea (presented in section 5.3) of the job dependent provenance capturing is to capture the input data of the process graph, the whole process graph and the output of the process graph. This is a common concept the back end providers can agree on, since it is not affecting the back end providers implementations much. It is also capable of giving the OpenEO user a simple overview of how the results are generated and what are differences between different job executions.To make the comparison of different execution behaviors more informative, the granularity of the capturing can be improved.
Therefore, the bench marking context model gets introduced. The idea is that not only the input data of the whole process chain and the output data gets captured, but also every data in between of every process of the process graph. So for every process in the process graph the input data, the output data and the code executing the process gets captured. This makes it easier for users to see where in the process chain the execution actually produced different results. But it also comes with higher implementation and

execution costs at the back end side. The affect on the performance of the execution will be more than the common context model. The possibility of the implementation of such a granularity is also highly dependent on the back end implementation. Not every back end might be able to implement this into the back end due to external tools where the execution and results of the single processes in the process graph are not distinct-able. If a back end can support it the granularity can even be higher. In an extreme example, the data and code could be captured for every line in the code. The flexibility of capturing granularity may define the context model design in the future.

List of Figures

List of Tables

Bibliography

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

[2] Prov model primer. *W3C Working Group Note*, 4 2013.

[3] *PROV-O: The PROV Ontology*. W3C Recommendation. World Wide Web Consortium, United States, 4 2013.

[4] Yin & yang: Demonstrating complementary provenance from noworkflow & yesworkflow. In *Provenance and Annotation of Data and Processes - 6th International Provenance and Annotation Workshop, IPAW 2016, Proceedings*, volume 9672 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 161–165, Germany, 1 2016. Springer Verlag.

[5] 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, 6 2016.

[6] crontab. crontab - files used to schedule the execution of programs. http://man7.org/linux/man-pages/man5/crontab.5.html. Accessed: 2018-10-11.

[7] Debian. About debian. https://www.debian.org/intro/about. Accessed: 2018-11-09.

[8] dpkg. dpkg - package manager for debian. http://man7.org/linux/man-pages/man1/dpkg.1.html. Accessed: 2018-10-11.

[9] I. Emsley and D. De Roure. A framework for the preservation of a docker container. Digital Curation Centre, 2017.

[10] EODC. Github repository of the openeo eodc back end driver. https://github.com/Open-EO/openeo-openshift-driver/tree/release-0.0.2. Accessed: 2018-10-05.

[11] 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. Villamizar, and X. Yu. Toward the geoscience paper of the future : Best practices for documenting and sharing research from data to software to provenance. 2016.

[12] R. HAT. What is openshift. `https://www.openshift.com/learn/what-is-openshift/`. Accessed: 2018-10-11.

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

[14] D. Huo, J. Nabrzyski, and C. Vardeman. Smart container: an ontology towards conceptualizing docker. In *International Semantic Web Conference*, 2015.

[15] inotify. inotify - monitoring filesystem events. `http://man7.org/linux/man-pages/man7/inotify.7.html`. Accessed: 2018-07-09.

[16] inotifywait. inotifywait - wait for changes to files using inotify. `https://linux.die.net/man/1/inotifywait`. Accessed: 2018-07-09.

[17] C. Knoth and D. Nust. Enabling reproducible obia with open-source software in docker containers. September 2016.

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

[19] M. Konkol, C. Kray, and M. Pfeiffer. The state of reproducibility in the computational geosciences. 04 2018.

[20] G. T. Konrad Hinsen, Konstantin Läufer.

[21] 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.

[22] T. Miksa, S. Pröll, R. Mayer, S. Strodl, R. Vieira, J. Barateiro, and A. Rauber. Framework for verification of preserved and redeployed processes. In *iPRES*, 2013.

[23] 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.

[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 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 71–83. Springer Verlag, 2015.

[25] OpenEO. Architecture of the openeo project. https://open-eo.github.io/openeo-api/v/0.3.1/arch/. Accessed: 2018-09-25.

[26] OpenEO. The core api of openeo in version 0.0.2 . https://open-eo.github.io/openeo-api/v/0.0.2/apireference/. Accessed: 2018-09-25.

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

[28] E. Pebesma, W. Wagner, M. Schramm, A. Von Beringe, C. Paulik, M. Neteler, J. Reiche, J. Verbesselt, J. Dries, E. Goor, and et al. Openeo - a common, open source interface between earth observation data infrastructures and front-end applications. Nov 2017.

[29] 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*, IPAW 2016, pages 199–203, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

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

[32] 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.

[33] A. Rauber, A. Asmi, D. van Uytvanck, and S. Pröll. Identification of reproducible subsets for data citation, sharing and re-use. 2016.

[34] A. Rauber, T. Miksa, R. Mayer, and S. Pröll. Repeatability and re-usability in scientific processes: Process context, data identification and verification. In *DAMDID/RCDL*, 2015.

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

[36] C. Schubert. Ccca data centre: Github repository. https://github.com/ccca-dc. Accessed: 2018-11-25.

[37] 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.

[38] 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, June 2017.

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

[40] S. Thomsen. *Cryptographic Hash Functions*. PhD thesis, 2 2009.

[41] 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, Oct 2011.