



TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG

The University of Resources. Since 1765.

Reproducible geoscientific modelling with hypergraphs

By the Faculty of Geosciences, Geoengineering and Mining
of the Technische Universität Bergakademie Freiberg

approved

Doctoral Thesis

to attain the academic degree of

Doctor rerum naturalium
(Dr. rer. nat.)

submitted by **M.Sc. Georg Semmler**

born on the February 11, 1993 in Zschopau

Assessor: Prof. Helmut Schaeben
Prof. Heinrich Jasper
Prof. Martin Breuning

Date of the award: Freiberg, 23st May 2023

Acknowledgements

I would like to express my deepest appreciation to my supervisor Prof. Helmut Schaebe. He mentored this work after his retirement. Without his guidance, advice and support none of this work would have been possible.

This endeavour would not have been possible without the support from my co-supervisor Prof. Heinrich Jasper. We had numerous discussions about content of this work. Without his input this work would not be as detailed as it is today.

Many thanks to the Electromagnetic and Seismic Working Group of the Department of Geophysics at TU Bergakademie Freiberg for providing code, instructions and datasets used as part of the BHMZ case study. I want to thank Dr. Mathias Scheunert for answering my questions about how to combine all the provided data.

Thanks should also go to the Landesamt für Umwelt, Landwirtschaft und Geologie Sachsen for providing data and instructions used as part of the Kohlberg case study. I would like to thank Dr. Ines Görz and Sascha Görne for guiding me through the construction of the subsurface model.

I am also thankful to Dr. Volkmar Dunger for the uncomplicated way in which he provided the source code of Bowahald, which is used as part of the hydrologic balance model case study.

Thanks should also go to my colleagues at GiGa Infosystems for their recurring input to the topic of this thesis. Especially I would like to thank Aleksey Zholobenko for helping to correct the language of this thesis and Paul Gabriel for providing the possibility to finish thesis.

I also wish to thank Anatoly Zelenin and Richard Gootjes for many hours of discussion about the subject of this work and for their encouragement to complete this work.

Thanks should also go to all the other colleagues from university who provided environment which made this work possible.

I would like to acknowledge the financial support to work on this topic provided by the IAMG Computers and Geosciences Research Scholarships and the by the European Social Fund.

I gratefully acknowledge the assistance of the student assistance that help me by the implementation of the presented prototype.

I am also grateful to my partner Sophie for her support and continued encouragement to finish this thesis. Last but not least, I want to thank all my friends and my family for their support, inspiration and encouragement.

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Hilfe eines Promotionsberaters habe ich nicht in Anspruch genommen. Weitere Personen haben von mir keine geldwerten Leistungen für Arbeiten erhalten, die nicht als solche kenntlich gemacht worden sind. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

11. August 2023

M.Sc. Georg Semmler

Declaration

I hereby declare that I completed this work without any improper help from a third party and without using any aids other than those cited. All ideas derived directly or indirectly from other sources are identified as such.

I did not seek the help of a professional doctorate-consultant. Only those persons identified as having done so received any financial payment from me for any work done for me. This thesis has not previously been published in the same or a similar form in Germany or abroad.

23st May 2023

M.Sc. Georg Semmler

Contents

1. Introduction	1
1.1. Survey on Reproducibility and Automation for Geoscientific Model Construction	2
1.2. Motivating Example	3
1.3. Previous Work	7
1.4. Problem Description	7
1.5. Structure of this Thesis	8
1.6. Results Accomplished by this Thesis	9
2. Terms, Definitions and Requirements	11
2.1. Terms and Definitions	11
2.1.1. Geoscientific model	11
2.1.2. Reproducibility	12
2.1.3. Realisation	12
2.2. Requirements	13
3. Related Work	15
3.1. Overview	15
3.2. Geoscientific Data Storage Systems	15
3.2.1. PostGIS and Similar Systems	15
3.2.2. Geoscience in Space and Time (GST)	16
3.3. Geoscientific Modelling Software	18
3.3.1. gOcad	18
3.3.2. GemPy	21
3.4. Experimentation Management Software	23
3.4.1. DataLad	23
3.4.2. Data Version Control (DVC)	24
3.5. Reproducible Software Builds	26
3.6. Summarised Related Work	28
4. Concept	29
4.1. Construction Hypergraphs	29
4.1.1. Reproducibility Based on Construction Hypergraphs	34
4.1.2. Equality definitions	35
4.1.3. Design Constraints	37
4.2. Data Handling	37
5. Design	41
5.1. Application Structure	41
5.1.1. Choice of Application Architecture for GeoHub	44
5.2. Extension Mechanisms	47
5.2.1. Overview	47
5.2.2. A Shared Library Based Extension System	48
5.2.3. Inter-Process Communication Based Extension System	50
5.2.4. An Extension System Based on a Scripting Language	51

5.2.5.	An Extension System Based on a WebAssembly Interface	52
5.2.6.	Comparison	53
5.3.	Data Storage	54
5.3.1.	Overview	54
5.3.2.	Stored Data	55
5.3.3.	Potential Solutions	60
5.3.4.	Model Versioning	63
5.3.5.	Transactional security	65
6.	Implementation	75
6.1.	General Application Structure	75
6.2.	Data Storage	75
6.2.1.	Database	75
6.2.2.	User-provided Data-processing Extensions	78
6.3.	Operation Executor	80
6.3.1.	Construction Step Descriptions	80
6.3.2.	Construction Step Scheduling	82
6.3.3.	Construction Step Execution	85
7.	Case Studies	89
7.1.	Overview	89
7.2.	Geophysical Model of the BHMZ block	89
7.2.1.	Provided Data and Initial Situation	89
7.2.2.	Construction Process Description	90
7.2.3.	Reproducibility	92
7.2.4.	Identified Problems and Construction Process Improvements	92
7.2.5.	Recommendations	94
7.3.	Three-Dimensional Subsurface Model of the Kolhberg Region	94
7.3.1.	Provided Data and Initial Situation	94
7.3.2.	Construction Process Description	95
7.3.3.	Reproducibility	97
7.3.4.	Identified Problems and Construction Process Improvements	99
7.3.5.	Recommendations	100
7.4.	Hydrologic Balance Model of a Saxonian Stream	100
7.4.1.	Provided Data and Initial Situation	100
7.4.2.	Construction Process Description	102
7.4.3.	Reproducibility	107
7.4.4.	Identified Problems and Construction Process Improvements	108
7.4.5.	Recommendations	108
7.5.	Lessons Learned	109
8.	Conclusions	111
8.1.	Summary	111
8.2.	Outlook	112
8.2.1.	Parametric Model Construction Process	112
8.2.2.	Pull and Push Nodes	112
8.2.3.	Parallelize Single Construction Steps	113
8.2.4.	Provable Model Construction Process Attestation	113

Appendix	125
A. Construction Steps for the BHMZ model	125
A.1. stacking	125
A.2. meshing	125
A.3. DC inversion	126
A.4. seismic inversion	127
A.5. visualise BHMZ	128
B. Provided manual for the Kohlberg dataset	136
B.1. 3D-Modell Kohlberg	136
B.2. Kartendaten vorbereiten	138
B.3. Schnittdaten importieren:	139
C. Construction Steps for the Kohlberg model	140
C.1. xlsx -> csv	140
C.2. reproject to utm33	140
C.3. extract (Quartär)	141
C.4. extract (Prequartär)	142
C.5. triangulate	142
C.6. extrude	145
C.7. project	147
D. Construction step definitions for the Hydrologic balance model	149
D.1. Calculate bounding box	149
D.2. Clip DEM	150
D.3. Clip BK50	150
D.4. Calculate aspect	151
D.5. Calculate slope	151
D.6. Calculate slope length	152
D.7. Calculate avg height	153
D.8. Calculate slope per hydrotope	153
D.9. Calculate aspect per hydrotope	154
D.10. Calculate maximal slope length per hydrotop	155
D.11. Calculate hydrotop area	156
D.12. Lookup soil type	156
D.13. Preprocess climate data	157
D.14. Generate bwmhydro files	159
D.15. Generate bwmuse files	162
D.16. Generate bwmlayer files	164
D.17. Run Bowahald	171
D.18. Plotting results	172
D.19. Summarise	175
E. GeoHub User Manual	177
E.1. Overview	177
E.2. Project View	179
E.3. Version View	181
E.4. Operations View	182
E.5. Executor View	185
E.6. Executor configuration	187
E.7. File Kind View	188
E.8. Plugin View	189
E.9. User View	193
E.10. Graph View	194
E.11. Graph Schedule View	201

E.12.	View mode	203
E.13.	Edit mode	204
E.14.	gOcad History Export tool	205

List of Figures

1.1.	Version information of the used gOcad version	3
1.2.	Input parameters for each of the construction steps	3
1.3.	Intermediate and final results as visualised in gOcad	5
1.4.	Two realisations of the described workflow. One surface is coloured grey, the other one red	6
1.5.	Project settings dialog of gOcad	6
3.1.	Example subsurface model in GST Web [38]	17
3.2.	Exemplary subsurface model in gOcad	18
3.3.	A simple example workflow consisting of three construction steps shown as a directed acyclic graph as generated by the <code>dvc dag</code> command on the command line The boxes represent nodes of the graph, while the * * * form edges. Each node represents a construction step, while each edge represents a dependency between these steps. According to the documentation, the visualization assumes a top to bottom directionality.	25
4.1.	Example construction steps	30
4.2.	Order and direction of construction steps is meaningful	30
4.3.	Construction steps are not edges, as shown by the fact that a construction step may require more than one input dataset	31
4.4.	A construction process cannot be represented as a tree, as shown by the fact that the same <i>Spatial context</i> input dataset is used by several construction steps	31
4.5.	Example construction hypergraph for a simplified geoscientific model construction	34
4.6.	Example construction hypergraph for a simplified geoscientific model construction	38
5.1.	Different software architectures	42
5.2.	GeoHub application structure GeoHub uses a micro-service architecture that splits the presentation layer and domain and data source layers between different services. Each part of the domain layer is implemented by a different application. Extensions for equality checking, metadata extraction and construction step execution are provided as different services as part of the application backend.	46
5.3.	Different data kinds stored by GeoHub	54
5.4.	Example construction hypergraph for a simplified geoscientific model construction	57
5.5.	Insert a single dataset This sequence diagram shows how first the input datasets are written to the corresponding datastorage location. Then, each affected construction step is executed to generate the corresponding intermediate and output datasets	66

5.6.	State transition diagram for construction step states	
	The execution of a construction step always starts in the Scheduled state. As soon as the executor starts executing a construction step, the state changes to Started . Depending on the result of this execution, the state changes to Completed if the execution was successful or to Failed if the execution was aborted due to errors.	67
5.7.	Load a dataset	
	This sequence diagram shows how datasets are loaded in two steps. The first step loads the required metadata from the relational database system, while the second step loads the actual geoscientific dataset from the file storage . .	68
5.8.	Update a dataset	
	This sequence diagram shows that similar to a dataset creation first the input datasets are written to the corresponding datastorage location. then, each affected construction step is executed to produce the corresponding intermediate and output datasets. The notable difference to a dataset creation is that only a part of input datasets is needed.	69
5.9.	Delete a dataset	
	This sequence diagram shows the multi step procedure used for performing a hard delete operation. First, the references to all datasets to be deleted are loaded from the relational database. In the next step, this data is removed from the relational database. Finally, the corresponding data is removed from the file storage.	70
5.10.	Simultaneous access to a newly created dataset	
	This sequence diagram shows how GeoHub handles two simultaneous accesses from Alice (creating a new dataset) and Bob (loading this dataset). The sequence diagram splits the backend service into two instances to clarify which operations belong to which user requests.	72
5.11.	Simultaneous access to a deleted dataset	
	This sequence diagram shows how GeoHub processes two simultaneous accesses by Alice (hard deleting a dataset) and Bob (loading this dataset). The sequence diagram splits the backend service into two instances to clarify which operations belong to which user requests.	73
6.1.	Database schema	
	Orange connections represent versioned dependencies between tables black connection represent ordinary table dependencies	76
6.2.	Example construction hypergraph	82
7.1.	A geometric representation of the tunnel surrounding the BHMZ block	90
7.2.	Construction hypergraph for the BHMZ block model	91
7.3.	A visualisation of the interpretations of both geophysical measurement	93
7.4.	Overview map of the study area	95
7.5.	Construction hypergraph for the Kohlberg model	96
7.6.	Calculated point distance between two realisations of the triangulate construction step from the Kohlberg model	98
7.7.	Triangulation differences between two different realisations of the triangulate construction step of the Kohlberg model	98
7.8.	Study area for the Leubsdorf dataset	101
7.9.	Study area for the Langenau dataset	101
7.10.	Construction hypergraph for the Hydrologic balance model	103
7.11.	Median monthly evaporation per hydrotop for the Langenau dataset	104

8.1. Example construction hypergraph for a simplified geoscientific model construction	111
E.2. Main View	177
E.3. Settings drop down menu	177
E.4. Change password dialog	178
E.5. About GeoHub dialog	178
E.6. Side panel	179
E.7. Project View	179
E.8. Create project	180
E.9. Project Info Dialog	180
E.10. Project Edit Dialog	181
E.11. Delete project Dialog	181
E.12. Version view	182
E.13. Operations View	182
E.14. Operations info dialog	182
E.15. Operations edit dialog	183
E.16. Operations delete dialog	183
E.17. Operations create dialog	184
E.18. Executor view	185
E.19. Register executor dialog	186
E.20. File Kind View	188
E.21. Delete File Kind Dialog	188
E.22. Create File Kind Dialog	189
E.23. Plugin View	190
E.24. Plugin Info Dialog	190
E.25. Plugin Add Dialog	190
E.26. User View	193
E.27. Add user dialog	194
E.28. Graph view (Overview)	194
E.29. Graph legend	195
E.30. Graph Export Panel	195
E.31. Graph view with version slider	195
E.32. Graph view (View mode)	196
E.33. Node info dialog	197
E.34. Edge info dialog	197
E.35. Graph view (Edit mode)	198
E.36. Graph edit bar	198
E.37. Upload archive dialog	199
E.38. Add node dialog	199
E.39. Add edge dialog	200
E.40. Edit node dialog	200
E.41. Edit edge dialog	200
E.42. Save changes dialog	201
E.43. Discard changes dialog	201
E.44. Graph Schedule View (Overview)	202
E.45. Revision panel	202
E.46. Graph Schedule View (View Mode)	203
E.47. Node Info Dialog	204
E.48. Edge Info Dialog	204
E.49. Graph Schedule View (Edit Mode)	205
E.50. Manage File dialog	206

E.51.Save revision dialog	206
-------------------------------------	-----

List of Tables

1.1. Answers to question 1 and 3	2
1.2. Answers to question 2 and 4	2
5.1. Three Principal Layers according to Fowler [73] and Haffner [74]	41
5.2. Possibility to implement Requirements based on application architecture. (+ Possible, – Not Possible, 0 Not Relevant)	45
5.3. Comparison of the different approaches based on our criteria (+ Positive, – Negative, 0 Neutral)	53
5.4. Chosen extension mechanisms per extension API	54
5.5. Example key value metadata schema	55
5.6. Incidence matrix belonging to the example construction hypergraph shown in figure 5.4	57
5.7. Required criteria per data kind (+ Required, – Not Required, 0 As Required, x Avoid if Possible)	62
5.8. Criteria per storage solution (+ Positive, – Negative, 0 Neutral)	62
5.9. Chosen Storage solutions for each data kind	63
5.10. Cases considered for concurrent data access patterns	70
7.1. Characteristic hydrologic values, according to BoWaHald	105
7.2. Annual statistical summary of characteristic hydrologic values per hydrotop as produced for the Langenau dataset. All values are in $\frac{mm}{year}$	106
7.3. Yearly statically summary of observed precipitation All values are in $\frac{mm}{year}$	106

List of Listings

1.1. Input points used for the example. This is the literal content of the <code>points.xyz</code> file	4
3.1. Example gOcad macro for performing a DSI interpolation of an imported point set	20
3.2. Example Python script for creating a GemPy model	22
3.3. Example data recorded by <code>datalad run</code>	23
5.1. Example <code>Dockerfile</code> from one of the Docker containers used as part of the case studies	58
6.1. Example query to load all nodes and their generating construction steps belonging to a given construction hypergraph. Bind parameter 1 (\$1) corresponds to the entity ID of the construction hypergraph.	77

6.2.	A query to check if all enforced metadata keys are set for a given JSONB value. Bind parameter 1 (\$1) corresponds to the node entity ID of the dataset to be checked, bind parameter 2 (\$2) corresponds to the version of the construction hypergraph, and bind parameter 3 (\$3) represents the JSONB value containing the attributes to be checked. For the CHECK constraint, bind parameter 3 is replaced with the <code>attribute</code> column of the <code>datasets</code> table.	78
6.3.	Equality check extension interface	79
6.4.	Example construction step description	81
7.1.	Excerpt from the text file with the first arrival times of the seismic measurement	92
7.2.	List of files generated by a first run of the <code>run bowahald</code> construction step as contained in the result TAR archive	107
7.3.	List of files generated by a second run of the <code>run bowahald</code> construction step as contained in the result TAR archive	107
A.1.	Definition of the stacking construction step	125
A.2.	Definition of the meshing construction step	126
A.3.	Definition of the DC inversion construction step	127
A.4.	Definition of the seismic inversion step	128
A.5.	Definition of the visualise BHMZ construction step	136
C.6.	Definition of the <code>xlsx -> csv</code> construction step definition	140
C.7.	Definition of the <code>reproject to utm33</code> construction step	141
C.8.	Definition of the <code>extract (Quartär)</code> construction step	142
C.9.	Definition of the <code>extract (Quartär)</code> construction step	142
C.10.	Definition of the <code>triangulate</code> construction step	144
C.11.	Definition of the <code>extrude</code> construction step	147
C.12.	Definition of the <code>project</code> construction step.	149
D.13.	Definition of the <code>calculate bounding box</code> construction step	150
D.14.	Definition of the <code>clip DEM</code> construction step	150
D.15.	Definition of the <code>clip BK50</code> construction step	151
D.16.	Definition of the <code>calculate aspect</code> construction step	151
D.17.	Definition of the <code>calculate slope</code> construction step	152
D.18.	Definition of the <code>calculate slope</code> construction step	152
D.19.	Definition of the <code>calculate avg height</code> construction step	153
D.20.	Definition of the <code>calculate slop per hydrotop</code> construction step	154
D.21.	Definition of the <code>calculate aspect per hydrotop</code> construction step	155
D.22.	Definition of the <code>calculate maximal slope length per hydrotop</code> construction step	155
D.23.	Definition of the <code>calculate hydrotop area</code> construction step	156
D.24.	Definition of the <code>lookup soil type</code> construction step	157
D.25.	Definition of the <code>preprocess climate data</code> construction step	159
D.26.	Definition of the <code>generate bwmhydro files</code> construction step	162
D.27.	Definition of the <code>generate bwmuse files</code> construction step	164
D.28.	Definition of the <code>generate bwmlayer files</code> construction step	171
D.29.	Definition of the <code>run bowahald</code> construction step	172
D.30.	Definition of the <code>plotting results</code> construction step	175
D.31.	Definition of the <code>summarise</code> construction step	177
E.32.	Listing: Exemplary construction step definition	184
E.33.	Exemplary executor configuration	187
E.34.	API definition for the Plugin interfaces	193

List of Algorithms

1.	Construction step scheduling for the initial execution scenario	83
2.	Construction step scheduling for the updated execution scenario	85
3.	GeoHub executor main loop	87

List of Definitions

2.1.	Definition (Geoscientific Model)	12
2.2.	Definition (Reproducibility)	12
2.3.	Definition (Realisation)	12
3.1.	Definition (Reproducible Build)	26
4.1.	Definition (Datasets)	31
4.2.	Definition (Construction step)	32
4.3.	Definition (Construction process of a geoscientific model)	32
4.4.	Definition (Directed Hypergraph)	32
4.5.	Definition (Construction hypergraph)	32
4.6.	Definition (Path)	33
4.7.	Definition (Input Dataset of the Construction Process)	33
4.8.	Definition (Output Dataset of the Construction Process)	33
4.9.	Definition (Reproducible construction hypergraphs)	35
4.10.	Definition (Bitwise equality)	35
4.11.	Definition (Structural equality)	36
4.12.	Definition (Distance-based equality)	36
4.13.	Definition (Geological equality)	36
5.1.	Definition (Version)	64
5.2.	Definition (Revision)	65

1. Introduction

As more computing power has become available in recent years, researchers are increasingly turning to digital models to answer important questions. The International Journal of Geomathematics [1] summarises the reasoning for the usage of digital models in their journal scope as follows:

“As in many areas of science, mathematics has also gained an unprecedented importance in geosciences. The complexity of the processes within the Earth, at its surface, and in the atmosphere can only be described, modelled, mapped, and understood by means of modern mathematical methodologies”

Digital models turn these mathematical descriptions into virtual representations. They are easier to inspect and manipulate than their physical natural counterpart. However, they make it difficult for others to reproduce research results based on such models. The ability to reproduce the research results of others is one of the cornerstones of modern science. As a result, some speak of a reproducibility crisis [2]–[5], as it has become difficult to reproduce research results based on digital models. The main problem is that in addition to reproducing the actual research, the underlying digital models must first be reproduced. However, this raises several potential problems. Specifically, it requires the availability of the information used to build these models. The data used to design the model is not the only factor that can affect the result. Both the software used and computational environment can contribute to variability too.

Geoscientific models come in a wide variety [6]. Common to all is the use of georeferenced data to understand and model the inner workings of an earth system. Common examples are:

- Three-dimensional subsurface models [7], [8]
- Reservoir models [9], [10]
- Geophysical inverse models [11]–[13]
- Hydrogeological flow models [14], [15]

Each of these models is built by combining various partially georeferenced input datasets using domain-specific operations.

To understand how a particular model was constructed, we need to answer the following key questions:

- What data were used to design a particular geoscientific model?
- How are these data combined into the geoscientific models?

Konkol et al. [5] tried to reproduce the results of 41 geoscientific publications. They already limited their selection to publications where code and data were openly available and where R [16] was used as the programming language. They state that they encountered serious problems with the provided code in 22 publications, with anything that required a deeper understanding to solve was classified as a serious problem. In addition, at least five publications contained system-dependent code, which means they could not run the provided code in their

runtime environment. They were able to replicate 97 figures from 28 publications. Of these 97 figures, 46 figures contained differences at a deeper level. That is, they contained different curves or other important features did not appear as expected. In our opinion, this study highlights a major problem with the current way of publishing results, especially since the study already limits the set of publications to a subset where it seemed possible to reproduce anything at all.

1.1. Survey on Reproducibility and Automation for Geoscientific Model Construction

To gather more evidence that reproducing geoscientific models is a real problem, we conducted a small non-representative survey among participants at a conference about 3D subsurface model construction. The participants work for various geological surveys and universities across Europe. A total of 16 people participated in our survey. We asked the following questions:

1. How important is reproducibility when creating geomodels?
2. How important is automation when creating geomodels?
3. Are you already using software to automate the creating of geomodels?
4. Do you already use software to test the reproducibility of self-made geomodels?

Table 1.1.: Answers to question 1 and 3

Question	Important	Not Important	Did not answer
1	14	1	1
3	16	0	0

Table 1.2.: Answers to question 2 and 4

Question	Looking for a solution	Existing software	Manual testing
2	8	0	3
4	8	8	0

Table 1.1 lists the responses on how important reproducibility and automation are to the participants. Table 1.2 breaks down how many participants already use existing software in the relevant area and who is looking for solutions.

Almost all participants agree that reproducibility is an important topic. Interestingly, most participants who indicated that reproducibility is important were looking for a solution to check their models. At least three participants appear to perform manual checks. Automatic model construction is also considered important by all participants. About half of the participants already use existing software in this area. Participants cite gOcad [17], ESRI ArcGIS [18], GemPy [19], Python [20] and Microsoft Excel as possible solutions. We consider the consensus about automation unsurprising at all, as our participants work regularly on creating geoscientific models using the same methods repeatably. We assume that this might not the case for applications, where geoscientific models are created as part of research, such as testing out geological hypothesis by constructing complex subsurface models.

Overall these results confirm our assumption that verifying the reproducibility of geoscientific model construction is an unsolved problem that deserves to be addressed.

1.2. Motivating Example

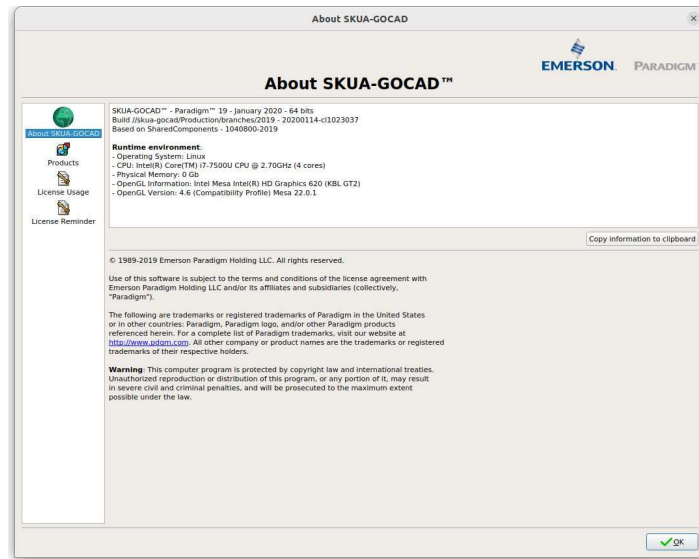


Figure 1.1.: Version information of the used gOcad version

The solution to the outlined problem is anything but simple. In the context of this section, we would like to present a small case study that shows how difficult it can be to provide a workflow description that contains all relevant details in order for the construction process to be reproducible. Our example refers to a discrete smooth interpolation (DSI) [21] using gOcad [17]. This operation is commonly used to generate 3-D subsurface models. We document the entire process with as much information as possible in the form of text and screenshots.

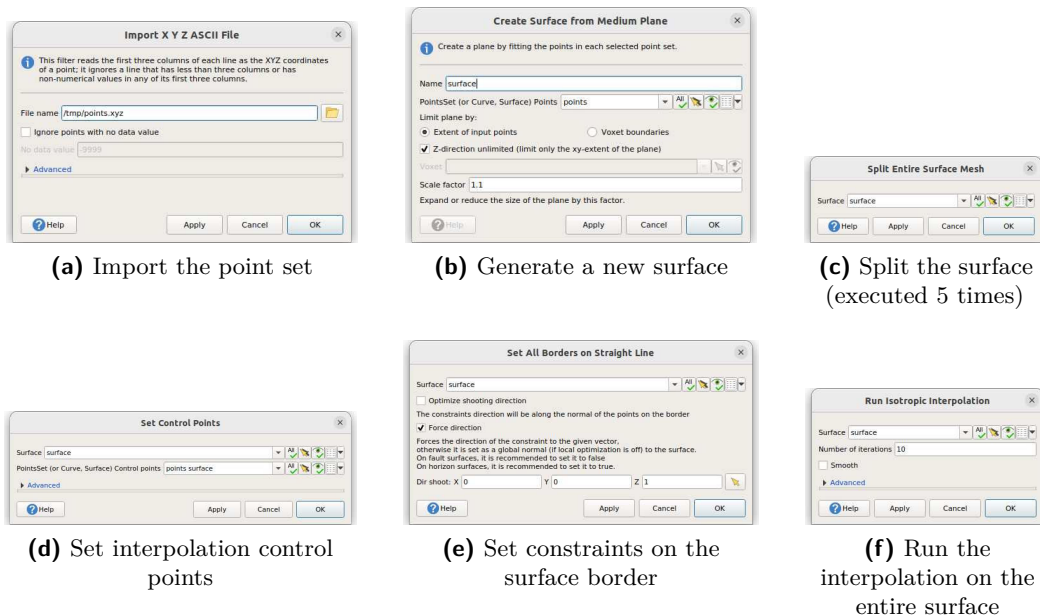


Figure 1.2.: Input parameters for each of the construction steps

```

424312.000000 5643124.000000 145.369995
425124.000000 5643812.000000 184.960007
424374.000000 5643688.000000 137.070007
424810.000000 5643124.000000 179.240005
424622.000000 5642124.000000 197.339996
424874.000000 5642686.000000 183.910004
425624.000000 5643186.000000 168.070007
424810.000000 5643624.000000 157.919998
424374.000000 5642812.000000 154.649994
425124.000000 5643312.000000 171.550003
425686.000000 5643626.000000 130.320007
425874.000000 5642186.000000 189.839996
425810.000000 5642622.000000 170.589996
424124.000000 5642186.000000 173.429993
425186.000000 5642124.000000 203.550003
425812.000000 5642874.000000 161.440002
425312.000000 5642374.000000 192.509995
424186.000000 5642624.000000 158.220001
425124.000000 5642812.000000 179.830002
425874.000000 5643936.000000 129.149994

```

Listing 1.1.: Input points used for the example. This is the literal content of the `points.xyz` file

We have performed all described steps with SKUA-GOCAD version 19. The exact version information can be found in figure 1.1. Listing 1.1 contains the content of the file `points.xyz`, which was used as the input for our example.

To build a triangulated surface based on this set of point, we performed the following construction steps:

1. Import point file via File → Import → Horizon Interpretations → X Y Z (Figure 1.2a)
2. Generate a new surface via Surface → New → Points Medium plane (Figure 1.2b)
3. Split the surface into smaller triangle segments (5 times) Surface → Tools → Split → All... (Figure 1.2c)
4. Add Control point to the surface via Surface → Constraints → Control Points → Set Control Points (Figure 1.2d)
5. Add Border constraints to the surface via Surface → Constraints → Constraints on Border → (Set on straight Line) All Borders (Figure 1.2e)
6. Run discrete smooth interpolation (DSI) via Surface → Interpolation → On Entire Surface (Figure 1.2f)

Figures 1.3a, 1.3b, 1.3c, 1.3d contain the intermediate results after the corresponding construction steps. Figure 1.3e contains the final surface after interpolation.

In order to check whether the described construction process is reproducible, we repeated all steps in the currently existing gOcad instance. Figure 1.4 contains images of the two surfaces in the same view. Figure 1.4a indicates that the two surfaces do not overlap perfectly. Figure 1.4b shows an example gap between the two surface realisations. This result shows that even when using the exact same environment, the same construction steps, and the same input data, it is not possible to reproduce the exact same result due to a non-deterministic construction step.

As a second way to check the reproducibility of the construction steps, we tried to perform the same construction steps on another computer with the same gOcad version. The first

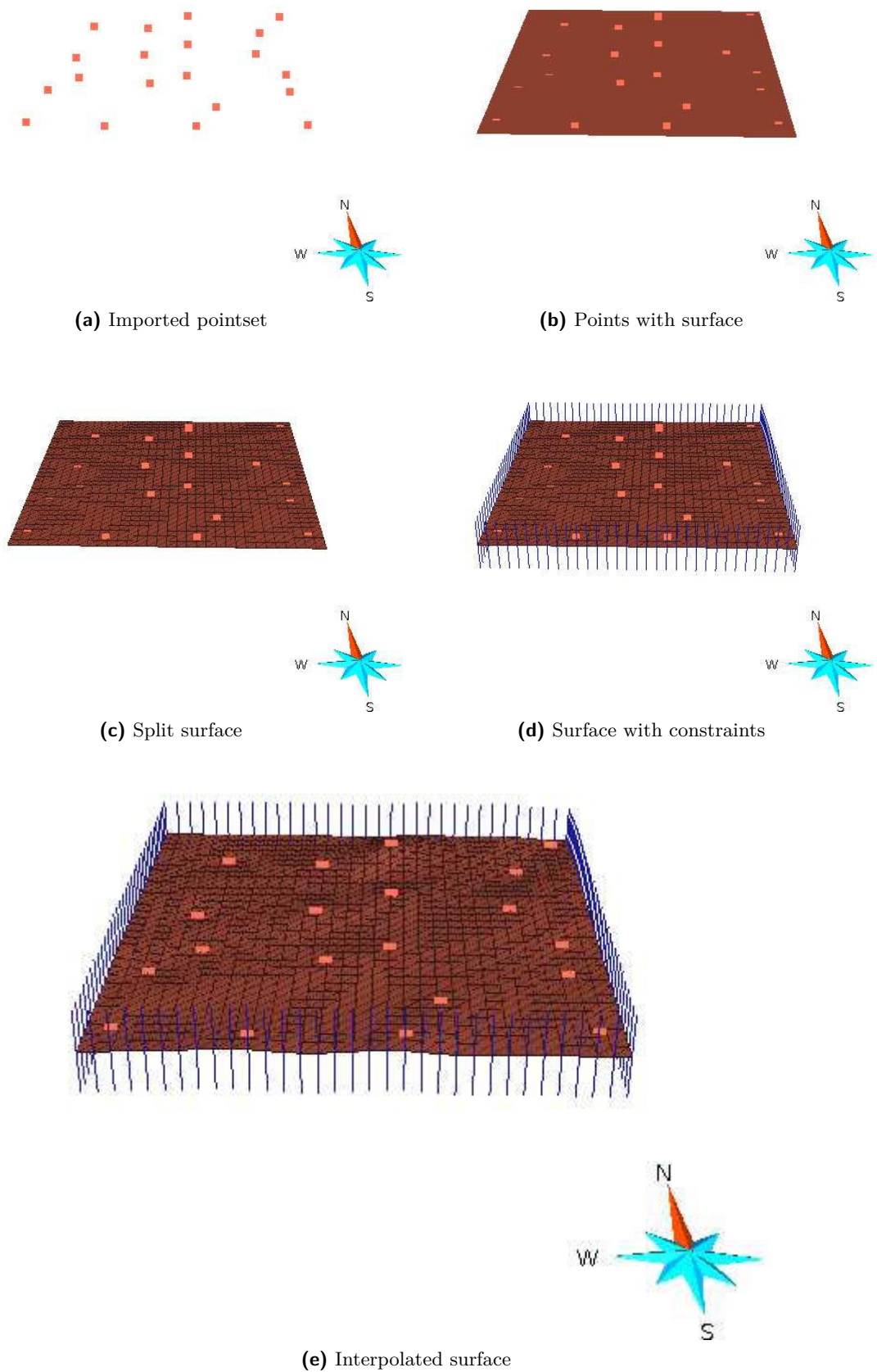


Figure 1.3.: Intermediate and final results as visualised in gOcad

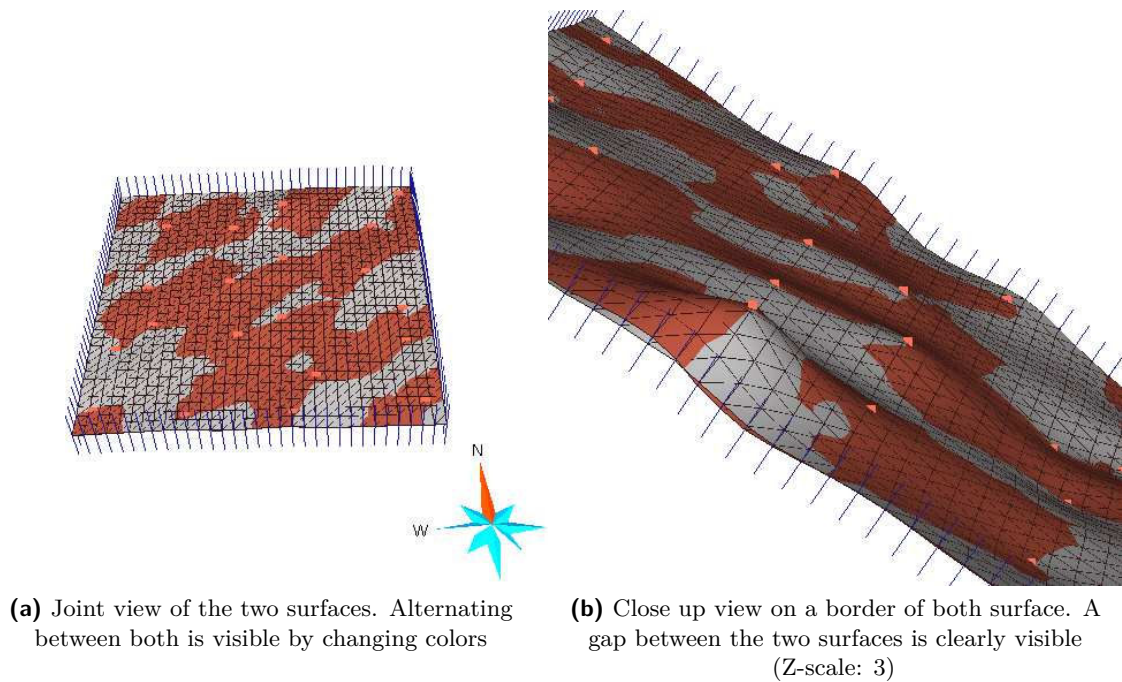


Figure 1.4.: Two realisations of the described workflow. One surface is coloured grey, the other one red

problem occurs as soon as we configure a new gOcad project. Figure 1.5 shows the project settings dialog of gOcad. In this dialog we can make various settings for the general setup of gOcad projects. Each of these settings has a major impact on how data is interpreted by gOcad. We have not documented these settings above.

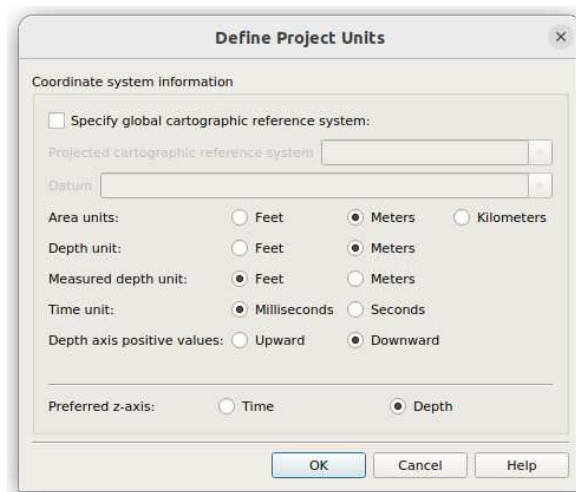


Figure 1.5.: Project settings dialog of gOcad

Overall, this example shows how easy it is to miss important input data even with extensive documentation of the individual construction steps. Furthermore, it is shown that even if all data are available, an attempt to reproduce the model can fail because the construction process itself is not deterministic.

1.3. Previous Work

The current state of the art in reproducible geoscientific model construction comprises of nothing more than comprehensive documentation of the work performed to construct the model. This, in addition to providing the appropriate input data, may be sufficient to enable third parties to reproduce a geoscientific model construction. As our small example has shown, there is plenty of room for mistakes to happen. Such errors include missing data, incomplete documentation of the performed construction steps, or different software versions.

In the field of geoscientific modelling there are many software solutions. Within this thesis, we will present different solutions for the storage and construction of geoscientific models. For each type of software, two typical examples are presented. As the previous example has shown, it is necessary to have both components in order to reproduce the construction of a geoscientific model construction. Such reproductions require access to the appropriate input data and to a detailed description of the construction steps to be performed. None of the presented software solutions supports all requirements in one package.

In addition to geoscientific software packages, we present work from different application areas to show how a similar problem is solved in a different setting. These areas include machine learning, medical computing, and software development. Each of the software solutions presented is specifically tailored to the application domain. While this specialisation makes it difficult to reuse these software packages for geoscientific model construction, they can still be used as inspiration for implementing a solution for geoscientific models.

1.4. Problem Description

Different sources define reproducibility in different ways [5], [22]. They all agree on the following basic points regarding digital models:

- The initial set of data must be provided
- Repetitions with the same or a different software must produce the same result.

Consequently, this information must be available in some way to reproduce the construction of a geoscientific model.

The usual approach to answering these questions is to write a report that includes all relevant details on how the geoscientific model was constructed. This report may include references to the published data and a detailed description of the methods used to create the model. Depending on the author, this information may be incomplete. It usually contains everything that seems relevant from the author's point of view and may or may not contain all information relevant to the actual reproduction of the models. As Konkol et al. [5] and our small example in section 1.2 show, there is a high probability that important information is missing.

In addition, written reports have the problem that it is difficult to verify that a given set of instructions and data is actually sufficient to reproduce the construction of a model. In most cases, one has to sit down and manually repeat the construction based on the given description. Then, the results need to be compared. Such a manual reproduction is a cumbersome process that is often not carried out.

A better solution provides an automatic indicator of whether a geoscientific model construction is reproducible. Such a solution must manage all the information required to ensure

reproducibility. In addition, it is helpful to provide detailed information on the reason for potential non-reproducibility.

With this thesis, we want to present a framework that is able to:

- Provide information about the provenance of a geoscientific model by showing the corresponding input data
- Represent reproducible geoscientific model construction workflows in a generic way
- Manage all required data, including input data, information about the construction workflow, and possible results for later comparison
- Verify the reproducibility of a stored construction workflows without the need for manual reconstruction of the corresponding geoscientific model

1.5. Structure of this Thesis

This thesis consists of 8 chapters. The overall work presents a framework for describing reproducible geoscientific construction processes based on construction hypergraphs. Furthermore, a prototypical software implementing this framework is presented.

The Introduction (Chapter 1) provides a general overview of the research area. Furthermore, the main research problem of this thesis is presented here. Finally, this is where the structure of the work and the results are summarised.

In chapter 2, “Terms, Definitions and Requirements,” important terms and definitions are introduced. In addition, the most important requirements for the to-be-developed framework are presented.

In chapter 3 “Related Work” we present specific software packages, including several examples of established geoscientific data storage systems, geoscientific modelling software, and experimentation management software. In addition, we present existing work for reproducible software builds because the underlying problem is similar.

Chapter 4 “Concept” introduces the general idea of how to represent reproducible geoscientific model construction processes as hypergraphs. In addition, we present a mechanism to verify that a geoscientific construction process described as a construction hypergraph is reproducible.

Chapter 5 “Design” presents various ways in which our concept can be turned into a working software implementation. We examine various possible application structures, extension mechanisms and solutions for storing data. Finally, we present a way to keep data stored in different locations synchronised in order to always present a consistent view of the stored data.

Chapter 6 “Implementation” contains details about the practical implementation of our prototype. This chapter contains information about how the data is stored and how we automatically check the reproducibility of construction processes.

In chapter 7 “Case Studies,” three examples of geoscientific construction processes based on real use cases are presented. Each of the case studies was implemented using the prototype software presented in the previous chapters. The first case study involves data analysis of geophysical field measurements. The second case study presents a classical subsurface 3-D model construction using gOcad. The final third case study deals with the construction of a hydrological balance model.

The last chapter 8 “Conclusions” contains a summary of this work. In addition, several possible improvements are described.

In addition to the main work, the Appendix contains additional information. Appendix A, C, D contain detailed information on the individual construction step used in the three case studies presented. Appendix B contains the instruction provided by LfULG Saxony as the foundation for the second case study. Appendix E contains a user manual for the presented prototype software.

1.6. Results Accomplished by this Thesis

As major contribution of this work, we present a framework to describe and record the construction process of geoscientific models in a reproducible way. Our framework is designed to:

- Describe how data are combined to build a complex geoscientific model
- Record the performed steps used to construct the model such that it is possible to repeat them later
- Perform automated checks to ensure that the same result can be reproduced at a later time
- Build a reproduction of a geoscientific model

Combined with a strict separation between data and methods, this allows the definition of standardised workflows for the construction of well-specified geoscientific models based on different data. This capability allows some degree of automation of the construction process, as the same workflow can be repeated with different data.

With GeoHub, we provide a prototype implementation of the presented framework. We have used this prototype to apply the presented framework to several real-world case studies.

2. Terms, Definitions and Requirements

2.1. Terms and Definitions

In the following section, a number of terms are introduced and defined to accurately describe the outlined problem and our solution.

2.1.1. Geoscientific model

A model can be seen as a simplified description of a certain part of the real world. To achieve this goal, the model abstracts and approximates certain facts.

Houlding [23] notes that geoscientist “wear many hats, e.g. geologist, hydrogeologist, geophysicist, environmental engineer, and geotechnical engineer.” Each of these groups has its own area of expertise, and each group uses different types of models. Structural geologists may build three-dimensional subsurface models to test specific geological theories or to visualise subsurface structures. Such models are usually represented as geometries and topologies [24]. Hydrogeologists build flow simulations, that show how a particular fluid, in most cases water, propagates in a given environment. In addition to geometric information, such models include physical parameters that control the fluid simulation. An environmental engineer might be interested in models that describe what amount of precipitation will cause a significant runoff in order to design an appropriate detention basin. Models used in this application area typically include information about the structure of the surrounding terrain. A geotechnical engineer might be interested in whether or not a particular subsurface can support their building. To answer these questions, they build simulations based on soil and rock properties. Geophysicists, on the other hand, try to identify structures in the subsurface based on their measurements. They parameterise geometric models to perform simulations. These results are then compared with measured values to verify that the theoretical model can explain a particular measurement.

A common theme of geoscientific models is their spatial and possibly temporal reference frame [25]. In addition to these commonalities, geoscientific models vary widely. Some may involve complex equations, others may be simplified black-box models, some may be built solely for visualisation purposes only, and still others may serve as the basis for further calculations or interpretations [23].

Others approach the definition of geoscientific models based on their use case, as in [26], slide 4:

“A model is a construct (of equations, relationships, imagined bodies, made of plastic etc...), which has one or more (interesting) aspect(s) of reality. A model is always a simplification, it depends on the fact that the aspect of reality that is to be modelled is mapped.”

In the context of this work, we focus on digital geoscientific models, which are digital representations of geoscientific models.

For this work, we define a geoscientific model as follows:

Definition 2.1 (Geoscientific Model). A geoscientific model is a simplified description of a real-world geoscientific system that contains all necessary information to represent the part of reality under study. \square

We will explicitly not focus on any particular type of geoscientific model, such as three-dimensional subsurface models. Our goal is to develop a general solution that applies to as many different geoscientific models as possible.

2.1.2. Reproducibility

Reproducibility is defined by different sources in different, slightly inconsistent ways. The Association for Computing Machinery (ACM) [27] lists the following definitions:

- The same result can be replicated by the original author using the same tools. This is called *Repeatability* by the ACM.
- The same result can be replicated by others using the same tools. This is called *Replicability* by the ACM.
- The same result can be replicated by others without using the same tools. This is called *Reproducibility* by the ACM.

Plessner [22] points out that different authors and institutions use slightly different definitions here. Claerbout and Karrenbach [28], for example, uses the terms *Reproducibility* and *Replicability* with reverse meaning compared to the ACM.

A common criterion of all definitions is the statement that the same result can be achieved by repeating the construction process. The various definitions differ in how the construction process is repeated.

In the context of this work, we define reproducibility as follows:

Definition 2.2 (Reproducibility). A geoscientific model is reproducible if it is possible to construct the same model with the same input data and a documented construction process. \square

This definition follows the definition of *Reproducibility* given by Claerbout and Karrenbach [28] as an appropriate definition for geoscientific applications. It is consistent with the ACM definition of *Replicability*. Using ACM *Reproducibility* definition would require the replication of the same process using different tools, which in turn requires scientific work by those attempting to reproduce a model. Since we want to provide an automated process to check whether a model meets our definition of *Reproducibility*, the ACM definition is not appropriate.

2.1.3. Realisation

Definition 2.3 (Realisation). A **Realisation** describes the execution of a certain construction process with a specific set of input datasets. \square

2.2. Requirements

In the following section we will break down our abstract goal of describing reproducible construction processes for geoscientific models into concrete requirements.

The first question is the degree of reproducibility of construction processes of digital geoscientific models. What exactly should be reproducible, by whom, in which environment and on which time scales?

We have already established that we want to make the entire process of building a geoscientific model reproducible. This process starts with a set of input datasets and produces a final geoscientific model. This goal, of course, requires that all information is stored to make reproductions possible in the first place. Without this information, you cannot even attempt to reproduce anything. We call this requirement **Store Information**.

Ideally, we would enable anyone to reproduce a given construction process without any precondition. The use of digital geoscientific models limits the number of qualified people to those who can work with a computer. Since we present our results as prototype software, we must assume that people have access to our software. They must either interact with the software to start a new reproduction of an existing construction process, or they must extract the necessary information to perform the reproduction outside of our software. This limits the user base to people who have expertise in using data management software. We refer to this requirement as **Skilled Personnel**.

The geoscientific community has already developed numerous software packages to construct geoscientific models. Well known examples include gOcad [17], GemPy [19], the RingMesh framework [29], BoWaHald [30], ArcGIS [18] and many more. Replacing all this software packages to provide an integrated solution for reproducible geoscientific modelling workflows will not work, especially since users are already familiar with these established tools. Therefore, we need to be able to integrate these software packages into the representation of reproducible construction processes. We call this requirement **Integration of Other Software**.

The combination of the requirements for integrating other software into a construction process and the requirement that the construction process must be reproducible by different people results in another hidden requirement. Different people may work and live in different places. Therefore, it cannot be assumed that all people attempting to reproduce a construction process of a particular model have physical or virtual access to the same computer. Since a construction process usually depends on a number of software packages, we need to provide these software packages in some way so that people can reproduce a construction process. There are two possible solutions here: Either require that all people attempting to reproduce a specific construction process have access to the same environment that was used to construct the initial realisation, or require that a construction process also include all necessary information about the software environment used as part of the information stored. Using the same environment for all reproductions creates a trust problem: Users must trust that the provided environment will not manipulate the results, and the environment provided must trust that users will not manipulate the environment to prevent future reproductions. Incorporating information about the used environment circumvents both problems and makes the process more robust against losing access to a particular environment which contains exactly the right versions of the required software. Thus, we require that a construction process can be reproduced regardless in any computational environment independently of the installed geoscientific software packages. A necessary constraint here is that the computational environment must be able to run any software required by the construction process. It is explicitly not required that the computational environment of our software solution provides

the software required for the construction process. We call this requirement **Computational Environment Independence**.

Both the construction processes and the datasets used as the basis for a particular construction process change over time. These changes are a common experience in scientific work and can occur for a variety of reasons. A better understanding of the underlying natural processes can employ an improved algorithm to interpret the measured data. Refined measurements can result in higher quality data that can improve the overall geoscientific model. Algorithmic improvements in numerical code used to process datasets can lead to a reduction in errors in geoscientific models. Our framework should be able to account for any of these time dependent changes to the construction process and datasets. Recording these changes in terms of existing construction processes adds context to the change itself and can reveal important details about the construction process itself. We assume that any description of a geoscientific model construction process may evolve, and we need to record these changes. This includes all information stored in relation to the construction process, such as relevant data and metadata. We call this requirement **Recording Changes**.

Finally, we postulate the following basic requirements for our software:

Requirement 2.1 (Store Information). We must store all the information necessary to reproduce a construction process at a later time. □

Requirement 2.2 (Skilled Personnel). A construction process should be reproducible by anyone who has some domain-specific knowledge. □

Requirement 2.3 (Integration of Other Software). The construction process may include other software packages □

Requirement 2.4 (Computational Environment Independence). A reproducible construction process description must include information about the computational environment required for this particular construction process. □

Requirement 2.5 (Recording Changes). Our framework must record changes to the construction process over time. □

3. Related Work

3.1. Overview

With GeoHub, we propose a framework for advanced documentation of how a digital geoscientific model was specified and realised with software that processes the given data and metadata. While many established applications solve parts of the presented problem, there is, no known solution for all requirements listed in the previous chapter. We decided to present some established software packages, focusing on interesting features that could be useful for the development of our solution. Some of them are related to geosciences, some are from other research areas.

In section 3.2 we present solutions for storing geoscientific datasets in a database system. These solutions serve as inspiration for the design of GeoHub’s data storage architecture.

In section 3.3 we present software that produces subsurface models. Geoscientists typically use these software packages to build their models. We present them here to demonstrate existing workflows and to show possible integration points in our new system.

In section 3.4, we present tools used in other research areas to describe data processing workflows. We present these solutions to get ideas for the general design of our system. However, due to minor and major differences in size, structure, and processing steps involved, none of the existing solutions can meet all the previously mentioned requirements.

Last but not least, in section 3.5, we present work on reproducible software builds. This work solves a similar problem in a different research area. Therefore, we can use these solutions to problems occurring in both application areas as inspiration. At the same time, however, we cannot take the actual applications because they do not work at all with geoscientific model construction workflows.

3.2. Geoscientific Data Storage Systems

3.2.1. PostGIS and Similar Systems

PostGIS [31] is an extension of the well known database system PostgreSQL [32]. PostGIS extends the data types inherently supported by PostgreSQL with data types specifically suited for storing geometric and geographic information. The extension includes support for storing points, lines, polygons, meshes, and grids in two and three dimensions. In addition, GIS functions, such as calculating distances or intersecting geometries, are implemented using custom operators and SQL functions. Similar extensions exist for other well-known database systems, such as SpatiaLite [33] for SQLite or Oracle Spatial [34] for the Oracle database system. We present PostGIS here as an exemplary representative, but most of the arguments apply to similar extensions as well.

Technically, PostGIS is an ordinary native PostgreSQL extension [35], that defines a set of user-defined data types, functions, and operators. The advantage of this approach is that it integrates well with PostgreSQL's existing SQL interface, so it can be used by any tool that works with PostgreSQL.

PostgreSQL and its extension PostGIS is a flexible database system that allows to store almost any data. Thus, it is possible to model specific reproducible model construction processes such that they can be stored in a PostgreSQL database. PostGIS provides support for geometric datasets to facilitate the representation of such data. Other types of datasets might require additional extensions or custom solutions. A solely database system based approach can only handle the storage of all datasets involved. All information about how these data need to be processed to build the final model must be stored as an explicit description. Since these descriptions may reference external tools, the database itself cannot validate that they are correct. Generalising this design into something that can be used to store reproducible construction processes generally requires a lot of design work and the development of several external auxiliary tools. These tools can then be used to check certain invariants of the construction process, such as whether it works as designed and whether it is reproducible. The resulting tools would use PostgreSQL and perhaps PostGIS internally for data storage, while the logic for handling a reproducible construction process would be implemented outside the database. The design and implementation of such an application might end up with a similar solution to the one presented in this thesis.

- PostgreSQL with the PostGIS extension is a flexible database system that allows storing geoscientific data
- As a flexible database system, PostgreSQL allows storing any information. This data can contain information about the construction workflow and other required details. The use of this information would require the development of an additional extension/software similar to GeoHub.

3.2.2. Geoscience in Space and Time (GST)

Geoscience in space and time (GST) [36] is a software suite to manage three-dimensional subsurface geological models. GST is a commercial software suite developed by GiGa infosystems. GST consists of a database service for managing subsurface models, a web application for presenting subsurface models stored inside the database, and a desktop management application for interacting with the database. At the time of writing, GST supports storage of the following geometric types:

- Point sets based on several points in three-dimensional space
- Line sets based on several lines in 3D
- Triangulated surfaces in 3D
- Tetrahedral meshes in 3D
- Regular grids in 3D
- Stratigraphic grids in 3D
- Geological profiles in 3D

GST supports the storage of geometry metadata for each geometry type. For this purpose, a key-value based system is used to represent arbitrary metadata schemes where the user can specify any desired value. In addition, each geometry supports the storage of properties within its primitive components. Point sets support storage of point-based property values, line sets, triangulated surfaces, and tetrahedral meshes support storage of property values

attached to their points and primitive elements. Cell-based property values are supported for regular grids and stratigraphic grids.

There are several motivations for the multi-component setup:

- Provide a central platform for collaboration with others on the same model
- Archive subsurface models in a central location
- Presentation of subsurface models to a broad audience via a web interface

Each of these use cases results in its own workflow when using GST, which affects the functionality provided by the platform. In order to work collaboratively on a model, it is necessary for users to be able to check out, modify, and update small portions of the model simultaneously. This feature requires the database system to ensure that no conflicting updates are allowed. GST solves this problem by implementing a checkout-based locking system [37], that allows a specific model volume to be marked as under modification by a specific user. To archive subsurface models, GST must process a large amount of data at once and store it in a way that makes it available for a longer period of time. In addition, GST must be able to import subsurface models from various sources and represent those models in its database without losing information. To display existing subsurface models via a web interface, GST may need to load and render complex three-dimensional geometries in a performant manner. Depending on the size of the displayed subsurface models, this may require downscaling the model to a coarse resolution. Figure 3.1 shows an example of a geoscientific model visualised in GST web.

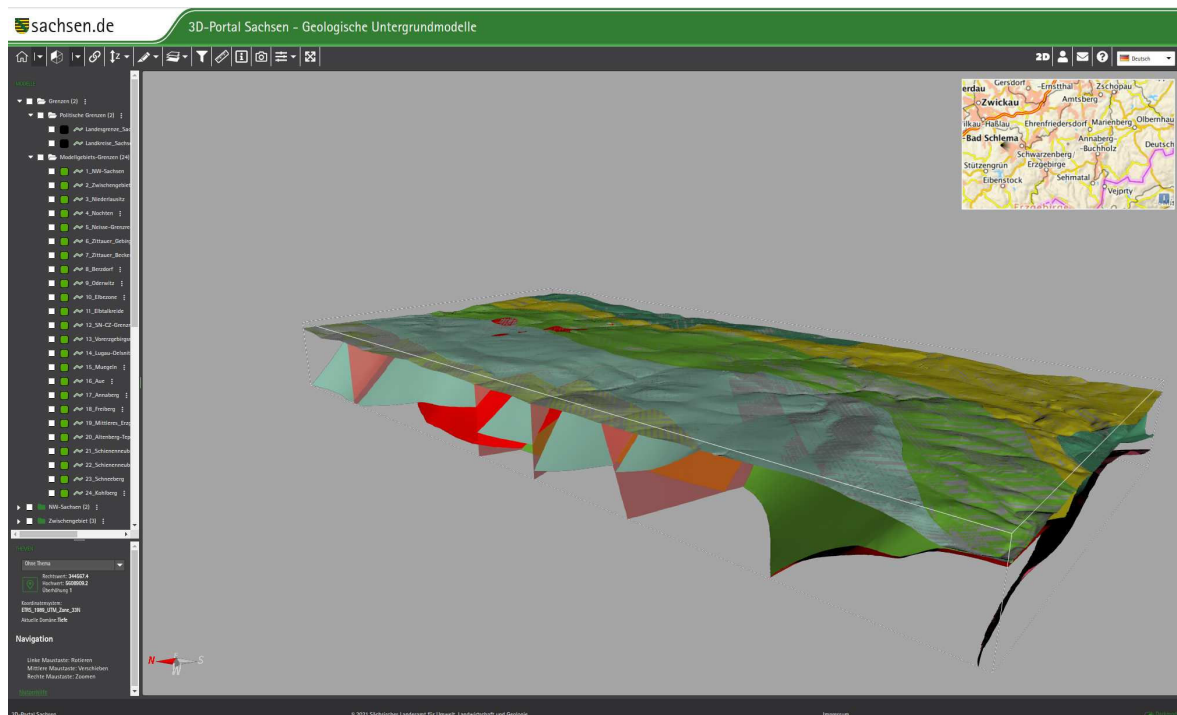


Figure 3.1.: Example subsurface model in GST Web [38]

GST is implemented as a classic client-server application [39]. The database functionality is provided by a central server, while the web interface and desktop management applications are designed as client applications that control the state on the server. The server application stores all data in a central location. Internally the data is stored in two different locations. The actual geometric information of the subsurface models is stored in its own file format directly on the hard disk. Metadata and associated information is stored in a

relational database. This allows for high-performance access to potentially large subsurface models while ensuring that there is no concurrent conflicting access to the data. Both client applications can query the central server to obtain subsurface models. The web application displays these models. The desktop client application allows further use of this data by saving it to the local disk for later editing and uploading an updated version.

GST enables versioning of the data stored in its central database. This feature makes it possible to track how a dataset evolves over time. Each version has a linked description, where users can provide additional information about what has been changed since the last version. This system is conceptually based on the commit system used by Git [40]. Describing entire model-building processes in GST is nearly impossible. When only geometric data are involved, strict guidelines are required to store every information in the database and provide descriptions of any processing steps as part of the version descriptions. This approach fails as soon as the model building process involves anything other than subsurface geometries, since GST cannot represent anything else.

- GST does not store the complete construction workflow of geoscientific models, but only the result of the construction
- There is no possibility to repeat a construction process because there is no information describing such a process

3.3. Geoscientific Modelling Software

3.3.1. gOcad

Geological Object Computer Aided Design (gOcad) [17] is an established commercial application suite specially designed to create complex subsurface three-dimensional models using various input data via a graphical user interface. gOcad originally implemented an explicit modelling approach, meaning that the user could manually constructs models based on different geometry primitives. In more recent versions of gOcad, an implicit modelling approach has been added via the new SKUA workflow [41]. Thus, it is possible to build coherent models based on a set of input data in a more automatic way.

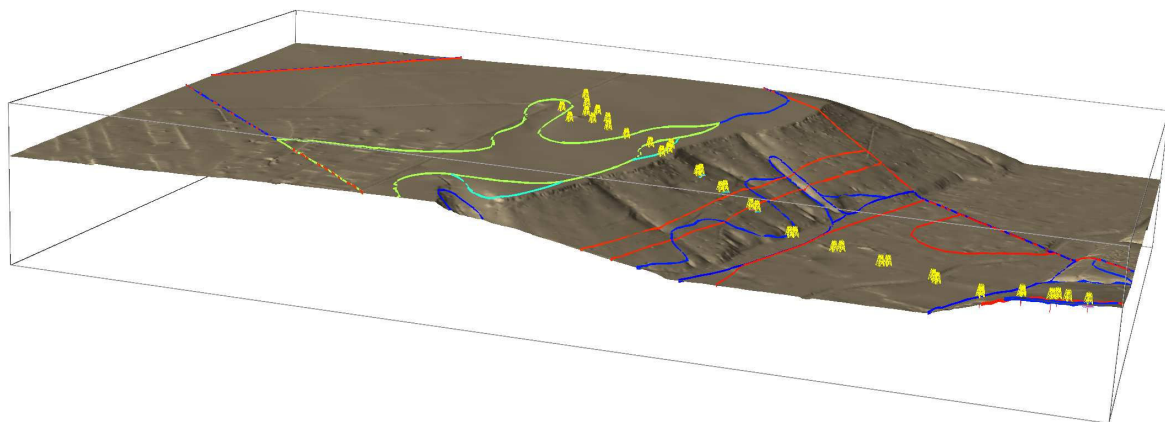


Figure 3.2.: Exemplary subsurface model in gOcad

A classic use of gOcad starts with the initialisation of a new project. Thereby a set of files is created on the local hard disk. Then, as a user, you can import various datasets, such as geological cross-sections, borehole data, or seismic datasets. Based on these data, gOcad

provides different workflows to construct a subsurface model. Some require more manual work, such as the extraction of cross-section lines from georeferenced images. Others work mainly automatically, such as the creation of a triangulated surface based on a set of points. The information about each processing step is stored in the gOcad project files. The format of these files is not documented and is version dependent. This means that it is difficult to extract information from these files outside of the appropriate gOcad version. Figure 3.2 shows an example model produced by gOcad. gOcad supports the export of datasets to different partially specified file formats, such as gOcad ASCII files [42] or even simple CSV files [43]. These exports allow other programs to use models constructed with gOcad. For example, GST (see section 3.2.2) uses such exported datasets to insert or update the corresponding datasets in its central database. The export process focuses on the data and excludes most of the metadata. As a result, the exported dataset contains only a fraction of the information of the entire project. In particular, the entire processing history is lost as part of the export.

In addition to the manual, point-and-click workflow described earlier, gOcad provides tools to automate parts of the workflow via a macro pipeline. These descriptions can be exported for later use. Listing 3.1 contains an example of a gOcad macro that performs a discrete smooth interpolation (DSI) [21] based on an imported set of points. These macros can be easily composed from the gOcad graphical user interface by selecting which already performed construction steps should be part of the macro. A macro can be exported from there as a Javascript file for later use. According to the gOcad documentation, it is possible to later include normal Javascript statements in addition to the generated calls. The execution of a macro is possible via the graphical user interface or by passing a command-line flag when starting the application.

The use of recorded macros in gOcad allows later repetitions of the model construction. These macros can be easily composed in gOcad itself based on the recorded edit history. It does not allow the user to easily reason about the reproducibility of a construction workflow, since there is no easy way to compare the results of different realisations. Moreover, this process is highly dependent on the following factors:

- Different gOcad versions contain different functionality. Therefore, different versions may not be able to perform the required actions to reproduce a result.
- The computational environment must provide the same input data in exactly the same location as before. For example, see line 3 of listing 3.1, which explicitly specifies the path of the point set to import. For each reproduction, a file with the same name must exist in exactly the exact location.
- gOcad does not provide any tools to check the equality of two models.

```

1  var skua = PDGM.require('skua');
2  skua.run('ImportXYZFile',
3    { 'File_name': ["/path/to/file.xyz"],
4      'category': "Horizons",
5      'cvn': "Domain=Default_depth",
6      'ignore_points_with_no_data_value': "false",
7      'no_data_value': -9999 },
8    { blocking: false, typed: true });
9  skua.run('TSurfCreateFromPlane',
10    { 'Clip_with_XYZ': "true",
11      'Clip_with_voxel': "false",
12      'Do_not_clip_Z': "true",
13      'name': "DGM",
14      'points': ["/gobj:file"],
15      'scale': 1.1,
16      'voxel': "none" },
17    { blocking: false, typed: true });
18  skua.run('TSurfSplitAll', { 'on': "/gobj:DGM" }, { blocking: false, typed: true });
19  skua.run('TSurfSplitAll', { 'on': "/gobj:DGM" }, { blocking: false, typed: true });
20  skua.run('TSurfSplitAll', { 'on': "/gobj:DGM" }, { blocking: false, typed: true });
21  skua.run('TSurfSplitAll', { 'on': "/gobj:DGM" }, { blocking: false, typed: true });
22  skua.run('TSurfSplitAll', { 'on': "/gobj:DGM" }, { blocking: false, typed: true });
23  skua.run('TSurfSetBOSLOnAllBorders',
24    { 'dir_shoot': [0., 0., 1.],
25      'force_direction': "true",
26      'on': "/gobj:DGM",
27      'optimize_shooting_direction': "true" },
28    { blocking: false, typed: true });
29  skua.run('TSurfSetFcp',
30    { 'control_points': ["/gobj:file"],
31      'dir_shoot': [0., 0., 1.],
32      'on': "/gobj:DGM",
33      'optimize_shooting_direction': "false" },
34    { blocking: false, typed: true });
35  skua.run('TSurfRunIsotropicDsiGeo',
36    { 'conjugate': "false",
37      'fitting_factor': 2.0,
38      'nbiter': 10,
39      'on': "/gobj:DGM",
40      'smooth': "false" },
41    { blocking: false, typed: true });

```

Listing 3.1.: Example gOcad macro for performing a DSI interpolation of an imported point set

3.3.2. GemPy

GemPy [19] is an open-source software package for three-dimensional subsurface modelling. It is distributed as a Python library. GemPy implements an implicit modelling workflow.

Listing 3.2 shows an example model construction Python script. Such a workflow can be easily extended by using any Python library to perform additional actions, such as retrieving files from remote sources, waiting until certain conditions are met, or anything else that can be expressed by a Python script. This concept allows for a large number of use cases, but on the other hand, it limits its use to a user group that can write Python scripts.

Representing the model as script makes it theoretical easy to repeat the model construction later, since only the execution of the corresponding script has to be repeated. This concept requires the storage of the construction script and all datasets used. Also, each script has an implicit dependency on the environment that executes the script. A compatible Python interpreter is required to execute the model construction script. The interpreter must have access to the specific GemPy version used in order to execute GemPy related statements. Any difference in the GemPy version between the original and the reproduced environment may have an impact on the execution result, as it may change the implementation of certain functions. These considerations also apply to any other Python dependency used by the model construction script. In addition, each of the Python dependencies may have internal dependencies on other libraries. All of these can affect the result of the computation. See the Compatibility section of the NumPy project change log [44] for numerous examples of incompatibilities between different versions.

In addition to these requirements for the runtime environment used to reproduce a construction process, it is necessary to check if a construction process is reproducible. In general, every operation cannot be expected to produce deterministic results. GemPy does not have an explicit API to compare produced models. Instead, it provides a comparison operator implementation based on Python's default operator implementation [45]. Since GemPy does not provide specialised implementations, these are based on object identity instead of a value-based approach. Such a definition of equality is not a good way to compare two model instances. This implementation always results in two realisations not being equal because they are represented by two different object instances.

GemPy is a software package for constructing subsurface models. It cannot be used as a stand-alone tool to collaborate on the construction of geoscientific models. In conjunction with version control systems such as Git [40] such collaboration becomes possible.

- GemPy allows construction processes to be described as scripts, such that the construction can be repeated
- GemPy is limited to a specific area of geoscientific model construction
- GemPy does not provide any good way to compare the results of two repetitions of the construction process
- Changes in the software environment can lead to changed results
- GemPy does not provide a solution for managing data of any kind related to the construction workflow

```

1  """
2  Greenstone.
3  =====
4  """
5
6  # Importing gempy
7  import gempy as gp
8
9  # Aux imports
10 import numpy as np
11 import matplotlib.pyplot as plt
12 import os
13
14 print(gp.__version__)
15
16 geo_model = gp.create_model('Greenstone')
17 data_path = 'https://raw.githubusercontent.com/cgre-aachen/gempy_data/master/'
18
19 # Importing the data from csv files and set extent and resolution
20 geo_model = gp.init_data(geo_model,
21                          [696000, 747000, 6863000, 6930000, -20000, 200],
22                          [50, 50, 50],
23                          path_o=data_path +
24                          "/data/input_data/tut_SandStone/SandStone_Foliations.csv",
25                          path_i=data_path +
26                          "/data/input_data/tut_SandStone/SandStone_Points.csv")
27
28 gp.map_stack_to_surfaces(geo_model, {"EarlyGranite_Series": "EarlyGranite",
29                                       "BIF_Series": ("SimpleMafic2", "SimpleBIF"),
30                                       "SimpleMafic_Series": "SimpleMafic1",
31                                       "Basement": "basement"})
32 geo_model.add_surface_values([2.61, 2.92, 3.1, 2.92, 2.61])
33 gp.set_interpolator(geo_model,
34                     compile_theano=True,
35                     theano_optimizer='fast_compile',
36                     verbose=[])
37 gp.compute_model(geo_model, set_solutions=True)
38
39 # Display the model
40 gp.plot_3d(geo_model)
41
42 # Saving the model
43 # ~~~~~
44 #
45 np.save('greenstone_ver', geo_model.solutions.vertices)
46 np.save('greenstone_edges', geo_model.solutions.edges)
47 gp.save_model(geo_model)

```

Listing 3.2.: Example Python script for creating a GemPy model

3.4. Experimentation Management Software

3.4.1. DataLad

DataLad [46] is a distributed data management platform that provides combined analysis code and data management. The software originates from the field of computational neuroscience and is developed as an open-source project. DataLad describes itself as a completely domain independent general-purpose data management tool.

Technically, DataLad builds on the well-known version control system Git [40]. Like Git, DataLad provides a command-line interface to interact with various datasets. In addition, it uses git-annex [47] to store large files, that are otherwise not supported by Git itself. Using a version control system as a foundation allows DataLad to easily track the evolution of a dataset over time. Using Git as the primary layer allows DataLad to follow a distributed and decentralised storage model where each node can hold all the data. The basic assumption here is that each managed dataset is just a Git repository, where DataLad's definition of a dataset corresponds to our definition of a model.

DataLad allows users to track datasets and code simultaneously in the same Git repository. It also provides an interface to record individual executed commands that process the data. This information is recorded as part of Git's commit messages. Listing 3.3 shows an example of the recorded information. This information includes the command executed (as the `cmd` field), a list of input and output files (as `inputs` and `outputs`), the current working directory (as the `pwd` field), and some other internal information. DataLad can repeat the execution of recorded commands by using the supplied `datalad rerun` command. This command allows a single step to be repeated by specifying the concrete commit hash of the operation, to be repeated. It can also repeat several steps in the correct order by specifying a commit range as input. In this case, any command recorded within one of these commits will be executed in the order of the commits. DataLad does not provide tools to compare the output of these reproductions. It refers here to standard diffing tools, that only work well for textual data with similar internal structure. Consequently, there is no easy way to check whether or not a set of commands produces the same output on several runs. This circumvents the problem of defining equality between different files representing scientific datasets.

```
{
  "chain": [],
  "cmd":
    "convert -resize 400x400 feed_metadata/salt.jpg recordings/salt_small.jpg",
  "dsid": "8e04afb0-af85-4070-be29-858d30d85017",
  "exit": 0,
  "extra_inputs": [],
  "inputs": [
    "recordings/longnow/.datalad/feed_metadata/salt.jpg"
  ],
  "outputs": [
    "recordings/salt_small.jpg"
  ],
  "pwd": "."
}
```

Listing 3.3.: Example data recorded by `datalad run`

DataLad only uses information about input and output data to check whether the corresponding file exists when executing a command. This information are then used to provide helpful error messages for missing input data or when a command would overwrite an output dataset. This information is not used to identify the flow of data through a processing pipeline, although this information could be extracted from the appropriate commit messages. An ideal system could use this information to optimise the construction of updated models by skipping operations that affect only unchanged datasets.

Furthermore, the documentation of DataLad [48, Ch. 7.2] states that DataLad does offer possibilities to store both data and code. This may not be sufficient to reproduce the results at a later time or with a different computer, because the same software environment must be used for the reproduction as for the original realisation. The DataLad documentation explicitly lists possible dependencies here:

- The operating system
- Exact versions of the installed software
- Configuration of the installed software

Later in the mentioned chapter, the documentation presents the `datalad-containers` extension as a solution to this problem. This extension allows the user to specify a container image that should be used as an environment for executing commands. Containers are special virtual machines that combine software and configuration in a compact unit. This approach solves the presented problem by transforming the implicit dependency on the software environment into an explicit one.

In addition, DataLad provides support for metadata extraction and aggregation [46]. DataLad provides several built-in modules for extracting metadata. For example, DataLad provides a module for extracting metadata for images, that collects information based on the embedded EXIF data [49]. In addition to the built-in modules, it is possible to provide specially tailored extensions that provide additional metadata extraction capabilities. The extracted metadata is then stored in a private DataLad folder as a JSON document. This approach allows the representation of arbitrarily complex metadata schemes. DataLad follows the JSON-LD W3C Community Draft [50] for representing metadata as a JSON document.

- DataLad provides tools for recording construction processes, datasets used and associated metadata
- DataLad does not ensure that only datasets and programs contained in the current repository are used, since `datalad-containers` is an optional dependency
- DataLad couples the code used to construct the model directly with the input datasets used in the same repository.
- DataLad does not provide tools to check the equality of two datasets.

3.4.2. Data Version Control (DVC)

Data Version Control (DVC) [51] is a version control system for machine learning projects. DVC is a command-line tool that allows you to record an entire data processing pipeline, including data, code, and model definitions. The primary goal of DVC is to make machine learning models shareable and reproducible, although the software can presumably be applied in other application areas as well.

Technically, DVC is built on top of traditional version control systems such as Git [40] or Mercurial [52]. This allows DVC to track a versioned state of the current working directory,

which contains code, training data, and additional information. Unlike DataLad, DCV implements its own data storage mechanism for managing large binary files. This data is stored in a remote location, such as Amazon’s S3 system [53] or any network storage system, and is referenced by a unique identifier in the version control system. This approach essentially mirrors the behaviour of Git LFS [54] in a way that is independent from the underlying version control system. Just like the behaviour described, Git LFS uses an additional service to store large binary data and only references this data later in the Git repository.



Figure 3.3.: A simple example workflow consisting of three construction steps shown as a directed acyclic graph as generated by the `dvc dag` command on the command line. The boxes represent nodes of the graph, while the * * * form edges. Each node represents a construction step, while each edge represents a dependency between these steps. According to the documentation, the visualization assumes a top to bottom directionality.

In addition to files used as part of the machine learning workflows, DVC also enables the recording of the workflows themselves. A workflow can consist of more than one step. DVC records the workflow structure as a directed acyclic graph of workflow steps, where each node represents a workflow step, and each edge represents a dependency between two workflow steps. Figure 3.3 shows an example graph. According to the manual, DVC assumes that edges are orientated from the top to the bottom. An edge pointing from the *prepare* workflow step to the *featurize* workflow step can be read as the *featurize* workflow step, depending on output of the *prepare* workflow step. Each workflow step is described as a command to be executed. In addition, each workflow step can have an optional dependency on one or more files.

Based on the recorded processing workflow, DVC provides a `dvc repro` command. This command repeats the recorded processing steps using the current state of the working directory. However, it does not check whether the execution gives the same result as previous executions.

In our view, DVC offers the most complete approach for the reproducible construction of a model based on data, although there are some shortcomings.

In particular DVC does not enforce recorded dependencies between workflow steps and data. Any workflow step can declare an explicit dependency on data by using the corresponding DVC command line flag. Such referenced data can be a file that exists in the current working directory at execution time or in any other folder on the current computer or even on the Internet. DVC’s documentation [55] indicate that you should follow these rules:

- Read/write only from/to the specified dependencies and outputs (including parameter files, metrics, and plots).
- Rewrite the output in its entirety. Do not append to or edit existing files.
- Stop reading and writing files when the command exits.

Similarly to DataLad, the documentation also states that the code must be deterministic, i.e. it must produce the same output for the same input to achieve a reproducible workflow. In particular, it is recommended to avoid code that uses random numbers, time functions, or hardware dependencies, but DVC does not provide automatic checks to detect such operations.

- DVC records how transformations are performed via workflow steps. These descriptions contain little information about which applications were used as part of a workflow step. DVC’s documentation [55] states that executed code may only work on certain operating systems or require that certain software packages must be installed.
- The `dvc repro` command repeats the execution of a defined processing workflow, but does not check whether the results match. This approach circumvents the problem of defining equality between datasets. However, the user has no easy way to check whether the provided description of the workflow is sufficient to reproduce a given result.
- DVC is designed for use in the context of machine learning. Therefore, the software contains functionalities that are specifically tailored to this application area. This includes features such as plotting result metrics of training runs. Another limitation is that DVC lacks functionalities such as explicit metadata collection that may be useful for a geoscientific applications.
- DVC couples versioning of data and workflow to the same time scale by storing everything in the same underlying version control system. This makes it harder to understand what changes have actually changed the data or the processing pipeline.
- As a command-line tool that executes self-written scripts as workflow steps, DVC is only accessible to skilful personnel. Geologists may be overwhelmed to use its interface to develop their workflow steps. For them a more suitable approach would be to separate the definition of workflow steps from the actual workflow composition, such that more computationally skilled individuals can devise step descriptions. Then, less computationally proficient people can use these steps to represent their workflow.

3.5. Reproducible Software Builds

Software developers face a similar reproducibility problem as the applied science community. The problem there is: “Was the provided binary artifact generated from the provided source code, or is there a mismatch?” Answering this question is important for security-critical applications, as any unidentified change to the software, or dependency in the build chain [56], such as compilers, linkers or interpreters, can introduce backdoors or other security related vulnerabilities.

The Reproducible Builds project aims at an independently verifiable path from source to binary code and explicitly defines a reproducible software build as follows[57]:

Definition 3.1 (Reproducible Build). A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts. □

Here the Reproducible Builds project specifies the “source code” as the starting point of the build process. In addition, information about the build environment is required. It usually includes exact information about the version of the operating system used and all tools involved in the build process. Furthermore, each build process produces one or more artifacts, which may include executable files, distribution packages, or file images, and excludes build logs or similar output. Typically, these artifacts are compared at the bit level to determine if they are equal. In addition, the Reproducible Builds project indicates that such information must be provided on a case-by-case basis by the individual author who wishes to distribute reproducible software builds.

Reproducible builds are monitored by some Linux distributions for their entire package ecosystem. Examples are:

- Debian [58] states that 94.3% of packages can be built in a reproducible way for the amd64 target.
- ArchLinux [59] states that 81.9% of packages can be built in a reproducible manner.
- NixOS [60] states that 100.0% of packages can be built in a reproducible way for the x86_64-linux target.

All distributions listed track the percentage of packages that can be built reproducibly. These numbers present the situation at the time of writing and may change in the future. The reproducibility of the build process is checked by performing two independent builds and comparing the results. Some monitoring systems also change various potential sources of non-determinism between the two separate build processes. It includes things like changing the build path, the build user, and the time of the build system.

There are various potential causes for non-reproducible build results. The Debian project has detailed statistics about causes and how often they occurred based on Debian’s package list [61]. Common causes for differences between two artifacts are:

- Mismatched directory or file paths embedded in the artifact
- Inconsistencies in the timestamps embedded in the artifact
- Non-matching values of environment variable values embedded in the artifact
- Non-deterministic code generation resulting in different artifacts

A common approach to avoid the first three issues is to define a standard build environment, that is documented in detail. This includes information about the exact configuration of the operating system and all installed software. In addition, the Debian project suggests several workarounds in its documentation [62]:

1. Try to remove the information from the artifact in a postprocessing step using the `strip-nondeterminism` command [63].
2. Try to modify the corresponding build tool that produces the non-reproducible information so that it no longer generates any non-reproducible parts at all
3. Try to change the corresponding build tool that generates the non-reproducible information to use other reproducible information instead.
4. Try modifying the corresponding build tool to respect special environment variables developed by the Debian project for the use case of reproducible builds. These environment variables then contain a trimmed down version of the previous non-reproducible information to make the build reproducible.
5. Try to modify the built environment such that it does not contain the non-reproducible information

Building software in a reproducible manner is similar to the problem of building reproducible geoscientific models. Common problems identified by the above projects are likely to be encountered in our research domain as well. Common workarounds provide us with a basis for working around these problems, but not all can be readily applied to the task of building geoscientific models. For example, most build tools involved in software building processes are available for free, which is especially true for open source projects, such as those used by the Linux distribution mentioned above. In this situation, it is easy to modify the software used in such a way that reproducible results are produced, as suggested in workarounds 2 - 4. Geoscientific software is usually harder to modify. For example, it is near to impossible to remove sources of non-determinism from gOcad because users generally do not have access to the source code of a commercially distributed software package. Workaround 1 is also not easily transferable due to the variety of file formats involved. Artifacts, which are produced by software building processes are typically binary executable files or shared libraries. With ELF [64] and Win32/PE [65], only two file formats describe exactly what information can be recorded and where. For geoscientific models, there are a variety of fundamentally different file formats, and all of them may contain non-deterministic information in different ways. Therefore it seems hard to develop a tool that removes this information in a post-processing step.

3.6. Summarised Related Work

None of the presented software packages provides a complete solution to the problem of describing reproducible geoscientific model constructions. However, each of the presented packages solves some parts of the problem in a unique way. These solutions might be combined to provide a more general solution.

Geoscientific data storage solutions such as PostGIS and GST specialise in storing different types of geoscientific datasets. They lack the infrastructure to store information about the processing steps applied to each dataset.

Geoscientific model construction software such as gOcad and GemPy specialise in composing specific geoscientific models. These software packages provide the ability to see what construction steps have been performed. However, this information can not generally be transferred to construction processes involving the use of several software packages.

Experimentation management software such as DataLad and DVC provide tools for recording complex data processing pipelines. While both can describe a potentially reproducible construction process, they lack essential features that ensure independence from a particular computational environment and reproducibility of a construction process. Furthermore, these software packages focus on different application areas and make different assumptions about the data processing steps used and the datasets involved.

Last but not least, the existing infrastructure for providing reproducible software builds already solves the equivalent problem in another application area. Therefore, the knowledge gained there can be useful here. These insights include in particular strategies for checking reproducibility by repeating the corresponding process and strategies for circumventing potential problems when comparing the output of two realisations due to non-deterministic results.

4. Concept

The major objective of this thesis is the development of software to manage reproducible construction processes for geoscientific models, taking into account the requirements specified in section 2.2. This chapter turns the abstract requirements into an implementable algorithms.

As a key functionality, our application needs to represent the construction process of a geoscientific model in a algorithmic way. Section 4.1 presents a conceptual model of how such construction processes can be described as hypergraphs. This approach satisfies requirements 2.2 **Skilled Personnel**, requirement 2.3 **Integration of Other Software**, and requirement 2.4 **Computational environment independence**.

A tool that enables reproducible construction of geoscientific models must be able to handle various geoscientific data, as specified in requirements 2.1 **Store Information** and 2.5 **Recording Changes**. In section 4.2 we present some challenges and possible solutions for how our application handles data.

We will use a simplified construction process of a geoscientific model as a motivating example for both parts of this chapter. Our example workflow uses three different datasets to build an abstract geoscientific model. A geologic cross-section is provided as an image, a set of borehole data is provided as a file comprising a set of points, and a spatial context containing information about the target coordinate system of our model. The construction of the model involves three distinct operations. In the first step, the geologic cross-section is digitised. Then, both the digitised cross-section and the borehole data are projected into the target coordinate system. Finally, a triangulated surface is generated by a meshing operation. We consider this triangulated surface as the result of our construction process.

4.1. Construction Hypergraphs

The fundamental unit of a construction process is a single construction step. In our abstract model, each construction step is a transformation that converts a set of input datasets into a single output dataset. There is a wide variety of construction steps. Given requirement 2.3, **Integration of Other Software**, we assume that the construction steps are implemented by using other software packages. A common theme of construction steps is the execution of a processing operation, which can usually be described with a verb. Our example construction workflow consists of three different construction steps:

- Digitise geological cross-sections
- Transform coordinates
- Compute a mesh of points and lines

Digitising a geologic cross section means to turn a geologic cross-section available as an image into a line dataset in 3D space. Figure 4.1a contains a graphical representation of this construction step. The green ellipse represents the input to this construction step, while the

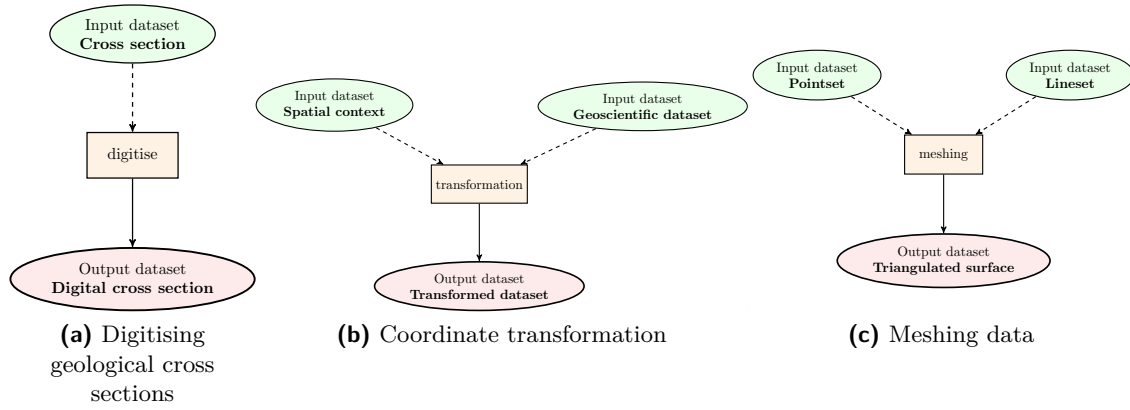


Figure 4.1.: Example construction steps

red ellipse represents the output. The operation itself is represented by the orange rectangle. The arrows indicate in which direction the data flow through the process.

A coordinate transformation projects a spatial object from one coordinate reference system to another reference system. This operation depends on two different types of input data. It uses the specification of the target coordinate system and the spatially referenced dataset itself. The result is a projected version of the spatially referenced input dataset. Figure 4.1b contains a graphical representation of this construction step.

Meshing combines a set of point and a set of lines to form unified three-dimensional triangulated surface. Figure 4.1c contains a graphical representation of this construction step.

The construction of a geoscientific model involves a sequence of construction steps. The following paragraphs contain example construction step sequences, highlighting essential characteristics of construction processes.

Each construction step can depend on the results of previous construction steps. These dependencies impose an inherent order of the construction steps. Figure 4.2a shows an example sequence of two construction steps. It illustrates that a change in the order of the construction steps involved leads to a corrupted construction process. The processed input data and the generated output data would then no longer match the corresponding construction steps.

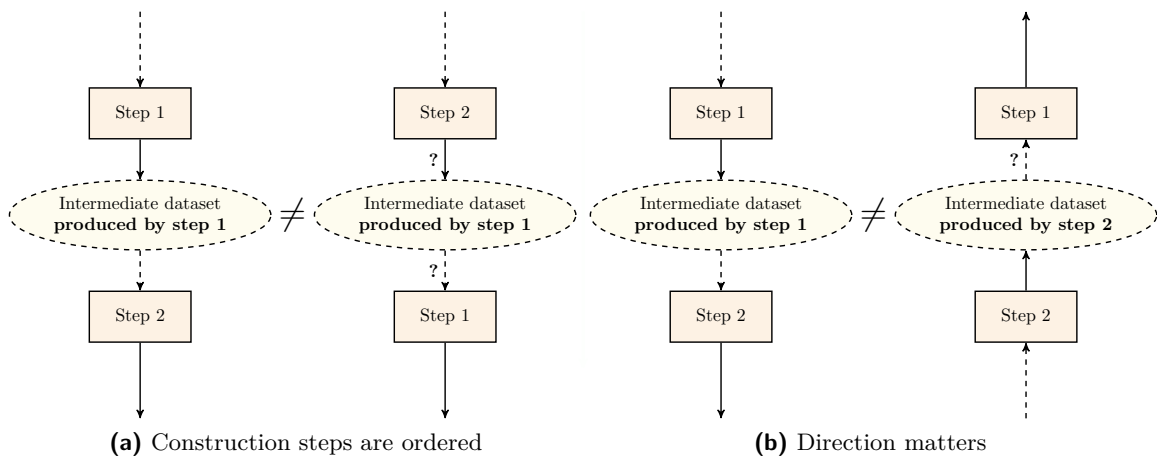


Figure 4.2.: Order and direction of construction steps is meaningful

A sequence of construction steps in a construction process must not only be ordered, but it also has a direction. It is not possible to reverse the sequence of construction steps in a construction process, for reasons similar to the order constraint. Figure 4.2b illustrates this case graphically. We could correctly perform the first construction step of a reverse construction process, given the correct input data. However, this is not true for the second step in the reverse sequence, since it would now have to process incompatible input data generated by the earlier second construction step.

A single construction step can use more than one dataset as input. A construction step needs access to all input datasets simultaneously to perform the underlying data transformation. This implies that the dependency between input data and output data is more complex than a simple one-to-one dependency. Figure 4.3 illustrates this case graphically. Consequently, construction steps cannot be represented as edges in a graph.

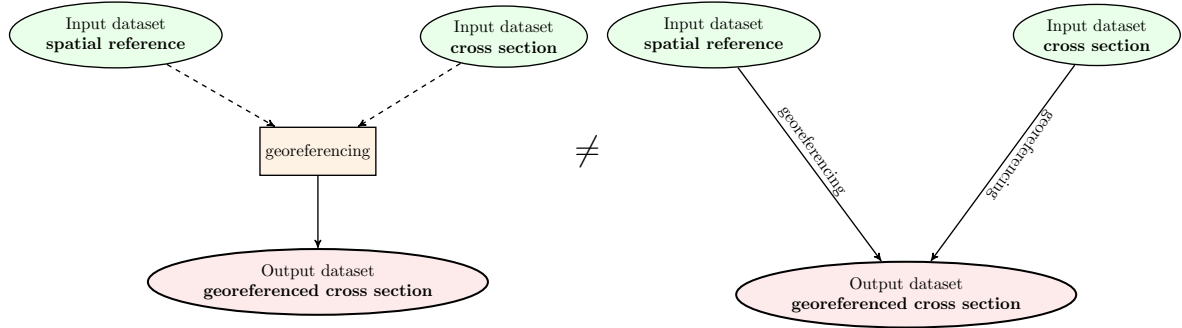


Figure 4.3.: Construction steps are not edges, as shown by the fact that a construction step may require more than one input dataset

The concatenation of several construction steps, each of which accepts more than one input dataset, provides another important insight. The dataset and construction steps involved in a construction process cannot form a simple tree relationship because of their interdependencies. Figure 4.4 illustrates this case graphically. For example, the *spatial context* dataset is used as an input dataset for more than one construction step, whereas in a tree-like relationship it could be used in only one construction step.

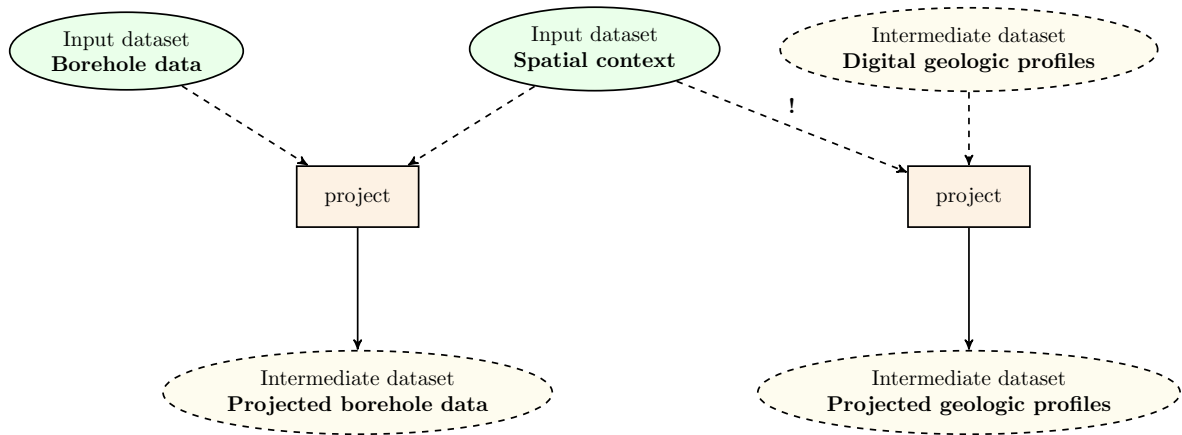


Figure 4.4.: A construction process cannot be represented as a tree, as shown by the fact that the same *Spatial context* input dataset is used by several construction steps

For this thesis we define dataset, construction step and construction process as follows:

Definition 4.1 (Datasets). We call a set of related data a dataset. These data are grouped by a logical dependency. A dataset may consist of several parts, which are grouped into one or more files. \square

Definition 4.2 (Construction step). We denote a construction step as a transformation $F : (D_1^I, \dots, D_n^I) \rightarrow D^O$ where D_k^I $k = 1, \dots, n$, $n \geq 1$, denotes the input datasets and D^O the output dataset. \square

Definition 4.3 (Construction process of a geoscientific model). We define a construction process as a sequence of construction steps. \square

From the above observations, we derive the following facts about appropriate data structures for representing construction processes:

- Construction steps are inherently ordered.
- Two successive construction steps have a well defined relationship to each other, where the output of one step is used as input for another step. This implies a directed dependency between datasets and construction steps and between different construction steps.
- A construction step may have more than one input dataset and the input datasets must be used together. This implies that all of the construction steps in a construction process cannot form a simple sequence or graph.
- A single dataset can be used as input for two different construction steps. This means that the totality of construction steps of a construction process cannot be described as a tree.

Given these observations, a directed hypergraph is an appropriate data structure to represent a construction process.

Definition 4.4 (Directed Hypergraph). We define a directed hypergraph H as a pair $H = (V, E)$ where $V = V_0, \dots, V_j$ is a set of vertices and $E = E_0, \dots, E_k$ a set of directed hyperedges. A hyperedge E connects a subset of vertices V_e of V that contains at least two vertices. A directed hyperedge further divides the vertices in V_e into head and tail vertices, where a directed hyperedge points from the set of tail vertices to the set of head vertices. \square

For our use case, a complete hypergraph corresponds to a specific construction process of a geoscientific model, a vertex corresponds to a dataset, and a hyperedge corresponds to a construction step. Unlike general directed hypergraphs, construction hypergraphs are subject to additional constraints. Since hyperedges represent construction steps, each hyperedge must have one and only one head vertex and at least one tail vertex. Another practical constraint is that the construction graph must be acyclic. Otherwise, a construction hypergraph with a fixed number of construction steps cannot be guaranteed to yield a complete valid model in a finite number of steps.

Definition 4.5 (Construction hypergraph). We define a construction hypergraph as an acyclic directed hypergraph consisting only of hyperedges having one and only one head vertex and at least one tail vertex.

We use $E_{(V_{i_1}, \dots, V_{i_n}) \rightarrow V_o}$ as notation for a hyperedge connecting the set $(V_{i_1}, \dots, V_{i_n})$ of tail vertices with the head vertex V_o . A hyperedge E represents a construction step F , such that

$E = E_{(V_{i_1}, \dots, V_{i_n}) \rightarrow V_o}$. This implies that each vertex V_{i_ℓ} that is part of the tail vertex set V_{i_1}, \dots, V_{i_n} corresponds to an input dataset $D_{i_\ell}^I$ of F . The head vertex V_o corresponds to the output dataset of F , such that $V_o = D^O$. \square

Definition 4.6 (Path). We define a sequence of hyperedges to be a path if at least one head vertex of a preceding hyperedge is an element of the set of tail vertices of the following hyperedge. \square

The representation of a construction process as a construction hypergraph allows us to reason about the structure of the construction process itself. We can infer dependencies between different construction steps and datasets based on the structure of the construction hypergraph. We call two construction steps F_i and F_j independent if there is no path connecting the output datasets ${}^iD^O$ and ${}^jD^O$. We say that the construction step F_i depends on F_j if there is a path between ${}^jD^O$ and one of the n input datasets ${}^iD_k^I$ for $k = 1 \dots n$ of F_i . We call construction steps F_f without dependencies, dependency-free. Their input datasets ${}^fD_k^I$ are input datasets of the construction hypergraph itself.

Definition 4.7 (Input Dataset of the Construction Process). A dataset is an input dataset of the construction process if no construction step generates it as an output dataset. \square

At the other end of the construction process, there are construction steps F_o , which are not used as dependencies of another construction step. We call these datasets the output datasets of the construction hypergraph. Depending on the exact structure of the construction hypergraph it may possess one or more output datasets.

Definition 4.8 (Output Dataset of the Construction Process). A dataset is an output dataset of the construction process if no construction step uses it as an input dataset. \square

Any other dataset involved in the construction process is called an intermediate dataset. Intermediate datasets are both, an output dataset ${}^lD^O$ of the construction step F_l , and an input dataset ${}^{l+1}D_k^I$ of the construction step F_{l+1} .

Figure 4.5 shows an example construction hypergraph. Oval nodes represent different datasets D_j , while rectangles represent different construction steps F_j . Dashed edges indicate that a particular dataset D_k^I is used as input for a particular construction step, while solid edges indicate that a construction step generates a dataset D_k^O . A hyperedge E_j is visualised by the construction step rectangle and all incoming and outgoing edges. Green nodes represent the input datasets of the construction process, while the red node represents the datasets generated by the construction process. Finally, yellow nodes represent intermediate datasets generated and processed by the construction steps of the construction process.

The example construction hypergraph represents our example construction process. It consists of 7 nodes representing the datasets used by the construction process. *Borehole data*, *Spatial context* and *Geological profile* represent the input datasets, which are provided as the basis for model construction. *Digital geological profiles*, *Projected borehole data*, and *Projected geological profiles* are intermediate datasets generated and processed as part of the construction process. Finally, the *Triangulated surface* dataset is the output dataset of our construction process. It is generated by the specified sequence of construction steps applied to the input datasets.

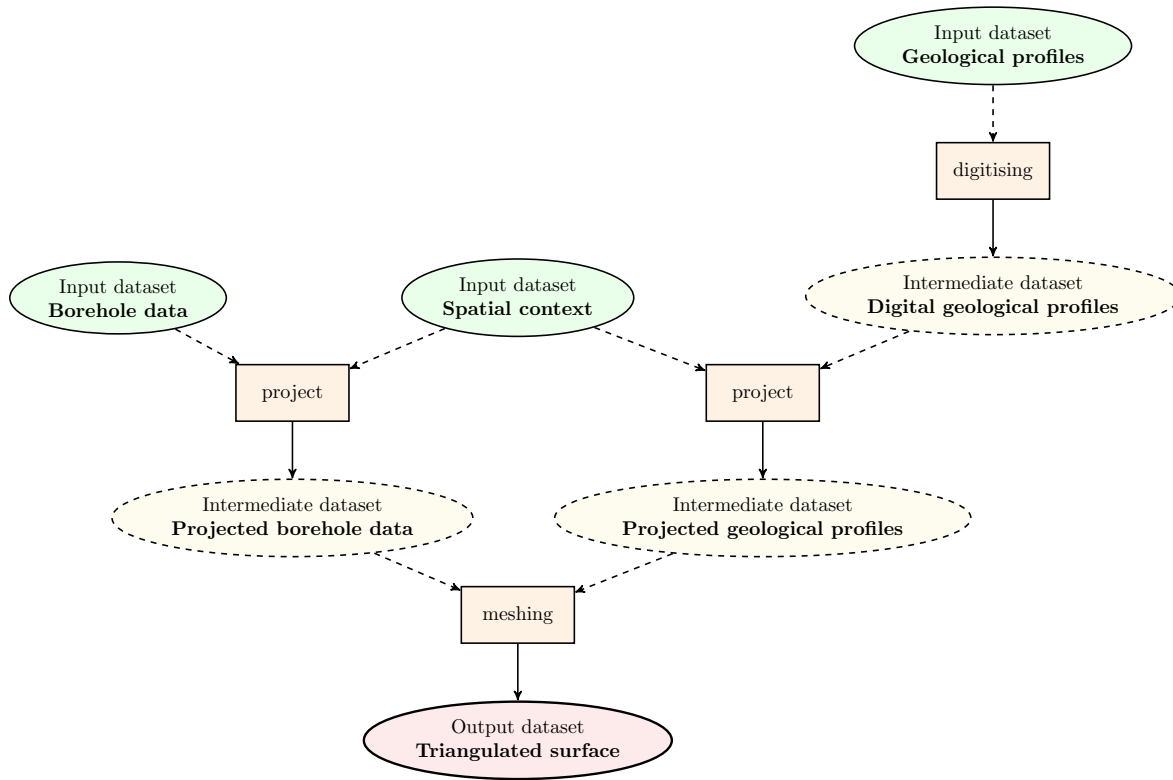


Figure 4.5.: Example construction hypergraph for a simplified geoscientific model construction

The construction hypergraph contains four construction steps. The *digitise*, and *meshing* construction step are performed once, while the *project* construction step is applied twice to different input data.

4.1.1. Reproducibility Based on Construction Hypergraphs

A construction hypergraph contains an unique sequence of construction steps that are used to construct a particular geoscientific model. In addition, it contains information about the relationships between datasets and construction steps. We intend to use this information to verify that a construction process is reproducible. An empirical method for checking reproducibility is to repeat the construction process with the same input datasets and compare all intermediate and output datasets produced by the repeated construction with the initially supplied datasets. If all intermediate and output datasets match the initially provided datasets, we can be reasonably confident that the construction process produces the same results and is therefore reproducible. This approach is inspired by the strategy used for reproducible software builds as presented in section 3.5.

Performing such a verification requires the following steps:

1. Store any datasets generated or processed by the construction process
2. Repeat the construction process with the stored input datasets from step 1
3. Compare all intermediate and output dataset with their originally provided counterparts.

The first step, storing datasets, is necessary because these datasets are used in later steps of the verification procedure. Input datasets are used as the basis for reproducing the result.

They are identical across different realisations of the same construction process as each realisation is based on the same input datasets. Intermediate and output datasets are used to compare the original construction results with the newly generated realisations. It is impossible to repeat the construction without providing all input datasets. This, in turn, makes it impossible to check that the same output datasets are generated. This fact implies that our system must be able to store a possibly diverse amount of different datasets.

The second step, repeating the construction process, requires access to the corresponding construction hypergraph, that encodes the structure of the construction process. Therefore, our application must store the construction hypergraph together with the required datasets. To enable the repetition of the process later without the need of human interaction, we must record the entire process such that a computer can execute. Such a recording enables the execution of a construction hypergraph, which requires each construction step to be provided in a computer-executable format. In general, computers can execute any compiled program or script written in a programming language compatible with the current execution environment. Section 5.2 describes several designs for such a system in detail. As part of our requirements, we stated that we must be able to use other programs in our construction workflow. This means that these programs, and any program used to execute the construction step description itself, must be available in the runtime environment to repeat a construction step later. We call a construction step description self-contained if it includes all necessary information about all programs used and a complete description of the actual actions to be executed.

As the final third step of the verification procedure it is required to compare any produced dataset with their originally provided counterpart. This requires defining equality between geoscientific datasets. Section 4.1.2 discusses this problem in detail.

All in all we define reproducible construction hypergraphs as follows:

Definition 4.9 (Reproducible construction hypergraphs). We define the construction workflow to be reproducible if all generated intermediate and output datasets match their initial counterpart. This can be defined as $\forall j = 1 \dots n, {}^jD_{R_0}^O = {}^jD_{R_1}^O$ where R_0 is the originally provided realisation and R_1 is the reproduced realisation of the output dataset ${}^jD^O$ of the j step of the construction hypergraph. \square

4.1.2. Equality definitions

Using the information provided by the construction hypergraph, it is possible to repeat the construction of the same geoscientific model with the same input data.

Definition 4.9 raises an important question: How should equality of geoscientific datasets be reasonably and verifiably defined? A conclusive answer does not seem to exist [66]–[68], as each specific use case has its own specific requirements. We describe four general approaches to define equality of geoscientific datasets:

Definition 4.10 (Bitwise equality). We consider two datasets D_a and D_b as equal if:

- The number of files in D_a and D_b are equal
- For each file $x_a \in D_a$ there is a file $x_b \in D_b$ with the same name and size
- For each file $x_a \in D_a$ the corresponding file $x_b \in D_b$ consists of the same sequence of bytes

□

Bitwise equality is independent of the data format used, since all data is represented as files. It makes it possible to reason about equality without considering the details of specific file formats. The definition of bitwise equality, in turn, can classify datasets as not equal, that would otherwise be considered equal in many practical applications. One example for which bitwise equality is well suited are images. It is reasonable to expect that images representing the result of a complex construction process will be identical. On the other hand, this approach fails as soon as a timestamp or other source of pseudo-randomness is included in the resulting datasets.

Definition 4.11 (Structural equality). We consider two datasets D_a and D_b to be equal if for each part $P_a^i \in D_a$ (as defined in definition 4.1) there exists an equal part $P_b^i \in D_b$ for a list of defined required parts P^i . □

The list of required parts depends on the specific format of the dataset and may include only a specific part of the dataset. This definition allows to ignore parts of the dataset, which are considered irrelevant or disturbing at this moment. A common example are timestamps generated as part of the model construction process. It also removes the constraint that each part must occur in the same order in both datasets D_a and D_b . For triangulated surfaces, for example, the definition of structural equality allows two datasets to be considered as equal even if the triangles are stored in different order in the corresponding files.

Definition 4.12 (Distance-based equality). We consider two datasets D_a and D_b as equal if $|D_a - D_b| < \epsilon$ for a given threshold $\epsilon \geq 0$ and a given distance metric $|\dots|$. □

This definition is based on the established practice of comparing floating point numbers by allowing a minimal difference. The commonly used equality definition considers two floating point numbers to be equal if their difference is less than some small threshold. The distance-based equality definition extends this approach to more complex datasets. Compared to bitwise equality, this approach allows small differences between the resulting datasets by choosing a threshold ϵ greater than zero. Similar to structural equality, the distance based equality definition allows potentially irrelevant parts of the dataset, such as timestamps or folder names, to be ignored by simply excluding them when defining the distance function. Conversely, this approach to defining equality requires a distance function that is specific to a particular datasets format. This function must include and possibly exclude information from the appropriate files. In addition, an appropriate value for ϵ must be chosen, which in turn may depend on the actual datasets. Simulation results are a major example of using this approach to define equality. Most mathematical models already provide a distance function that is used in the underlying numerical simulation. We can apply the same function to check whether two datasets generated by independent model realisations are considered to be equal.

Definition 4.13 (Geological equality). We consider two datasets D_a and D_b to be equal if they support the same geological interpretation. □

Unlike the other three approaches, this definition is somewhat fuzzy. This fuzziness makes it hard to impossible to perform an automatic check based on geological equality. It is ongoing research [67]–[69] on how to transform this definition of equality into something more suitable for automatic checking. We consider this approach to be of limited use for use cases that require automatic equality checking. Geological equality may be useful with respect to complex subsurface models consisting of many elements. For non-geological geoscientific models, such as a hydrological balance model, this approach is not useful.

Deciding which equality definition is most appropriate in practice must be done on a case-by-case basis. The following factors influence this decision:

- The software used to produce the dataset, as some software may introduce factors such as timestamps or internal information into the produced datasets. In these cases, bitwise equality will not work well.
- The application for which the geoscientific model is designed. Is the model built to be used later as the basis for a numerical simulation, or is it intended to visualise something? In the first case, the order and position of triangles matter. In the second case, it may be acceptable to use completely different triangles as long as the model looks the same. In the first case bitwise equality seems to apply, while for the second case applying geological equality definition is sufficient.

4.1.3. Design Constraints

Overall this results in the following design constraints for our construction hypergraph, for the implementation of concrete construction steps, and for the definition of equality used to reason about reproducibility:

Design constraint 4.1 (Computer-Executable Construction Steps). Each design step must be in a computer-executable form, either as a stand-alone program or as a description that can be executed by a specific program. □

Design constraint 4.2 (Self-Contained Construction Steps). Each construction step must be self-contained. Requirement 2.4, “Computation environment independence” states, that we cannot assume that we have access to the computer with which the model was originally constructed. □

Design constraint 4.3 (Customisable Equality Definition). There must be a way to customise the equality definition for datasets formats. If there is no custom implementation, the application can use the most restrictive bitwise equality to directly compare the output dataset files. □

4.2. Data Handling

In the previous section, we focused on construction processes that consist of construction steps. Now, we will take a closer look at these datasets and the requirements they impose on the design of our system. To illustrate the general workflow we will use our simple example construction process.

Figure 4.6 contains the graphical representation of the corresponding construction hypergraph. There you can see that the construction process starts with three different sets of input data:

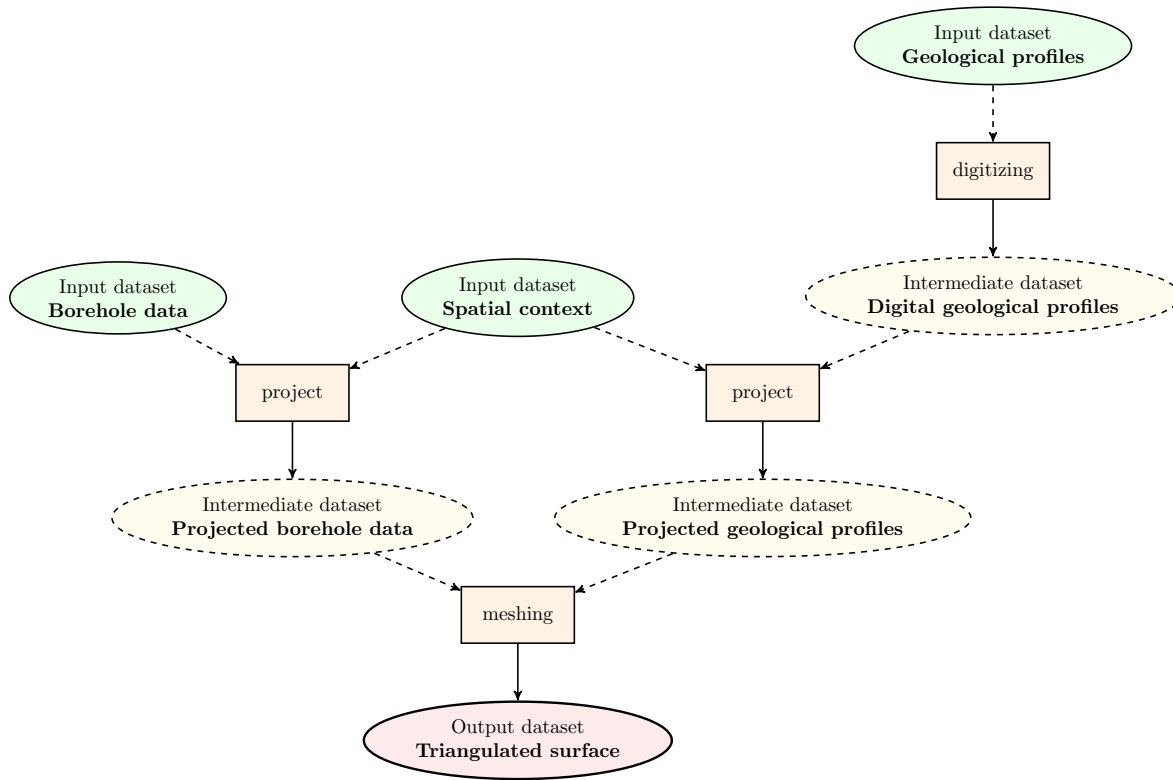


Figure 4.6.: Example construction hypergraph for a simplified geoscientific model construction

1. Borehole data
2. Spatial context
3. Geological cross-section

We assume that the user provides these datasets. They represent the starting point from which a construction process can be reproduced. Ideally, these data should be provided as unprocessed as possible in order to include as many relevant construction steps as possible in the reproducible construction process. Construction hypergraphs only provide information about the reproducibility of the construction process they represent, not about other parts of the process not represented by the construction hypergraph. This restriction is relevant for cases where preprocessing steps are already applied to input datasets. For our example, we would presume that both the borehole data and the geological cross-section data are taken from an archive, which means both likely represent the result of some preprocessing work. On the other hand, the spatial context represents a target parameter of the geoscientific model to be constructed. This allows the user to communicate this choice as an explicit input.

To verify that the construction process is reproducible, we must repeat the construction process using the same input datasets. To make these datasets available for these repetitions, we need to store them.

Next, these input datasets are processed by different construction steps. Each construction step is provided as a user-defined description using specialised programs. Consequently, we need to provide the datasets for each construction step in the format provided by the user as otherwise the construction step may fail due to an incompatible format of the provided dataset.

Each executed construction step generates a result dataset. To check whether a construction process is reproducible, we have to compare different realisations of these result datasets to

each other. Of course, this assumes that more than one realisation of a given result dataset is available at the time of comparison. Similar to the input datasets, one solution here is to store every dataset that is generated as a result of a construction step. This includes all intermediate and output datasets of the construction process.

The datasets themselves do not necessarily contain all relevant information. Sometimes additional information is provided as metadata. For example, the geological cross section input dataset is likely to be provided as an image. Images usually do not contain information about their spatial reference or orientation. Such information is usually provided by the user as metadata. This means that our software must store metadata associated with each stored dataset.

Our geological profile image is then used by the *digitise* construction step to result in a digital geological profile that may contain a spatial reference and orientation. This means that the construction steps need to have access to all metadata associated with their input datasets. A reverse construction step that generates an image from a digital geological profile will generate information about spatial location and orientation as metadata. This means that our application must provide metadata associated with the input datasets to the construction steps and apply the metadata generated by the construction steps to associate them with the corresponding output dataset.

Storing metadata in conjunction with specific datasets raises another difficult problem: What metadata is considered relevant? There is an ongoing debate about what metadata is relevant to geoscientific data [70]–[72]. Based on these discussions and the different needs and standards of different geoscientific domains, we decided to allow users to specify metadata schemes that meet their specific needs.

Last but not least, allowing users to define their metadata schema for construction workflows, has implications for all datasets that are part of this workflow. It explicitly includes intermediate and resulting datasets of the construction process. Now, if a user specifies that certain metadata is required for all datasets, that metadata for the datasets automatically generated as part of the construction process, must come from somewhere. Sometimes, the appropriate metadata is already included in the actual dataset. To verify its existence, we need allow users to automatically extract metadata as part of the construction process.

This results in the following design constraints for the data storage layer of our application:

Design constraint 4.4 (Store all Input Datasets). We need to store the user supplied datasets because construction steps are also user supplied and assume that their input datasets have certain properties □

Design constraint 4.5 (Opaque Datasets). We cannot assume much about the size and structure of datasets. We need to be able to handle small and large datasets. □

Design constraint 4.6 (Store Metadata). We need to store metadata for each input dataset provided by a user. □

Design constraint 4.7 (Pass Metadata). Metadata must be passed as input to construction steps, along with the corresponding datasets, and construction steps need to propagate metadata from their input data. □

Design constraint 4.8 (Automatic Metadata Extraction). The construction process needs to be able to automatically extract metadata from intermediate and result datasets of the construction process. □

Design constraint 4.9 (Metadata Schema Independence). The data storage layer needs to permit different metadata schemes for different construction processes, since there is no unique metadata schema for all geoscientific datasets. □

5. Design

In this chapter, we explore various options for implementing an application for our proposed concept. In section 5.1, we provide a general overview of possible application structures. In section 5.2, we will then explore how the construction steps can be represented based on the requirements and the design constraints outlined in the previous chapters. Finally, in section 5.3, we discuss how the data storage can be designed.

5.1. Application Structure

The goal of this thesis is to present a theoretical foundation and provide a concrete implementation of the theoretical ideas. In this chapter, we will present several possible application architectures and evaluate them based on our requirements from chapter 2.2 and our design constraints from chapter 4. We will also compare the presented architectures to the applications described in chapter 3. Each application must have a design that fits the tasks of the application. As mentioned by Fowler [73, p. 5] it is important to note that no solution can be used for all types of applications. Different applications require different application designs depending on their requirements.

Table 5.1.: Three Principal Layers according to Fowler [73] and Haffner [74]

Layer	Responsibilities
Presentation	Provides an interface for users to interact with the application
Domain	Implements logic for interaction and manipulation of data and application state
Data Source	Manages data and application state

Table 5.1 contains the classical three-layer approach for designing applications. Each layer performs different tasks. There are various ways to implement applications based on this principle. In this section, we will cover the following software architectures:

- Monolithic Desktop applications
- Client-Server architecture
- Micro-Service based architecture
- Fully distributed architecture

We will describe how they implement the three different principal layers, how this affects the overall application design, and how the different approaches compare.

Monolithic Desktop applications implement all three fundamental layers within the same application. Figure 5.1a shows a schematic overview of how the different application layers interact with each other in this type of application. Applications that use this pattern are self-contained, so they do not need to interact with other applications to provide their functionality. An example of such an application from Chapter 3 is gOcad [17]. The presentation

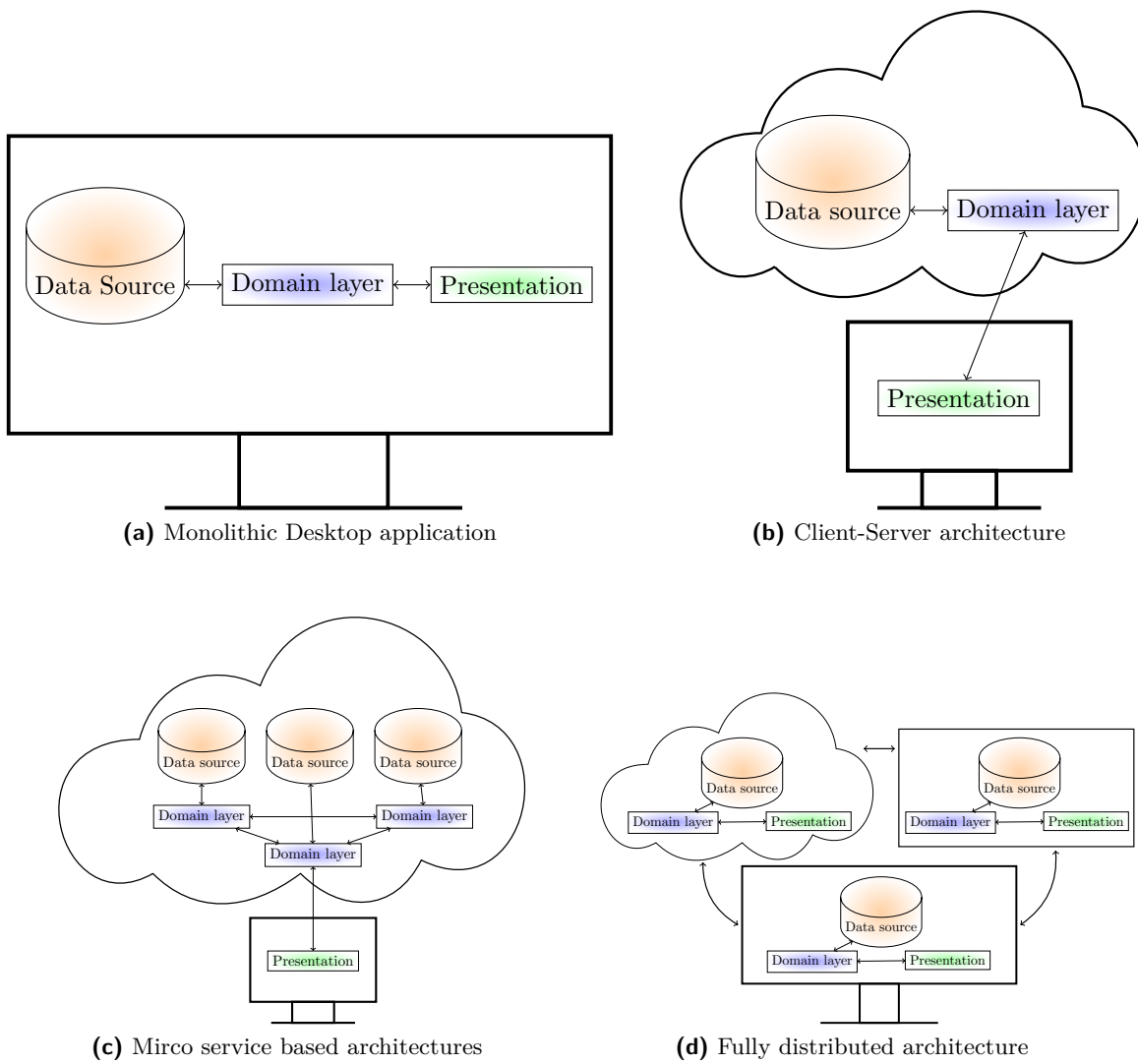


Figure 5.1.: Different software architectures

layer of desktop applications is usually tied to the frameworks provided by the platform on which the application is to be deployed. There are also cross-platform frameworks, such as QT [75] or electron.js [76], that provide abstraction across different desktop environments. The data source layer of such applications typically interacts with file-based data sources. In most cases, the domain layer is tightly coupled with the presentation and data source layers. Since each instance of a monolithic desktop application is independent of other instances, it is not easy to enable collaboration between different users. A common solution is to allow different application instances to share files stored in the data source layer. However, this approach usually assumes that only one instance is using the stored files at the same time. This limitation makes simultaneous collaboration difficult.

For **Client-Server-based applications** the three fundamental layers are distributed across two separate applications. Figure 5.1b shows a schematic overview of how the different application layers interact in this type of application. Examples of applications that use this pattern from Chapter 3 include PostGIS [31] and GST [36]. In all designs, the presentation layer runs as a different application than the data source layer. However, the domain layer can be shared by both applications. The application running the presentation layer is usually called the client, while the application running the data source layer is usually called the server. There are several ways to deploy a presentation layer. So-called thick clients implement the presentation layer using platform-specific or cross-platform frameworks, similar to monolithic desktop applications. Thick clients can also implement parts of the domain layer. Thin clients, on the other hand, typically expose the presentation layer as an HTML-based Web application. Thin clients typically provide only the presentation layer. There are several approaches to storing data as part of the data source layer as part of the server application. Commonly used approaches include relational or non-relational databases and file-based solutions. Depending on the use case, both the client and server application can implement parts of the domain layer. The communication between client and server application is done via a well-defined API (Application Programming Interface). Such a communication API can use different interfaces, for example, a network interface or file-based Unix sockets. A well-defined standard like HTTP [77] or an application specific protocol can be used to communicate over these interfaces. The characteristic client-server architecture allows real-time data exchange between multiple users, since several clients can interact with the same server simultaneously. Since the server application knows about all connected clients, it can provide mechanisms to synchronise the data modified by several clients simultaneously.

Compared to the other architectural patterns presented, the **Micro-Service-based Architecture** is a relatively new addition to common software architectures. Similar to the client-server architecture, micro-services distribute the presentation and the data source layers among different applications. However, instead of providing a monolithic server application that handles the entire data source layer and large portions of the domain layer, micro-services-based architectures divide these components into several smaller applications. Figure 5.1c shows a schematic overview of how the different application layers interact with each other in this type of application. According to the canonical pattern described by Fowler [78], each backend application handles one task. They all communicate with the client applications through a central API. Both the data source and domain layers can consist of many independent backend applications, each with a well-defined task. The presentation layer of a micro-service-based architecture can be implemented similarly to client-server applications. This means that either a thick-client application with platform-specific or cross-platform frameworks or a thin-client deployed as an HTML-based Web application can be used as the presentation layer. Splitting the server side of large applications has the theoretical advantage of making it easy to scale a single component of the system without having to duplicate the entire server-side of the application on several computers. Another advantage

of micro-service-based architectures is that it is easy to split the service into several smaller parts. This facilitates compliance with Conway’s Law [79], which states that any application design is, by and large, a copy of the organisation’s communication structure. Micro-service-based software architectures keep each service and its corresponding development team small, resulting in a lower organisational overhead.

In a **Fully distributed architecture**, there is no clear separation between the client and server parts of the application. Each instance of the application implements all three principal layers. According to Unmesh [80], a system is considered as a distributed architecture as soon as it consists of at least three separate instances. Each of these instances must provide the same functionality and can interact with the other instances. Figure 5.1d shows a schematic overview of how the different application layers interact with each other in this type of application. Examples for applications from Chapter 3 include DataLad [46] and DVC [51]. Each instance of a distributed application is a client application and a server application at the same time. To interact with other instances, a given instance must negotiate the global application state with other instances. Once they agree on the current overall state of the application, they can exchange data and interact with each other. A significant advantage of fully distributed systems is that they can be easily scaled to handle a much larger load. All that is required is to set up more instances of the same application. In addition, such systems are generally considered to be highly fault-tolerant, since a single missing instance can easily be replaced by one of the other instances. On the other hand, negotiating the current global application state with other application instances usually involves complexity.

5.1.1. Choice of Application Architecture for GeoHub

In the following section, we will evaluate the presented architectures based on requirements and design constraints. Table 5.2 contains an overview of which requirement can be satisfied with which architectural pattern.

Requirement 2.1 **Store Information** requires the storage of various types of data. Combined with design constraint 4.4 **Store all Input Datasets**, this places some requirements on our data source layer. It means that our application must provide all the necessary datasets required to reproduce the construction process when needed. An application architecture with a data source managed in a central location, makes it easier to satisfy these requirements. Client-Server or micro-service based applications can meet this requirement because both approaches share the same central data source layer for different client applications.

Requirement 2.2 **Skilled Personnel** does not impose any direct requirements on the software architecture of our application. We can argue here that a person with domain-specific knowledge should be able to install or interact with any scientific application from their domain.

Requirement 2.3 **Integration of Other Software** demands that our application interact with a large number of other software. Since there is no standard API for this interaction, we must create our own abstraction layer here. Much of this interaction occurs within the domain layer where our software must evaluate construction hypergraphs to perform constructions of geoscientific models. Monolithic desktop applications and client-server-based architectures have a strict design of their domain layer, that makes it part of the application. Since we want to control many different applications provided by our users, these designs do not fit very well. On the other hand, micro-service based architectures assume that independent applications provide the domain and data source layer. In our opinion, this is a much better solution satisfy this requirement.

Requirement 2.4 **Computational Environment Independence** imposes some requirements on possible application structures. Since we cannot assume that the computer of the person who constructed our geoscientific model is still available, we must at least move the data source layer to a location that is under the control of our application. In all architectures, that allow separation of the principal application layers between different applications, the data source layer naturally resides in a central part of the application infrastructure. These include client-server and micro-services-based architectures. For monolithic desktop applications, this requires additional functionality such as network share to move application data to a central location. Fully distributed applications, by definition, have no central location to for data storage. Depending on the application, all data may be distributed across some or all instances of the application. On the one hand, this can make the application much more resilient to data loss. On the other hand, it also makes it more complex to keep data in sync between all application instances. In our opinion, this complexity is unnecessary for solving the given task, and we do not consider a distributed application design to be appropriate for our use case.

Requirement 2.5 **Recording Changes** also does not place direct requirements on the overall software architecture. For the data storage layer of the application, it is necessary to allow changes to the stored geoscientific models over time. This can be supported with all four possible designs presented.

Table 5.2.: Possibility to implement Requirements based on application architecture. (+ Possible, – Not Possible, 0 Not Relevant)

Requirement	Monolithic Desktop	Client-Server	Micro Service Based	Fully distributed
Store Information	–	+	+	+
Skilled Personnel	0	0	0	0
Integration of Other Software	–	–	+	–
Computation Environment Independence	–	+	+	–
Recording Changes	0	0	0	0

We chose to implement our application with a micro-services-based architecture based on our requirements. In addition, we chose to implement our user interface as a thin client based on a single-page HTML application, as this makes the application independent of the client environment.

Our domain layer consists of several parts. First, there are components that are responsible for the executing construction steps. Due to the requirement 2.3 **Integration of Other Software**, this component must interact with other software. In addition, the domain layer provides functionality for comparing the results of construction processes. As described in section 4.1.2, equality depends on the actual structure of the dataset. Therefore, the user can provide custom equality definitions as described in design constraint 4.3 **Customisable Equality Definition** design constraint. Design constraint 4.8 **Automatic Metadata Extraction** requires automatic extraction of metadata. Similar to the implementation of equality functions, this component must provide a user-extensible mechanism for extracting metadata, since we cannot assume much about data formats. In section 5.2 we present possible solutions for designing an extension mechanism, and we will identify a good solution

for each use case. In addition, the domain layer must provide common functionality, such as restricting access to stored data to authenticated users and forwarding information from the data source layer to the presentation layer and vice versa.

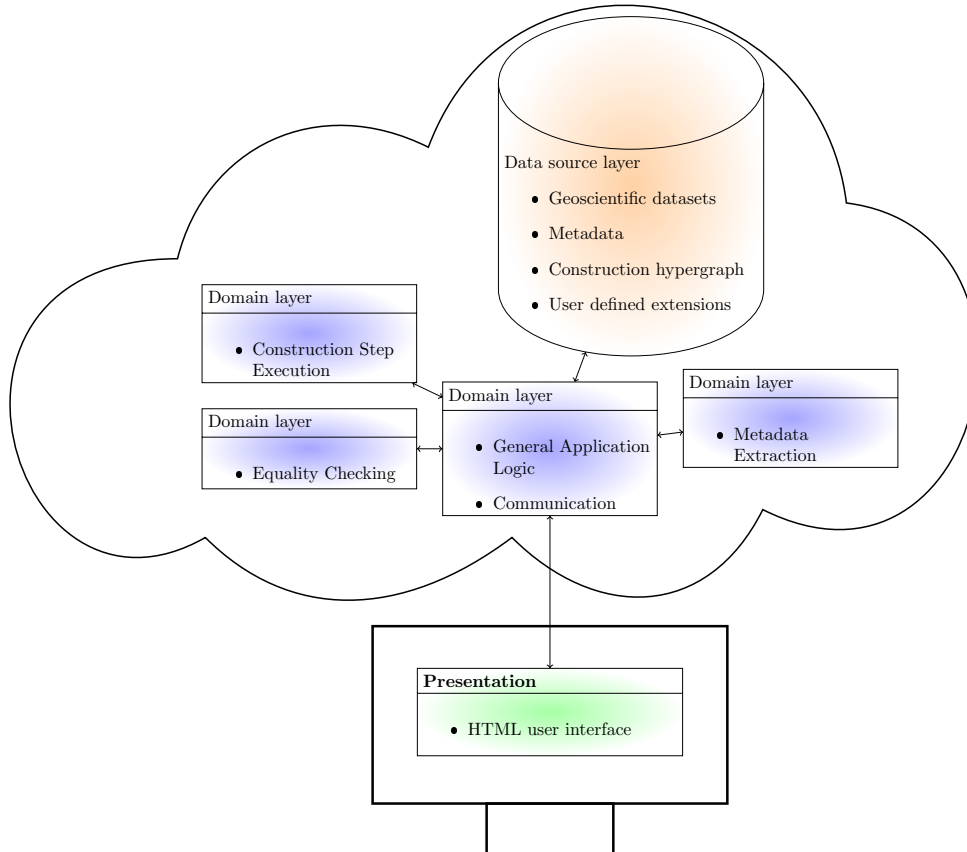


Figure 5.2.: GeoHub application structure

GeoHub uses a micro-service architecture that splits the presentation layer and domain and data source layers between different services. Each part of the domain layer is implemented by a different application.

Extensions for equality checking, metadata extraction and construction step execution are provided as different services as part of the application backend.

Our data source layer needs to store different types of data, including our geoscientific datasets, any existing metadata for those datasets, and the construction hypergraphs for each construction workflow. Each type of data has different properties. Consequently, each type of data must be handled differently. In addition, the data source layer must ensure that all stored data is consistent at all times. This means that each of the other layers can only access the data source layer in such a way that there is a consistent view from the stored data of the different parts of the data source layer. Section 5.3 provides details about the actual design of the data source layer.

This leads to the application structure shown in figure 5.2.

5.2. Extension Mechanisms

5.2.1. Overview

Our system must have several components where users can provide descriptions for performing a particular action. We intend to use custom extensions to automatically extract metadata from datasets, to check whether two datasets are considered to be equal, and to provide model-specific construction steps. All of these components will be used by the backend components of our service.

There are different strategies for implementing such an extension API (Application Programming Interface). In the following section, we define the requirements for these extension API, present some possible implementation variants, and compare them.

Each of our three use cases has slightly different requirements for such an extension system.

The metadata extraction API must take a dataset as argument and return a structured record with all relevant metadata as response. We will refer to this extension API as the **Metadata Extraction API**.

An API for comparing the equality of two datasets takes the two datasets as arguments and returns a boolean value, indicating the equality of both datasets as a response. We will refer to this extension API as the **Dataset Equality API**.

An API for representing a construction step consumes a set of input datasets and their corresponding metadata and produces an output dataset with its associated metadata. We refer to this extension API as the **Construction Step API**.

Manero [81] lists four possible solutions for integrating extensions into an application:

1. A shared library based extension system via the Foreign Function Interface (FFI)
2. An Inter-process communication (IPC) based extension system
3. An extension system based on a scripting language
4. An extension system based on a WebAssembly Interface

We will evaluate these four extension systems based on the following criteria. The order of the presented criteria corresponds to their importance for our system.

Requirement 5.1 (Security). GeoHub is designed to store potentially large amounts of data, some of which is potentially confidential. Under no circumstances should a user-provided extension allow users to bypass the applications (and operating system's) established access rules for stored data. This includes privilege escalation for user accounts, accessing data without the proper permissions, or even manipulating data that cannot otherwise be accessed. The most important question to answer here should be: "What could a malicious user do to use the extension system to expand their otherwise limited privileges?" □

Requirement 5.2 (Stability). Extensions, like any software, can misbehave. This includes non-safety-critical problems such as memory leaks, extensive memory usage, or random aborts. Of course, none of these behaviours is desired. An extension system needs to have built-in measures to handle potentially misbehaving extensions before they can affect the stability of the host application. □

Requirement 5.3 (Reusability). Some users may want to reuse other applications or parts of their otherwise developed code bases as part of their extension. It is necessary to provide users with a development environment they are familiar with to make extension development as easy as possible. This includes the support of programming languages that users may already be familiar with. Such a language choice must be combined with an extension API that is designed to be used with existing applications or libraries. \square

Requirement 5.4 (Portability). The extension system should not impose any significant barriers from the perspective of a potential user. This means that it should be as easy as possible to build new extensions. Existing extensions should be as portable as possible in terms of requirements for the runtime environment, the operating system and, more generally, the hardware environment used. Or, to put it another way: “No extension should depend on specific properties of the computational environment provided.”. This necessity is a direct consequence of requirement 2.4 **Computational Environment Independence**. \square

Requirement 5.5 (Interoperability). The extension system should integrate easily with the main application. It should be possible to seamlessly call functions provided by possible extensions, get results back from the extensions, and transfer the corresponding data between our application code and the extension implementation. \square

Requirement 5.6 (Performance). Since we want our system to handle potentially large datasets, and each dataset might be processed by one or more user-provided extensions, the performance of the user-provided extensions is an important consideration. No one likes to wait for anything, so the extension system should allow users to write potentially fast extensions. An important factor here is that it is possible to pass data to extension efficiently, since for each request we need to pass a set of files to extract metadata. Another relevant aspect in this context is that the tool or programming language used to develop extensions must support the generation of efficient code in the first place. \square

5.2.2. A Shared Library Based Extension System

A shared library based extension system is a classical approach for extending applications developed in a compiled language [82]. The extensions are provided by the user as a shared library. These libraries are then loaded at runtime via `dlopen` [83, pp. 860–862] or similar interfaces. Functions, defined as part of the extension API are then dynamically searched and called via `dlsym` [83, pp. 862–865] or similar interfaces. Well-known examples of extension systems based on this principle are the native extension system of PostgreSQL [84] and kernel modules of the Linux kernel [85].

From the point of view of **security** (Requirement 5.1), an extension system based on shared libraries must be considered problematic. Each extension is considered to be part of the host application, which means that there are no barriers between the extension and the host application. This means that all code of all extensions is executed in the same context of the host application. Since our host application communicates with the client-side web interface through a network interface, each extension can establish a network connection. Also, since our host application stores data as part of the data source layer, any extension could access and modify the same data. This, in turn, would mean that a malicious user could write an extension that sends confidential datasets over the network or modifies or deletes valuable datasets without the host application even noticing. A common workaround for this problem is to restrict the registration of extensions to trusted users, as it is the case with PostgreSQL

or Linux. In our case, this would mean that only administrative users can register extensions, since their access rights would already allow them to perform all of the above actions. Another way to minimise this problem is to use existing sandbox mechanisms like Native Sandbox [86] or Vx32 [87] to limit the actions an extension can perform. However, applying such mitigation can significantly affect the performance of extensions.

As mentioned earlier, code loaded from shared libraries is considered part of the host application. Roughly speaking, this means that if the extension leaks memory, the host application also leaks that memory. If the extension allocates a large amount of memory, this is attributed to the host application. All of this would not be a problem in itself, but shared libraries in most cases contain code compiled into native machine code. For this reason, terminating a misbehaving extension from the host application requires access to low-level machine details. Freeing accidentally leaked resources is almost impossible there. In combination, this can have a negative impact on the **stability** (Requirement 5.2) of the main application.

A shared library based extension system generally allows **reuse** of code (Requirement 5.2) as long as the programming environment used by the existing code provides facilities for building a shared library. This is true for most compiled languages such as C, C++, or Fortran, but not for most languages that depend on an extensive language runtime, such as Java or Python. Users therefore can develop extensions based on code in any language that supports compilation to shared libraries. However, in most cases, this requires writing an intermediate code layer to connect the existing code to the extension API. Consequently, it is not possible to reuse entire applications, only libraries.

Reasoning about the **portability** (Requirement 5.4) of a shared library-based extension system is a complex task. Shared libraries have different structures depending on the operating system used. See, for example, the ELF specification [64] and the PE specification [65] for Linux and Windows specific library formats. Furthermore, shared libraries target a specific hardware platform such as AMD64 [88] or AARCH64 [89]. Moreover, a shared library may depend on one of the extensions of the processor hardware platform to improve their performance. Examples for AMD64 are AVX (Advanced Vector Extensions) or AVX2, an examples for AARCH64 is SVE (Scalable Vector Extensions). As an additional dimension, there are several calling conventions that describe how a symbol corresponding to a particular function is named in a shared library and how that function must be called. These conventions are referred to as ABI (Application Binary Interface). For example, they specify how a symbol name specified in the source code should be converted to a symbol name that is part of the final binary code, or what kind of values should be passed to a function in a particular CPU register, and which CPU register is used by a function to return values. An ABI depends on the processor architecture and the programming language used to develop the shared library [90]. In addition, most ABIs depend on the actual compiler implementation [91], [92]. A common choice to reduce this multidimensional compatibility matrix is to use an ABI based on the C standard [93], since it describes a language- and compiler- independent ABI. A consequence of such a design is that it limits the supported programming languages to languages that provide an ABI based on the C standard. Overall, an extension system based on shared libraries requires users to provide an extension that exactly matches several features of the host application runtime environment. Moving the application installation to a new system may change the requirements for extensions, as some of the above items may change. This may result in incompatibilities with existing applications. This feature is in direct conflict with the requirement 2.4 **Computational Environment Independence**

From the point of view of **interoperability** (Requirement 5.5), a shared library based extension system is easy to implement for a given platform. All major operating systems provide an API for dynamically loading shared library. Therefore, implementing an abstraction layer

based on these interfaces is possible as for example demonstrated by the Boost C++ library [94].

The **performance** (Requirement 5.6) of an extension system based on shared libraries can match the performance of the host application code because the extension is provided as a compiled binary. However, security measures such as sandboxing the shared library may add overhead by requiring additional code to be executed for certain operations.

5.2.3. Inter-Process Communication Based Extension System

One way to address the security aspects of an extension system based on shared libraries is to move the extension to a separate process. This process can then run with limited capabilities using tools provided by the operating system, such as virtualisation, or advanced sandboxing techniques. Consequently, the host application and the extension can no longer integrate directly with each other. They must use an explicitly defined channel to communicate with each other. This mechanism is called IPC (Inter-process communication) and can use different interfaces. Common choices are network interfaces, file-based sockets, or simply STDIN/OUT (Standard Input/Output Interface). Examples of extension systems based on external applications are third-party **cargo** subcommands [95] and **Pandoc** filters [96]. There are several ways to implement an extension system based on this principle. We present an approach where the extension is wrapped in a Docker container [97], as this suggested by Boettiger [98].

The **security** (Requirement 5.1) properties of an extension system based on external applications are superior to the properties of the approach based on shared libraries. The Docker based approach in particular offers numerous advantages here, as it essentially packages each extension application in its own separate virtual operating system. In addition, this approach allows to explicitly shares certain resources such as files or network sockets with the extension.

Offloading extensions from the host application into separate applications improves the **stability** (Requirement 5.2) of the host application, as malfunctioning extensions can no longer crash the host application. Furthermore, the Docker based solution has the additional benefit of limiting the resources used by extension applications, such as main memory and CPU time. This can further limit the impact of misbehaving extensions.

The Docker based approach makes it relatively easy to **reuse** existing applications (Requirement 5.3), because Docker images are designed to provide a runtime environment for almost any Linux-supported software. However, since the extension application must somehow communicate with the main application, the extension itself or a proxy application must implement the communication protocol defined by the host application. Thus, it is usually not possible to use existing software exclusively.

A Docker based extension system does not suffer from most of the system-specific **portability** issues (Requirement 5.4) listed for the shared library based extension system. Since the host application is not directly linked to the extension application, there is no need to consider ABI, extension structure, or extension dependencies. This is accomplished by packaging the extension into a Docker image. Extension dependencies on specific hardware platforms remain because the extension is essentially deployed as a compiled application running in a Linux virtual environment. Docker does not abstract the hardware itself.

An extension system based on external applications is more difficult to **integrate** into the host application than the approach based on shared libraries (Requirement 5.5). The host application must maintain the communication interface through which the extension application can be controlled. In addition, a communication protocol must be implemented that enables the use of all functionality provided by the extension application. In the case of a Docker based approach, the communication interface is partially provided by the Docker API, which allows controlling the environment of the external application.

Using external applications to implement an extension system has a larger impact on **performance** than a shared library based extension system (Requirement 5.6). When switching to another application, the operating system must perform a context switch. In addition, it is no longer possible to share resources such as memory or file handles between the main application and the extension application. All information needed by both applications must be explicitly transported via IPC. This results in an additional performance overhead compared to a shared library based system. Casalicchio and Perciballi [99] give an overview of how large the performance overhead is when running applications in a Docker container compared to running them natively. They conclude that CPU-bound workloads incur up to 10% overhead and IO bound workloads include up to 30% overhead. These results can serve as a lower bound for comparisons. However, the IPC mechanism used to control the extension application probably causes additional overhead.

5.2.4. An Extension System Based on a Scripting Language

Developing an extension system based on scripting languages is a common choice. Languages such as Lua [100], Python [101], or JavaScript [102] are often chosen. For example, gOcad, as one of the applications described in Chapter 3, provides a Python-based extension mechanism.

Script languages are executed within an interpreter, which means they have access only to the resources that the interpreter provides. Depending on the design and target audience of a scripting language, this may allow restricted access to critical resources, or it may generally allow access to all system resources. This property makes the **security** (Requirement 5.1) of a script language based extension system dependent on the actual language choice. Typically, common scripting languages such as Python or JavaScript allow by default access to various system resources.

Depending on the actual implementation, it may be possible to control the execution of extension code in detail. For example, some interpreters provide a feature that allows to restrict the resource usage. In addition, most implementations provide an interface through which the execution of running code can be interrupted. In conjunction with built-in resource metering in the main application, this allows termination of extension implementations that are not behaving correctly to ensure the **stability** of the main application (Requirement 5.2).

A scripting language based extension system allows users to **reuse** any code already written in that specific language (Requirement 5.3). Depending on the language chosen, this may include many relevant libraries. Unfortunately, most languages do not allow the reuse of code written in languages other than the scripting language itself.

Scripting languages are designed to be **portable** across different operating systems and hardware platforms (Requirement 5.4). These languages depend on an interpreter implementation that translates the language itself or an intermediate representation into executable machine code at runtime. Therefore, as long as a compatible interpreter is available, it is possible to

execute the same code regardless of the hardware platform or operating system used. This compatibility guarantee explicitly excludes platform-specific features and extensions.

The **interoperability** (Requirement 5.5) of a scripting language based extension system with the host application depends on the actual language choice. Fortunately, for most languages there exists some libraries that expose the actual language implementation behind a usable API. For example, language implementations such as Lua [100] are explicitly designed for integration into larger applications. On the other hand, JavaScript implementations such as the V8 Engine [103] are designed for use in web browsers. Such a design makes integration into different applications notoriously difficult.

Scripting languages are considered less **performant** than languages compiled to native code [104] because the code is translated at runtime instead of translating it in advance ahead of time and performing more complex optimisations there (Requirement 5.6).

5.2.5. An Extension System Based on a WebAssembly Interface

WebAssembly [105] (WASM) is an emerging technology originally developed as a compact bytecode format for use in web browsers. In recent years, browser-independent runtimes such as Wasmtime [106] and Wasmer [107] have been developed. They allow WebAssembly bytecode to be executed outside of web browsers in a sandboxed environment. WASI [108], a capability-based ABI design, enable interaction with system APIs. As with the shared library-based approach, the host application calls specific functions of a user-supplied WebAssembly binary.

From the point of view of **security** (Requirement 5.1), WASM-based interpreters are an excellent solution. As the name implies, WASM is a technology with its roots in the web development environment. Every part of this environment is concerned with the execution of untrusted code. For example, any code executed by a web browser as part of a web page must be considered untrusted code. With WASI, there is a fine-grained, capability-based framework that allows bytecode interpreters to constrain the capabilities of code executed within those interpreters. This framework makes it possible to restrict the capabilities of extensions to pure computations only.

The WebAssembly code is executed by a runtime, similar to the scripting language based solution. In addition, runtime systems usually provide the ability to control how many computational resources an extension uses. See, for example, the Wasmer Metering Middleware [109]. This functionality enables the termination of faulty extensions to ensure the **stability** of the host application (Requirement 5.2).

An important motivation for the development the WebAssembly bytecode format was the possibility to use this format as a compilation target for existing programming languages. Consequently, this format can be used as a compilation target for existing compiled languages such as C, C++, C#, Fortran, Rust, or Go. This allows **reuse** of code written in any of these languages (Requirement 5.3). Similar to shared library-based extensions, it is only possible to reuse code fragments. All these fragments must be integrated into the extension API defined by the host application.

WebAssembly is designed to be **portable** across different operating systems and hardware platforms, even when accessing system APIs via WASI. (Requirement 5.4)

From the point of view of **interoperability** with the host application (Requirement 5.5), a system based on a WebAssembly interpreter is similarly complex to an extension approach based on scripting languages. Standalone WebAssembly runtimes are provided as a library

that can be integrated into the host application. By using the functionality provided by these runtime libraries, it is possible to parse, interpret and execute WebAssembly code as part of the host application. Some technical work is required to pass the necessary data to the extensions. This integration work requires converting data to and from the ABI defined by the WebAssembly standard.

The Kripken team [110] claims that the **performance** of WASM-based bytecode (Requirement 5.6) is only up to two times slower than the same C code compiled into native optimised machine code. In general, we expect some performance overhead with this approach, since the extension system must translate the portable bytecode before executing it. However, we expect this overhead to be less than the overhead caused by external application based and scripting language based extension systems, based on the numbers cited.

5.2.6. Comparison

Table 5.3.: Comparison of the different approaches based on our criteria (+ Positive, – Negative, 0 Neutral)

	shared library based extension system	Inter-process communication based extension system	Script language based extension system	WebAssembly Interface based extension system
Security	BLOCKER	+	+	+
Stability	–	+	+	+
Reusability	+	+	0	+
Portability	–	0	+	+
Interoperability	+	–	0	0
Performance	+	–	0	0

Table 5.3 gives an overview of our six criteria and our four extension systems. Overall, this table does not give us a result that says: “This system is the best choice.” Each of the systems presented has its strengths and weaknesses. Therefore, we have to decide on a case-by-case basis which extension system is best suited for which use case in our application.

Based on our previous requirements list, we consider **security** and **stability** of the main application as essential features. This rules out an extension system based on shared libraries for all three extension APIs, as this approach has weaknesses in this respect.

The Metadata Extraction API and the Model Equality API must interact with datasets in specific geoscientific file formats. However, both extension APIs require code written explicitly for that specific task. This means that while users may want to reuse code, they are unlikely to reuse entire applications. A scripting language based extension system and a WebAssembly interface based extension system would allow this. We chose a WebAssembly based extension system because it is likely to allow more code to be reused as code written in multiple programming languages can be translated into WebAssembly bytecode. This is illustrated by the higher **Reusability** rating of the WebAssembly based extension system in table 5.3. Another potential advantage of using a WebAssembly based extension system is that the extension can be used as part of the HTML single-page application and as part of the various backend services.

Different requirements apply to the Construction Step API. The Requirement 2.3 **Integration of Other Software** explicitly states that it should be possible to reuse software. This requirement means that an extension system based on inter-process communication is the

only viable solution, since all other solutions do not allow the reuse of entire applications. As discussed in section 5.2.3 one common solution to implement such a system uses Docker images. These images allow software and all required dependencies to be combined into a single environment definition.

Table 5.4 gives an overview of which extension mechanisms are chosen for which extension API.

Table 5.4.: Chosen extension mechanisms per extension API

Extension API	Chosen extension mechanism
Metadata Extraction API	WebAssembly based
Dataset Equability API	WebAssembly based
Construction Step API	Inter-process based system (Docker)

5.3. Data Storage

5.3.1. Overview

In order to be able to repeat the construction process of a geoscientific model at a later point in time, it is necessary to store various data required for the construction process. Figure 5.3 provides an overview of the different kinds of data stored by GeoHub. We need to keep the following data:

- Geoscientific datasets processed and generated by the construction process
- Metadata for these geoscientific datasets
- An abstract representation of the construction hypergraph
- The environment for construction steps
- Implementation of user-defined equality definitions and metadata extractors

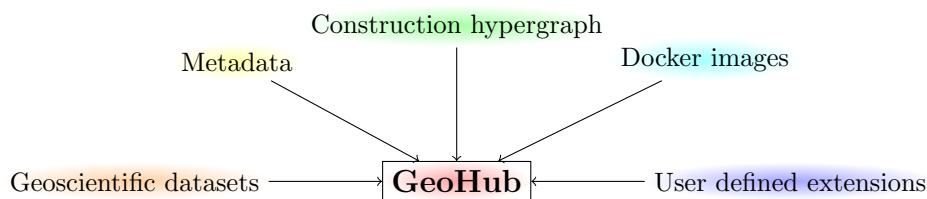


Figure 5.3.: Different data kinds stored by GeoHub

Each type of data requires a different strategy to store it efficiently. In this section, we will review different methods for storing this data. We will then evaluate each strategy based on various criteria and select a reasonable approach for each kind of data.

Criterion 5.1 (Flexibility to store data of different sizes and structures). For all the different types of data listed above, it is important that our system can store and retrieve the appropriated data. Depending on the exact nature of the data, this may require flexibility to process fundamentally different data in terms of size and structure. \square

Criterion 5.2 (Added complexity). From the software architecture point of view, a possible solution should add as little complexity as possible to the overall system. Additional complexity makes it more difficult to develop and maintain software. \square

Criterion 5.3 (Synchronisation guarantees). We design our system such that it can be used by more than one user collaboratively. As consequence, we need to synchronise the application state between different users. Any possible storage solution should make such synchronisations as simple as possible. \square

5.3.2. Stored Data

5.3.2.1. Geoscientific Input and Intermediate Datasets

The design constraint 4.4 **Store all input datasets** directly implies that we need to store geoscientific datasets that will be used as input to a construction step or that are produced by a construction step. These are fundamental building blocks of geoscientific models.

A construction process may depend on different datasets stored in different file formats. Well known examples include:

- Seismic measurement data, usually stored as “SEG Y” [111]
- Remote sensing images (can be stored as GeoTiff [112], HDF5 [113])
- 3D geometric data (can be stored as GoCad ASCII file [42], VTK file [114])
- 2D geometric data (can be stored as ESRI Shapefile [115])
- Tabular data (can be stored as CSV [43], Excel File [116])

As a result, our application must store data that may differ fundamentally in size and structure. Design constraint 4.5 **Opaque datasets** explicitly states that we cannot assume anything about size and structure of these datasets. We estimate that smaller datasets, such as, the result of a small-scale geoelectric survey as presented in section 7.2, have a data size in the order of several kilobytes. On the other hand, large-scale seismic surveys [117] can quickly produce data in the order of several terabytes.

In addition to the different data formats and sizes, we also need to consider how our application will use these data later. As shown in section 5.3.2.3, users can describe complete construction processes involving external software programmes as a construction hypergraph composed of construction steps. This means that the datasets must be provided in the expected format for the corresponding steps of the construction process.

In summary, this means that criterion 5.1 **flexibility to store data of different sizes and structures** is essential for storing geoscientific datasets.

5.3.2.2. Metadata Attached to Geoscientific Data

Table 5.5.: Example key value metadata schema

Key	Value
Author	Georg Semmler
Location	50.91°N, 13.33°E
Modified	Tue, April 16 2020 16:43

Metadata is commonly referred to as data that describes other data. Common examples are the author of a particular file, the date the file was last modified, or the location where a record was recorded. A common theme here is that a subject such as author, modification

date, or location is always combined with a concrete value. This inherent structure leads directly to the natural representation of metadata as key-value pairs. Each key represents the subject of the metadata, and each value represents the metadata itself. Table 5.5 contains an example key-value set.

There are ongoing discussions about which metadata keys are important for which areas of scientific research. Different competing standards list different sets of required metadata keys. The following lists contain well-known examples:

- ISO 19115, Geographic Information – Metadata [118]
- ISO 19119, Geographic Information – Services [119]
- Inspire Guidelines [120]
- DCAT application profile for data portals in Europe [121]
- GeoDCAT-AP: A geospatial extension for the DCAT application profile for data portals in Europe [122]
- USGIN Metadata Profile [123]

These standards describe which metadata must be stored for specific geoscientific datasets. For this purpose, they define a list of required metadata keys. In addition, specifications can be made for the structure of the associated values. For our example above, this could specify, for example, that “Author” and “Modified” must be present, while “Location” is optional. In addition, “Modified” must be a timestamp, while “Location” should represent a location in the form of coordinates.

Based on the design constraint 4.6 **Store Metadata**, we want to store metadata as it is provided by the user. Since we cannot know which metadata schema might be right for a particular user, we need to provide a flexible solution that supports more than one schema. This in turn means that the **flexibility** of the storage solution according to criterion 5.1 is essential for storing metadata.

Should we store metadata in a different location than the geoscientific datasets, we must ensure that both storage solutions are **synchronised** according to criterion 5.3.

The storage of metadata is an essential part of our application. For this reason, we can accept additional **complexity** (Criterion 5.2) caused by additional data storage solutions.

5.3.2.3. Construction Hypergraph Representation

The data stored for the construction hypergraph differs from the geoscientific datasets and their associated metadata in one crucial aspect: The format of the construction hypergraph is defined by our application. In contrast, the format of the former two types of data can be controlled by the user for the reasons described in the corresponding sections 5.3.2.1, 5.3.2.2. This fact means that we know a lot of details about the structure of the construction hypergraph itself. Based on this knowledge about the structure of the construction hypergraph, we can design the data format used such that the different parts of the application can use it optimally. As described in section 4, a hypergraph is a data structure consisting of a set of nodes connected by a set of hyperedges. A common representation of ordinary graphs are incidence matrices. Given a directed acyclic graph G with a set of nodes $N = \{N_0, \dots, N_n\}$ and edges $E = \{E_0, \dots, E_e\}$ the corresponding incidence matrix $A = [a_{ij}]$ for $i \in 0, \dots, n$ and $j \in 0, \dots, e$ is defined as follows:

$$a_{ij} = \begin{cases} 1, & \text{if } E_j = (N_i, N_x) \text{ with } x \in 0, \dots, n \\ 0, & \text{if } N_i \notin E_j \\ -1, & \text{if } E_j = (N_x, N_i) \text{ with } x \in 0, \dots, n \end{cases} \quad (5.1)$$

An additional constraint for regular graphs is that each column contains exactly two non-zero entries. Also, one of the entries must be positive and represent the head of the edge, while the other is negative and represent the tail of the edge. We can use this data structure to see which edges a node belongs to by looking at the non-zero entries of the corresponding row of the incidence matrix. Also, we can use this data structure to see which nodes are connected by an edge by looking at the corresponding column of the incidence matrix.

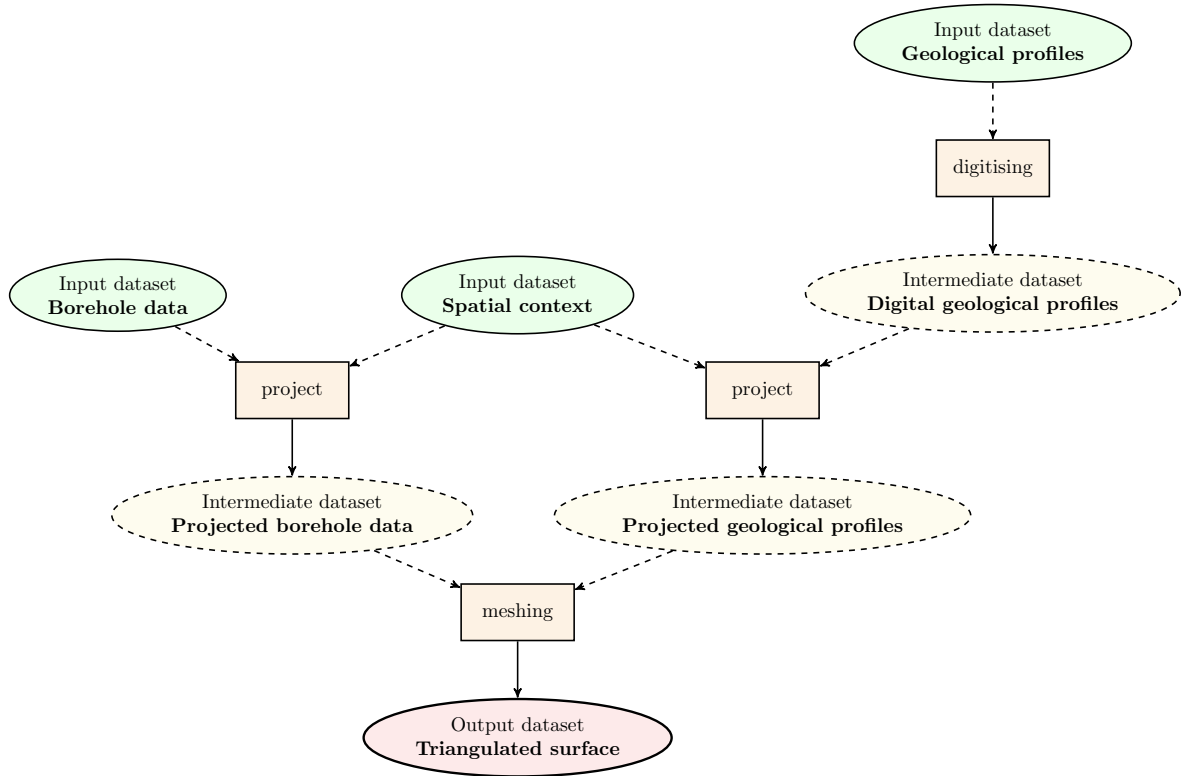


Figure 5.4.: Example construction hypergraph for a simplified geoscientific model construction

Table 5.6.: Incidence matrix belonging to the example construction hypergraph shown in figure 5.4

	digitise	project (1)	project (2)	meshing
Geological profiles	-1	0	0	0
Digitised geological profiles	1	-1	0	0
Borehole data	0	0	-1	0
Spatial context	0	-1	-1	0
Projected borehole data	0	0	1	-1
Projected geological profiles	0	1	0	-1
Triangulated surface	0	0	0	1

We can easily generalise incidence matrices for acyclic directed hypergraphs by changing the condition for non-zero entries to: Each column contains two or more non-zero entries, at least one of which is positive and at least one of which is negative. Since we additionally require

that each construction step (hyperedge in the construction graph) has exactly one result, we can tighten this condition to require that each column contains exactly one positive value and at least one negative value.

Figure 5.4 shows an example construction hypergraph. Table 5.6 contains the corresponding incidence matrix.

An incidence matrix can describe the structure of a construction hypergraph. However, our system must also store information about which geoscientific dataset is associated with a given node and which construction steps correspond to a given edge in a given construction hypergraph. In terms of data size, the representation of a construction graph is much smaller than a single geoscientific dataset, consisting only of the incidence matrix and a small set of references to the information stored elsewhere.

Since we can control the data structure used for construction hypergraphs, a possible data storage solution does not require much **flexibility** in data size and storage (Criterion 5.1). However, this component must work with the other parts of our system. Therefore, it is much more important that a possible storage solution provides good support for **synchronisation** as defined in criterion 5.3. Furthermore, to simplify the overall implementation, a storage solution for this type of data should not add additional **complexity** (Criterion 5.2) to our system. We should therefore prefer to use an existing data storage solution if possible.

5.3.2.4. Environment for Construction Steps

As described in section 5.2, we plan to use a Docker based system to execute construction steps. This is necessary to meet the requirement 2.4 **Computational Environment Independence**. Docker images provide a defined environment that can be used to execute software.

```

1 FROM r-base
2
3 RUN apt update && \
4     DEBIAN_FRONTEND="noninteractive" apt install -y openjdk-11-jdk && \
5     ln -s /usr/lib/jvm/java-1.11.0-openjdk-amd64/ /usr/lib/jvm/default-java
6
7 RUN R -e "install.packages('xlsx')" && \
8     R -e "install.packages('zoo')" && \
9     R -e "install.packages('dplyr')" && \
10    R -e "install.packages('ggplot2')"
```

Listing 5.1.: Example Dockerfile from one of the Docker containers used as part of the case studies

Docker images are usually built with a Dockerfile as environment description in text form. Listing 5.1 contains an example Dockerfile. Docker images are represented as file system overlays [124], that can be used to combine multiple layers into a final image. Each layer is created by executing a statement from the corresponding Dockerfile. Each layer is built on top of all previous layers. Our example Dockerfile contains three statements, each of them produces such a layer. The FROM statement defines a base layer. The two RUN statements both create a new layer based on the result of the given command executed within the previous layer. The Docker implementation defines the storage format of these images. Depending on the software included in the produced image, the size of an image can quickly reach several gigabytes.

Since the process of building a Docker is well described by **Dockerfiles**, it may seem reasonable to save only the **Dockerfile**. These files are much smaller, with a size of a few kilobytes. With this approach we would need to rebuild the Docker image each time that particular container is used. There is a significant difference between this approach and the approach of using stored Docker images directly. If only the **Dockerfile** is kept instead of the already built Docker images, the time at which the image is built shifts from before it is uploaded to the registry to each time a construction step dependent on that image is executed. Commonly used statements in **Dockerfiles** install software, sometimes by downloading the corresponding packages from the Internet. This is the case with both **RUN** statements in our example **Dockerfile**. If we repeat the build process later, we may download a newer version of the same software, leading to potential different model construction results. Another, even worse result is that the build process might fail because the server hosting the software used is no longer available. Both of these effects can result in construction processes that are not reproducible. For this reason, we need to store the images themselves to achieve our fundamental goal of providing reproducible construction processes.

A storage solution for storing Docker images does not need to be **flexible** according to criterion 5.1, since Docker images have a single defined format. However, it must be **synchronised** (Criterion 5.3) with the other parts of our application. Otherwise, a construction step could refer to a Docker image that does not exist. Since the construction steps and their environment are fundamental to our application, some additional **complexity** according to criterion 5.2 caused by an independent storage solution is acceptable.

5.3.2.5. Implementation of User Defined Extensions

As already described by the design constraint 4.3 **Customisable Equality Definition** and the design constraint 4.8 **Automatic Metadata Extraction** we need to deal with user-defined extensions. Section 5.2 concludes that we will use a WebAssembly based extension system for these use cases. These extensions need to be stored by the backend application for later use.

The WebAssembly standard [105] describes two possible representations: Textual and Binary. It also states that the textual representation is intended for teaching and debugging purposes, while the binary representation is intended for production use cases. Therefore, it can be assumed that the extensions are provided as a file in the binary WebAssembly format. WebAssembly code in binary form typically produces files ranging in size from kilobytes to megabytes. This gives us a defined size and structure of data, that needs to be stored. Consequently, we do not require **flexibility** according to criterion 5.1 in terms of data structure and size.

Storing user-defined extensions is by no means the main goal of our application. As consequence we want to avoid adding more **complexity** (Criterion 5.2) than necessary to the overall system to store these type of data.

The outcome of geoscientific model constructions will depend on such extensions, as they provide definitions for equality and may extract necessary metadata for later usage. This means that we must ensure that any concurrent access to the stored extensions is **synchronised** according to criterion 5.3 using appropriate techniques either by the data storage solution itself or by our application.

5.3.3. Potential Solutions

5.3.3.1. Raw File Storage

We define a raw file storage as a storage solution that allows arbitrary data to be stored as an opaque unit. Examples include the local file system, shared remote file systems, or cloud-based solutions such as AWS S3 [53] and similar services. Each unit stored in a raw file storage is identified by a unique identifier. This identifier can be used to access, modify, or delete the corresponding unit. File systems use file paths as identifier, while cloud-based solutions typically use specialised solutions such as Amazon's distinct object identified [125]. We focus on using local file storage as a storage solution for our design. However, future versions may replace the local file storage and use cloud-based solutions instead.

The raw file storage offers maximum **flexibility** in terms of the structure and size of the stored data (Criterion 5.1).

Using the raw file system to store data does not add additional **complexity** according to criterion 5.2 to our application, since a file system is already present in all environments.

File systems do not provide comprehensive guarantees for concurrent access to the same data [126]. The details depend on the operating system, the file system used, and even the underlying hardware. These dependencies make it difficult to develop an application that abstracts across different environments. Consequently, the application itself must **synchronise** concurrent accesses in a meaningful way according to criterion 5.3.

5.3.3.2. Relational Database Systems

Relational database systems are a popular choice for storing structured data. Implementations use interfaces based on dialects of the Structured Query Language (SQL) [127] to access and manipulate stored data. Relational database systems offer several advantages compared to raw file storage. One main advantage is the extensive guarantees for concurrent data accesses [128]. Relational database systems are designed to store large amounts of structured data. For concrete considerations, we focus on the open source database system PostgreSQL [32], since it provides a well established mature SQL implementation for free.

PostgreSQL provides a **flexible** storage solution (Criterion 5.1). It is well suited in configurations where the structure of the data is known in advance. In addition, PostgreSQL offers data types such as `JSONB` [129] or `HSTORE` [130] for storing unstructured data. These types assume a key-value based data structure, where keys and values are defined dynamically. In our own experience [39], relational database systems provide good performance as long as the stored information is accessed in small pieces. If larger contiguous chunks of data are accessed, performance degrades.

Using a relational database system increases the overall **complexity** of the designed system according to criterion 5.2. Using a PostgreSQL database to store data requires an additional application as part of the application setup.

PostgreSQL provides extensive guarantees for **synchronisation** of data access. These guarantees apply to all data stored within the relational database system. However, as soon as a relational database system is combined with other storage solutions, additional effort is required to synchronise the data between the different solutions to fulfil criterion 5.3.

5.3.3.3. NoSQL Document Stores

The commonality of NoSQL document storage systems is not so much a standardised design approach as a different design compared to relational databases. Consequently, different implementations offer different features and trade-offs. In the following considerations, we will focus on the open source document database MongoDB [131]. The MongoDB project describes the database as a document-based database system. The term “Document” refers to a data structure that contains a structured representation of information. A document-based database system is a database system that addresses a set of such documents with a unique key. MongoDB itself uses a document structure derived from JSON. This approach enables the storage of structurally different documents in MongoDB.

MongoDB provides the **flexibility** to store data with unknown internal structure (Criterion 5.1). Each document can contain its own set of key-value pairs. The size of a document stored in MongoDB is limited to 16 megabytes [132].

The use of MongoDB increases the overall **complexity** according to criterion 5.2 of the planned system, since an additional component is added.

In theory, MongoDB provides guarantees for **synchronisation** of concurrent data accesses according to criterion 5.3. In practice, however, these guarantees have proven to be insufficient on several occasions [133]–[136].

5.3.3.4. Neo4J

Neo4J [137] is the leading platform for managing graph-based datasets in a database. It provides a custom query language called Cypher [138] for querying, filtering and modifying graph-based datasets. In addition, Robinson et. al. [139, p. 54] show that it is possible to store complex dependencies and even hypergraphs in a Neo4j database.

Neo4J is a specialised solution for storing graph-based datasets. This fact limits the **flexibility** of storing arbitrary structured data to such graph-based data structures (Criterion 5.1).

As with previous database systems, adding Neo4J as additional component to our system increases the overall **complexity** according to criterion 5.2.

As a database, Neo4j provides similar guarantees for concurrent data access as PostgreSQL [139, p. 162]. These guarantees allow the **synchronisation** of concurrent access to data stored in Neo4J. Similar to a PostgreSQL based storage solution additional synchronisation according to criterion 5.3 may be required to keep data consistent between different storage locations.

5.3.3.5. Docker Image Repository

Docker registries are specialised software that store immutable Docker image instances. There are several open source implementations such as Gitlab’s Container Registry [140] or the self-hosted Docker Hub’s version [141]. Docker registries use a name and version tag to identify stored images. The image name and version tag allow the same environment to be started again as long as the image is stored in the registry.

Docker image repositories are designed for a single use case: Storing immutable instances of existing Docker images. Unfortunately, this limits the **flexibility** according to criterion 5.1 for storing any other type of data, as it only supports the storage of Docker images.

Using a Docker image repository increases the overall **complexity** of the designed system because it introduces another component (Criterion 5.2).

5.3.3.6. Comparison

Table 5.7 gives an overview of the different requirements for the data to be stored. Table 5.8 summarises how well the various solutions can meet specific requirements.

Table 5.7.: Required criteria per data kind (+ Required, – Not Required, 0 As Required, x Avoid if Possible)

	flexibility	complexity	synchronisation
Geoscientific datasets	+	0	+
Metadata	+	0	+
Construction hypergraph	–	x	+
Construction step environment	–	x	+
User defined extensions	–	x	+

Table 5.8.: Criteria per storage solution (+ Positive, – Negative, 0 Neutral)

	flexibility	complexity	synchronisation
File Storage	+	0	–
PostgreSQL	+	–	+
MongoDB	+	–	0
Neo4J	0	–	+
Docker registry	–	–	0

Storing geoscientific datasets requires a high degree of **flexibility** in terms of data structure and size. Table 5.8 indicates that only the local file storage, PostgreSQL, and MongoDB provide the required flexibility. MongoDB limits the maximum document size to 16 megabytes [132], which is too small for general geoscientific datasets. Given previous experiences [39] with storing large amounts of data outside of relational database systems, we choose file storage as the solution for storing geoscientific datasets.

Metadata also need a **flexible** storage solution. However, unlike the geoscientific datasets themselves, they only require flexibility in terms of structure, not data size. The relatively small size makes MongoDB and PostgreSQL an excellent choice for storing this type of data. We chose PostgreSQL as the data storage solution for metadata because it provides more comprehensive guarantees of data correctness and synchronised data access than MongoDB.

Our application controls the data structure used by the construction hypergraph. Consequently, our storage solution does not need to be **flexible** with respect to the stored data as long as it supports the storage of the appropriate data structure. All storage solutions expect Docker registries support storing the necessary data structures. Since we do not want to add more complexity to our application than necessary, we will reuse one of the existing data storage solutions here. Both PostgreSQL and the local file storage can store the required data structures. The required amount of data required to be stored per construction hypergraph

is relatively small. We chose PostgreSQL as storage solution for the construction hypergraph, following a similar reasoning as for the storage of metadata

Storing the user-defined environment to execute specific construction steps requires storing Docker images. These images have a well-defined structure. Docker image registries provide specialised software to store these images without re-implementing the needed data structures. For this reason, using an existing Docker image registry implementation is a suitable solution for storing Docker images.

User-defined extensions are provided in the binary WebAssembly format. Since we do not want to add more complexity to our application than necessary, we will reuse one of the existing data storage solutions here. Docker image registries are designed to store docker images, which means they cannot be used to store WebAssembly extensions. Both PostgreSQL and the local file storage solution can store user-provided extensions as binary objects. Given the potential size and the requirement to always load a full extension at once, we decided to reuse the local file storage based on the same arguments for geoscientific datasets.

Table 5.9 summarises which storage solution is selected for which type of data.

Table 5.9.: Chosen Storage solutions for each data kind

Data Kind	Storage Solution
Geoscientific dataset	File Storage
Metadata	PostgreSQL
Construction hypergraph	PostgreSQL
Construction step environment	Docker Image Registry
User defined extensions	File Storage

5.3.4. Model Versioning

Requirement 2.5 **Recording changes** states that our geoscientific model may change over time. In the context of this section, we want to evaluate which parts of the information stored in our application can change over time in which ways. For this, we consider the same five different types of data as used in the previous section:

- Geoscientific datasets
- Metadata for the geoscientific datasets
- Abstract representation of the construction hypergraph
- Environment for construction steps.
- Implementation of user-provided equality definitions and metadata extractors

We look at two different ways how information can change.

5.3.4.1. Construction Hypergraph Versioning

One axis of change is to change the construction process of the geoscientific model itself. As long as users improve the geoscientific model itself, they will also change the methods used to construct the model, for example, by replacing a naive mathematical inversion with a more advanced implementation that produces better results. These changes translate directly into changes to specific construction steps, which might also modify the environment and tools used in these steps in terms of stored information. The newer method may have

different requirements, such as a newer Matlab version. Such different tooling requirements for construction steps will result in an updated or wholly changed version of the corresponding Docker image. As described in section 5.3.2.4, Docker repositories already include a versioning mechanism that addresses different versions of the same image using an identifier. This mechanism implies that each construction step must refer to a specific version of the environment in which it is to be executed.

Another way to improve the geoscientific model is to better combine the available information. An example would be a precipitation correction based on an additional data set of historical precipitation data. This type of improvement changes the overall structure of the construction graph. We chose to summarise both types of changes together as changes to the construction process. We will refer to specific instances on this axis of changes as **version**. A version concerns the construction hypergraph itself and the specific environments used to execute the construction steps. Our relational database system stores references to both as part of different tables. All stored information must be addressable by a version identifier that allows tracking changes in information over time. We realise this identifier by a monotonically increasing sequence of versions. In section 6.2.1.1 we present the exact integration into our database schema.

Definition 5.1 (Version). A version tracks changes to the construction hypergraph, individual construction steps and the environment used by these construction steps. \square

5.3.4.2. Construction Hypergraph Realisation Versioning

We consider changes to the datasets used as the second axis of change for the construction process. A prominent example is the replacement of an erroneous dataset with a corrected version. These changes are different from changes to the construction graph itself, as they can also affect outdated versions of the construction graph. For example, it is conceivable that after viewing a constructed model, one may notice some errors and correct them later. Since model construction can take some time, it is possible that other users have already continued to evolve the construction process itself, for example, by adding more construction steps. Changing the input dataset to a corrected version and then repeating the construction is a simple way to correct an incorrect geoscientific model. The new realisation of the construction process builds an entirely separate instance of the geoscientific model, including its own reproduction check. This workflow is particularly useful when a completely different set of input data is used, as it allows the same construction process to be applied to different situations. This approach can save significant time in cases where the same measurements and subsequent complex modelling are applied to different study areas.

We will refer to specific instances on this axis of changes as **revision**. This axis of changes concerns the geoscientific datasets and their associated metadata. A particular revision always refers to a specific version of the construction hypergraph. Nevertheless, a single version of the construction hypergraph can be used to construct different revisions of the underlying geoscientific model with changed input data. All information stored on geoscientific datasets and their metadata must be labelled with both a version and a corresponding revision. Similar to version, a monotonically increasing sequence is used to represent revisions and their inherited order. Specific revisions can then be used to address metadata stored in the relational database or to look up references to geoscientific datasets stored in the file system by appropriately tagging the corresponding reference in the relational database. We will present the exact integration into our database schema in section 6.2.1.1.

Definition 5.2 (Revision). A revision tracks changes to datasets used to create construction graph realisation. Each revision always refers to a particular realisation of a geoscientific model of a known version of a particular construction hypergraph. \square

5.3.5. Transactional security

5.3.5.1. Transactional Safety for Non-Concurrent Accesses

As mentioned earlier, we need to provide mechanisms to ensure data integrity. These mechanisms are important for cases where data is used together but stored in different data storage solutions. In the context of this section, we will focus specifically on the interaction between data stored in the relational database system and data stored in the file system. This primarily concerns the interaction between the construction hypergraph and the attached geoscientific datasets. With respect to versioning of data, we will discuss revisions, as they specifically change the dependency between the construction hypergraph and attached geoscientific datasets. However, the versions themselves are less relevant, as they only change the data stored in the relational database. There, data consistency can be ensured by the database system itself. For our examples, let's assume that there are two users, Alice and Bob. Both are working on the same construction process, on the same dataset at the same time. It is assumed that Alice's work starts a little bit earlier. Using these two users, we will illustrate some concurrent access situations that can occur and that must be handled. In principle, there are four possible basic operations that each user can perform:

1. Create a dataset
2. Load an existing dataset
3. Modify an existing dataset
4. Delete an existing dataset

Here, a dataset refers to a geoscientific dataset associated with a known version of the construction hypergraph. Thus, each operation creates a new revision for a given construction hypergraph with a stable version. Each operation can access data stored in the relational database system or the file storage. Relational database systems provide extensive data consistency guarantees [142]. File systems, on the other hand, generally do not provide consistency guarantees [126], [143]. Since we plan to store data in both places, we need to ensure they agree on the same application state. As shown by Semmler [39], a promising approach is to use the relational database system as the central source of truth. This approach can be briefly summarised as follows: If a file reference exists in the relational database system, it is assumed to exist in the file storage as well. If it does not exist in the relational database system, it does not exist at all for the system. This means that the main application must ensure that only references to files that are actually written to the file store are inserted into the relational database. In addition, the backend application must have exclusive control over the file location, which explicitly means that no other application modifies this data. In the following sections, we present a simplified sequence of operations for each possible basic operation. We assume an access pattern to both storage locations, file system, and relational database. Implementation details such as the actual payload of the operations are intentionally omitted to simplify the presentation. As a general strategy, we use a copy-on-write approach [144]. That is, we always write new data to a new location and never modify existing data. An update would then look like writing the updated data to a new location while leaving the old data in the old location. This approach ensures that references to stored files never change and therefore cannot be invalidated by an operation.

In the next subsections, we will present the exact sequence of actions for each basic operation. After that we will focus on different combinations of simultaneous operations.

5.3.5.1.1. Create Operations

Storing a new construction process in GeoHub involves two steps:

1. Design the corresponding construction hypergraph
2. Creating a new realisation of the model by uploading a new set of input data

In the first step, several construction steps are combined into a construction hypergraph. The backend application then stores this hypergraph in a new version. This operation only involves the relational database system, since the construction hypergraph is stored there. The hypergraph itself can be referenced via a newly created version identifier.

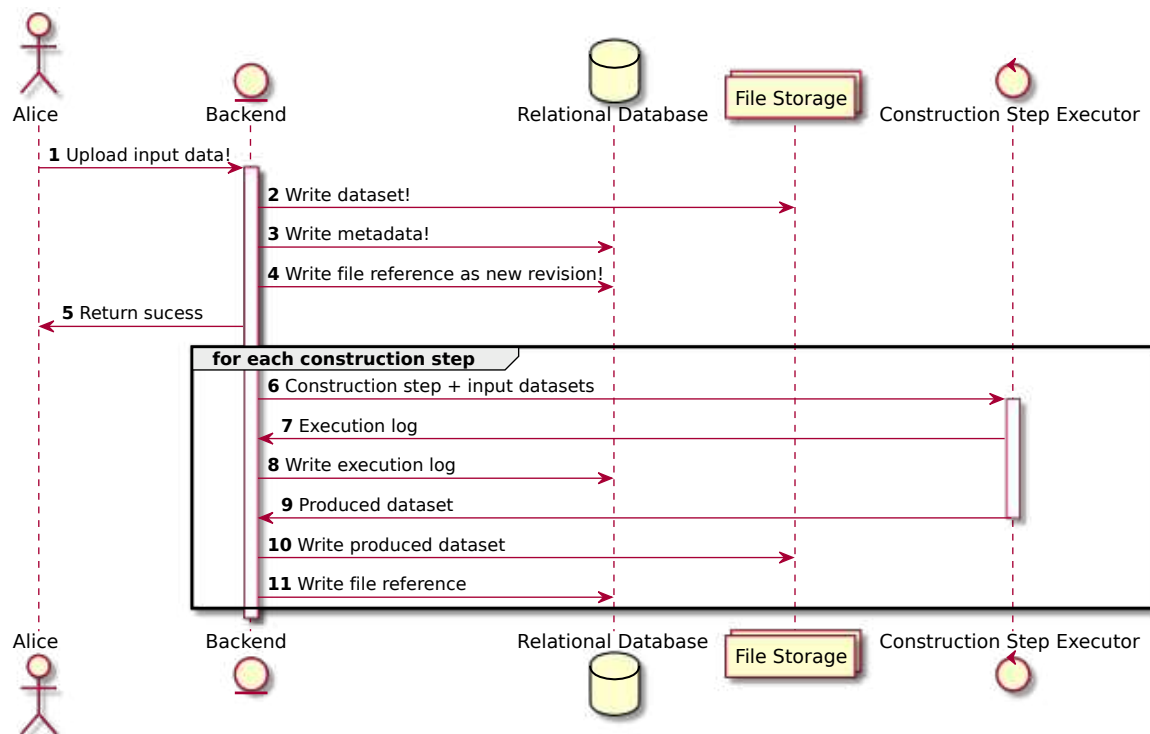


Figure 5.5.: Insert a single dataset

This sequence diagram shows how first the input datasets are written to the corresponding datastorage location. Then, each affected construction step is executed to generate the corresponding intermediate and output datasets

The second step includes other parts of our system. Figure 5.5 contains an example sequence for creating a new realisation of the stored construction process. Our user Alice sends a set of input datasets to the backend application (Step 1). This application now begins to distribute the data to the various storage solutions. First, the actual geoscientific datasets are written to the file storage (Step 2). Then, the backend application inserts the metadata and references to the geoscientific datasets stored in the file system into the relational database system (Step 3). The backend application creates a new revision to reference this data later (Step 4). Now, this stored data is available to other users. The backend application now reports the success back to Alice (Step 5).

After that, the backend-application starts executing construction steps depending on the provided input data. For this purpose, the construction step description and all associated

input data are sent to an executor (Step 6). The executor performs each action described in the construction step description. In addition, any command line output generated during the execution of the construction step is uploaded to the backend application and stored in the relational database (Step 7 + 8). This command line log can be used later for debugging purposes. The executor uploads the newly generated geoscientific dataset to the backend application once the execution of the construction step is complete (Step 9). The storage of this data follows a similar strategy as for the input datasets. The backend application first stores the actual geoscientific datasets on the file system (Step 10). Then, the metadata and references to the geoscientific datasets are inserted into the relational database system (Step 11). Finally, this data is appended to the old revision created in step 4 instead of using a new revision. From this point on, the data becomes available to any user.

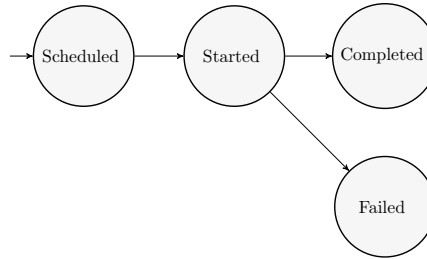


Figure 5.6.: State transition diagram for construction step states

The execution of a construction step always starts in the **Scheduled** state. As soon as the executor starts executing a construction step, the state changes to **Started**. Depending on the result of this execution, the state changes to **Completed** if the execution was successful or to **Failed** if the execution was aborted due to errors.

The backend application repeats this process for each construction step. This procedure requires that each construction step is performed exactly once. Otherwise, this process would generate duplicated data. We use the relational database system to ensure this, by keeping track of the state of each construction step for a given revision of a construction hypergraph. Figure 5.6 contains the state transition diagram for the various states of the construction steps. This tracking includes the states **Scheduled**, to indicate that an operation is ready to be executed, **Started**, to indicate that an execution has started, **Completed**, to indicate that a construction step execution has been completed, and **Failed** to indicate that a construction step execution has failed.

5.3.5.1.2. Load Operations

Figure 5.7 shows an example download procedure. Our user Alice requests an existing geoscientific model from the backend application (Step 1). The backend application receives a request with information about which dataset at which version and revision is requested. For both the revision and version a set of unique identifiers is used. The processing of a download request is divided into two phases. First, all requested metadata and references of all attached datasets is loaded from the relational database system (Step 2 + 3). The backend application then uses these references in step 4 to load the linked records from the file storage. This data is returned to our user Alice in step 5 and 6. No stored data or version or revision identifier are changed in this process. Since the files stored in the file storage are supposed to be immutable once references are inserted into the relational database system, we assume that this operation cannot fail for data consistency reasons. For a complete list of potentially conflicting access patterns see section 5.3.5.1.5 on concurrent data access.

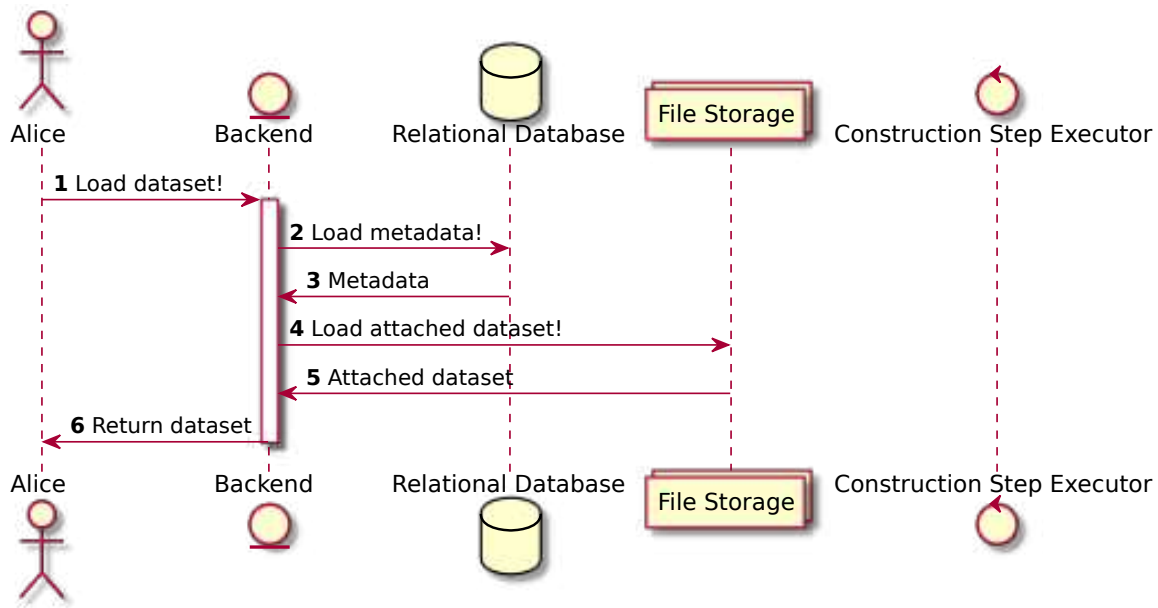


Figure 5.7.: Load a dataset

This sequence diagram shows how datasets are loaded in two steps. The first step loads the required metadata from the relational database system, while the second step loads the actual geoscientific dataset from the file storage

5.3.5.1.3. Update Operations

There are two different ways to update the data stored in our system. Alice could update the structure of the construction hypergraph, or she could upload new input datasets for the construction process and request to construct a new realisation of the stored geoscientific model using the existing construction process and new data.

For the first update case, a list of changes to the construction hypergraph and some unique identifiers, containing information about the construction graph to update, are sent to the backend application. Since the construction hypergraph is versioned, the backend application uses the provided information to create a new version of the construction hypergraph in the relational database. Older versions of the hypergraph stored in the database are not changed by this process. Instead, a new version of the construction hypergraph is created. The new version identifier identifies this new database entry. This type of operation only occurs in the relational database by appending new data, so the relational database system itself ensures data integrity. Data stored in the file system is not affected by this operation, since this operation does not affect file references or file storage. Since existing data, which is identified by an existing version and revision identifier, is not changed, it remains accessible.

The second update case assumes that a new set of input data is sent to the backend application. This data contains an updated version of a stored input dataset and a set of identifiers to determine the correct construction workflow. Figure 5.8 shows an example of such an update request. Conceptually, this involves the creation of a new realisation of the stored geoscientific model. This means that the process used is very similar to the creation process described in section 5.3.5.1.1. The only notable difference between an insertion request and an update request is that the update request does not require a complete set of input datasets. Instead, it can use data that has already been uploaded. At the end, the backend generates a new revision of the stored geoscientific model. The new data becomes available incrementally using the existing revision identifier.

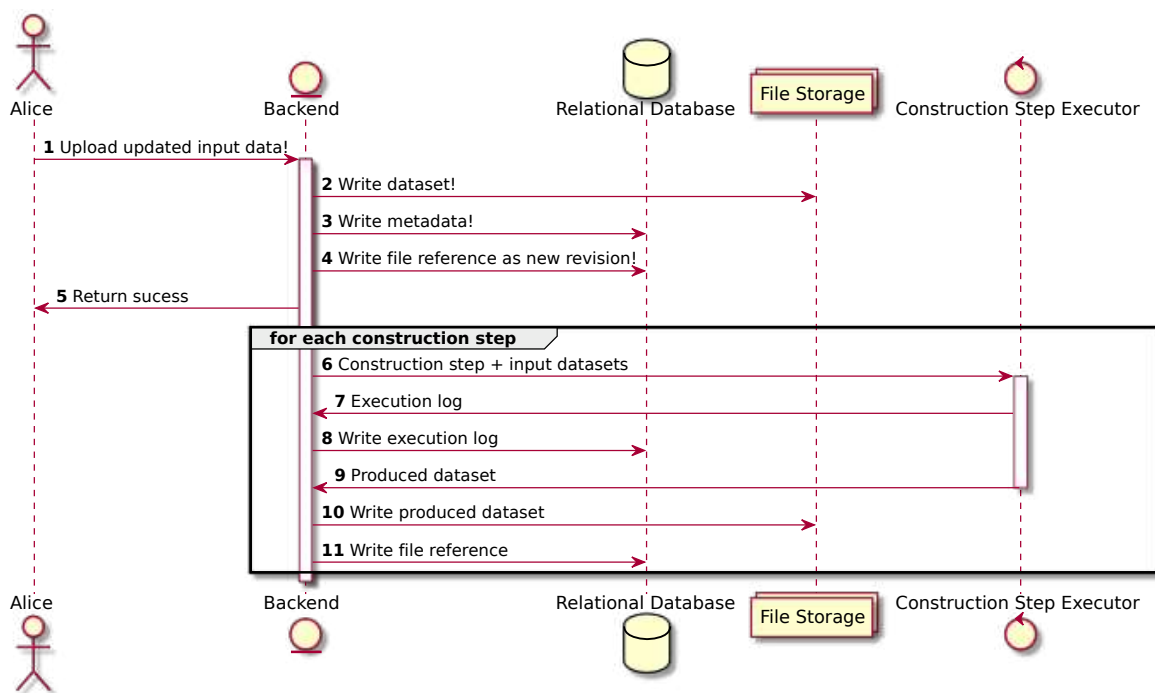


Figure 5.8.: Update a dataset

This sequence diagram shows that similar to a dataset creation first the input datasets are written to the corresponding datastorage location. then, each affected construction step is executed to produce the corresponding intermediate and output datasets. The notable difference to a dataset creation is that only a part of input datasets is needed.

5.3.5.1.4. Delete Operations

Depending on the use case, there are different delete operations. For example, let's say that Bob is a research assistant while Alice is an administrative user. Bob should be able to say something like: "I do not need this construction workflow anymore, so I want to remove it!" In theory, this could be done by actually deleting all the underlying data. Since our system aims to provide a platform for reproducible models, this seems to be a less than optimal solution, since this data would disappear in such a case. A better solution would be to mark this data as deleted in the application so that an administrative user like Alice could recover the data. To implement such a solution, the relational database needs to keep track of which data is marked as deleted and which is not. This is accomplished by an adding an additional field to the corresponding data tables. Data marked as deleted is not used by any other operations. Thus, from the point of view of such operation, this data does not exist. A delete operation affects only the relational database system, whose data integrity is guaranteed by the relational database itself.

A second use case for deleting data comes from our administrative user Alice. She might be required to remove data from the database because she is externally required to do so, e.g., by regulatory requirements to extract data or to free up hard disk space by removing unused data. From a conceptual perspective, such hard deletions should be avoided at all costs in a system intended to store reproducible geoscientific models. Since there are potential cases where such an operation may be required, our conceptual transaction model must be able to perform such an operation. Figure 5.9 shows an example sequence of a request, to remove data from the system. In the first step the backend application receives a request to delete a specific dataset or construction workflow in the first step. This request contains information about which data at which version and revision is to be deleted. The backend application

then queries the relational database system loading the references for all data that needs to be removed from the file system (Step 2). Subsequent steps remove the corresponding references (Step 4) and, if desired, the entire construction workflow from the relational database system. Both operations should be included in a database level transaction to ensure data integrity. Removing these references prevents future request from accessing this data. The final step is to remove all files from the previously loaded list from the file storage (Step 5). The backend application must ensure that there is no ongoing read access to these files. If this is not the case, it must postpone the actual deletion of these files until these requests are finished.

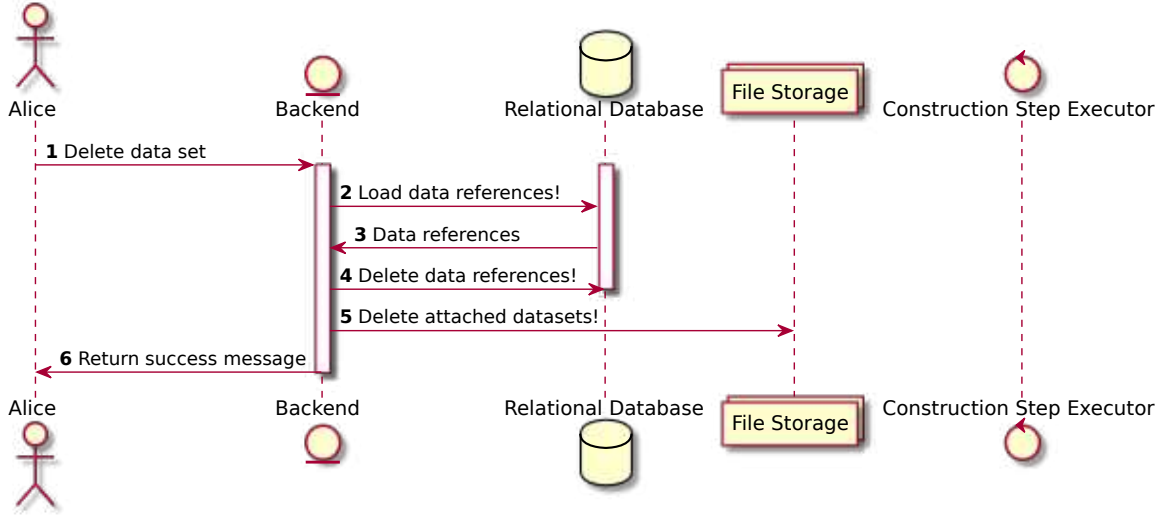


Figure 5.9.: Delete a dataset

This sequence diagram shows the multi step procedure used for performing a hard delete operation. First, the references to all datasets to be deleted are loaded from the relational database. In the next step, this data is removed from the relational database. Finally, the corresponding data is removed from the file storage.

5.3.5.1.5. Transactional Safety for Concurrent Data Accesses

This section will explain how GeoHub handles competing access patterns to the same datasets. Table 5.10 provides a complete list of cases that can occur when two parties attempt to execute an operation on the same dataset. For all cases, we assume that Alice's request is received by the backend application just before Bob's request arrives, but Bob's request arrives before Alice's request completes. In the next sections, we will classify these cases and present ways to prevent data corruption in all of these cases. For modifying operations we assume operations that create a new revision. For deleting operations we assume operations that remove all data. In the simpler versions of those operations only need to access the relational database system, so all potential integrity problems will be handled there.

Table 5.10.: Cases considered for concurrent data access patterns

Alice/Bob	Create	Load	Modify	Delete
Create	$C_A C_B$	$C_A \star B$	$C_A \star B$	$C_A \star B$
Load	$\star_A C_B$	$L_A L_B$	$L_A M_B$	$L_A D_B$
Modify	$\star_A C_B$	$M_A L_B$	$M_A M_B$	$M_A D_B$
Delete	$\star_A C_B$	$D_A \star B$	$D_A \star B$	$D_A \star B$

Case $C_A C_B$

For the case $C_A C_B$, Alice and Bob are trying to create a construction workflow. From a data consistency perspective, it is perfectly fine to create two separate workflow instances. This allows each request to be handled as described in paragraph 5.3.5.1.1 for a single operation. A separate version and revision identifier can be used later to refer to both workflows.

Case $C_A \star_B$

The case $C_A \star_B$ describes situations where Alice creates a construction workflow realisation and Bob tries to do something with that very realisation. As shown in Figure 5.5, each insert request first writes data to the file storage and then publishes references to that data in the relational database system. Figure 5.7 shows how a load request first obtains the necessary data from the relational database system that contains dataset references, and then accesses the file storage to load the actual datasets. Figure 5.10 shows how the backend application might handle the situation described for a load request. Here, Bob's read operation begins before Alice's write request is completed. Therefore, the relational database does not yet contain information about the data Alice uploads to the system. As a result, the backend application reports that no data corresponding to Bob's request was found. If the request is received from Bob while the backend application is running the construction workflow, the backend application can only return access to fully written datasets. We assume that this case is the same whether Bob is trying to load, update, or delete the record, since each of these operations requires that the appropriate dataset reference stored in the relational database is accessed first. These references are not available in the relational database system until the corresponding data has been completely written to the file system.

Case $\star_A C_B$

The case $\star_A C_B$ describes a similar situation to the case $C_A \star_B$ but with a slightly different timing. Alice is trying to access datasets in the backend application before Bob has created those datasets. From the backend perspective, this case is no different from other cases where a user attempts to access a non existing dataset, resulting in a "dataset not found" error being returned. Using similar reasoning as in the case $C_A \star_B$, we assume here that this case is the same regardless whether Alice attempts to load, update, or delete the dataset, since each of these operations requires first accessing the appropriate file references in the relational database system. These references are not found until Bob's request is complete.

Case $L_A L_B$

The case $L_A L_B$ describes a situation where Alice and Bob try to load the same dataset. From a data integrity perspective, such an operation is unproblematic because neither operation modifies existing data. Therefore, we can handle each of these requests independently, as described in paragraph 5.3.5.1.2.

Case $M_A L_B$

The case $M_A L_B$ describes a situation where Alice changes a dataset while Bob requests the corresponding dataset. As shown in Figure 5.8, an update request first writes data to the file storage and then publishes references to that data in the relational database system. In contrast, Figure 5.7 shows that a load request performs these operations in reverse order. If Bob requests the new revision of data that Alice is uploading, this operation order means

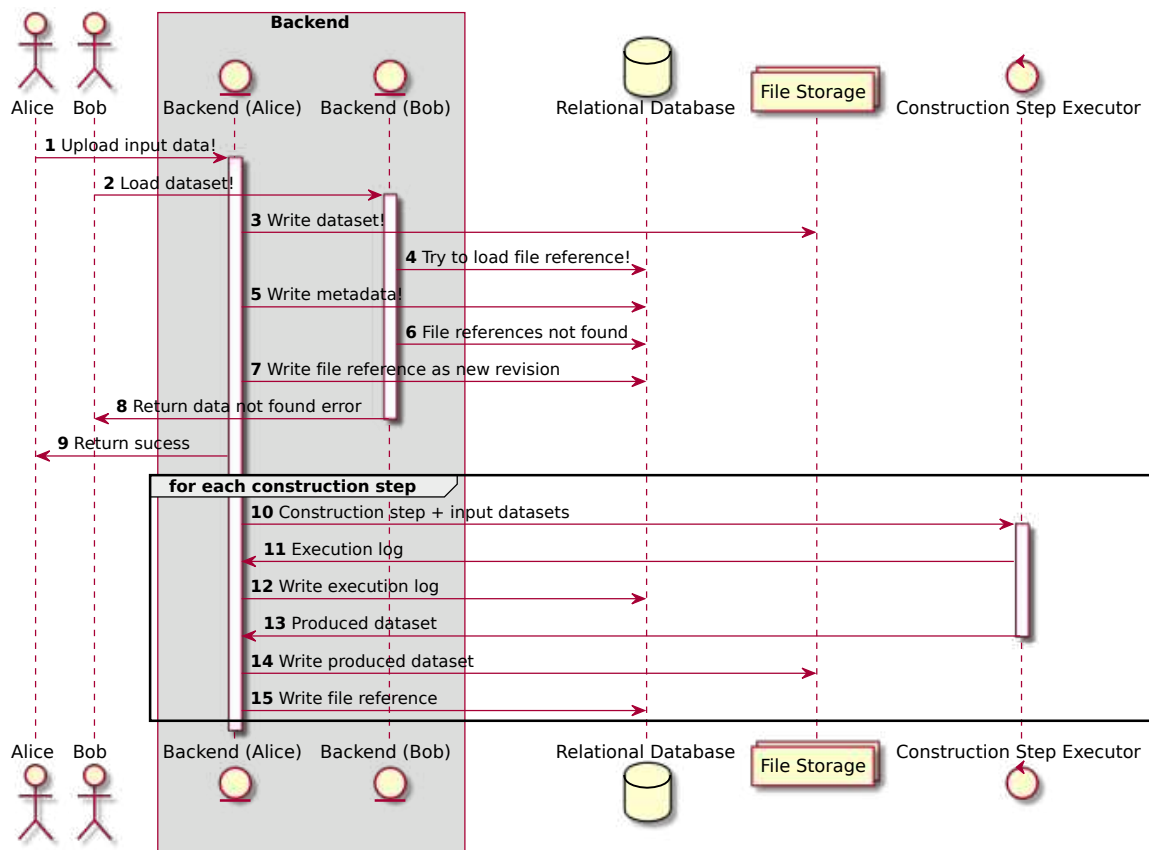


Figure 5.10.: Simultaneous access to a newly created dataset

This sequence diagram shows how GeoHub handles two simultaneous accesses from Alice (creating a new dataset) and Bob (loading this dataset). The sequence diagram splits the backend service into two instances to clarify which operations belong to which user requests.

that Bob’s request cannot see that data because Alice request has not been yet completed. Therefore, this case is treated similarly to the case $C_A \star_B$.

Case $L_A M_B$

The case $L_A M_B$ describes a similar situation where Bob starts an update after Alice has requested a specific dataset in a particular revision. Depending on the exact revision requested, either a “revision not found error” (for Bob’s new revision) or the data for the current revision (for the last revision before Bob’s update) is returned. The first variant follows the reasoning of the case $\star_A C_B$. In contrast, the second variant is based on the “Copy on Write” strategy, where the existing datasets are not changed, but a new dataset is written.

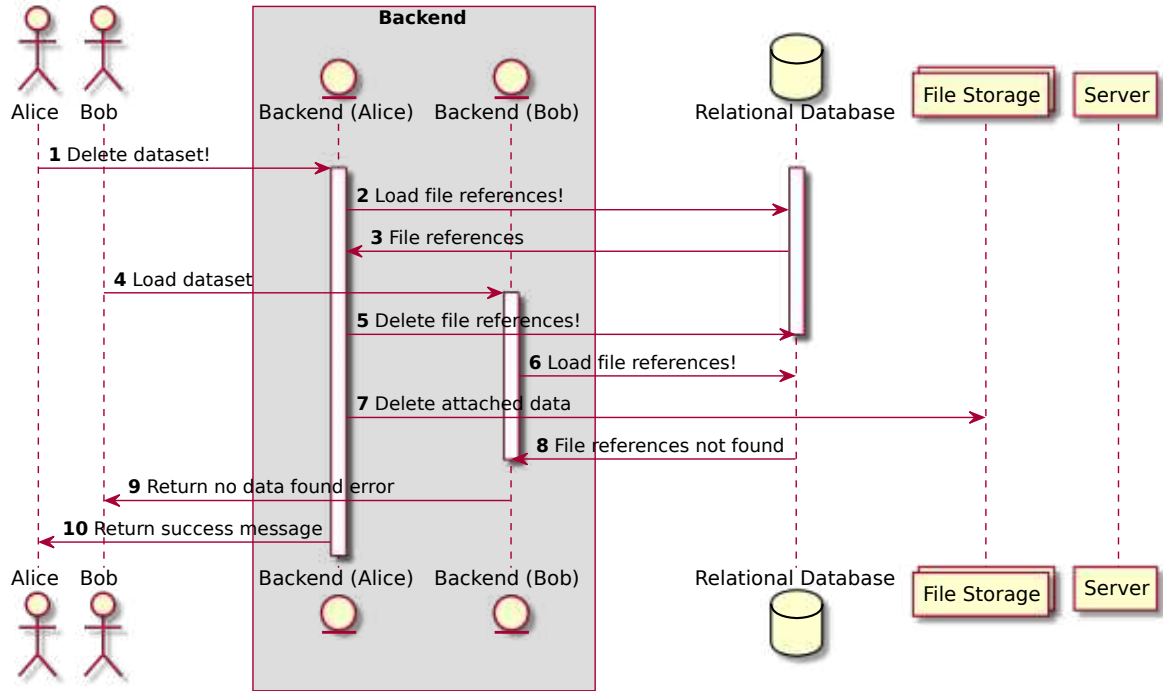


Figure 5.11.: Simultaneous access to a deleted dataset

This sequence diagram shows how GeoHub processes two simultaneous accesses by Alice (hard deleting a dataset) and Bob (loading this dataset). The sequence diagram splits the backend service into two instances to clarify which operations belong to which user requests.

Case $D_A \star_B$

Case $D_A \star_B$ describes a case where Alice deletes an existing dataset while Bob attempts to load, modify, or delete the dataset. Figure 5.11 shows an example timeline of events where Bob attempts to load an existing dataset. The request from Alice is received first. The backend application then reads the required metadata from the relational database and removes the references. This sequence of operations is wrapped in a database transaction to ensure integrity. Therefore, at the time Bob’s request is received, the relational database does not contain any references to the data. Bob’s request is then answered with a “no data found” error. In the meantime, the remaining data can be removed from the file system because the backend application has previously received a list of references to this data from the relational database system. We assume that this case is the same whether Bob is trying to load, modify, or delete the dataset because each of those operations accesses the metadata first.

Case L_AD_B

Case L_AD_B describes the variant where Alice tries to load a record while Bob removes it. Since the backend application receives Alice's request slightly earlier than Bob's, it will load the data. After receiving Bob's request, the backend application could go ahead and start deleting the corresponding dataset, but that might conflict with Alice's load request. The backend application could remove files that are required to complete Alice's request. As a result, the backend application must sort the operations accordingly. In the first step, all information can be removed from the relational database system because Alice's request has already loaded it. After this step, the dataset are not longer available for future request. Next, the backend application must ensure that Alice's request is completed before removing the stored files from the file storage. Once the request from Alice is complete, the backend application removes the attached files from the file storage.

Case M_AD_B

Case M_AD_B describes the variant where Alice tries to update a dataset while Bob removes it. Since the backend application receives Alice's request slightly earlier than Bob's, it starts the update operation as described in the paragraph 5.3.5.1.3. If the backend application now removes the dataset as described in paragraph 5.3.5.1.4, the two operations collide with each other. Alice's request inserts new data, while Bob's request has already deleted the parent version of the dataset. This result in a defective dataset being left behind. So instead of deleting directly, the backend application must first mark the dataset as inaccessible, for example, by using the corresponding `is_deleted` flag described in paragraph 5.3.5.1.4. This flag ensures that no other operation can access or modify the current dataset anymore. After that, the delete operation is postponed until Alice's update is complete. Once this is the case, the backend application uses the delete operation described in paragraph 5.3.5.1.4.

Case M_AM_B

The case M_AM_B describes the case where both Alice and Bob try to modify the same dataset. From a data integrity perspective, this case is quite simple to solve. The backend application first receives a request from Alice to update a particular dataset, and then receives a similar request from Bob, but with different data. Since the request from Alice arrives first, it will get a new revision first. Then the update sequence shown in figure 5.8 is used to update the dataset. Bob's request will be processed in the same way, but his update gets a later/newer revision identifier. Since both requests append only new data, besides incrementing some conceptual revision counter, they cannot interact with each other. The revision identifier can be incremented using existing sequence mechanisms in the relational database system. As a result, the system has two new revisions that were uploaded independently by different people. This approach can be problematic from an organisational perspective, where revisions represent a linear history like a timeline. Each revision is an improved version of the previous revision. From a technical perspective, such behaviour is acceptable because the data is in a consistent state. From an organisational perspective, a mechanism beyond this to avoid duplicate/colliding updates is desirable. As described by Le [145], locks are an established solution for this problem.

6. Implementation

6.1. General Application Structure

The GeoHub prototype presented in this thesis consists of three separate application parts:

- A server application that provides the central application state
- An HTML single page web application that serves as a user interface
- One or more Executor applications for executing construction steps.

The server application is written in Rust [146]. It uses a PostgreSQL [32] database as a relational database to store metadata. Geoscientific datasets are stored as files in a directory controlled by the application.

The HTML single page web application is developed using HTML [147] and ClojureScript [148]. It is used to provide a user interface such that users can easily interact with the software. The user interface provides graphical tools to create and modify construction hypergraphs, schedule new realisations by uploading new datasets, and review existing realisations by downloading the generated datasets.

The Executor application is written in Rust [146]. It executes user-supplied descriptions of construction steps in user-defined environments. A GeoHub server instance can provide construction steps to several independent Executors, while an Executor can execute construction steps provided by different Geohub instances. Each Executor can be run on a different computer to better utilise computing power. An executor uses Docker [97] internally to deploy custom environments and execute construction steps.

In this chapter, we will discuss the implementation of specific GeoHub components. In section 6.2, we introduce the relational database schema and our extension APIs for extracting metadata and comparing datasets. In section 6.3, we present how users can define construction steps, how the GeoHub server schedules a set of construction steps for execution, and how GeoHub executors run these construction steps.

6.2. Data Storage

6.2.1. Database

Figure 6.1 contains an overview of GeoHub’s database schema for storing data in the relational database. Large parts of the database schema follow database design best practices, so we will not discuss them in detail. Instead, we will focus on the cornerstones of the implementation.

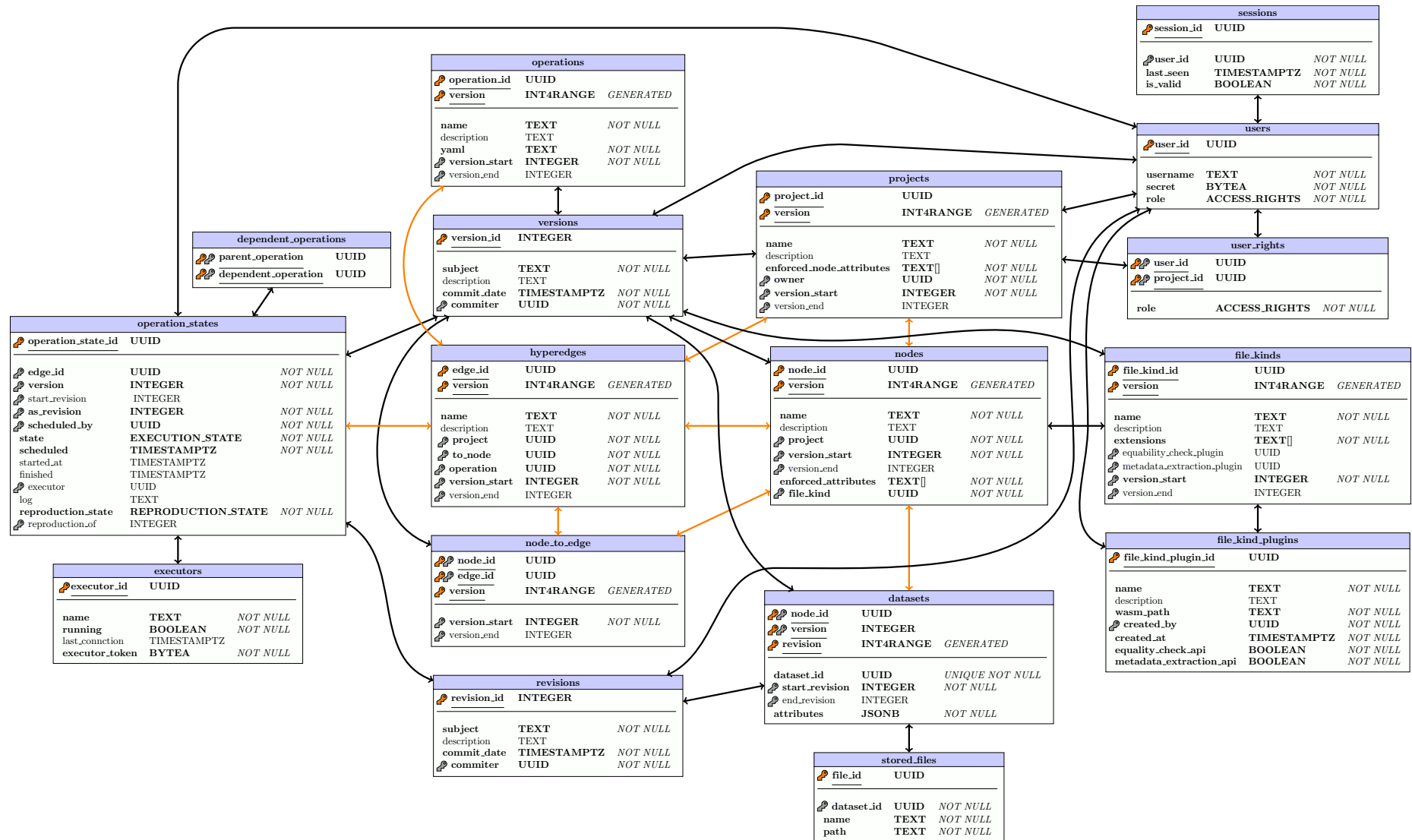


Figure 6.1.: Database schema
 Orange connections represent versioned dependencies between tables
 black connection represent ordinary table dependencies

6.2.1.1. Construction hypergraphs and versioning

GeoHub records a construction graph using five tables:

- **projects** which contains a list of existing construction hypergraphs
- **nodes** which contains a list of nodes for all construction hypergraphs
- **hyperedges** which contains a list of hyperedges for all construction hypergraphs
- **operations** which contains a list of construction steps that can be shared between different construction hypergraphs

GeoHub uses versions and revisions to track all parts that may change over time. In particular, a **version** describes how a construction workflow changes over time. A **revision** records a realisation generated by a particular set of input datasets and a specific construction workflow version. To update each component of the construction hypergraph independently, we need to track a version for each database entity. We decide to implement this using the following scheme: Each entity keeps track of which version it was updated and up to which version it is considered a valid entry. This information is stored in the **version_start** and **version_end** columns of the corresponding tables. Both columns are foreign keys of the corresponding table that refer to the **versions** table. When an entity is “updated,” a new table entry is created. The **version_end** value of the old entity is updated to match the **version_start** entry of the new entry. Deleting an entry simply sets the corresponding **version_end** value. This approach allows us to update individual entries independently, while making it possible to exactly reference a specific version of the construction hypergraph.

We use the **int4range** PostgreSQL type in combination with the generated columns feature to ensure data consistency. The **int4range** type describes an interval of integers. The generated column feature allows to always generate the content of a column based on a predefined expression. For our use case, we generate the **version** column of the corresponding tables based on the **version_start** and **version_end** values. Since the **version** column is always not null, we can use it as primary key in combination with the corresponding entry ID. We enforce that the version intervals for a given entity ID do not overlap by using an additional check constraint. This ensures a unique entry per entity ID and version.

To ensure the consistency of construction graphs, we introduce a customised database relation. We describe a dependency between two versioned entities as a **versioned dependency**. Each dependency must ensure that a matching entry with the corresponding entity ID and an overlapping version range exists. Such dependencies are indicated by orange lines in figure 6.1. This definition constrains the matching entries in the corresponding tables via **ON** clauses in SQL join statements, as shown in listing 6.1. GeoHub uses a **CHECK** constraint to ensure that a matching entity is always present for versioned dependencies.

```
SELECT nodes.*, hyperedges.*
FROM nodes
INNER JOIN hyperedges ON hyperedges.to_node = nodes.node_id
    AND hyperedges.version @> nodes.version
WHERE nodes.project_id = $1;
```

Listing 6.1.: Example query to load all nodes and their generating construction steps belonging to a given construction hypergraph. Bind parameter 1 (\$1) corresponds to the entity ID of the construction hypergraph.

6.2.1.2. Datasets and Metadata Schemas

In the section 5.3, we discussed in detail possible designs for storing metadata for datasets used as part of the construction process. We chose to represent this metadata as key-value types in our schema. The `datasets` table records metadata as key-value pairs for a particular dataset associated with a particular construction hypergraph node with a specific version and data revision. This table contains an `attribute` column of type `JSONB` for the purpose of storing metadata. `JSONB` is PostgreSQL’s improved implementation of a JSON [149] data type that allows the storage of arbitrary key-value pairs.

As mentioned earlier, there may be an institutional requirement to store certain metadata. The database schema shown in figure 6.1 provides support to enforce these requirements. Users can specify a set of required attributes via the `enforced_node_attributes` and the `enforced_attributes` columns of the `projects` and the `nodes` table. Both columns are of the type `TEXT []`, so they can contain zero, one, or more required attributes. Listing 6.2 contains a query for checking whether a `JSONB` key-value store contains all the required keys for a particular node of a specific construction hypergraph version. GeoHub uses a similar query as part of a `CHECK` constraint attached to the `attributes` column of the `datasets` table. This approach ensures that all required metadata is present in the database.

```
SELECT $3 ?& (nodes.enforced_attributes || projects.enforced_node_attributes)
FROM nodes INNER JOIN projects ON nodes.project = projects.project_id
      AND nodes.version @> $2 AND projects.version @> $2
WHERE nodes.node_id = $1
```

Listing 6.2.: A query to check if all enforced metadata keys are set for a given `JSONB` value. Bind parameter 1 (`$1`) corresponds to the node entity ID of the dataset to be checked, bind parameter 2 (`$2`) corresponds to the version of the construction hypergraph, and bind parameter 3 (`$3`) represents the `JSONB` value containing the attributes to be checked. For the `CHECK` constraint, bind parameter 3 is replaced with the `attribute` column of the `datasets` table.

6.2.2. User-provided Data-processing Extensions

GeoHub defines extension points such that users can provide custom implementations for performing equality checks between two datasets or to extract metadata from existing datasets. These extension mechanisms are required because the functions they provide depend on the exact internal structure of the datasets provided. Unfortunately, this structure is not known to GeoHub. In the section 5.2 we have chosen an extension system based on WebAssembly (WASM) [105]. In this section, we will give an overview of the implementation of such an extension system. We will use the equality check plugin extension API as an example. The same system, but with a different API, is also used by the metadata extraction extension API.

Our extension system consists of the following parts:

- A software solution for the execution of WebAssembly based extensions
- An interface definition for transferring data between the host application and each extension

The WASM standard [105] specifies a textual and a binary representation for WASM bytecode. Both are intermediate representations and both require additional software to translate the provided WASM bytecode into actual machine code. This translation can be done using an interpreter-based solution, a just-in-time compiler, or an ahead-of-time compiler to

translate the intermediate representation into machine code. In an interpreter-based solution, each instruction of the bytecode representation is directly translated into machine code at runtime. This approach leads to a simple implementation at the expense of execution speed. On the other hand, a just-in-time compiler adds additional optimisations by analysing the runtime behaviour of the executed code, but this leads to translation overhead since additional work is required. Finally, an ahead-of-time compiler performs an optimised translation in a separate step before the code can be executed. This approach leads to improved runtime performance, as more time can be spent on optimisation at the expense of more time spent on the translation itself. Fortunately, existing software packages provide implementations for all three approaches. For example, a popular solution from the Rust ecosystem is provided by the library `wasmer` [107]. GeoHub uses this library to execute WebAssembly extensions.

GeoHub defines an interface for extensions such that there is a known way to invoke user-provided functionality. The design of such an interface must be based on the use case of the extension. We focus here on the plugin interface for equality checking extension API. See appendix E for details about the metadata extraction extension API. Extensions need access to datasets to check the equality between two datasets. An equality check will either return the result that both datasets are equal or not, so a boolean value. In addition, there is a possibility that an error occurred during the execution of the extension. For example, that one of the datasets does not match an expected format. An extension must be able to pass an appropriate error message to the host application. Listing 6.3 contains a simplified version of an interface definition for this use case.

```

1
2  enum result_tag { Ok, Error };
3
4  struct check_equality_result {
5      result_tag kind;
6      union {
7          bool equality_check;
8          char* error_message;
9
10     } result;
11 };
12
13 struct file {
14     char* file_name;
15     unsigned char* data;
16 };
17
18 struct dataset {
19     int file_count;
20     file* files;
21 };
22
23 check_equality_result check_equality(
24     dataset dataset1,
25     dataset dataset2
26 );

```

Listing 6.3.: Equality check extension interface

The WebAssembly standard contains several restrictions regarding to the ABI of functions called from outside the WebAssembly environment. In particular, only the following types may be passed through this interface:

- Integers (either 32 or 64 bit wide)
- Floating-point numbers (either 32 or 64 bit wide)

These requirements further limits the ability to pass information to extensions. We worked around this limitation in GeoHub by writing data directly to the private memory area used by the WebAssembly extension and passing references to that data as pointers via a 32-bit integer through the ABI. This approach works because the WebAssembly standard defines the size of the pointer type as a 32-bit integer. Also, the memory area used by the WebAssembly extension is controlled exclusively by our WebAssembly runtime. This allows us to access and manipulate this memory area while the extension is not yet running.

6.3. Operation Executor

GeoHub executes user-defined construction steps in a user-defined environment. Section 5.2 describes general strategies for implementing such a feature. There, we decided to use a Docker-based approach.

To implement the execution of construction step in GeoHub, the following three components are used:

- A generic construction step description that allows the user to describe a construction step with all necessary details.
- Functionality that decides which specific construction steps are executed next and which construction step can be executed. This functionality is provided by the GeoHub back-end application.
- Functionality to execute a construction step. This functionality is provided by the GeoHub Executor tool

The following subsections provide details on the implementation of each component.

6.3.1. Construction Step Descriptions

In order to execute arbitrary construction steps, GeoHub needs a lot of information:

- Which datasets are used as input for the construction step
- What environment should be used for the execution of the construction step
- Where to place the input datasets in this environment
- What exactly should be done as part of this particular construction step
- What is the expected outcome of the current construction step

All this information is provided by the user creating the construction step. Information about which datasets are used as input for a particular construction step is embedded in the structure of the construction hypergraph. A construction step is represented there by a hyperedge, which points from the input datasets to the single output dataset. All other information listed above must be specified as part of the construction step definition.

We decided to design a way to describe this information with a single input structure. Various formats for describing continuous integration workflows description formats inspire the

format used. The exact format is based on YAML [150] . Listing 6.4 contains an example construction step definition. See appendix A, C, D for more examples. Appendix E.4.1 contains a specification of the exact format.

```

1 image: git.informatik.tu-freiberg.de:5050/semmlerg/bhmz_dc:latest
2 input_path: "/home/matlab/"
3 operation:
4   - command: |
5       mkdir bhmz/data_BERT
6       mv RZ01.dat bhmz/data_BERT/RZ01.dat
7       mv RZ02.dat bhmz/data_BERT/RZ02.dat
8       cd bhmz && matlab -nosplash -nodesktop -r "drive_stack_data"
9   displayName: Stack data
10 output:
11   - file: /home/matlab/bhmz/RZstack.dat

```

Listing 6.4.: Example construction step description

Each environment used to execute a construction step must provide all the software needed for that particular construction step. We chose to provide these environments using Docker images. These images allow users to bundle their software using preexisting tooling. On the other hand, this limits the environments used as construction steps to Linux. With a Docker image, it is possible to regenerate the same environment over and over again by launching a new image instance each time the environment is needed. Users can specify the Docker image that provides their environment in the definition of the construction step. Listing 6.4 includes this specification on line 1 using the `image:` key. Future GeoHub releases may extend this definition to allow different types of environments here, for example, to allow the use of other operating systems.

The construction step executor and the user-defined environment must agree on where the input datasets of a construction step is located. This problem can be solved either by convention, such that the executor always places the input records in the same location. Or it can be solved by specification, where a path is specified as part of the definition of the construction step. If the path is not specified, the definition of the construction step definition is simpler, but it may result in data being stored in locations that are not accessible to the construction step itself. We have allowed both variants by making the corresponding construction step definition entry optional. The `input_path:` key on line 2 allows the user to specify a path within the construction step run time environment where the input datasets should be placed. If this key is not specified, the executor places the data in the `/tmp` directory.

Given the environment and the required input datasets, the construction step executor needs to know what actions to perform. Inspired by continuous integration setups, we decided to represent this information in the form of a user-supplied shell script. Listing 6.4 contains an example script at lines 3 - 9. The `operation:` key specifies a list of command blocks. Each of the command blocks contain a shell script block. Each block is executed in a single pass. This layout allows the construction step to be internally structured and provides additional context for failed executions. Future GeoHub releases may extend this functionality to allow script processors other than shell scripts. This functionality would allow to use various scripting languages, such as Python or R, directly instead of manually calling the appropriate interpreter as part of the construction step definition.

Last but not least, the executor needs information about what to expect as a result after the execution of the construction step within the provided environment. This information

is explicitly represented in the definition of the construction step by listing files that are considered to be the output dataset of the construction step. The example shown in listing 6.4 contains this definition in lines 10 - 11. The **output:** key allows the user to specify a list of files with an absolute path. The executor will attempt to download each file after the execution of the construction step is completed.

6.3.2. Construction Step Scheduling

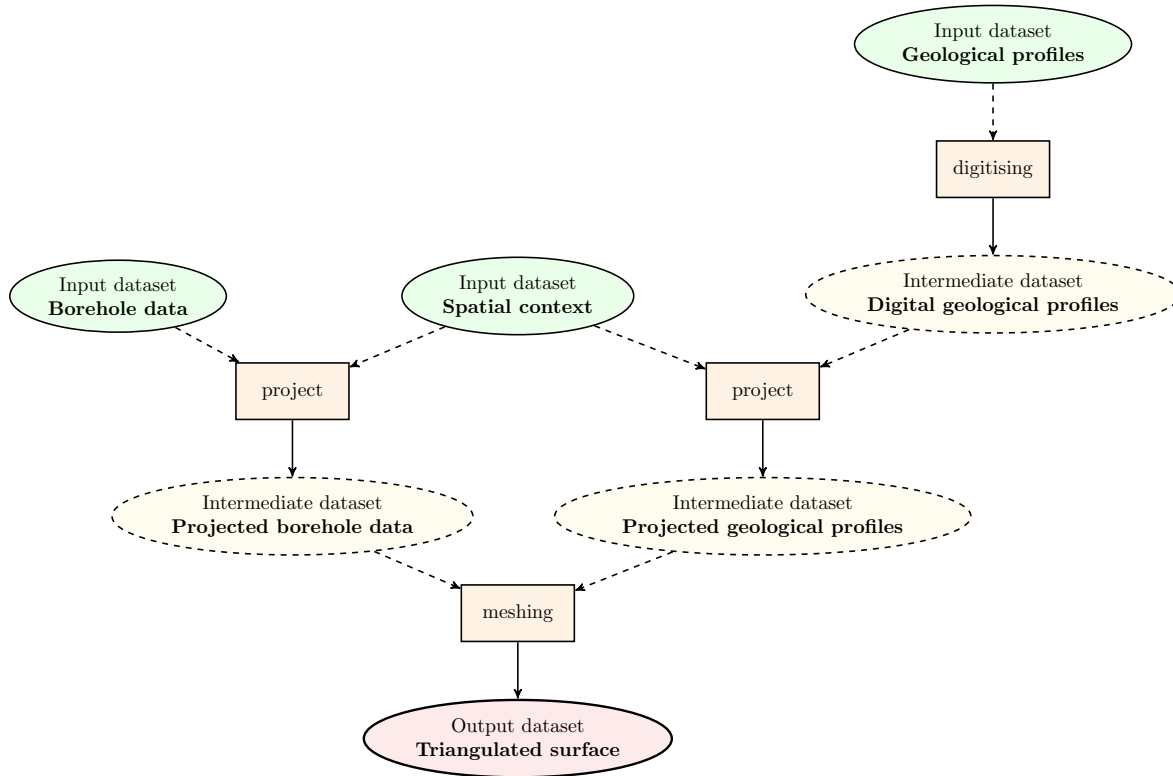


Figure 6.2.: Example construction hypergraph

Before a certain construction step is executed, GeoHub has to decide which construction steps must be executed in which order. This step is called construction step scheduling and is implemented as part of the GeoHub server. The result of the scheduling is an ordered list of construction steps that contains dependencies between the individual construction steps. This list is stored in the **operation_states** table in the relational database system, which acts as a persistent queue. The various executors use this information to determine which construction step needs to be executed next by querying this table for scheduled construction steps without dependencies or for those where the dependencies have already been executed.

We will look at the following two scenarios to better understand how scheduling construction steps is implemented in GeoHub. Our first scenario starts with a given construction hypergraph. A user provides a complete set of the input dataset such that each input node of the construction hypergraph has a matching dataset. We call this scenario **initial execution**. The second scenario assumes that there is already an existing realisation of a particular construction hypergraph. Therefore, only a subset of the input datasets is provided to generate a changed or updated realisation of the same geoscientific model. We call this scenario **updated execution**. Figure 6.2 contains a simple example construction hypergraph that we will use to illustrate the general process for both scenarios. The goal for both scenarios is to obtain a list of construction steps that need to be executed in the specified order.

Algorithm 1: Construction step scheduling for the **initial execution** scenario

Input : hypergraph
Output: scheduled construction steps

```

1 visitedNodes = []
2 scheduleList = []
3
4 forall inputDataset in hypergraph do
5   | scheduleSubGraph(hypergraph, inputDataset, NULL, visitedNodes, scheduleList)
6 end
7
8 Fn scheduleSubGraph(hypergraph, node, lastEdge, out visitedNodes, out
   scheduleList)
9   forall hyperedge in hypergraph.outgoingHyperedges(Node) do
10    | if hyperedge ∈ scheduleList then
11      |   if lastEdge != null then
12        |     | scheduleList[hyperedge].addDependency(lastEdge)
13      |   else
14        |     | scheduleList.addEntry(hyperedge, lastEdge)
15      |
16      | visitedNodes.markVisited(node)
17      | nextNode = hyperedge.output
18      |
19      | if nextNode ∉ visitedNodes then
20        |   | scheduleSubGraph(
21          |     | hypergraph,
22          |     | nextNode,
23          |     | hyperedge,
24          |     | visitedNodes,
25          |     | scheduleList
26          |     | )
27      |   |
28    | end
29  end
30  return

```

For the **initial execution** scenario, we assume that a complete set of input datasets is provided. In the case of our example construction graph, this means that the *Geological profiles*, *Spatial context*, and the *Borehole data* datasets are provided by a user. To compute the resulting list of construction step that needs to be executed, we iterate over all input datasets in the construction hypergraph. For each dataset, we obtain the construction steps corresponding to outgoing hyperedge from the construction hypergraph. These construction steps use the dataset as input dataset. Then, for each construction step, we check whether or not we have already included this step in the list of scheduled construction steps. Otherwise, we include the construction step and continue with the dataset that represents the output dataset of the construction step. Here we record a dependency on the previous construction step and find all construction steps that use this dataset as input. For these construction steps, we repeat the same procedure as for the previous ones, with the small difference that we add the recorded dependency to the list of scheduled construction steps. We repeat this until we get an output dataset of the construction hypergraph. Algorithm 1 contains a more formal representation of this procedure.

Applying this algorithm to our example construction hypergraph, starts the iteration at the *Borehole data* input dataset. From there, we visit the *project* construction step. This step is included as the first entry to our list of scheduled construction steps. This construction step generates the *Projected borehole data* dataset, which in turn is used by the *meshing* construction step. Since the *meshing* construction step is not yet included in the list of scheduled construction step, we add the *meshing* operation as the second entry. We also add the information that this construction step can only be performed after the *project* step has been completed. Since the *meshing* operation generates the output dataset of the construction hypergraph, we continue with the second input dataset, *Spatial context*. This dataset is the input dataset of two construction steps, *project* and *project*. The first construction step is already included in the list of scheduled construction step, while the second is still missing. Therefore, we add the second *project* construction step to our list of scheduled construction steps. This construction step generates the *Projected geological profiles* dataset, that will be used as input for the *meshing* construction step. The *meshing* construction step is already part of our list of scheduled construction steps, but this entry does not include the information that the *meshing* step cannot be performed until after the second *project* step. Therefore, we add this dependency to the existing *meshing* entry in the list of scheduled construction steps. Last but not least, we look at the *Geological profiles* dataset. This dataset is used as input by the *digitise* construction step. Since this construction step is not yet included in the list of scheduled construction steps, we add it to our list. The *digitise* construction step generates the *Digital geological profiles* dataset which is used as input dataset by the second *project* step. This construction step is already included in the list of scheduled construction steps but the dependency on the *digitise* step is missing. Therefore, we add this dependency information to the existing entry. At this point we reached the end of our algorithm as we processed all input datasets. The list of scheduled construction steps now contains the following entries: *project*, *meshing* (depending on *project* and *project*, *project* (depending on *digitize*), *digitize*.

In the **updated execution** scenario, things are different. In this scenario, only a part of the input datasets are provided. We could now load any missing dataset from a previous execution and use the same scheduling strategy as for the **initial execution** scenario. However, this strategy would require the execution of more construction steps than strictly necessary, missing a good opportunity for optimisations. Instead, we used a slightly modified version of the scheduling algorithm for the initial execution case described earlier. This version of the scheduling algorithm does not iterate over all the input datasets of the construction hypergraph, but instead relies on a list of modified input datasets. The other part of the algorithm remains the same. Algorithm 2 contains a formal description of this modified variant. It can be observed that the algorithm is the same as for the **initial execution** scenario when the list of `modifiedNodes` contains all input datasets of the construction hypergraph

We now apply our modified algorithm to our example construction hypergraph and assuming that the *Borehole data* dataset has been modified. We start with the modified dataset *Borehole data*. From there, we follow the dependency of the input dataset to the construction step *project*. This step will be the first entry in our list of scheduled construction steps. The *project* construction step generates the *Projected borehole data* dataset, which is then used as an input dataset by the *meshing* construction step. Therefore, we include the *meshing* step as the second entry in the list of scheduled construction steps. We also record that the execution of the *meshing* construction step depends on the earlier *project* step. Finally, the *meshing* construction step generates an output dataset of the construction hypergraph. We have finished the scheduling process because we have reached an output dataset and the list of modified input datasets is now empty. The schedule list now contains two entries: *project* and *meshing* (depending on *project*). Under the strategy described in the **initial**

Algorithm 2: Construction step scheduling for the **updated execution** scenario

Input : hypergraph
Input : modifiedNodes
Output: scheduled construction steps

```

1 visitedNodes = []
2 scheduleList = []
3
4 forall inputDataset in modifiedNodes do
5   | scheduleSubGraph(hypergraph, inputDataset, NULL, visitedNodes, scheduleList)
6 end
7
8 Fn scheduleSubGraph(hypergraph, node, lastEdge, out visitedNodes, out
   scheduleList)
9   forall hyperedge in hypergraph.outgoingHyperedges(Node) do
10    | if hyperedge ∈ scheduleList then
11    |   | if lastEdge != null then
12    |   |   | scheduleList[hyperedge].addDependency(lastEdge)
13    |   | else
14    |   |   | scheduleList.addEntry(hyperedge, lastEdge)
15    |   |
16    |   | visitedNodes.markVisited(node)
17    |   | nextNode = hyperedge.output
18    |   |
19    |   | if nextNode ∉ visitedNodes then
20    |   |   | scheduleSubGraph(
21    |   |   |   hypergraph,
22    |   |   |   nextNode,
23    |   |   |   hyperedge,
24    |   |   |   visitedNodes,
25    |   |   |   scheduleList
26    |   |   | )
27    |   |
28    |   | end
29    |   | return

```

execution scenario, the resulting schedule list would contain four entries. Thus, we reduced the computational effort required to compute an updated realisation of the described geoscientific model has been reduced by two construction step in this example case. For the general case, the reduction depends on the exact structure of the construction hypergraph and the set of provided input dataset and will therefore vary.

6.3.3. Construction Step Execution

The final component to automatically execute geoscientific model construction processes via construction hypergraph executions is the construction step execution functionality. This functionality is implemented outside of the GeoHub Server application to distribute the computational workload across multiple machines. The GeoHub Executor application provides this functionality. The implementation was inspired by Gitlabs CI Runner [151].

A GeoHub Server instance can offer construction steps for execution to multiple GeoHub

Executors. A GeoHub Executor can contact more than one GeoHub Server to obtain construction steps for execution. In this way, available computing resources can be used efficiently. GeoHub Executor uses an HTTP API to communicate with each configured GeoHub Server instance. A GeoHub Executor always initiates communication. This approach has the major advantage that no additional network configuration is required for computers running a GeoHub Executor instance. In particular, it is not necessary for a computer running GeoHub Executor to have a publicly accessible or even stable IP address. It is required that a GeoHub Executor installation can connect to the configured GeoHub Server instance via HTTP requests.

Each executor must be explicitly registered with a Geohub Server instance. As part of this process, the server and the executor generate a shared secret that is then used in any future communication to ensure that the executor is allowed to access certain information. Appendix E.5 contains detailed information about registering an executor with a server instance.

Each GeoHub Executor periodically queries GeoHub Server instance for construction steps that need to be executed. The GeoHub Server instance uses these requests to update the `last_seen` field of the `executors` table and performs a search in the `operation_states` table to find a construction step to execute. The response from the server instance contains either the information that there is no construction step waiting for execution. In this case, the executor pauses and tries again to get a construction step at a later time. When a construction step needs to be executed, all relevant information about the construction step is sent to the executor. This information includes the definition of the construction step, a list of references to the input datasets, and an identifier that allows the server to identify which construction step execution is executed by the executor later on. The executor then loads each required dataset from the server instance. In a next step, the executor starts a new Docker container instance based on the `image:` information provided by the construction step definition. This assumes that the executor has access to the local Docker installation and to the corresponding Docker registry that provides the image in question. After starting the Docker container, the executor uploads each input dataset into the container instance in the configured `input_path`. The next step is to execute each `command` block as a shell script within the Docker container. All command-line output generated by these commands is sent to the GeoHub Server instance as an execution log. After the execution of the commands is completed, the executor downloads all the files specified in the `output:` section of the construction step definition from the Docker container. Then, the container is stopped and removed from the executor's system. Finally, the executor uploads the output files to the server to finish the execution. Any error is reported to the server, and the execution of the respective construction step will be marked as failed. Algorithm 3 contains a simplified version of the executor's main loop.

Algorithm 3: GeoHub executor main loop

```

Input  : serversToContact
Input  : pollPeriod
Input  : dockerHandle

1 repeat
2   foundOp = false
3   forall server in serversToContact do
4     nextOp = server.fetchNextOp()
5     if nextOp then
6       foundOp = true
7
8       files = []
9       forall fileRef in nextOp.files do
10        | files.push(fileRef.fetch())
11      end
12
13      image = dockerHandle.fetchImage(nextOp.image)
14      container = dockerHandle.startContainer(image)
15
16      forall file in files do
17        | container.uploadIntoPath(nextOp.inputPath, file)
18      end
19
20      forall command in nextOp.commands do
21        | container.executeCommand("sh -c " + command + " ")
22      end
23
24      results = []
25      forall output in nextOp.output do
26        | results.push(container.fetchFile(output))
27      end
28
29      container.stopAndRemove()
30      server.uploadResult(results)
31    end
32    if not foundOp then
33      | sleep pollPeriod
34 until forever

```

7. Case Studies

7.1. Overview

Within this chapter, we will present three case studies to demonstrate the strengths and weaknesses of the implemented prototype. Each case study is based on a model construction process from a different area of geoscientific research. We present the following case studies:

- Analysis and interpretation of a geophysical measurement campaign
- Construction of a three-dimensional subsurface model with gOcad, including several pre-processing steps
- Development of a hydrologic balance model for the catchment area of a small stream.

Within the scope of this chapter, we aim to show that the provided prototype can solve the outlined problem while working in different application areas with diverse model construction methods. Therefore, we focus on presenting critical parts of the construction process implementation with GeoHub rather than discussing the scientific basis of each construction process.

For each case study, we first present the provided datasets and instructions. Then, the implementation of the construction process in GeoHub is described. In the next section, the reproducibility of the construction process is discussed. In the following section, possible improvements to the implemented construction process are described. In the last section, we try to give some general recommendations for the use of GeoHub in the described application area.

7.2. Geophysical Model of the BHMZ block

7.2.1. Provided Data and Initial Situation

The first case study uses data from two geophysical measurement campaigns to construct a unified interpretation of the BHMZ (Biohydrometallurgischen Zentrum) block in the mine “Reiche Zeche” in Freiberg. The case study is based on the tomographic seismic measurements performed by Sebastian Winter [152] and the tomographic geoelectric measurements performed by Daniel Pötschke [153], as part of their master theses. The necessary construction steps for processing and interpreting the individual measurements were provided by the Seismic and Electromagnetic Working Group of the Department of Geophysics at TU Bergakademie Freiberg, as well as instructions for using the tools and performing the model construction. The objective of this study was to investigate the internal structure of a rock formation in the mine “Reiche Zeche” in Freiberg and to gain a better understanding of its internal composition. The rock formation is located underground and has an extension of about 35×13 meters. A tunnel surrounds the rock formation in approximately one plane. A combined visualisation of the measurement interpretations in a unified reference frame will be generated as a final model.

A total of four different input datasets was provided. Two files contain the measurement results for the geoelectric measurement, one geometry file representing a laser scan of the tunnel geometry, and one file containing the pre-processed measurement results of the seismic measurements. In addition, the Geoelectric Working Group provided several Matlab [154] scripts for processing and interpreting the geoelectric measurement results [155]. These scripts generate a three-dimensional resistivity distribution by solving an inverse problem. The Seismic Working Group also provided code for processing the seismic measurements that generates a two-dimensional velocity distribution. A Python script converts the results and combines both distributions into a unified model. This script was developed as part of the implementation of this case study.

Both working groups provided internally developed code to process the data. The geoelectric working group made the code available through an existing Git repository. This repository contained a working continuous integration setup based on Docker.

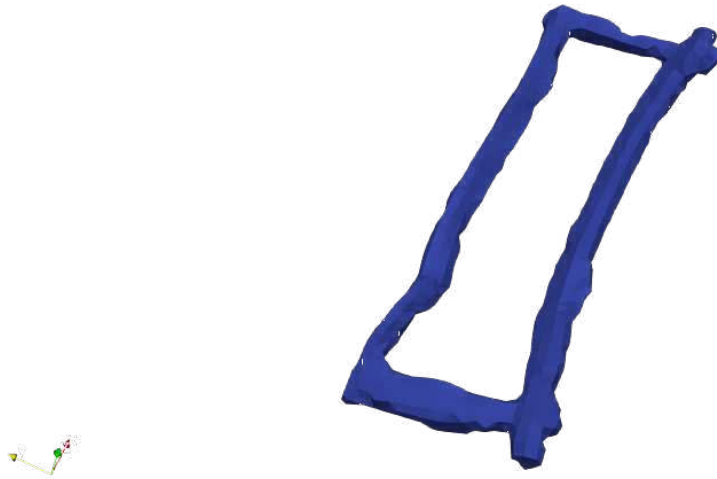


Figure 7.1.: A geometric representation of the tunnel surrounding the BHMZ block

7.2.2. Construction Process Description

Figure 7.2 shows a graphical representation of the construction hypergraph used for the implementation of this case study. The construction hypergraph consists of nine nodes representing datasets, four of which are input datasets, and one is the output dataset representing the combined visualisation. Five construction steps are required to perform the necessary construction work. Appendix A contains a detailed description of each construction step.

The construction process requires four different input datasets:

1. A geometric representation of the tunnel surrounding the BHMZ block as an STL geometry file [156]. See figure 7.1 for an image of the corresponding geometry.

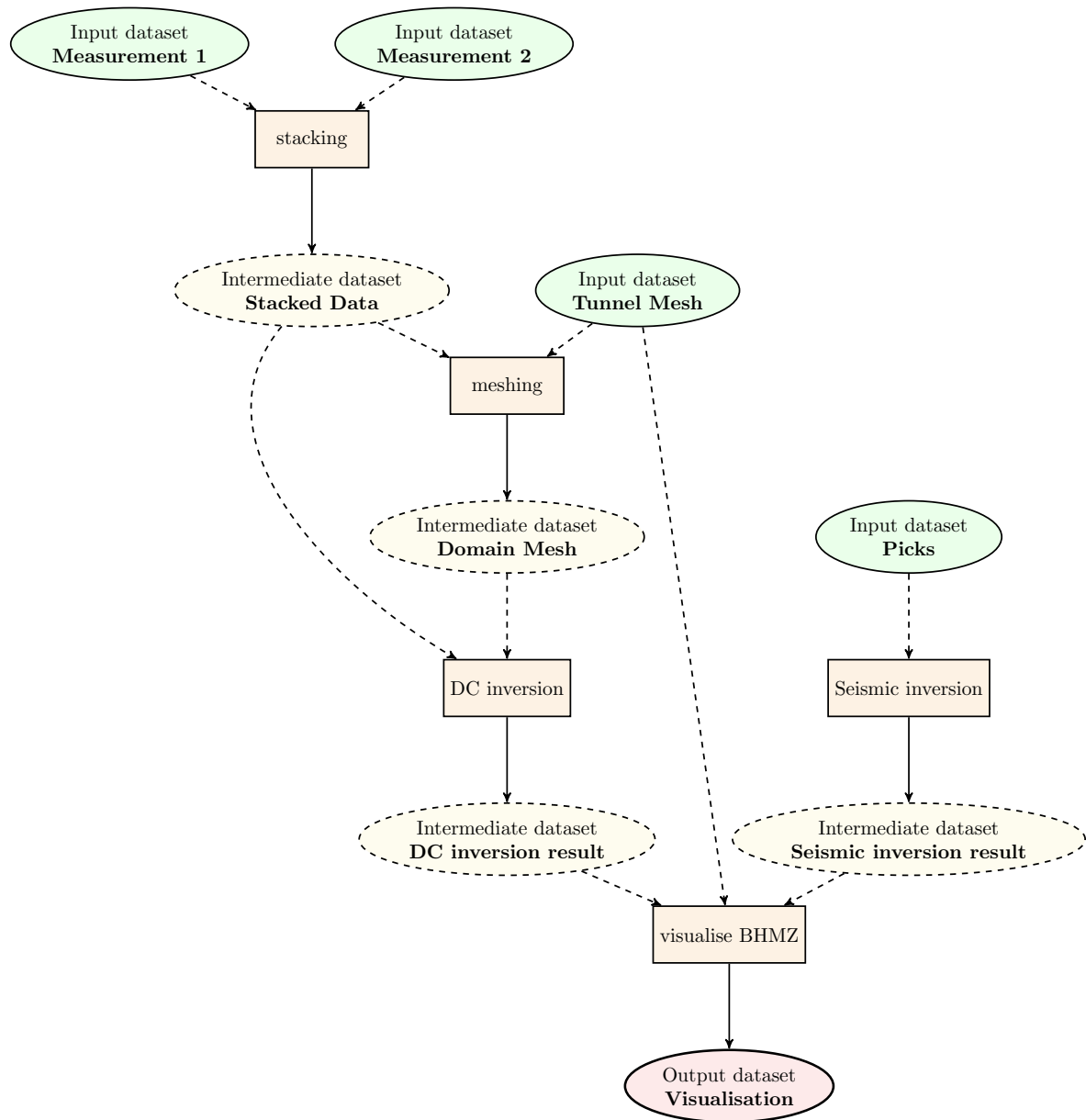


Figure 7.2.: Construction hypergraph for the BHMZ block model

2. A text file containing the first arrival times for the seismic measurement in a space-separated format. See listing 7.1 for an excerpt from this file. The file contains the following columns:
 - Source ID: Numerical identifier of the seismic source
 - Receiver ID: Numerical identifier of the seismic receiver
 - Travel time in nanoseconds, where -1000 represents a no data value
 - X coordinate of the source location
 - Y coordinate of the source location
 - X coordinate of the receiver location
 - Y coordinate of the receiver location
3. Two .DAT files with the measurement results of the geoelectric measurements as Matlab matrices as provided by the geoelectric working group.

The construction process generates a single output dataset. This result represents the unified visualisation of all measurement results. The visualisation is provided in the form of a Paraview Python script that sets up the Paraview view to display all results in the correct location. Figure 7.3 contains an image of this visualisation.

```
101 5 -1000 3.082000 2.087000 7.462000 2.768000
101 6 -1000 3.082000 2.087000 8.570000 3.025000
101 7 553 3.082000 2.087000 9.889000 3.295000
101 8 1029 3.082000 2.087000 11.118000 3.431000
101 9 1183 3.082000 2.087000 12.439000 3.658000
101 10 1152 3.082000 2.087000 13.540000 3.577000
```

Listing 7.1.: Excerpt from the text file with the first arrival times of the seismic measurement

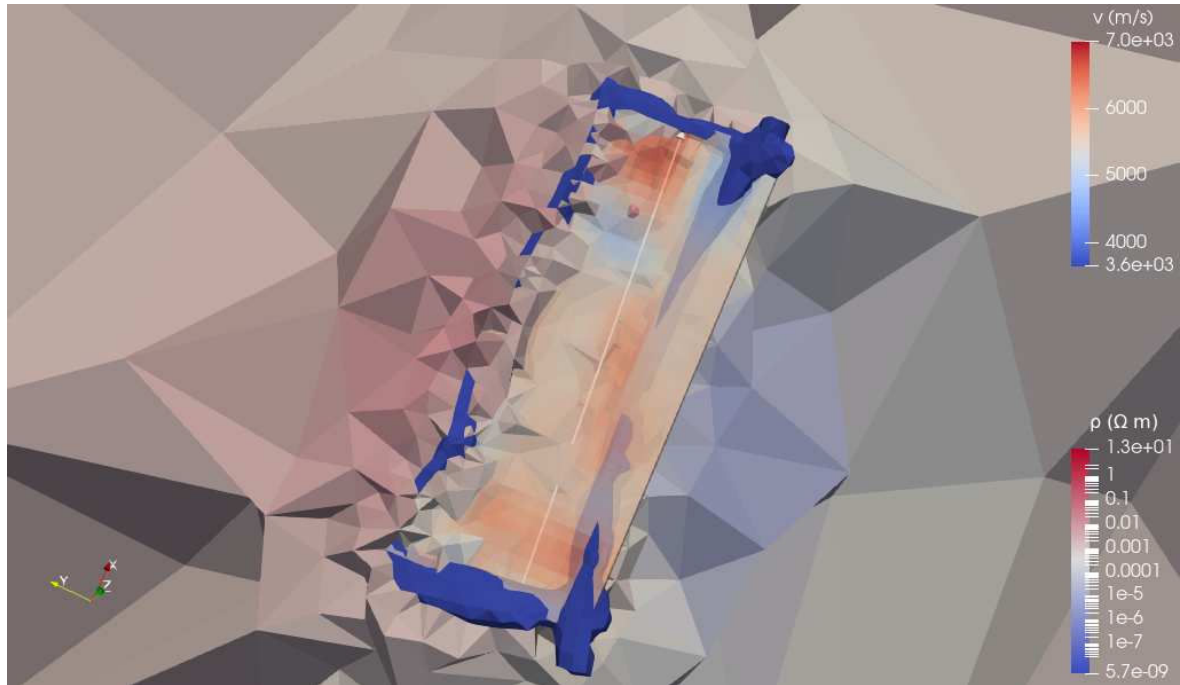
7.2.3. Reproducibility

All five construction steps of the presented construction hypergraph are reproducible using the bitwise equality definition. This result means that independent realisations of the construction process with the same input datasets generates identical results at the byte level.

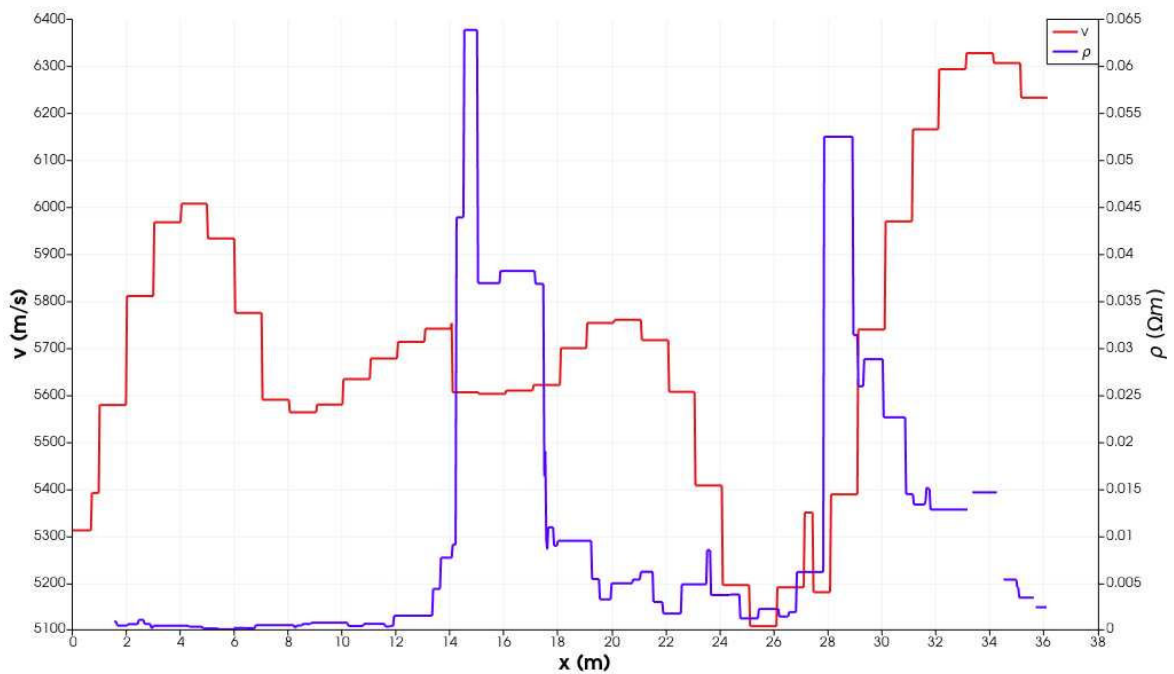
7.2.4. Identified Problems and Construction Process Improvements

By implementing this construction process in GeoHub, we noticed several issues and opportunities to improve the construction process.

When implementing the unified visualisation of all measurement results, we found that the geoelectric measurements misses information about their coordinate reference system with respect to the original laser scan of the tunnel. While the geoelectric working group provided a dataset containing transformed version of the tunnel geometry, it was difficult to unify both the seismic and the geoelectric inversion results into one unified visualisation. By calculating the required transformation between the seismic and geoelectric measurements points, we were able to reconstruct an approximate version of the required coordinate system transformation. However, we were unable to reconstruct the exact transformation. The measurement points used by the geoelectric model were slightly shifted by an unknown, non-constant offset. In the corresponding master thesis, this shift is justified to solve numerical instabilities due to singularities in the inversion model. Unfortunately, the exact values of



(a) Three-dimensional representation of the result including the tetrahedral DC-inversion mesh, the regular seismic inversion mesh, the tunnel (blue geometry) and the plot line (white line)



(b) Plotted seismic velocity and electric resistivity values along a line through the model

Figure 7.3.: A visualisation of the interpretations of both geophysical measurement

the shifts are not documented. Since both datasets are primarily used for methodological research, this issue is probably not relevant for the daily work of both working groups. The implementing of the construction process in GeoHub helped us to identify this inconsistency. This result shows that implementing construction processes in GeoHub can prevent such data inconsistencies from occurring in the first place.

The current construction process implementation is tailored to this specific dataset. This is partly due to the included scripts, which expect only certain datasets as input, and partly due to the general structure of the construction process, which hard-codes some assumptions, such as the reverse-engineered version of the coordinate system transformation. For a practical application of the implemented construction process, it may be helpful to generalise all parts of the construction process in such a way that it can be easily reused in similar measurement setups. Such generalisation requires improving the code used as part of the construction steps in order not to assume certain properties of the measurement by replacing hard coded values. An example of this is the hard coded transformation in the `visualise BHMZ` construction step Appendix A.5.

7.2.5. Recommendations

Scientists at the Department of Geophysics at TU Bergakademie Freiberg are researching methods for the efficient interpretation of measurement results. In doing so, they focus on the processing of the data and may ignore this documentation of the provenance of the input datasets. While this is fine for method-based research, it can limit further use of the data. A system like GeoHub could help by enforcing standards for what information needs to be minimally collected. Since both working groups are developing their own software, it should not be difficult for them to provide the software as a construction step. One working group has already organised its code in a Git repository and is testing changes via a Docker-based continuous integration setup. Generating the appropriate environment can then be easily integrated into their overall software development workflow as another build step as part of the continuous integration setup. Such a setup can easily be used to automatically build a Docker image based on a specific software release. This image can then be used as the basis for definition of construction steps. Since the software used in possible construction steps is developed internally, it is easy to provide any necessary interface to allow the software to run as part of a construction step. We assume that this potential user group will be able to compose their own construction processes, including custom operations and environments. Such a defined construction process can then be used as documentation for existing approaches to processing certain types of measurements.

7.3. Three-Dimensional Subsurface Model of the Kolhberg Region

7.3.1. Provided Data and Initial Situation

The second case study presented implements a classical explicit three-dimensional subsurface model construction process, including data pre-processing. The presented model [157] was constructed in the context of the planning of a bypass road south of Pirna/Saxony. The Geological Survey of Saxony provided all datasets used to construct this model, as well as detailed instructions on how to combine the datasets into a final three-dimensional subsurface model. See Appendix B for the original instructions. The model has a size of about 3km × 1.3 km. Figure 7.4 shows an overview map of the study area.

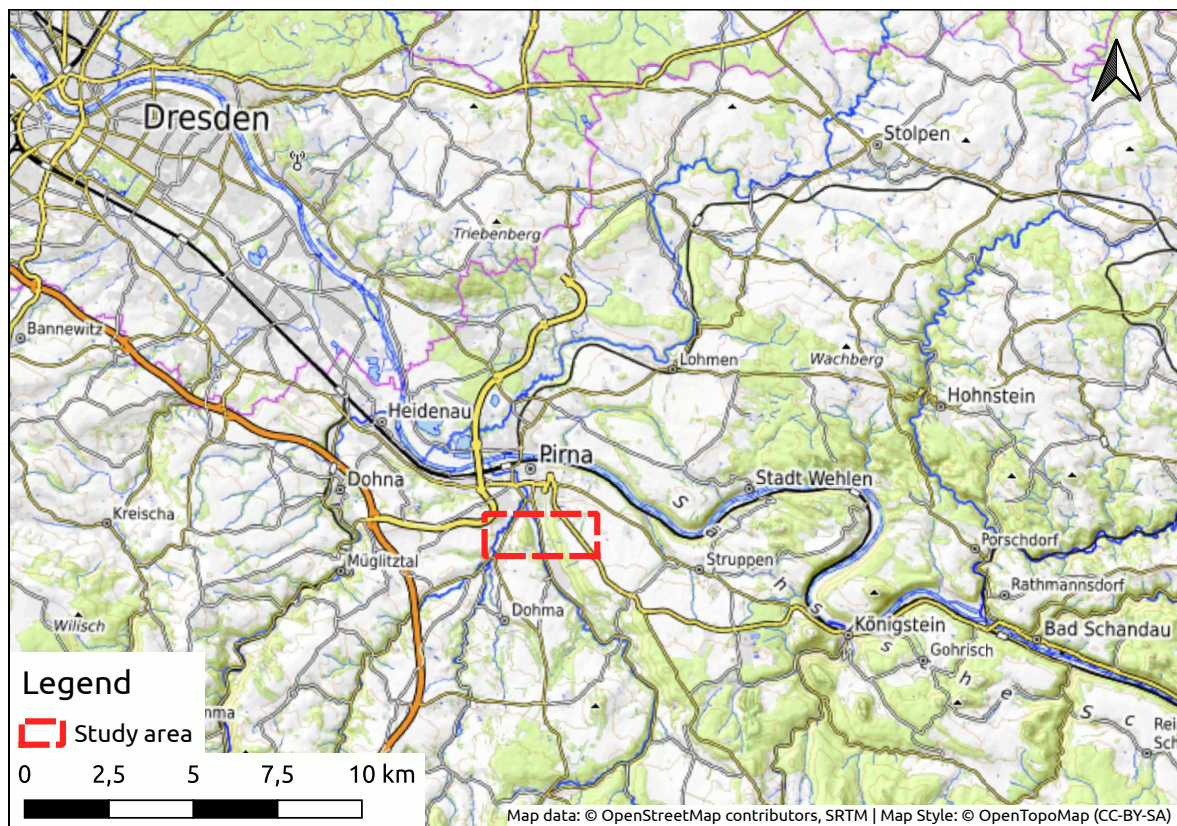


Figure 7.4.: Overview map of the study area

7.3.2. Construction Process Description

Figure 7.5 shows a graphical representation of the construction hypergraph used to implement this case study. The model construction process consists of 14 datasets and seven construction steps. Of these 14 datasets, seven are provided as input datasets, three are generated as output datasets, and the other four are intermediate datasets. The following seven datasets are provided as input datasets:

1. A Microsoft Excel file with information about Boreholes in the area of interest.
2. Digital elevation model as XYZ file.
3. An empty gOcad project that is used to document general gOcad project settings.
4. An ESRI shapefile containing a cross-section path
5. An ESRI shapefile containing the potential bypass route
6. An ESRI shapefile containing the boundaries of the study area
7. An ESRI geodatabase archive containing a geological map of the study area.

The construction process generates three output datasets:

1. Two CSV files containing the borehole data projected to the UTM33 coordinate reference system
2. A set of gOcad ASCII files representing the extruded “Streckenachse”
3. A set of gOcad ASCII files containing the outlines of the geological map, which are projected onto the surface of the terrain

The construction process includes several pre-processing steps. These steps prepare the provided data for later use. Subsequently, the datasets are combined into a final subsurface

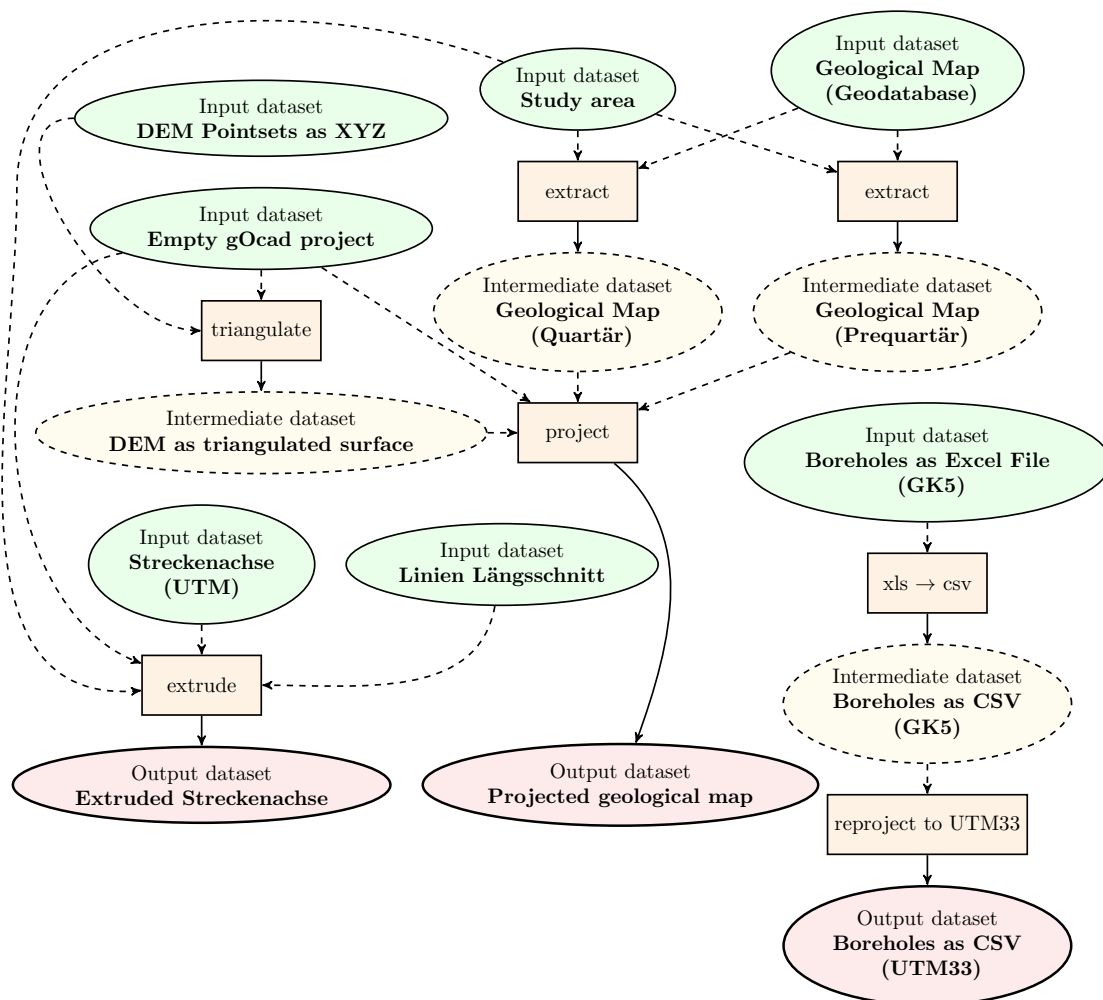


Figure 7.5.: Construction hypergraph for the Kohlberg model

model using gOcad [17]. We decided to deviate from the instructions for all pre-processing steps here, due to the fact that these steps are much easier to implement using Q-GIS [158] instead of ArcGIS [18]. Q-GIS supports scripting and running in a Docker container. Construction steps using gOcad were implemented using gOcad’s macro support. This feature allows the scripts used as part of the construction step to be easily recorded within gOcad. Information about gOcad macros corresponding to any action in an existing gOcad project is available as part of the gOcad project files. This information can be extracted later, either manually or in an automated way. Our GeoHub prototype, provides a helper tool to extract all construction steps performed within a user-specified session and combines those construction steps as a construction hypergraph. This construction hypergraph description can then be imported into GeoHub. Combined with the original input datasets, this construction hypergraph can be used to easily repeat existing gOcad projects and verify whether they are reproducible. See appendix E for details on how to use the corresponding tool.

We refrain from a detailed discussion of individual construction steps. See appendix C for details on specific construction steps.

7.3.3. Reproducibility

Of the seven construction steps involved in the construction process of the Kohlberg model, three construction steps are reproducible using the bitwise equality definition. The **triangulate**, **extrude** and **extract** construction steps are not reproducible according to the bitwise equality definition.

The **triangulate** construction step generates a triangulated surface based on a set of point. Appendix C.5 presents the construction step in detail. This construction step performs the following three gOcad operations in the given order:

1. Merge the provided separate pointsets of the digital elevation model into one combined pointset
2. Perform a Delaunay triangulation based on the merged pointsets
3. Simplify the triangulated surface with gOcad’s decimate operation

Running the **triangulation** construction step several times and comparing the results, we found that each run generates a triangulated surface with a different number of vertices and triangles. One example run produced 38,902 vertices and 77,222 triangles, while another run produced only 38,822 vertices and 77,062 triangles. Figure 7.6 shows the triangulated surface generated by the **triangulation** construction step. The figure shows the calculated distance to the nearest point in the other realisation. The distance appears to be close to zero for large portions of the triangulated surface, but there are a few locations where a larger distance is observed. Visual inspection of these areas revealed relatively large triangles at these locations. Figure 7.7 shows the triangulation of one of the locations with large distance values for both realisations. The triangulation displayed in these images are different.

Further investigation has shown that the decimate operation introduces these differences. We used a customised version of the **triangulate** construction step that skips the final decimate operation to check whether it continues to produce different files. Upon inspection, it appeared that these files contained descriptions of exactly the same geometries, unlike before. Only the order of triangles and vertices in the file changed. Consequently, triangulated surfaces generated with the modified **triangulate** construction step could be considered equal according to the structural equality definition.

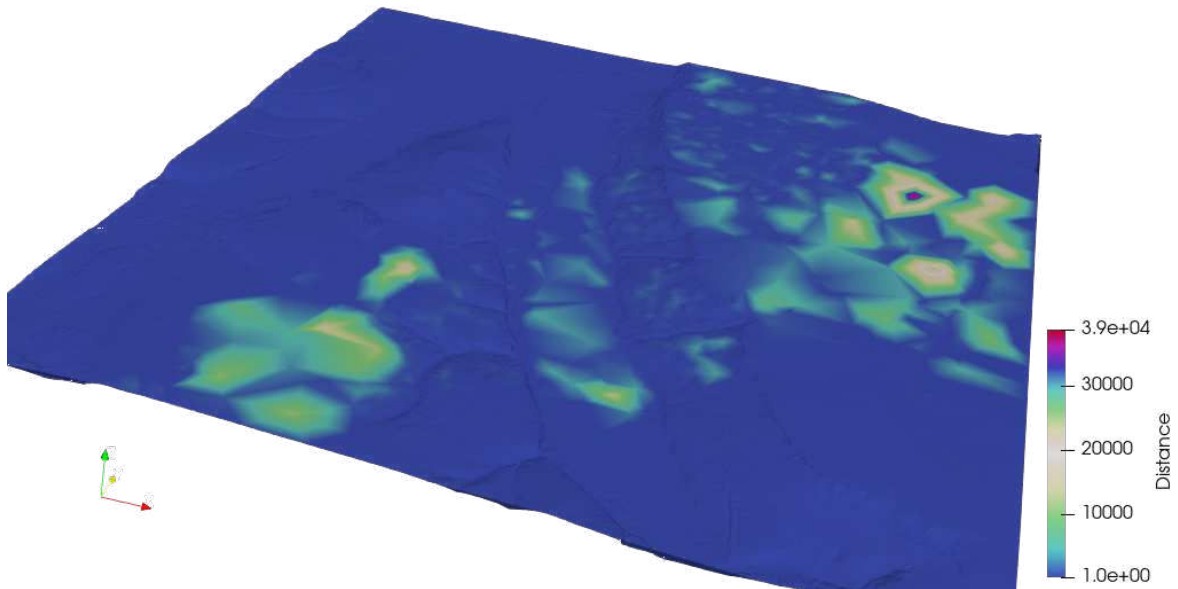


Figure 7.6.: Calculated point distance between two realisations of the `triangulate` construction step from the Kohlberg model

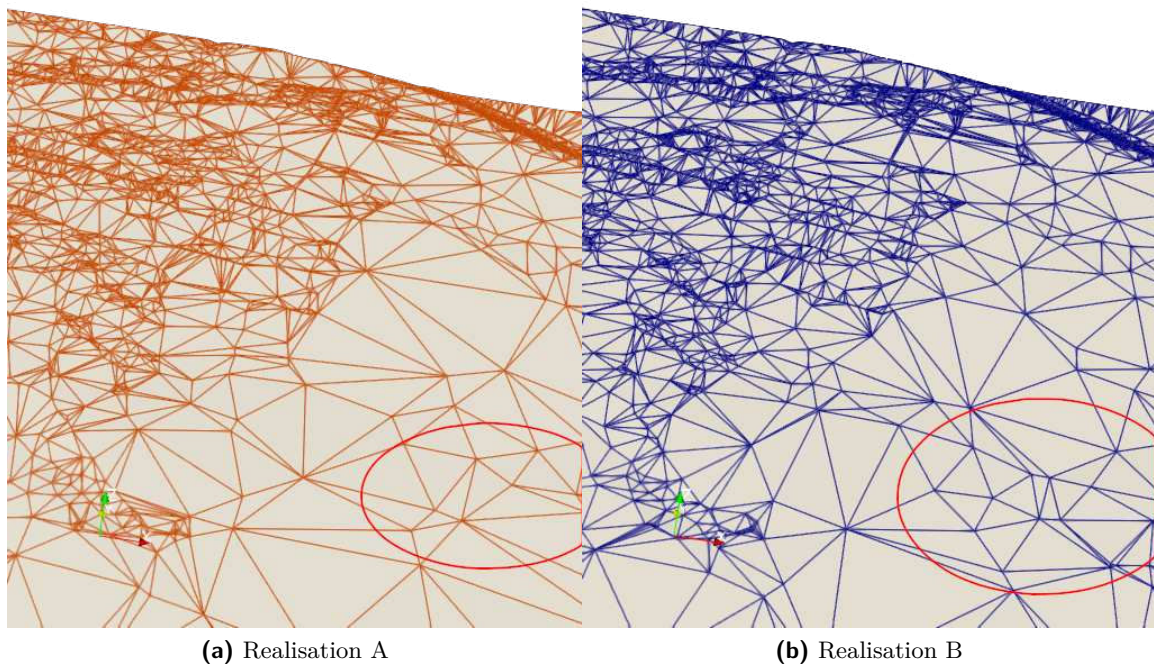


Figure 7.7.: Triangulation differences between two different realisations of the `triangulate` construction step of the Kohlberg model

Depending on the intended use of the final model, the output of different realisations of the original **triangulate** construction step (including the decimate operation) could be considered equal based on a distance based equality definition. Since the described surfaces are nearly identical, we can determine the distance used in approach by using the maximum distance of any point of the reproduction to the original surface. If this value is less than a certain reasonable threshold, the surfaces are considered equal. This definition is only useful if the triangulation has no influence on further construction steps.

The **extrude** construction step generates several output files containing different geometries. See Appendix C.6 for a detailed description of this construction step. One of the generated geometries is a triangulated surface. Similarly to what we have already described for the **triangulate** construction step, the generated result contains the same geometry, only the order of triangles changed in some realisations. These datasets could be considered equal according to the structural equality definition.

The **extract** construction steps extract parts of a geological map given as Geodatabase. Appendix C.3, C.4 contains detailed descriptions of these construction steps. On the first look these steps appear to be reproducible according to the technical equality definition. This is no longer the case as soon as the two realisation used to check the reproducibility were created on different days. This result suggests some sort of dependency onto the execution date of the construction step. Each of these construction steps generate a ESRI Shapefile [115]. A Shapefile consists of multiple separate files in different formats. The mismatch appears between the DBF [159] files of two realisations. The specification of this file format requires to embed the date of the last update as byte 1-3 in the generated file, which is causing the difference. We can consider such datasets as equal according to the structural equality definition. In addition it might be possible to modify the corresponding construction step in such a way that they always set the execution date to a fixed value.

7.3.4. Identified Problems and Construction Process Improvements

By implementing this model construction process using GeoHub, we noticed several issues and opportunities for improvements.

The first minor problem we noticed was that the instructions for constructing the model provided by the Geological Survey of Saxony were incomplete. The instructions were missing important details, such as the configuration of the gOcad project that serves as the basis for the model construction. These settings include several options that can affect the outcome of the modelling process. For example, the definition of the direction of the Z-axis influences the result of many gOcad operations.

A major issue with this case study is the behaviour of certain gOcad operations. We have shown that the gOcad decimate operation can produce different triangulation's for the same input dataset. Further experiments have shown that other operations can also produce different results on repeated runs using the same input dataset. These includes well-known major gOcad operations such as the discrete smooth interpolation algorithm [21], where different realisations result in slightly shifted point coordinates at the edge of the interpolation domain. It seems possible to define a specialised interpretation of equality for all these cases, which may be useful for certain model applications. In our opinion, this is a fundamental problem. Therefore, we suggest to systematically classify the operations provided by gOcad as reproducible or not, given the presented definition of equality.

In addition to these problems, we identified a potential for improvement in the way construction steps are defined in our prototype. As it is currently implemented, a construction step

contains a shell script that describes the specific details. While this is sufficient to perform all required operations, additional code may be required to implement certain operations. In particular, gOcad operations require some additional setup at the beginning and a special command to “close” gOcad at the end of the script. A future version of GeoHub could add support for custom command steps that specify a script processor to automatically perform these steps at the beginning and end of the user provided scripts.

7.3.5. Recommendations

Members of the geomodelling community are experts in building complex geometric representations of the subsurface. These people have advanced skills in working with well-known tools such as gOcad. These tools provide simple interfaces for capturing and automating certain operations. It is relatively easy to extract the information needed to compose new construction steps. Composing a construction steps requires two components: A Docker image of the environment in which the construction step is executed and the shell script to describe the construction step itself. The first part is challenging and requires technical knowledge of operating systems and software. Fortunately, this part only needs to be performed once per application. The second part can be done simply by recording the required steps as a macro via the provided interface of the corresponding software. On this basis, we assume that the more technical skilled part of the geomodelling community will be able to compose their own construction step definitions. Tools such as the gOcad History Import Tool presented in Appendix E can further simplify the composition of construction hypergraphs for specific software packages such gOcad. In addition, they enable non-technical users to build complex construction hypergraphs. They would primarily work in their familiar environment to construct such models and only later export them to GeoHub.

All users should be able to combine existing construction steps into a construction hypergraph that describes how a particular model was built. This simply requires mapping the existing construction description to a construction hypergraph by formalising the construction steps and their inputs. A proven reproducible construction process appears to be an essential feature for projects like the nuclear waste deposit or even planning a major bypass road.

7.4. Hydrologic Balance Model of a Saxonian Stream

7.4.1. Provided Data and Initial Situation

This case study is based on an exercise required to complete the syllabus “Wasserhaushalt/Wasserhaushaltsmodellierung” at the TU Bergakademie Freiberg. In this lecture, students are required to calculate a hydrologic balance model using the BoWaHald software package [30] for a study area of their own choice. This exercise includes the following steps:

1. Subdivision of the study area into smaller parts, called Hydrotopes. This subdivision groups together areas with similar properties. This includes properties like land cover, land use, soil structure, and terrain information.
2. Collecting the required data for each hydrotop. This data include the following information:
 - Climatic data
 - Land cover and land use

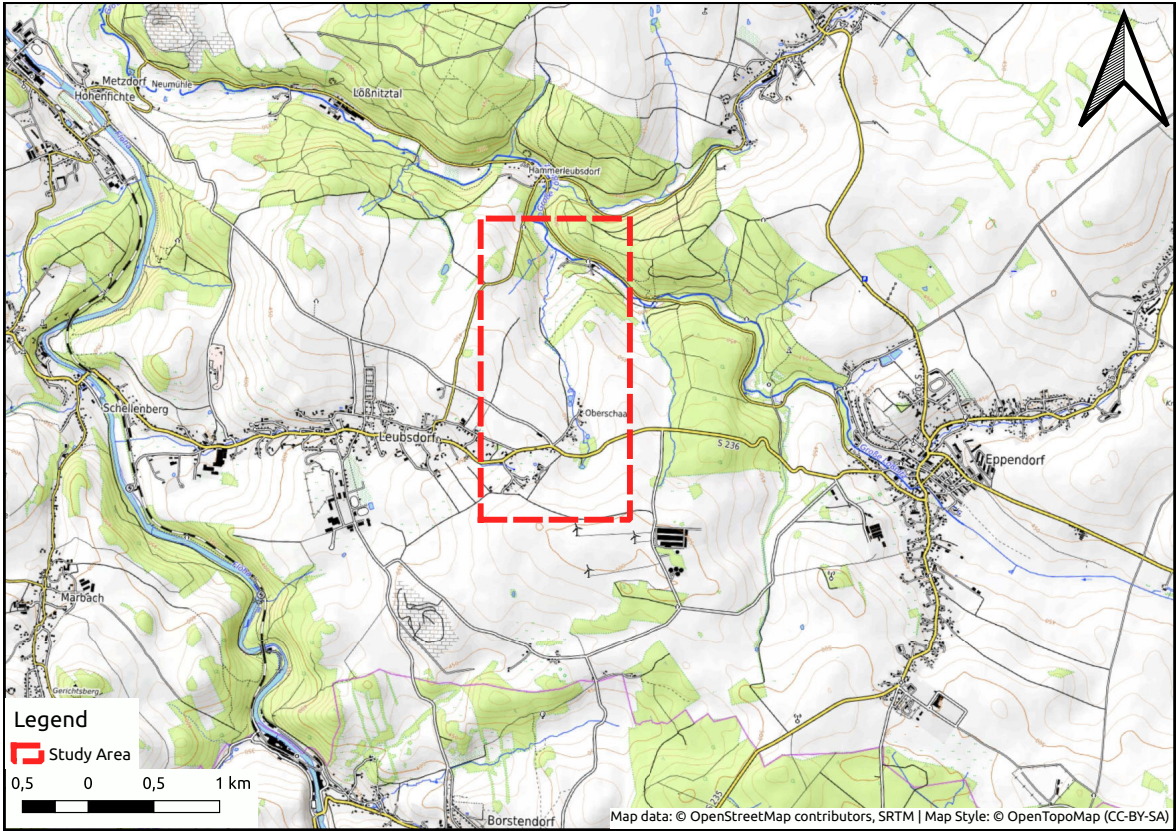


Figure 7.8.: Study area for the Leubsdorf dataset

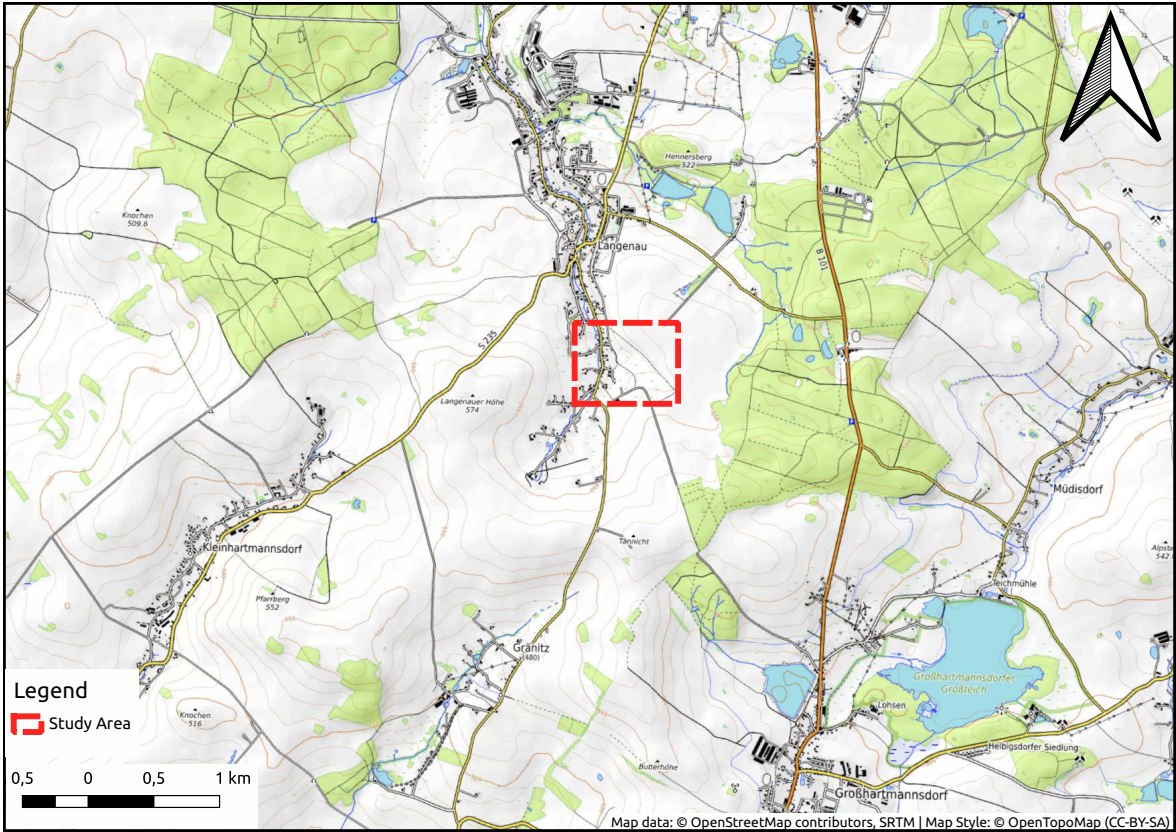


Figure 7.9.: Study area for the Langenau dataset

- Soil structure
 - Terrain information
3. Based on these data, generate the corresponding hydrotop files with BoWaHald. At least three files per hydrotop must be filled with information:
 - A hydrotop file with general topographic information (such as location, height, area) for each hydrotop
 - A soil layer definition file that describes the different soil layers per hydrotop
 - A land-use file with a description of the land use and land cover per hydrotop.
 4. In addition, climate data must be pre-processed to fit the required format of a Microsoft EXCEL table.
 5. Calculate a hydrologic balance model for each hydrotop with BoWaHald
 6. Summarise the calculated models to provide general statements about the hydrologic balance of the study area.

The work done as part of these steps can be divided into two categories: Some of the work is guided by scientific decisions that influence how the actual model should be constrained. Examples of this type of work include deciding how the study area should be subdivided into hydrotopes or what land use should be assumed for a particular hydrotop. The other part of the work involved in calculating such a hydrologic balance requires a lot of manual, repetitive data transformations. Examples include any kind of data processing to get the data into a format accepted by BoWaHald. Since this second type of work is very time-consuming, it lends itself well to automation. Therefore, this case study shows how the tedious steps can be automated for a well-defined workflow.

In this case study a generalised version of this model construction process is implemented. It aims to automate the repetitive steps of model construction while allowing the user to actively influence key parameters of the model. Furthermore, this case study aims to present a unified workflow that allows the same model construction process to be repeated in any study area in Saxony.

The construction process was evaluated with two different model realisations: One describes a small stream near the village of Leubsdorf/Saxony, the other dataset describes meadows and fields in the vicinity of Langenau (a village near Freiberg). Figure 7.8 and figure 7.9 show an overview of the respective study area. The Leubsdorf dataset contains 19 different hydrotopes. The Langenau dataset contains five different hydrotopes. For both datasets, the model is calculated for the time period from 1.11.1990 to 31.10.2020.

7.4.2. Construction Process Description

Figure 7.10 shows a graphical representation of the construction hypergraph used to implement this case study. The construction process for this model consists of 25 datasets and 19 construction steps. Of these 25 datasets, six are provided as input datasets, and two are generated as output datasets. All of the construction steps in this construction hypergraph are designed to be used with different datasets. This design allows the same model construction process to be applied to different datasets to evaluate the underlying model construction process in other geographic areas. This feature is currently limited to areas in Saxony because the input dataset needed to determine the soil type information is only available there in the required format.

The following six datasets must be provided as input datasets:

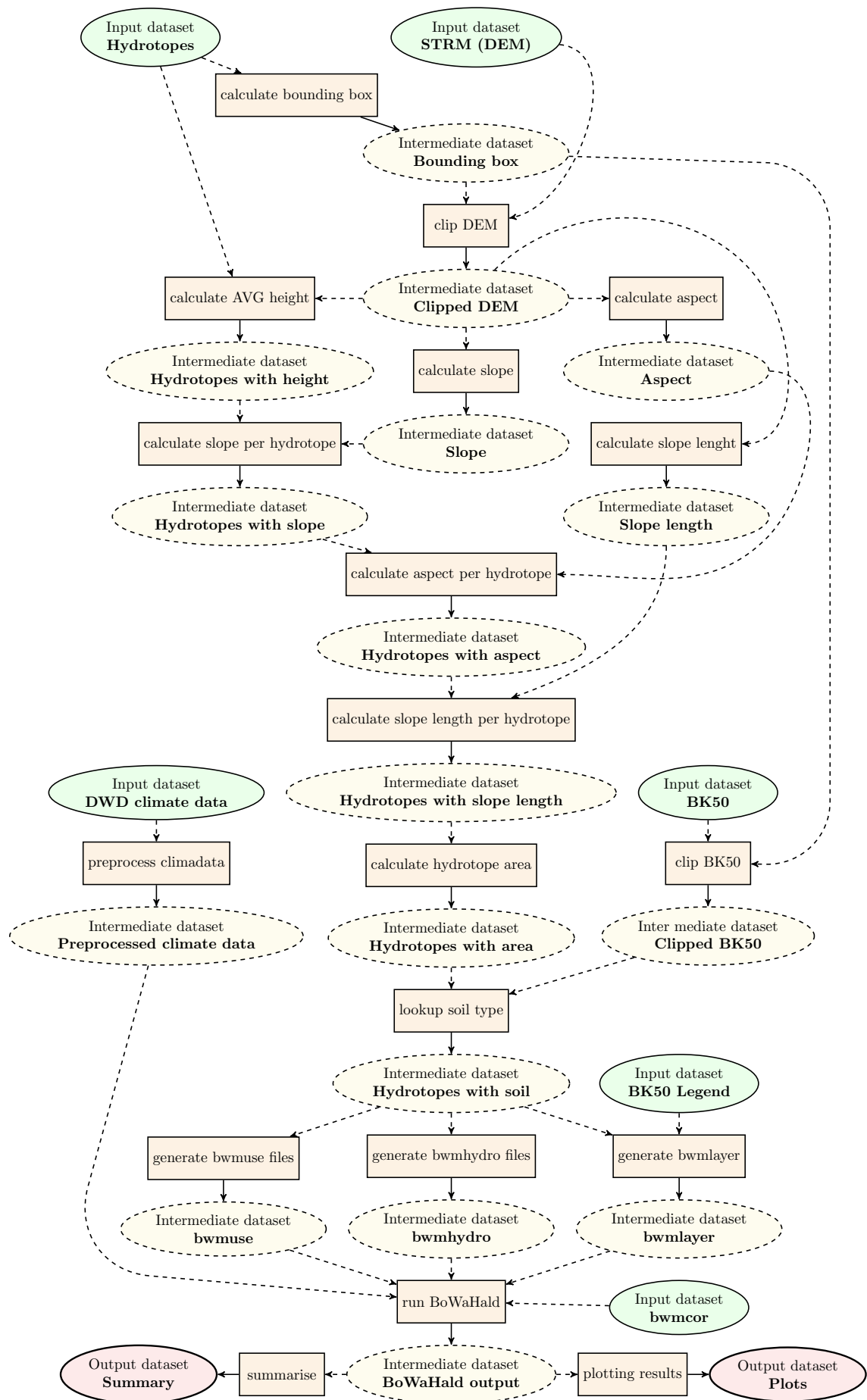


Figure 7.10.: Construction hypergraph for the Hydrologic balance model

1. A copy of the soil map BK50 from the Geological Survey of Saxony [160] as ESRI shapefile
2. The legend of the soil map BK50 from the Geological Survey of Saxony as Microsoft Excel file
3. Climate data of the study area as a CSV file as provided by the German Meteorological Service (DWD)
4. An ESRI shapefile containing the outlines of any hydrotop as polygons.
5. A Shuttle Radar Topography Mission (SRTM) raster tile for the study area, as provided by NASA [161]
6. A BoWaHald [30] precipitation correction file that describes how climate data must be adjusted.

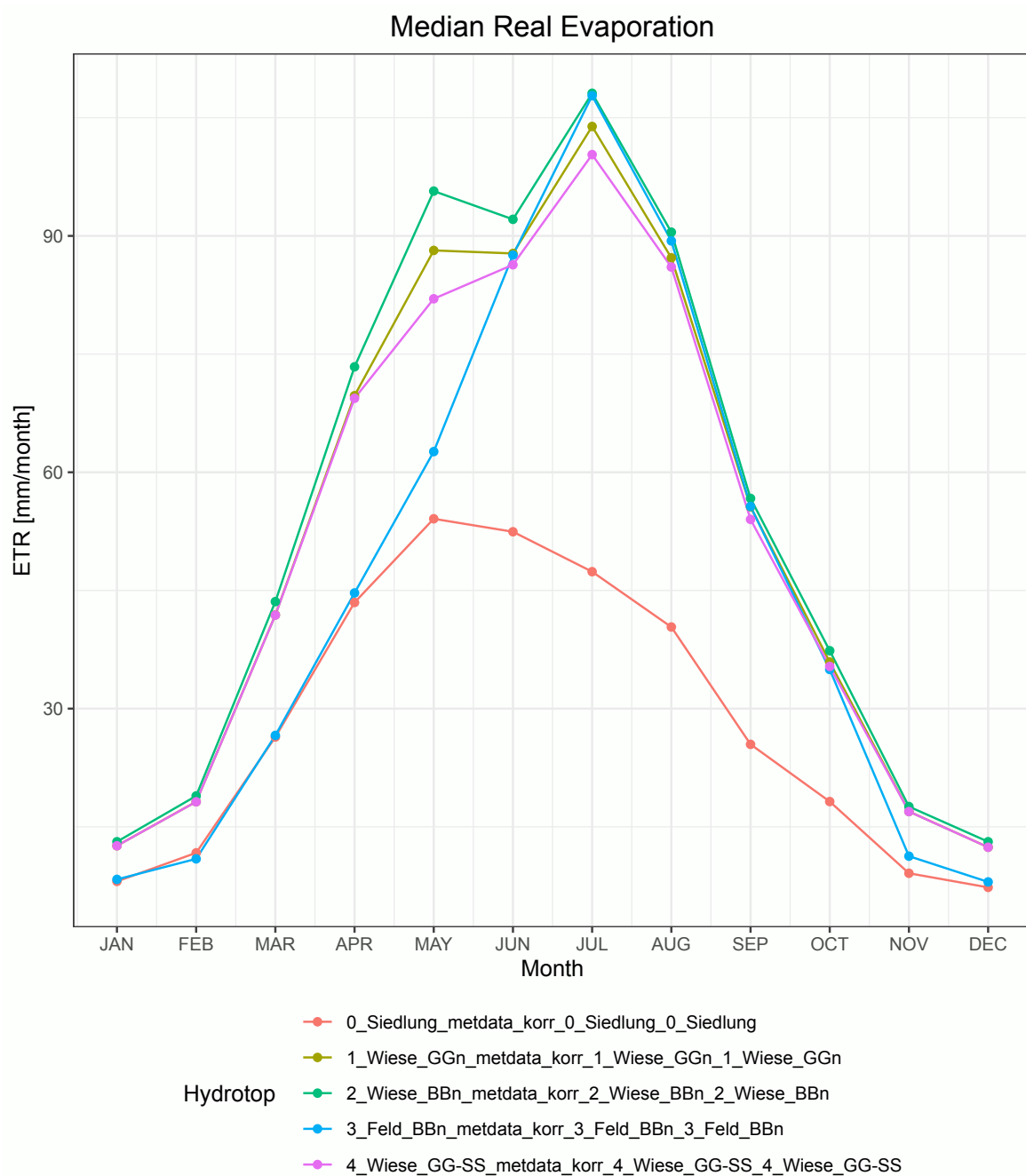


Figure 7.11.: Median monthly evaporation per hydrotop for the Langenau dataset

The construction process generates two output datasets:

1. Five plots showing the monthly median of characteristic hydrologic values per hydrotop. Figure 7.11 shows an example plot for the median monthly evaporation from the Langenau dataset.
2. A statistical summary with the minimum, median and maximum values of characteristic hydrological values per hydrotop. Table 7.2 contains an example result for the Langenau dataset. Table 7.1 contains a description of the relevant values.

The construction process was verified with both the Leubsdorf and Langenau datasets. Appendix D contains a detailed description of all construction steps. The general structure of the construction steps was provided by domain experts, in this case as content of the corresponding university lecture.

Table 7.1.: Characteristic hydrologic values, according to BoWaHald

hydrologic value	long variant	meaning
ETR	real evaporation	Actual amount of water that evaporates in given period of time
ETP	potential evaporation	Theoretical maximum amount of water that will evaporate in a given period of time assuming that this amount of water is available
RO	surface runoff	Amount of water flowing off at the surface
RU	subsurface runoff	Amount of water reaching the bottom of the modelled soil layer sequence
P	precipitation	Amount of water that rains or snows on the modelled area

Table 7.2.: Annual statistical summary of characteristic hydrologic values per hydrotop as produced for the Langenau dataset. All values are in $\frac{mm}{year}$

Hydrotop	ETR_{min}	ETR_{median}	ETR_{max}	ETP_{min}	ETP_{median}	ETP_{max}	RO_{min}	RO_{median}	RO_{max}	RU_{min}	RU_{median}	RU_{max}
Siedlung	194	345	370	364	453	530	330	653	1336	0	36	82
Wiese	490	633	710	549	683	803	0	4	134	86	407	766
GGn												
Wiese	516	660	744	572	710	836	0	0	109	67	379	766
BBn												
Feld	434	544	625	479	594	701	44	123	417	111	351	599
BBn												
Wiese	471	620	697	549	683	803	0	5	144	100	420	763
GG-SS												

Table 7.3.: Yearly statically summary of observed precipitation All values are in $\frac{mm}{year}$

P_{min}	P_{median}	P_{max}
643	1037	1463

7.4.3. Reproducibility

Ten of the 19 construction steps of this model construction process are reproducible according to the bitwise equality definition. The following nine construction steps are not reproducible:

- calculate bounding box
- calculate AVG height
- calculate slope per hydrotop
- calculate aspect per hydrotop
- calculate slope lenght per hydrotop
- calculate hydrotop area
- lookup soil type
- clip BK50
- run bowahald

The first eight construction steps in this list generate a ESRI Shapefile [115]. As already outlined in the previous case study the DBF [159] file format, which is used to store parts of the data, requires to embed the date of the last update as part of the file. This in turn changes the result of these construction steps depending on the execution date. It seems to be reasonable to use the structural equality definition for all of these construction steps.

The `run bowahald` construction step cannot be successfully reproduced using bitwise equality. Listing 7.2 and Listing 7.3 contain a portion of the file list in the TAR archive that is the output of the `run bowahald` construction step. It can be seen that the list of files is different, but all names follow the same general structure. The underlying problem is that BoWaHald inserts the execution timestamp into all output file names as the first part of the file name.

```

1 2021-11-23_10-07__0_Wald_PPn_N_metdata_korr_0_Wald_PPn_N_0_Wald_PPn_N_false_false.xls
2 2021-11-23_10-08__1_Wiese_BBn_S_metdata_korr_1_Wiese_BBn_S_1_Wiese_BBn_S_false_false.xls
3 2021-11-23_10-08__2_Wiese_BBn_N_metdata_korr_2_Wiese_BBn_N_2_Wiese_BBn_N_false_false.xls
4 2021-11-23_10-08__3_Feld_GG-SS_N_metdata_korr_3_Feld_GG-SS_N_3_Feld_GG-SS_N_false_false.xls
5 2021-11-23_10-08__4_Aue_GGn_metdata_korr_4_Aue_GGn_4_Aue_GGn_false_false.xls
6 2021-11-23_10-08__5_Wald_GGn_N_metdata_korr_5_Wald_GGn_N_5_Wald_GGn_N_false_false.xls
7 ...

```

Listing 7.2.: List of files generated by a first run of the `run bowahald` construction step as contained in the result TAR archive

```

1 2021-11-23_10-12__0_Wald_PPn_N_metdata_korr_0_Wald_PPn_N_0_Wald_PPn_N_false_false.xls
2 2021-11-23_10-13__1_Wiese_BBn_S_metdata_korr_1_Wiese_BBn_S_1_Wiese_BBn_S_false_false.xls
3 2021-11-23_10-13__2_Wiese_BBn_N_metdata_korr_2_Wiese_BBn_N_2_Wiese_BBn_N_false_false.xls
4 2021-11-23_10-13__3_Feld_GG-SS_N_metdata_korr_3_Feld_GG-SS_N_3_Feld_GG-SS_N_false_false.xls
5 2021-11-23_10-13__4_Aue_GGn_metdata_korr_4_Aue_GGn_4_Aue_GGn_false_false.xls
6 2021-11-23_10-13__5_Wald_GGn_N_metdata_korr_5_Wald_GGn_N_5_Wald_GGn_N_false_false.xls
7 ...

```

Listing 7.3.: List of files generated by a second run of the `run bowahald` construction step as contained in the result TAR archive

If we match the corresponding files by name, ignoring the included timestamp, we can compare the contents of each file. This comparison shows a difference in a single spreadsheet cell for each pair of files. On closer inspection, this cell does not contain a simulation result, but the name of the result file. This result, in turn, means that the difference is also caused by a timestamp contained in the result file name. Again, it is reasonable to consider the results as equal according to a suitable structural equality definition. We define equality between a set of BoWaHald result files as follows:

Two result file sets D_a , and D_b are equal

- If they contain the same number of files
- For each file F_a in D_a , there is a matching file F_b in the data set D_b , without consideration of the timestamp at the beginning of each file name
- Each file F_b is identical to the corresponding file F_a , except for the single spreadsheet cell that contains the file name

Using this definition of equality for the BoWaHald result files, the entire model construction process is reproducible.

7.4.4. Identified Problems and Construction Process Improvements

Although the presented construction graph implements the construction process for calculating a hydrologic balance model as thought in the lecture “Wasserhaushalt/Wasserhaushaltsmodellierung,” there is always room for improvement. We identified several minor and more major opportunities for improvement for the case study presented.

The `generate bwmlayer file` construction step presented in appendix D.16 contains an explicit lookup table for soil properties based on “Bodenkundliche Kartieranleitung (5. Auflage)” [162]. These values could be provided as an additional input dataset for the construction process. Using an additional input dataset for this information would make it easier to provide modified values for specific soil properties. In addition, this change would allow users to modify soil properties simply by modifying an input dataset, rather than modifying the construction step itself.

As described in the previous section, the construction process is not reproducible using the bitwise equality definition because BoWaHald itself embeds execution timestamps at various locations. BoWaHald could be improved to omit the timestamp or use a general placeholder for the stricter bitwise equality definition. Alternatively, this problem could be circumvented in the definition of the `run bowahald` construction step by manually setting the execution environment time to a fixed value as part of the construction step script.

7.4.5. Recommendations

This case study serves two purposes: It demonstrates that reproducibility is possible outside the realm of subsurface modelling, and it illustrates how workflow automation can significantly reduce the amount of work required to build a new model based on a existing construction process.

We recommend the use of a predefined construction process for this user group, namely persons who want to calculate a hydrological balance model. To build such a predefined construction process, it seems reasonable for an expert from the scientific field to work with knowledgeable GeoHub users to define and implement the general workflow. This construction process can then be reused by almost any other user, as they only need to provide a set of input files as documented by the people who originally created the construction process. In addition, the automated solution presented in this case study dramatically reduces the time and effort required to construct a hydrologic balance model. We estimate that the time required to construct a hydrologic balance model is reduced from about one work week to about half a day. Finally, automation eliminates several potential sources of mistakes, such unit conversions or implicit dependencies between multiple input values.

7.5. Lessons Learned

During the implementing of our three case studies, we gained experience in how real-world construction processes work and how GeoHub can be used in different environments. In the context of this subsection, we would like to provide some general advice on the usage of GeoHub.

We have worked with a variety of people who use the software in different ways to implement the presented construction processes. In general, speaking, we see the following user groups for GeoHub:

- Administrative users
- Construction step designers
- Construction process designers
- Construction process users

An administrative user is responsible for running GeoHub as a service. This work includes setting up the required environment, updating all parts regularly, and performing investigations in case of feedback from other users. This group of users does not need much knowledge about constructing geoscientific models. Instead, they need advanced knowledge in system administration.

Construction step designers define new construction steps from scratch. Therefore, members of this user group need a thorough understanding of the software and its dependencies that they use for their construction step. In addition, knowledge of how to build Docker images is required to define the runtime environment for executing specific construction steps.

Construction process designers use existing definitions of construction step to construct a new construction hypergraph. First, members of this user group must have a deep understanding of the actual construction process. Next, they must be able to answer questions about what data should be combined by which construction step. Finally, they bring the necessary expertise to construct a scientifically sound model.

Construction process users use existing construction processes to generate new realisations of the constructed model by providing a new set of input data. Members of this group do not need to know exactly how a particular model is designed. Instead, they are interested in the results generated by some predefined construction process.

In practice, there is no such clear distinction between users. One and the same user can be a member of several of the defined user groups. This group membership can even depend on concrete construction processes. For example, a user could provide construction steps that are used in construction process A, so that they would belong to the group of construction step designers. The same user could use construction process B only to create new realisations without caring about the details of the construction process implementation.

In addition to the different user groups, we gathered some experience about the actual operation of the software. The current prototype consists of two main components:

- One backend server
- One or more construction step executors

Each component has its own requirements for the environment. For example, the backend server requires access to the network to provide the graphical user interface. In addition, write access to a local directory is required to store all provided datasets, as well access to a PostgreSQL database.

A construction step executor does not need write access to a database or local directory. However, it does need network access to interact with the backend server and any used Docker image repository. In addition, a local Docker runtime must be provided to execute construction step definitions. Finally, adequate computing power must be provided to execute construction steps. The exact amount of computing power required depends on the construction steps themselves. More complex construction steps require more computing power.

Provision of detailed instructions for running a GeoHub service in a production environment remains the subject of future work.

Finally we have gathered some experience about reproducibility of geoscientific construction steps. We have seen examples of reproducible construction steps and we have seen examples of construction steps which produce different results according to some definitions of equality. The most common cause of non-reproducibility that was encountered during the implementation of the presented cases studies was some form of timestamp embedding. Such embedded time information causes bitwise equality to consider two realisations generated at different times as unequal. It seems to be reasonable to consider datasets with such embedded time information as equal for practical applications, as the time information does not change any other of the stored information. This can be achieved either by a suitable structural equality definition or by adjusting the corresponding construction steps such that a predefined timestamp is used in place of the actual execution timestamp. Another two construction steps generated results where the order of elements stored in the generated files changed. While this does not change the represented data, it does change the files representing those data. Again these data could be considered equal according to a suitable structural equality definition. Changing the construction step to not produce these non-deterministic ordering seems to be harder in these cases, as that would require modifying proprietary software. Finally, we encountered two cases of construction steps, that produced different results for the same input. Depending on the actual use case of the geoscientific model this might be problematic.

8. Conclusions

8.1. Summary

In this thesis, we have shown that geoscientific construction processes represented as hypergraphs can be easily repeated. By comparing different realisations of the same construction process, we are able to evaluate the reproducibility of the construction process itself. Figure 8.1 shows an example of a hypergraph representing the construction of a simplified geoscientific model.

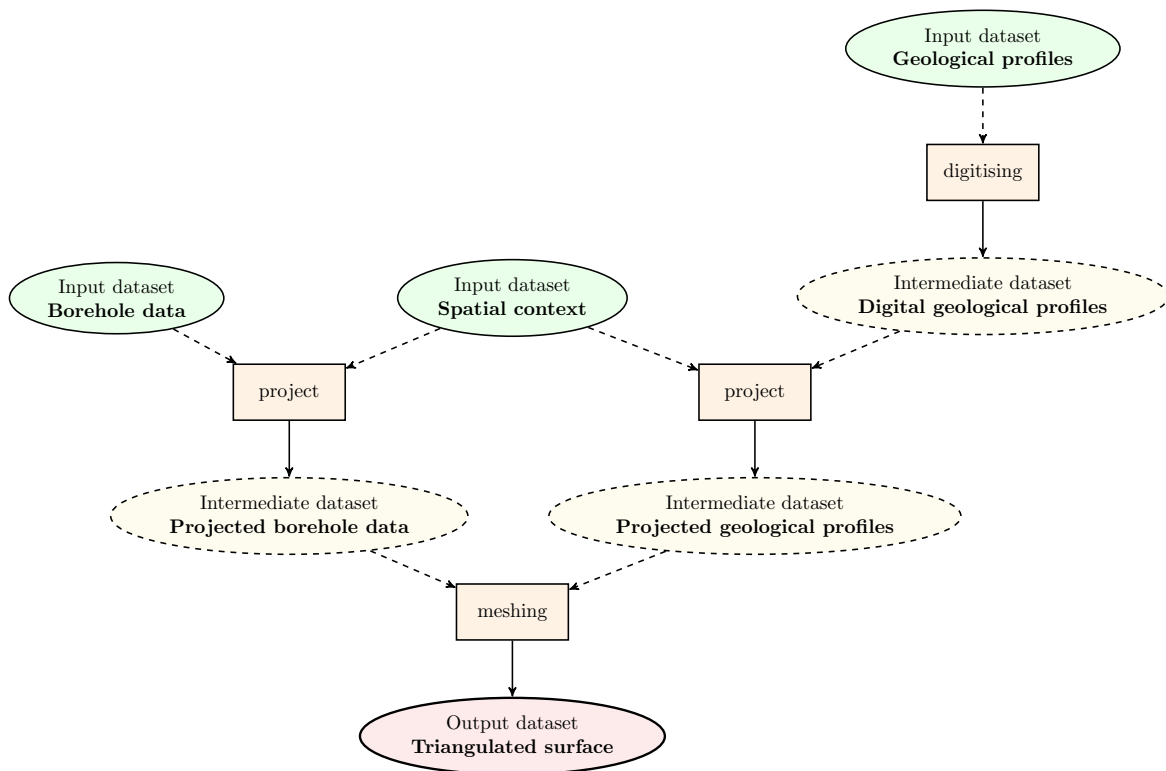


Figure 8.1.: Example construction hypergraph for a simplified geoscientific model construction

Our construction hypergraphs represent the dependencies between datasets and construction steps as directed hypergraphs. The construction steps are represented as hyperedges and the datasets are represented as nodes of the construction hypergraph. The construction hypergraph contains dependencies between different construction steps as explicit information.

In order to verify the reproducibility of a construction process, we have presented a software-independent format for describing construction steps. This format allows us to repeat construction steps, which in turn allows us to repeat an entire construction process. By comparing the results of different realisations of the same construction process, we can draw conclusions about the reproducibility of the overall process. By comparing specific intermediate or output datasets generated by a non-reproducible construction process, we can attribute the non-reproducibility to specific construction steps.

In order to be able to compare different datasets, we have introduced different definitions of dataset equality for different use cases. Bitwise equality enables the comparison of datasets regardless of their data format by comparing them at the byte level. Structural equality allows comparing datasets based on their internal structure. Distance-based equality definitions allow small differences to be included in the comparison. Finally, the geological equality definition allow two datasets to be considered as equal if they support the same geological interpretation.

In addition to the conceptual framework for representing construction processes of geoscientific models as construction hypergraphs, we present a prototype implementation of this concept. For this prototype, we explore different ways to implement components of the software and finally decide on a particular architecture.

Finally, we used the developed prototype to demonstrate the concept using several case studies. These case studies include a geophysical inversion, the construction of three-dimensional subsurface model using gOcad, and the calculation of a hydrologic balance model. These examples demonstrate that our prototype can be used with a variety of other software while providing helpful insights into the reproducibility of these construction processes.

As with all software prototypes, nothing is ever finished. As part of the next section we present a number of possible improvements for GeoHub.

8.2. Outlook

8.2.1. Parametric Model Construction Process

GeoHub allows the description of a specific process for the construction of a geoscientific model to include any detail necessary to repeat the construction. With such a construction process, users can verify that the construction process is reproducible. They can also repeat the same construction process for different datasets to automate time-consuming tasks. Some of these construction process can have adjustable parameters. The hydrologic balance model construction process presented in section 7.4 is an example of such a construction process. The presented construction process is specifically configured to calculate a hydrologic balance model for the period between 1.11.1990 and 31.10.2020. It may be helpful to allow users to efficiently customise this time period.

Theoretically, this could already be implemented today by using another input dataset. However, we believe that it is semantically the wrong place to include information that affects the entire construction process as an ordinary input dataset. Instead, a better approach to handle such information would be to parameterise the entire construction process with this information. For example, the implementation of such a feature could introduce a construction process wide variable. These would need to be configured for each construction process realisation and would be stored as part of the realisation. Each configured construction process variable could then be passed as an environment variable into the runtime environment of each construction step during the construction process execution.

8.2.2. Pull and Push Nodes

There are various solutions for storing geoscientific models. They are specialised for specific use cases and may already contain large amounts of datasets. Examples include databases for borehole logs and subsurface models provided by geological surveys, climate data provided

by the various meteorological surveys, or Earth observation data provided by space agencies. These data sources are well established. It would be beneficial for any new application to integrate these existing data sources. Such a integration is especially meaningful for GeoHub, as it can provide automated construction processes to build models based on datasets.

An important functionality for integration with these data sources is the ability to automatically fetch data from a defined source or write it to a defined destination. In this context, this means fetching data from one of the data storage solutions via a defined API to use it as an input dataset for a construction process, or writing an output dataset to one of the existing storage solutions. Support for both can be integrated into GeoHub's concept of input or output datasets by including different types of data nodes in the construction hypergraph. GeoHub fetches an input dataset from a remote location according to a defined procedure for pull input nodes. For push output nodes, the resulting output dataset is written to a specified remote location.

This alone would better integrate GeoHub into existing setups. With the additional improvement of providing an external API to trigger construction process executions or allow GeoHub to wait for dataset changes on one of the pull input nodes, these enhanced input and output node definitions could largely automate data processing for use cases where the same construction process need to be executed over and over again for continuously updated data.

8.2.3. Parallelize Single Construction Steps

The current GeoHub prototype assumes that a dataset consists of a defined set of files. Each construction step is then defined to consume several datasets and produce a single dataset. This assumption is not always true, as several steps of the hydrologic balance case study demonstrate. This is particularly evident for the `generate bwmhydro files` (Appendix D.14), `generate bwmuse files` (Appendix D.15), `generate bwmlayer files` (Appendix D.16), and `run bowahald` (Appendix D.17) construction steps. These construction steps operate on datasets containing a variable number of files, depending on the contents of the specific input datasets. They emulate the handling of a dynamic number of files per input/output dataset by using a TAR archive. Such an archive can contain several files in a single archive represented as file. This additional step requires that the definitions of these construction steps contain certain amount of code that handles the (un)packing of these archives and possibly the repetition of an operation for each file contained in these archives.

GeoHub could more effectively support such use cases by allowing construction hypergraphs to signal that a node represents not just a single dataset, but a collection of datasets of the same type that may contain a variable number of entries. This concept would require different types of construction step definitions. Those that support only a single dataset input/output node, those that have either a dataset collection as input or output, and those that have both a dataset collection as input and output. At least for the last new type of construction step, it appears to be possible to execute the construction step per entry in the input dataset collection. This means that for an input dataset collection with n entries, the construction step is executed n times to generate an output collection with n entries.

8.2.4. Provable Model Construction Process Attestation

A reproducible construction process can be essential and even required for certain use cases. GeoHub provides a way to describe construction processes in a reproducible manner. There is

no way to confirm that no person has later maliciously modified any part of the construction process implementation. Depending on the use case, there may be considerable interest in changing the results subsequently, whether due to personal opinions, outside pressure, or even political influence. Again, a motivating example is the search for a nuclear waste deposit site. It must be presumed that any decision on the location of the deposit site will be challenged in court. In such a situation, it might be important to show not only how a particular model was built, but also that the presented realisation is consistent with the realisation that the decision was based on.

Modern cryptography offers tools which might be helpful here. Cryptographic hash functions are designed to provide a unique, non-guessable mapping from a corpus of data to a fixed-size value. This value can then be used to identify that specific data corpus. Applied to GeoHub, this means that we could implement a system that systematically hashes all relevant information for a given construction process realisation. This hash can then be published, for example as part of a final report on some important decisions. Since this information is public from that moment on, it can be used as evidence in a subsequent dispute to prove that the specific construction process realisation has not been changed since the time of the publication.

Cryptographic signature methods can be used to prove that a specific person, who has a cryptographic secret, has signed a specific set of data. In combination with the hash procedure described earlier, this can be used to prove that a specific person signed a construction process realisation at a specific time. This signature can also be published as part of the final report and later used as proof that this person or institution signed this realisation. It is important to note that the signature should contain all relevant data, including the hash value mentioned above. The general guidelines for keeping cryptographic key material secret must be followed in this case.

Our general proposal is to provide a multi-stage procedure for both the hash and signature scheme. Each dataset D_i that is part of the construction process realisation W_i should have a separate hash value that depends at least on the following information:

- The exact content of all files of the dataset
- The definition of the construction steps of the incoming hyperedge
- The construction step environment of the incoming hyperedge
- Each hash value of the direct input datasets of the construction step corresponding to an incoming hyperedge

For the input datasets of the construction process, only the first bullet point applies. Any other dataset requires additional information about the incoming hyperedge to later confirm the correct version of the full construction process. It is important to only include information that will be available later such that a third party can independently verify these hash values from datasets, construction step descriptions, and construction step environments downloaded from a GeoHub instance. This means that at least a minimal implementation of the underlying hash scheme should be available as public code. Since a third party can use all available information to independently build a new realisation of the same model and compare the generated hash values, it seems essential to allow only bitwise equality in this context. According to the scheme suggested above, any other definition of equality would alter the recorded hash values. The exact amount of information contained in the hashes and the exact implementation of the suggested scheme should be done by a person educated in cryptography.

References

- [1] Springer Berlin Heidelberg, “GEM - International Journal on Geomathematics”, *Scope* [Online]. Available: <https://link.springer.com/journal/13137>; [Accessed: Aug. 02, 2022]
- [2] M. J. Steventon, C. A.-L. Jackson, M. Hall, M. T. Ireland, M. Munafo, and K. J. Roberts, “Reproducibility in Subsurface Geoscience”, *Earth Science, Systems and Society*, 2022, doi: 10.3389/esss.2022.10051.
- [3] C. Collberg, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, “Measuring Reproducibility in Computer Systems Research”, 2014 [Online]. Available: <http://reproducibility.cs.arizona.edu/tr.pdf>. [Accessed: Aug. 11, 2022]
- [4] M. Baker, “1,500 scientists lift the lid on reproducibility”, *Nature News*, May 2016, doi: 10.1038/533452a.
- [5] M. Konkol, C. Kray, and M. Pfeiffer, “Computational reproducibility in geoscientific papers: Insights from a series of studies with geoscientists and a reproduction study”, *International Journal of Geographical Information Science*, Aug. 2018, doi: 10.1080/13658816.2018.1508687.
- [6] G. Caumon, “Geological objects and physical parameter fields in the subsurface: A review”, in *Handbook of Mathematical Geosciences*, Springer, Cham, 2018 [Online]. Available: <https://link.springer.com/content/pdf/10.1007/978-3-319-78999-6.pdf>. [Accessed: Aug. 11, 2022]
- [7] C. A.-L. Jackson and A. Rotevatn, “3D seismic analysis of the structure and evolution of a salt-influenced normal fault zone: A test of competing fault growth models”, *Journal of Structural Geology*, Sep. 2013, doi: 10.1016/j.jsg.2013.06.012.
- [8] S. Perrouy, M. D. Lindsay, M. W. Jessell, L. Aillères, R. Martin, and Y. Bourassa, “3D modeling of the Ashanti Belt, southwest Ghana: Evidence for a litho-stratigraphic control on gold occurrences within the Birimian Sefwi Group”, *Ore Geology Reviews*, Dec. 2014, doi: 10.1016/j.oregeorev.2014.05.011.
- [9] M. J. Fetkovich, “Decline Curve Analysis Using Type Curves”, in *Fall Meeting of the Society of Petroleum Engineers of AIME*, Sep. 1973, doi: 10.2118/4629-MS.
- [10] D. S. Oliver and Y. Chen, “Recent progress on reservoir history matching: A review”, *Computational Geosciences*, Jan. 2011, doi: 10.1007/s10596-010-9194-2.
- [11] M. Bosch, T. Mukerji, and E. F. Gonzalez, “Seismic inversion for reservoir properties combining statistical rock physics and geostatistics: A review”, *GEOPHYSICS*, Sep. 2010, doi: 10.1190/1.3478209.
- [12] M. Moorkamp, B. Heincke, M. Jegen, R. W. Hobbs, and A. W. Roberts, “Joint Inversion in Hydrocarbon Exploration”, in *Integrated Imaging of the Earth*, American Geophysical Union (AGU), 2016.
- [13] P. K. Fullagar, N. A. Hughes, and J. Paine, “Drilling-constrained 3D gravity interpretation”, *Exploration Geophysics*, 2000, doi: 10.1071/eg00017.

- [14] P. Krause, D. P. Boyle, and F. Bäse, “Comparison of different efficiency criteria for hydrological model assessment”, in *Advances in Geosciences*, Dec. 2005, doi: 10.5194/adgeo-5-89-2005.
- [15] H. J. Henriksen, L. Troldborg, P. Nyegaard, T. O. Sonnenborg, J. C. Refsgaard, and B. Madsen, “Methodology for construction, calibration and validation of a national hydrological model for Denmark”, *Journal of Hydrology*, Sep. 2003, doi: 10.1016/S0022-1694(03)00186-0.
- [16] R Core Team, *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing, 2021 [Online]. Available: <https://www.R-project.org/>. [Accessed: Aug. 11, 2022]
- [17] J.-L. Mallet, P. Jacquemin, and N. Cheimanoff, “GOCAD project: Geometric modeling of complex geological surfaces”, in *1989 SEG Annual Meeting*, 1989, doi: 10.1190/1.1889515.
- [18] “ArcGIS Desktop”. Environmental Systems Research Institute, Redlands, CA, 2022 [Online]. Available: <https://www.esri.com/en-us/arcgis/products/index>
- [19] M. de la Varga, A. Schaaf, and F. Wellmann, “GemPy 1.0: Open-source stochastic geological modeling and inversion”, *Geoscientific Model Development*, vol. 12, Jan. 2019, doi: 10.5194/gmd-12-1-2019.
- [20] G. van Rossum, “Python Reference Manual”, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, CS-R9525, May 1995 [Online]. Available: <https://ir.cwi.nl/pub/5008/05008D.pdf>. [Accessed: Aug. 11, 2022]
- [21] J.-L. Mallet, “Discrete smooth interpolation”, *ACM Transactions on Graphics*, Apr. 1989, doi: 10.1145/62054.62057.
- [22] H. E. Plesser, “Reproducibility vs. Replicability: A Brief History of a Confused Terminology”, *Frontiers in Neuroinformatics*, vol. 11, 2018, doi: 10.3389/fninf.2017.00076.
- [23] S. Houlding, *3D Geoscience Modeling: Computer Techniques for Geological Characterization*. Springer Science & Business Media, 1994.
- [24] F. Wellmann and G. Caumon, “3-D Structural geological models: Concepts, methods, and uncertainties”, in *Advances in Geophysics*, C. Schmeltzbach, Ed. Elsevier, 2018.
- [25] A. Bokulich and N. Oreskes, “Models in Geosciences”, in *Springer Handbook of Model-Based Science*, L. Magnani and T. Bertolotti, Eds. Cham: Springer International Publishing, 2017.
- [26] H.-J. Götz, “TiPot – Towards the Integrative Interpretation of Potential Field Data by 3D Modelling & Visualization”. Freiberg, Oct. 2020 [Online]. Available: https://tu-freiberg.de/sites/default/files/media/iamg-student-chapter-34575/slides_20201015.pdf. [Accessed: Aug. 11, 2022]
- [27] “Artifact Review and Badging”. Aug. 2020 [Online]. Available: <https://www.acm.org/publications/policies/artifact-review-and-badging-current>. [Accessed: Aug. 11, 2022]
- [28] J. F. Claerbout and M. Karrenbach, “Electronic documents give reproducible research a new meaning”, *SEG technical program expanded abstracts 1992*, 1992, doi: 10.1190/1.1822162.
- [29] J. Pellerin *et al.*, “RINGMesh: A programming library for developing mesh-based geomodeling applications”, *Computers & Geosciences*, Jul. 2017, doi: 10.1016/j.cageo.2017.03.005.

- [30] V. Dunger, “Entwicklung und Anwendung des Modells BOWAHALD zur Quantifizierung des Wasserhaushaltes oberflächengesicherter Deponien und Halden”, Habilitation, TU Bergakademie Freiberg, Freiberg, 2007 [Online]. Available: <https://doi.org/10.23689/fidgeo-668>. [Accessed: Aug. 11, 2022]
- [31] PostGIS project, “PostGIS - Spatial and Geographic objects for PostgreSQL”. 2022 [Online]. Available: <http://postgis.net/>. [Accessed: Sep. 30, 2022]
- [32] PostgreSQL project, “PostgreSQL”. 2022 [Online]. Available: <https://www.postgresql.org/>. [Accessed: Sep. 30, 2022]
- [33] A. Furieri, “SpatiaLite”. Feb. 2021 [Online]. Available: <https://www.gaia-gis.it/fossil/libspatialite/index>. [Accessed: Nov. 03, 2022]
- [34] Oracle Corporation, “Oracle Spatial and Graph”. Jan. 2019 [Online]. Available: <https://www.oracle.com/de/database/spatial/>. [Accessed: Nov. 03, 2022]
- [35] “Chapter 38. Extending SQL”, *PostgreSQL Documentation*. Sep. 2021 [Online]. Available: <https://www.postgresql.org/docs/14/extend.html>. [Accessed: Nov. 04, 2021]
- [36] P. Gabriel and J. Gietzel, “Geosciences in Space and Time”. Giga Infosystems, Freiberg, 2022 [Online]. Available: <http://giga-infosystems.com/information>. [Accessed: Sep. 30, 2022]
- [37] J. Gietzel, “Eine datenbankbasierte Verwaltung begrenzungsfreier und hoch aufgelöster CAD-Modelle”, {{MSc Thesis}}, Technische Universität Bergakademie Freiberg, 2011.
- [38] Sächsisches Landesamt für Umwelt, Landwirtschaft und Geologie, “Geomodel Mittleres Erzgebirge (Rohsa 3.1)”. Apr. 2022 [Online]. Available: <https://www.umwelt.sachsen.de/umwelt/infosysteme/gst3/webgui/gui2.php?viewHash=22b40023bf64ff49387596dc18c7f3d7>. [Accessed: Apr. 08, 2022]
- [39] G. Semmler, “A database system for large, dynamically expanding three dimensional geomodels”, {{MSc Thesis}}, Technische Universität Bergakademie Freiberg, 2017.
- [40] S. Chacon and B. Straub, *Pro Git*. Apress, 2014 [Online]. Available: <https://github.com/progit/progit2/releases/download/2.1.223/progit.pdf>
- [41] J.-L. Mallet, *Elements of mathematical sedimentary geology: The GeoChron model*. EAGE publications, 2014.
- [42] P. Bourke, “GoCad: ASCII file data formats”. Apr. 2008 [Online]. Available: <http://paulbourke.net/dataformats/gocad/gocad.pdf>. [Accessed: Jul. 14, 2022]
- [43] Y. Shafranovich, “Common format and MIME type for comma-separated values (CSV) files”, Oct. 2005, doi: 10.17487/RFC4180.
- [44] “Release notes — NumPy v1.22 Manual”. 2022 [Online]. Available: <https://numpy.org/doc/stable/release.html>. [Accessed: Jan. 20, 2022]
- [45] “6. Expressions — Python 3.10.0 documentation”. 2021 [Online]. Available: <https://docs.python.org/3/reference/expressions.html#comparisons>. [Accessed: Nov. 05, 2021]
- [46] Y. Halchenko *et al.*, “DataLad: Distributed system for joint management of code, data, and their relationship”, *The Journal of Open Source Software*, Jul. 2021, doi: 10.21105/joss.03262.
- [47] J. Hess, “Git-annex”. Jun. 2022 [Online]. Available: <https://git-annex.branchable.com/>. [Accessed: Jul. 14, 2022]
- [48] A. S. Wagner *et al.*, “The DataLad handbook”, *Zenodo*, Apr. 2022, doi: 10.5281/zenodo.6463273.

- [49] Japan Electronics and Information Technology Industries Association, “Exchangeable image file format for digital still cameras: EXIF version 2.2”. JEITA CP-3451, Apr. 2002 [Online]. Available: <https://web.archive.org/web/20131019050323/http://www.exif.org/Exif2-2.PDF>. [Accessed: Sep. 30, 2022]
- [50] Benjamin Young and JSON-LD Working Group, Eds., “JSON-LD 1.1”. May 2020 [Online]. Available: <https://w3c.github.io/json-ld-syntax/>. [Accessed: Nov. 02, 2021]
- [51] Ruslan Kuprieiev, skshetry, Paweł Redzyński, Dmitry Petrov, and Peter Rowlands, “DVC”. Iterative, Oct. 2021 [Online]. Available: <https://github.com/iterative/dvc>. [Accessed: Oct. 07, 2021]
- [52] Olivia Mackall, “Mercurial SCM”. Jul. 2022 [Online]. Available: <https://www.mercurial-scm.org/>. [Accessed: Jul. 14, 2022]
- [53] “Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3)”. Amazon, 2021 [Online]. Available: <https://aws.amazon.com/s3/>. [Accessed: Jun. 16, 2021]
- [54] “Git Large File Storage”. GitHub Inc., 2021 [Online]. Available: <https://git-lfs.github.com/>. [Accessed: Oct. 07, 2021]
- [55] “DVC Command Reference (run)”. 2022 [Online]. Available: <https://dvc.org/doc/command-reference/run>. [Accessed: Jan. 21, 2022]
- [56] K. Thompson, “Reflections on trusting trust”, *Communications of the ACM*, Aug. 1984, doi: 10.1145/358198.358210.
- [57] C. Lamb, H. Levsen, M. Rizzolo, and V. Cascadian, “Reproducible Builds”. 2021 [Online]. Available: <https://reproducible-builds.org/docs/definition/>. [Accessed: Oct. 28, 2021]
- [58] “Overview of various statistics about reproducible builds”. 2021 [Online]. Available: <https://tests.reproducible-builds.org/debian/reproducible.html>. [Accessed: Oct. 29, 2021]
- [59] “Reproducible archlinux ?!”. 2021 [Online]. Available: <https://tests.reproducible-builds.org/archlinux/archlinux.html>. [Accessed: Oct. 29, 2021]
- [60] “Is NixOS Reproducible?”. 2021 [Online]. Available: <https://r13y.com/>. [Accessed: Oct. 29, 2021]
- [61] “Known issues related to reproducible builds”. 2021 [Online]. Available: https://tests.reproducible-builds.org/debian/index_issues.html. [Accessed: Oct. 29, 2021]
- [62] “ReproducibleBuilds/StandardEnvironmentVariables - Debian Wiki”. 2021 [Online]. Available: <https://wiki.debian.org/ReproducibleBuilds/StandardEnvironmentVariables#Checklist>. [Accessed: Oct. 29, 2021]
- [63] C. Lamb, A. Ayer, H. Levsen, M. Rizzolo, and M. Herbert, “Reproducible Builds / strip-nondeterminism”. Debian, 2022 [Online]. Available: <https://salsa.debian.org/reproducible-builds/strip-nondeterminism>. [Accessed: Jan. 21, 2022]
- [64] *Executable and linking format (ELF) specification*. TIS Committee; Tool Interface Standard (TIS) Committee, 1995 [Online]. Available: <https://refspecs.linuxfoundation.org/elf/elf.pdf>
- [65] A. Singh, “Portable Executable File Format”, *Identifying Malicious Code Through Reverse Engineering*, Jan. 2009, doi: 10.1007/978-0-387-89468-3.
- [66] J. W. Hunt and M. D. MacIlroy, “An algorithm for differential file comparison”, *Computing science technical report*, vol. 41, 1975 [Online]. Available: <https://www.cs.dartmouth.edu/~doug/diff.pdf>. [Accessed: Oct. 01, 2022]

- [67] A. Schaaf, M. de la Varga, F. Wellmann, and C. E. Bond, “Constraining stochastic 3-D structural geological models with topology information using approximate Bayesian computation in GemPy 2.1”, *Geoscientific Model Development*, Jun. 2021, doi: 10.5194/gmd-14-3899-2021.
- [68] S. T. Thiele, M. W. Jessell, M. Lindsay, J. F. Wellmann, and E. Pakyuz-Charrier, “The topology of geology 2: Topological uncertainty”, *Journal of Structural Geology*, Oct. 2016, doi: 10.1016/j.jsg.2016.08.010.
- [69] S. T. Thiele, M. W. Jessell, M. Lindsay, V. Ogarko, J. F. Wellmann, and E. Pakyuz-Charrier, “The topology of geology 1: Topological analysis”, *Journal of Structural Geology*, Oct. 2016, doi: 10.1016/j.jsg.2016.08.009.
- [70] M. Sen and T. Duffy, “GeoSciML: Development of a generic GeoScience Markup Language”, *Computers & Geosciences*, Nov. 2005, doi: 10.1016/j.cageo.2004.12.003.
- [71] A. Shaon *et al.*, “Long-term sustainability of spatial data infrastructures: A metadata framework and principles of geo-archiving.”, in *Proceedings of the 8th International Conference on Digital Preservation, iPRES 2011, Singapore, November 1-4, 2011*, 2011, doi: 11353/10.294224.
- [72] X. Specka *et al.*, “The BonaRes metadata schema for geospatial soil-agricultural research data – Merging INSPIRE and DataCite metadata schemes”, *Computers & Geosciences*, Nov. 2019, doi: 10.1016/j.cageo.2019.07.005.
- [73] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st edition. Boston: Addison-Wesley Professional, 2002.
- [74] E. G. Haffner, *Informatik für Dummies, Das Lehrbuch*. Weinheim: Wiley VCH, 2017.
- [75] T. Q. Company, “Qt | Cross-platform software development for embedded & desktop”. [Online]. Available: <https://www.qt.io>. [Accessed: Mar. 16, 2021]
- [76] “Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS”. [Online]. Available: <https://www.electronjs.org/>. [Accessed: Mar. 16, 2021]
- [77] R. Fielding *et al.*, “RFC 2616, hypertext transfer protocol – HTTP/1.1”, Jun. 1999, doi: 10.17487/RFC2616.
- [78] M. Fowler, “Microservices”, *martinfowler.com*. Mar. 2014 [Online]. Available: <https://martinfowler.com/articles/microservices.html>. [Accessed: Feb. 16, 2021]
- [79] M. E. Conway, “How do committees invent?”, *Datamation*, Apr. 1968 [Online]. Available: <http://www.melconway.com/research/committees.html>. [Accessed: Oct. 01, 2022]
- [80] U. Joshi, “Patterns of Distributed Systems”, *martinfowler.com*. Aug. 2020 [Online]. Available: <https://martinfowler.com/articles/patterns-of-distributed-systems/>. [Accessed: Mar. 17, 2021]
- [81] M. O. Manero, “Plugins in Rust: The Technologies”. May 2021 [Online]. Available: <https://nullderef.com/blog/plugin-tech/>. [Accessed: Jun. 01, 2021]
- [82] D. M. Beazley, B. D. Ward, and I. R. Cooke, “The inside story on shared libraries and dynamic loading”, *Computing in Science Engineering*, Sep. 2001, doi: 10.1109/5992.947112.
- [83] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st edition. San Francisco: No Starch Press, 2010.
- [84] “PostgreSQL: Documentation: 12: 37.10. C-Language Functions”. [Online]. Available: <https://www.postgresql.org/docs/current/xfunc-c.html>. [Accessed: Jul. 30, 2020]

- [85] “Kernel modules — The Linux Kernel documentation”. [Online]. Available: https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html. [Accessed: Jul. 30, 2020]
- [86] B. Yee *et al.*, “Native Client: A Sandbox for Portable, Untrusted X86 Native Code”, in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, doi: 10.1109/SP.2009.25.
- [87] B. Ford and R. Cox, “Vx32: Lightweight, User-level Sandboxing on the X86”, in *ATC’08: USENIX 2008 Annual Technical Conference*, Jun. 2008 [Online]. Available: https://www.usenix.org/legacy/event/usenix08/tech/full_papers/ford/ford_html/. [Accessed: Jun. 01, 2021]
- [88] “AMD64 Architecture Programmer’s Manual”, Advanced Micro Devices, Oct. 2020 [Online]. Available: <https://www.amd.com/system/files/TechDocs/24592.pdf>. [Accessed: Apr. 14, 2021]
- [89] D. Seal and D. Jagger, *ARM Architecture Reference Manual*, Second. Addison-Wesley Professional, 2001.
- [90] H. J. Lu, M. Matz, M. Grikar, J. Hubička, A. Jaeger, and M. Mitchell, Eds., “System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0”. Dec. 2018 [Online]. Available: <https://gitlab.com/x86-psABIs/x86-64-ABI>. [Accessed: Oct. 01, 2022]
- [91] J. MacCall, R. Smith, J. Merrill, T. Honermann, M. Herrick, and R. Anguiano, “Itanium C++ ABI”. 2021 [Online]. Available: <https://itanium-cxx-abi.github.io/cxx-abi/abi.html>. [Accessed: Jul. 30, 2020]
- [92] M. Ilseman, B. Wilson, and J. Holdsworth, “Swift ABI Stability Manifesto”, 2020 [Online]. Available: <https://github.com/apple/swift/blob/143fcd44c8ef6fb433b9b875dbec18acfccee0c/docs/ABISecurityManifesto.md>. [Accessed: Jul. 30, 2020]
- [93] ISO Central Secretary, Ed., “Information technology — Programming languages — C”, International Organization for Standardization, Geneva, CH, Standard ISO/IEC TR 9899:2018, Jun. 2018 [Online]. Available: <https://www.iso.org/standard/74528.html>
- [94] R. T. Forti, A. Polukhin, and K. Morgenstern, “Chapter 12. Boost.DLL - 1.75.0”. [Online]. Available: https://www.boost.org/doc/libs/1_75_0/doc/html/boost_dll.html. [Accessed: Apr. 14, 2021]
- [95] “Third Party Cargo Subcommands”, *GitHub*. [Online]. Available: <https://github.com/rust-lang/cargo/wiki/Third-party-cargo-subcommands>. [Accessed: Jul. 30, 2020]
- [96] J. MacFarlane, “Pandoc filters”. [Online]. Available: <https://pandoc.org/filters.html>. [Accessed: Jul. 30, 2020]
- [97] “Docker”. Docker Inc., Palo Alto, CA, Jun. 2022 [Online]. Available: <https://www.docker.com/>. [Accessed: Jul. 15, 2022]
- [98] C. Boettiger, “An introduction to Docker for reproducible research”, *ACM SIGOPS Operating Systems Review*, Jan. 2015, doi: 10.1145/2723872.2723882.
- [99] E. Casalicchio and V. Perciballi, “Measuring Docker Performance: What a Mess!!!”, in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, Apr. 2017, doi: 10.1145/3053600.3053605.
- [100] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, “The Implementation of Lua 5.0.”, *Journal of Universal Computer Science*, 2005, doi: 10.3217/jucs-011-07-1159.
- [101] “The Python Language Reference — Python 3.10.0a7 documentation”. [Online]. Available: <https://docs.python.org/dev/reference/index.html>. [Accessed: Apr. 19, 2021]

- [102] J. Harband and K. Smith, Eds., “ECMAScript® 2020 language specification”, ECMA International, Geneva, Jun. 2020 [Online]. Available: <https://www.ecma-international.org/wp-content/uploads/ECMA-262.pdf>. [Accessed: Oct. 02, 2022]
- [103] L. Bak, “V8 JavaScript engine”. Google Inc., Jan. 2022 [Online]. Available: <https://v8.dev/>. [Accessed: Jul. 15, 2022]
- [104] “Which programming language is fastest? | Computer Language Benchmarks Game”. [Online]. Available: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>. [Accessed: Apr. 19, 2021]
- [105] A. Rossberg, Ed., “WebAssembly Core Specification Version 2.0”, W3C, Apr. 2022 [Online]. Available: <https://www.w3.org/TR/wasm-core-2/>. [Accessed: Oct. 02, 2022]
- [106] A. Crichton and D. Gohman, “Wasmtime — a small and efficient runtime for WebAssembly & WASI”. [Online]. Available: <https://wasmtime.dev/>. [Accessed: Jun. 08, 2021]
- [107] “Wasmer - The Universal WebAssembly Runtime”. [Online]. Available: <https://wasmer.io/>. [Accessed: Jun. 08, 2021]
- [108] P. Hickey *et al.*, “WASI: WebAssembly System Interface”, Dec. 2020, doi: 10.5281/zenodo.4323447.
- [109] “Wasmer_middlewares::metering - Rust”. [Online]. Available: https://docs.rs/wasmer-middlewares/2.0.0-rc1/wasmer_middlewares/metering/index.html. [Accessed: Jun. 08, 2021]
- [110] “WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing”. [Online]. Available: <https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html>. [Accessed: Jul. 28, 2020]
- [111] R. Hagelund, S. A. Levin, and SEG Technical Standards Committee, Eds., *SEG-Y revision 2.0 Data Exchange format*. Society of Exploration Geophysicists, 2017 [Online]. Available: https://seg.org/Portals/0/SEG/News%20and%20Resources/Technical%20Standards/seg_y_rev2_0-mar2017.pdf. [Accessed: Oct. 02, 2022]
- [112] M. Ruth *et al.*, “Geotiff format specification geotiff revision 1.0”, vol. 1, Dec. 2000 [Online]. Available: <http://geotiff.maptools.org/spec/geotiffhome.html>. [Accessed: Oct. 02, 2022]
- [113] M. Folk, A. Cheng, and K. Yates, “HDF5: A file format and I/O library for high performance computing applications”, in *Proceedings of supercomputing*, Nov. 1999.
- [114] Kitware, Ed., “VTK File Formats Specification”. 2021 [Online]. Available: <https://kitware.github.io/vtk-examples/site/VTKFileFormats/>. [Accessed: Jul. 02, 2021]
- [115] ESRI, Ed., “ESRI Shapefile Technical Description”, p. 34, Jul. 1998 [Online]. Available: <https://www.esri.com/Library/Whitepapers/Pdfs/Shapefile.pdf>. [Accessed: Oct. 02, 2022]
- [116] ECMA, *ECMA-376: Office Open XML file formats*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 2016 [Online]. Available: <http://www.ecma-international.org/publications/standards/Ecma-376.htm>. [Accessed: Oct. 02, 2022]
- [117] “Disclosing all Volve data - Disclosing all Volve data - equinor.com”. [Online]. Available: <https://www.equinor.com/en/news/14jun2018-disclosing-volve-data.html>. [Accessed: Feb. 04, 2021]

- [118] ISO Central Secretary, Ed., “Geographic information — Metadata”, International Organization for Standardization, Geneva, CH, ISO 19115-1, 2014 [Online]. Available: <https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/37/53798.html>. [Accessed: Jul. 09, 2021]
- [119] ISO Central Secretary, Ed., “Geographic information — Services”, International Organization for Standardization, Geneva, CH, Standard ISO 19119:2016, 2016 [Online]. Available: <https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/92/59221.html>. [Accessed: Jul. 09, 2021]
- [120] European Commission, Ed., “Technical Guidance for the implementation of INSPIRE dataset and service metadata based on ISO/TS 19139:2007”, Mar. 2017 [Online]. Available: <https://inspire.ec.europa.eu/id/document/tg/metadata-iso19139>. [Accessed: Jul. 09, 2021]
- [121] European Commission, Ed., “DCAT application profile for data portals in Europe. Version 1.1”, European Commission, 2015 [Online]. Available: <https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/dcat-application-profile-data-portals-europe/release/11>. [Accessed: Jul. 09, 2021]
- [122] European Commission, Ed., “GeoDCAT-AP: A geospatial extension for the DCAT application profile for data portals in Europe”, European Commission, Standard, 2016 [Online]. Available: <https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/geodcat-application-profile-data-portals-europe/release/101>. [Accessed: Jul. 09, 2021]
- [123] Arizona Geological Survey, Ed., “USGIN Metadata Profile: Use of ISO metadata specifications to describe geoscience information resources”, USGIN Standards and Protocols Drafting Team, Standard 1.3, Jun. 2018 [Online]. Available: https://usgin.github.io/usginspecs/USGIN_ISO_Metadata.htm. [Accessed: Jul. 09, 2021]
- [124] “Use the OverlayFS storage driver”, *Docker Documentation*. Feb. 2022 [Online]. Available: <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>. [Accessed: Feb. 11, 2022]
- [125] “ObjectIdentifier - Amazon Simple Storage Service”. 2020 [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/API_ObjectIdentifier.html. [Accessed: Jun. 22, 2021]
- [126] D. Luu, “Files are fraught with peril”. 2019 [Online]. Available: <https://danluu.com/deconstruct-files/>. [Accessed: Aug. 04, 2020]
- [127] ISO Central Secretary, Ed., “Information technology database languages — SQL”, International Organization for Standardization, Geneva, CH, ISO/IEC 9075-15:2019, Jun. 2019 [Online]. Available: <https://www.iso.org/standard/67382.html>. [Accessed: Oct. 02, 2022]
- [128] M. Stonebraker and G. Kemnitz, “The POSTGRES next generation database management system”, *Communications of the ACM*, Oct. 1991, doi: 10.1145/125223.125262.
- [129] “8.14. JSON Types”, *PostgreSQL Documentation*. May 2021 [Online]. Available: <https://www.postgresql.org/docs/13/datatype-json.html>. [Accessed: Jul. 13, 2021]
- [130] “F.16. hstore”, *PostgreSQL Documentation*. Feb. 2022 [Online]. Available: <https://www.postgresql.org/docs/14/hstore.html>. [Accessed: Feb. 24, 2022]
- [131] “MongoDB”. 2021 [Online]. Available: <https://www.mongodb.com>. [Accessed: Jul. 13, 2021]
- [132] “Documents”, *MongoDB Manual*. [Online]. Available: <https://docs.mongodb.com/manual/core/document/>. [Accessed: Feb. 24, 2022]

- [133] K. Kingsbury, “Jepsen: MongoDB 4.2.6”, May 2020 [Online]. Available: <https://jepsen.io/analyses/mongodb-4.2.6>. [Accessed: Aug. 17, 2020]
- [134] K. Patella, “Jepsen: MongoDB 3.6.4”, Oct. 2018 [Online]. Available: <https://jepsen.io/analyses/mongodb-3-6-4>. [Accessed: Jul. 13, 2021]
- [135] K. Kingsbury, “Jepsen: MongoDB 3.4.0-Rc3”, Feb. 2017 [Online]. Available: <http://jepsen.io/analyses/mongodb-3-4-0-rc3>. [Accessed: Jul. 13, 2021]
- [136] K. Kingsbury, “Jepsen: MongoDB stale reads”, Apr. 2015 [Online]. Available: <https://aphyr.com/posts/322-jepsen-mongodb-stale-reads>. [Accessed: Jul. 13, 2021]
- [137] “Neo4j”. 2021 [Online]. Available: <https://neo4j.com/>. [Accessed: Jul. 14, 2021]
- [138] “The Neo4j Cypher Manual v4.3”, *Neo4j Cypher Manual*. 2021 [Online]. Available: <https://neo4j.com/docs/cypher-manual/4.3/>. [Accessed: Jul. 14, 2021]
- [139] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data (English Edition)*. O’Reilly Media, 2015.
- [140] “GitLab Container Registry | GitLab”. [Online]. Available: https://docs.gitlab.com/ee/user/packages/container_registry/. [Accessed: Jul. 15, 2021]
- [141] “Docker Hub registry”, *Docker Documentation*. Jul. 2021 [Online]. Available: <https://docs.docker.com/registry/deploying/>. [Accessed: Jul. 15, 2021]
- [142] D. R. K. Ports and K. Grittnner, “Serializable snapshot isolation in PostgreSQL”, *Proceedings of the VLDB Endowment*, Aug. 2012, doi: 10.14778/2367502.2367523.
- [143] D. Luu, “Filesystem error handling”. [Online]. Available: <https://danluu.com/filesystem-errors/>. [Accessed: Aug. 04, 2020]
- [144] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, “The zettabyte file system”, *Proceedings of the USENIX Conference on File and Storage Technologies*, 2003 [Online]. Available: <https://www.cs.hmc.edu/~rhodes/courses/cs134/fa20/readings/The%20Zettabyte%20File%20System.pdf>. [Accessed: Oct. 02, 2022]
- [145] H. H. Le, “Spatio-temporal Information System for the Geosciences: Concepts, Data models, Software, and Applications”, PhD thesis, Technische Universität Bergakademie Freiberg, 2014.
- [146] G. Hoare *et al.*, “The Rust Programming Language”. The Rust Programming Language, Nov. 2021 [Online]. Available: <https://github.com/rust-lang/rust>. [Accessed: Oct. 07, 2022]
- [147] I. Hickson, S. Pieters, A. van Kesteren, P. Jägenstedt, D. Denicola, and T. Berners-Lee, Eds., “HTML Living Standard”, WHATWG, 7.10.22 [Online]. Available: <https://html.spec.whatwg.org/>. [Accessed: Oct. 07, 2022]
- [148] R. Hickey, D. Nolen, M. Fikes, A. N. Monteiro, and B. Ashworth, “Clojure/clojurescript”. Clojure, Dec. 2021 [Online]. Available: <https://github.com/clojure/clojurescript>. [Accessed: Dec. 06, 2021]
- [149] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format”, Dec. 2017, doi: 10.17487/RFC8259.
- [150] O. Ben-Kiki, C. Evans, and I. dot Net, “YAML ain’t markup language (YAML) (tm) version 1.2”, YAML.org, 1.10.21 [Online]. Available: <https://yaml.org/spec/1.2.2/>. [Accessed: Oct. 07, 2022]
- [151] K. Trzciński, T. Maczugin, S. Azzopardi, P. Pombeiro, A. Caiazza, and G. N. Georgiev, “Gitlab runner”. 2021 [Online]. Available: <https://gitlab.com/gitlab-org/gitlab-runner>. [Accessed: Dec. 09, 2021]

- [152] S. Winter, “Seismische Tomographie am BHMZ Versuchsstand des Lehr- und Forschungsbergwerks ”Reiche Zeche””, {MSc Thesis}, Technische Univesität Bergakademie Freiberg, Freiberg, 2016.
- [153] D. Pötschke, “Geoelektrische Tomographie an einem Erzgang im Bergwerk Reiche Zeche (Freiberg)”, {MSc Thesis}, Technische Univesität Bergakademie Freiberg, 2017.
- [154] C. Moler, “MATLAB”. The MathWorks Inc., Natick, Massachusetts, 15.9.22 [Online]. Available: <https://www.mathworks.com/products/matlab.html>. [Accessed: Oct. 07, 2022]
- [155] M. Scheunert, L. Bräunig, G. Semmler, R. Gootjes, and J. Blechta, “Abschlussbericht der Nachwuchsforschergruppe GEOSax”, Apr. 2021.
- [156] L. Roscoe, “Stereolithography interface specification”, America-3D Systems Inc.
- [157] Landesamt für Umwelt, Landwirtschaft und Geologie Sachsen, Ed., “Subsurface Model for the road bypass near Pirna/Saxony”. 2021 [Online]. Available: <https://www.umwelt.sachsen.de/umwelt/infosysteme/gst3/webgui/gui2.php?viewHash=08766d8226186d1fc1db6d5689edc2c9&filter=true>. [Accessed: Nov. 15, 2021]
- [158] QGIS Development Team, “QGIS geographic information system”. QGIS, 15.07.22 [Online]. Available: <https://github.com/qgis/QGIS>. [Accessed: Aug. 19, 2022]
- [159] “dBASE .DBF File Structure”. [Online]. Available: http://www.dbase.com/Knowledgebase/INT/db7_file_fmt.htm. [Accessed: Nov. 25, 2022]
- [160] A. Bräunig, “Bodenkarte 1 : 50.000”. Sächsisches Landesamt für Umwelt, Landwirtschaft und Geologie, 2021 [Online]. Available: <http://www.boden.sachsen.de/digitale-bodenkarte-1-50-000-19474.html>. [Accessed: Nov. 19, 2021]
- [161] Earth Resources Observation And Science (EROS) Center and U.S. Geological Survey, Eds., “Shuttle radar topography mission (SRTM) 1 arc-second global”, 2017, doi: 10.5066/f7pr7tft.
- [162] H. Sponagel *et al.*, *Bodenkundliche Kartieranleitung*, 5th edition. Hannover: Bundesanstalt für Geowissenschaften und Rohstoffe, 2005.
- [163] C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities”, *International Journal for Numerical Methods in Engineering*, Sep. 2009, doi: 10.1002/nme.2579.
- [164] “XDMF Model and Format - XdmfWeb”. 2021 [Online]. Available: https://xdmf.org/index.php/XDMF_Model_and_Format. [Accessed: Nov. 12, 2021]
- [165] D. Temirkhodjaev, “Xlsx2csv”. Nov. 2021 [Online]. Available: <https://github.com/dilshod/xlsx2csv>. [Accessed: Nov. 16, 2021]
- [166] “ESRI Arc Geodatabase (file-based)”. Jun. 2020 [Online]. Available: <https://www.loc.gov/preservation/digital/formats/fdd/fdd000294.shtml#useful>. [Accessed: Nov. 16, 2021]

Appendix

A. Construction Steps for the BHMZ model

A.1. stacking

The stacking operation transforms two measurement files, named `RZ01.dat` and `RZ02.dat` provided in an application specific matlab matrix dat format into a combined stacked matlab matrix dat file. Listing A.1 contains the construction step definition as used by the case study presented in section 7.2. It is based on use case specific matlab code provided by the geoelectric working group of the department of geophysics at TU Bergakademie Freiberg. The script performed as part of this construction step consists the following operations:

1. Create a required input directory (Line 5)
2. Move the input files `RZ01.dat` and `RZ02.dat` into the created directory, so that the stacking matlab script can correctly read the corresponding data. (Line 6-7)
3. Execute the actual matlab stacking script. This script is provided by the software environment, that is based on the code provided by the geoelectric working group. (Line 8)

```

1 image: local-registry:5000/bhmz_dc:latest
2 input_path: "/home/matlab/"
3 operation:
4   - command: |
5       mkdir bhmz/data_BERT
6       mv RZ01.dat bhmz/data_BERT/RZ01.dat
7       mv RZ02.dat bhmz/data_BERT/RZ02.dat
8       cd bhmz && matlab -nosplash -nodesktop -r "drive_stack_data"
9   displayName: Stack data
10 output:
11   - file: /home/matlab/bhmz/RZstack.dat

```

Listing A.1.: Definition of the stacking construction step

A.2. meshing

The meshing operation creates a tetrahedral mesh, which is later used for the inversion of the geoelectric measurement results. The inversion mesh is based on the mesh describing the tunnel and the measurement data, that contain number and location of the measurement points. This construction step takes a tunnel mesh description as `tunnel_mesh.stl` file in the common geometry format STL [156] and the stacked measurement data as `RZstack.dat` as Matlab matrix dat file as input. It produces a tetrahedral mesh based on this input as GMSH [163] geometry. This file format consists of two related files. Listing A.2 contains the

construction step definition as used by the case study presented in section 7.2. The script performed as part of this construction step consists of the following operations:

1. Create a required input directory (Line 5)
2. Move the tunnel mesh file to the correct location, so that the corresponding matlab script can access the data (Line 6)
3. Create another required input directory (Line 7)
4. Move the stacked measurement data to the correct location, so that the corresponding matlab script can access the data (Line 8)
5. Run the meshing operation by executing the corresponding matlab script. This script is provided by the software environment, that is based on the code provided by the geoelectric working group (Line 11)
6. Move the created tetrahedral mesh to the output location. (Line 12-23)

```

1  image: local-registry:5000/bhmz_dc:latest
2  input_path: "/home/matlab"
3  operation:
4    - command: |
5        mkdir bhmz/mesh
6        mv tunnel_mesh.stl bhmz/mesh/
7        mkdir bhmz/data_BERT
8        mv RZstack.dat bhmz/data_BERT/
9    displayName: Copy data
10   - command: |
11       cd bhmz && matlab -nosplash -nodesktop -r "create_mesh"
12       mv /home/matlab/gy-apps/+meshing/vol_in_fs.geo /home/matlab/bhmz/mesh.geo
13       mv /home/matlab/gy-apps/+meshing/vol_in_fs.msh /home/matlab/bhmz/mesh.msh
14  output:
15    - file: /home/matlab/bhmz/mesh.geo
16    - file: /home/matlab/bhmz/mesh.msh

```

Listing A.2.: Definition of the meshing construction step

A.3. DC inversion

The DC inversion operation solves a inverse problem based on a tetrahedral mesh used for the numeric computation and a stacked measurement file used to constrain the problem. This construction step takes a tetrahedral mesh `mesh.geo/mesh.msh` in GMSH format and the stacked measurements `RZstack.dat` in Matlab's matrix dat format. It produces a complex dataset in the XDMF format [164] containing the inversion results for different sensitivity values. This file format is designed as output for numerical computations and can represent different kinds of geometries. Listing A.3 contains the construction step definition as used by the case study presented in section 7.2. The script performed as part of this construction step consists of the following operations:

1. Create a required input directory (Line 5)
2. Move the stacked measurement data to the right location, so that the matlab script can access the data later on (Line 6)
3. Create another required input directory (Line 7)
4. Move the tetrahedral inversion mesh to the right location, so that the matlab script can access the mesh later on (Line 8)

5. Increase the limit for the maximal number of open files to prevent a crash (Line 11)
6. Run the inversion by executing a matlab script provided by the environment. This script is provided by the geoelectric working group (Line 12)

```

1 image: local-registry:5000/bhmz_dc:latest
2 input_path: "/home/matlab/"
3 operation:
4   - command: |
5       mkdir bhmz/data_BERT
6       mv RZstack.dat bhmz/data_BERT/RZstack.dat
7       mkdir bhmz/mesh
8       mv mesh.msh bhmz/mesh/mesh.msh
9       displayName: Copy data
10  - command: |
11      ulimit -n 10000
12      cd bhmz && matlab -nosplash -nodesktop -r "drive_bhmz"
13      displayName: Run inversion
14 output:
15   - file: /home/matlab/bhmz/bhmz_3D.xdmf
16   - file: /home/matlab/bhmz/bhmz_3D_1.vtu
17   - file: /home/matlab/bhmz/bhmz_3D_2.vtu
18   - file: /home/matlab/bhmz/bhmz_3D_3.vtu
19   - file: /home/matlab/bhmz/bhmz_3D_4.vtu
20   - file: /home/matlab/bhmz/bhmz_3D_5.vtu
21   - file: /home/matlab/bhmz/bhmz_3D_6.vtu
22   - file: /home/matlab/bhmz/bhmz_3D_7.vtu
23   - file: /home/matlab/bhmz/bhmz_3D_8.vtu
24   - file: /home/matlab/bhmz/bhmz_3D_9.vtu
25   - file: /home/matlab/bhmz/bhmz_3D_10.vtu
26   - file: /home/matlab/bhmz/bhmz_3D_11.vtu
27   - file: /home/matlab/bhmz/bhmz_3D_12.vtu
28   - file: /home/matlab/bhmz/bhmz_3D_13.vtu
29   - file: /home/matlab/bhmz/bhmz_3D_14.vtu

```

Listing A.3.: Definition of the DC inversion construction step

A.4. seismic inversion

The seismic inversion operation solves a inverse problem based on a regular grid and observed seismic travel times through the rock formation. The inversion is performed by a python translation of a matlab script provided by the seismic working group at the department of geophysics at TU Bergakademie Freiberg. This construction step takes a text file `picks.txt` containing the observed travel times between different measurement locations as input. It produces multiple grid geometries stored in Paraview's VTK format [114] as result. Listing A.4 contains the construction step definition as used by the case study presented in section 7.2. The script performed as part of this construction step consists of the following operations:

1. Perform the seismic inversion by calling the python script provided by the software environment. This script is based on a matlab script provided by the seismic working group at the department of geophysics at TU Bergakademie Freiberg. (Line 5)

```

1 image: local-registry:5000/geosax-bhmz-seismic-python
2 input_path: "/"
3 operation:
4   - command: |
5       python Seismic\ Tomographie.py --picks /picks.txt
6       displayName: Run the inversion
7 output:
8   - file: /image_1000000.0.vti
9   - file: /image_439397.0560760795.vti
10  - file: /image_193069.77288832495.vti
11  - file: /image_84834.28982440726.vti
12  - file: /image_37275.93720314938.vti
13  - file: /image_16378.937069540647.vti
14  - file: /image_7196.856730011521.vti
15  - file: /image_3162.2776601683795.vti
16  - file: /image_1389.4954943731375.vti
17  - file: /image_610.5402296585327.vti
18  - file: /image_268.2695795279727.vti
19  - file: /image_117.87686347935866.vti
20  - file: /image_51.794746792312125.vti
21  - file: /image_22.758459260747887.vti
22  - file: /image_10.0.vti

```

Listing A.4.: Definition of the seismic inversion step

A.5. visualise BHMZ

The visualise BHMZ operation is the final construction step of the BHMZ model workflow. It combines the output produced by the seismic inversion and the DC inversion construction step into a unified visualisation. This is done by using a Paraview functionality, which allows to describe a visualisation as python script. This construction step creates the corresponding python script based on the provided inputs. This was developed as part the implementation of the case study presented in section 7.2. This construction step takes a XDMF geometry (`bhmz_3D.xdmf`) containing the geoelectric inversion result, a number of VTK grid files (`image_*.vti`) and the tunnel geometry (`tunnel_mesh.stl`) as STL input. It produces a Paraview visualisation script, which allows to present a unified visualisation in Paraview using the provided input files. Listing A.5 contains the construction step definition as used by the case study presented in section 7.2. The script performed as part of this construction step consists of the following operations:

1. Definition of the visualisation script. This script references all input files assuming they are located inside of the same directory as the script. Additionally it sets up the Paraview view port in such a way, that it shows the results at the right location. As of this it needs to apply different transformations to different parts of the input geometry. (Line 6-233)

```

1 image: alpine:latest
2 input_path: "/"
3 operation:
4   - command: |
5       cat << EOF > /bhmz_visualisation.py
6       # state file generated using paraview version 5.7.0
7
8       # -----
9       # setup views used in the visualization
10      # -----
11
12      # trace generated using paraview version 5.7.0
13      #
14      # To ensure correct image size when batch processing, please search
15      # for and uncomment the line
16
17      ##### import the simple module from the paraview
18      from paraview.simple import *
19      import os
20
21      script_path = os.path.dirname(os.path.abspath(__file__))
22      ##### disable automatic camera reset on 'Show'
23      paraview.simple._DisableFirstRenderCameraReset()
24
25      # Create a new 'Line Chart View'
26      lineChartView1 = CreateView('XYChartView')
27      lineChartView1.ViewSize = [1601, 1873]
28      lineChartView1.LegendPosition = [1506, 1814]
29      lineChartView1.LeftAxisTitle = 'v (m/s)'
30      lineChartView1.RightAxisTitle = '$\\rho$ ($\\Omega$ m)'
31      lineChartView1.LeftAxisRangeMinimum = 5340.0
32      lineChartView1.LeftAxisRangeMaximum = 5880.0
33      lineChartView1.BottomAxisTitle = 'x (m)'
34      lineChartView1.BottomAxisRangeMaximum = 38.0
35      lineChartView1.RightAxisRangeMaximum = 0.011600000000000001
36      lineChartView1.TopAxisRangeMaximum = 6.66
37
38
39      # Create a new 'Render View'
40      renderView1 = CreateView("RenderView")
41      renderView1.CenterOfRotation = [
42          18.264817256598004,
43          7.145154959697277,
44          2.6588124866578147,
45      ]
46      renderView1.StereoType = "Crystal Eyes"
47      renderView1.CameraPosition = [
48          -7.686134444102471,
49          12.29685262218834,

```

```

50         86.37064174318
51     ]
52     renderView1.CameraFocalPoint = [
53         18.264817256597993,
54         7.145154959697269,
55         2.6588124866578324,
56     ]
57     renderView1.CameraViewUp = [
58         0.8702899506852618,
59         0.4281717056944392,
60         0.24344279036143018
61     ]
62     renderView1.CameraFocalDisk = 1.0
63     renderView1.CameraParallelScale = 22.72257617910356
64     renderView1.Background = [0.32, 0.34, 0.43]
65
66
67     # -----
68     # setup view layouts
69     # -----
70
71     # create new layout object 'Layout #1'
72     layout1 = CreateLayout(name='Layout #1')
73     layout1.SplitHorizontal(0, 0.500000)
74     layout1.AssignView(1, renderView1)
75     layout1.AssignView(2, lineChartView1)
76
77     # -----
78     # restore active view
79     SetActiveView(renderView1)
80     # -----
81
82     # -----
83     # setup the data processing pipelines
84     # -----
85
86     # create a new 'STL Reader'
87     tunnel_meshvtk = STLReader(
88         FileNames=["%s/tunnel_mesh.stl" % script_path],
89         guiName="Tunnel"
90     )
91
92     # create a new 'XML Image Data Reader'
93     image_ = XMLImageDataReader(
94         FileName=[
95             "%s/image_1000000.0.vti" % script_path,
96             "%s/image_439397.0560760795.vti" % script_path,
97             "%s/image_193069.77288832495.vti" % script_path,
98             "%s/image_84834.28982440726.vti" % script_path,
99             "%s/image_37275.93720314938.vti" % script_path,
100            "%s/image_7196.856730011521.vti" % script_path,

```



```

101         "%s/image_16378.937069540647.vti" % script_path,
102         "%s/image_3162.2776601683795.vti" % script_path,
103         "%s/image_1389.4954943731375.vti" % script_path,
104         "%s/image_610.5402296585327.vti" % script_path,
105         "%s/image_268.2695795279727.vti" % script_path,
106         "%s/image_117.87686347935866.vti" % script_path,
107         "%s/image_51.794746792312125.vti" % script_path,
108         "%s/image_22.758459260747887.vti" % script_path,
109         "%s/image_10.0.vti" % script_path,
110     ],
111     guiName="Seismic inversion result",
112 )
113
114 transform1 = Transform(
115     Input=image_,
116     guiName="Rotated seismic inversion result"
117 )
118 transform1.Transform = "Transform"
119
120 # init the 'Transform' selected for 'Transform'
121 transform1.Transform.Translate = [0.0, 0.0, 3.5]
122 transform1.Transform.Rotate = [-15.0, 0.0, 0.0]
123
124
125 image_.CellArrayStatus = ["v"]
126 image_ = transform1
127
128 # create a new 'Xdmf3ReaderS'
129 bhmz_3D1xdmf = Xdmf3ReaderS(
130     FileName=["%s/bhmz_3D.xdmf" % script_path],
131     guiName="DC inversion result",
132 )
133
134 # create a new 'Extract Cells By Region'
135 extractCellsByRegion1 = ExtractCellsByRegion(
136     Input=bhmz_3D1xdmf,
137     guiName="Clipped DC inversion result"
138 )
139 extractCellsByRegion1.IntersectWith = "Plane"
140
141 # init the 'Plane' selected for 'IntersectWith'
142 extractCellsByRegion1.IntersectWith.Origin = [
143     340.166259765625,
144     -32.744140625,
145     2.65869140625,
146 ]
147 extractCellsByRegion1.IntersectWith.Normal = [0.0, 0.0, 1.0]
148
149 # create a new 'Plot Over Line'
150 plotOverLine1 = PlotOverLine(
151     registrationName='PlotOverLine1',

```

```

152         Input=transform1,
153         Source='Line'
154     )
155
156     # init the 'Line' selected for 'Source'
157     plotOverLine1.Source.Point1 = [0.0, 4.8, 2.3]
158     plotOverLine1.Source.Point2 = [36.0, 7.6, 2.3]
159
160     # create a new 'Plot Over Line'
161     plotOverLine2 = PlotOverLine(
162         registrationName='PlotOverLine2',
163         Input=bhmz_3D1xdmf,
164         Source='Line'
165     )
166
167     # init the 'Line' selected for 'Source'
168     plotOverLine2.Source.Point1 = [0.0, 4.8, 2.3]
169     plotOverLine2.Source.Point2 = [36.0, 7.6, 2.3]
170
171     # -----
172     # setup the visualization in view 'lineChartView1'
173     # -----
174
175     # show data from plotOverLine1
176     plotOverLine1Display = Show(
177         plotOverLine1,
178         lineChartView1,
179         'XYChartRepresentation'
180     )
181
182     # trace defaults for the display properties.
183     plotOverLine1Display.CompositeDataSetIndex = [0]
184     plotOverLine1Display.XArrayName = 'arc_length'
185     plotOverLine1Display.SeriesVisibility = ['v']
186     plotOverLine1Display.SeriesLabel = [
187         'arc_length', 'arc_length', 'v', 'v',
188         'vtkValidPointMask', 'vtkValidPointMask', 'Points_X',
189         'Points_X', 'Points_Y', 'Points_Y', 'Points_Z',
190         'Points_Z', 'Points_Magnitude', 'Points_Magnitude'
191     ]
192     plotOverLine1Display.SeriesColor = [
193         'arc_length', '0', '0', '0',
194         'v', '0.89', '0.10', '0.11',
195         'vtkValidPointMask', '0.22', '0.49', '0.72',
196         'Points_X', '0.30', '0.69', '0.29',
197         'Points_Y', '0.6', '0.31', '0.64',
198         'Points_Z', '1', '0.50', '0',
199         'Points_Magnitude', '0.65', '0.34', '0.16'
200     ]
201     plotOverLine1Display.SeriesPlotCorner = [
202         'Points_Magnitude', '0',

```

```

203         'Points_X', '0',
204         'Points_Y', '0',
205         'Points_Z', '0',
206         'arc_length', '0',
207         'v', '0',
208         'vtkValidPointMask', '0'
209     ]
210
211     # show data from plotOverLine2
212     plotOverLine2Display = Show(
213         plotOverLine2,
214         lineChartView1,
215         'XYChartRepresentation'
216     )
217
218     # trace defaults for the display properties.
219     plotOverLine2Display.CompositeDataSetIndex = [0]
220     plotOverLine2Display.XArrayName = 'arc_length'
221     plotOverLine2Display.SeriesVisibility = ['u']
222     plotOverLine2Display.SeriesLabel = [
223         'arc_length', 'arc_length', 'u', '$\\rho$',
224         'vtkValidPointMask', 'vtkValidPointMask', 'Points_X',
225         'Points_X', 'Points_Y', 'Points_Y', 'Points_Z',
226         'Points_Z', 'Points_Magnitude', 'Points_Magnitude'
227     ]
228     plotOverLine2Display.SeriesColor = [
229         'arc_length', '0', '0', '0',
230         'u', '0.3333333333333333', '0', '1',
231         'vtkValidPointMask', '0.22', '0.49', '0.72',
232         'Points_X', '0.30', '0.69', '0.29',
233         'Points_Y', '0.6', '0.31', '0.64',
234         'Points_Z', '1', '0.50', '0',
235         'Points_Magnitude', '0.65', '0.34', '0.16'
236     ]
237     plotOverLine2Display.SeriesPlotCorner = [
238         'Points_Magnitude', '0',
239         'Points_X', '0',
240         'Points_Y', '0',
241         'Points_Z', '0',
242         'arc_length', '0',
243         'u', '1',
244         'vtkValidPointMask', '0'
245     ]
246
247
248     # -----
249     # setup the visualization in view 'renderView1'
250     # -----
251
252     # show data from tunnel_meshvtk
253     tunnel_meshvtkDisplay = Show(tunnel_meshvtk, renderView1)

```

```
254
255     # trace defaults for the display properties.
256     tunnel_meshvtkDisplay.Representation = "Surface"
257
258     # show data from extractCellsByRegion1
259     extractCellsByRegion1Display = Show(extractCellsByRegion1, renderView1)
260
261     # get color transfer function/color map for 'u'
262     uLUT = GetColorTransferFunction("u")
263     uLUT.RGBPoints = [
264         5.659940960356391e-09,
265         0.231373,
266         0.298039,
267         0.752941,
268         0.0003293811915883703,
269         0.865003,
270         0.865003,
271         0.865003,
272         12.95422935485838,
273         0.705882,
274         0.0156863,
275         0.14902,
276     ]
277     uLUT.UseLogScale = 1
278     uLUT.ColorSpace = 'RGB'
279     uLUT.ScalarRangeInitialized = 1.0
280
281     # get opacity transfer function/opacity map for 'u'
282     uPWF = GetOpacityTransferFunction("u")
283     uPWF.Points = [
284         5.65994096035638e-09,
285         0.0,
286         0.5,
287         0.0,
288         12.954229354858398,
289         1.0,
290         0.5,
291         0.0
292     ]
293     uPWF.ScalarRangeInitialized = 1
294
295     # trace defaults for the display properties.
296     extractCellsByRegion1Display.Representation = "Surface"
297     extractCellsByRegion1Display.ColorArrayName = ["CELLS", "u"]
298     extractCellsByRegion1Display.LookupTable = uLUT
299     extractCellsByRegion1Display.SelectScaleArray = "u"
300
301
302     # show data from image_
303     image_Display = Show(image_, renderView1)
304     image_Display.Opacity = 0.7
```

```
305
306     # get color transfer function/color map for 'v'
307     vLUT = GetColorTransferFunction("v")
308     vLUTAutomaticRescaleRangeMode = "Never"
309     vLUT.RGBPoints = [
310         3645.8730641588027,
311         0.231373,
312         0.298039,
313         0.752941,
314         5322.936532079401,
315         0.865003,
316         0.865003,
317         0.865003,
318         7000.0,
319         0.705882,
320         0.0156863,
321         0.14902,
322     ]
323     vLUT.ScalarRangeInitialized = 1.0
324
325     # get opacity transfer function/opacity map for 'v'
326     vPWF = GetOpacityTransferFunction("v")
327     vPWF.Points = [3645.8730641588027, 0.0, 0.5, 0.0, 7000.0, 1.0, 0.5, 0.0]
328     vPWF.ScalarRangeInitialized = 1
329
330     # trace defaults for the display properties.
331     image_Display.Representation = "Surface"
332     image_Display.ColorArrayName = ["CELLS", "v"]
333     image_Display.LookupTable = vLUT
334     image_Display.ScalarOpacityFunction = vPWF
335
336     # setup the color legend parameters for each legend in this view
337
338     # get color legend/bar for uLUT in view renderView1
339     uLUTColorBar = GetScalarBar(uLUT, renderView1)
340     uLUTColorBar.Title = "$\\rho$"
341     uLUTColorBar.ComponentTitle = "($\\Omega$ m)"
342     uLUTColorBar.HorizontalTitle = 1
343
344     # set color bar visibility
345     uLUTColorBar.Visibility = 1
346
347     # get color legend/bar for vLUT in view renderView1
348     vLUTColorBar = GetScalarBar(vLUT, renderView1)
349     vLUTColorBar.WindowLocation = "UpperRightCorner"
350     vLUTColorBar.Title = "v"
351     vLUTColorBar.ComponentTitle = "(m/s)"
352     vLUTColorBar.HorizontalTitle = 1
353
354     # set color bar visibility
355     vLUTColorBar.Visibility = 1
```

```

356
357     # show color legend
358     extractCellsByRegion1Display.SetScalarBarVisibility(renderView1, True)
359
360     # show color legend
361     image_Display.SetScalarBarVisibility(renderView1, True)
362
363     # -----
364     # finally, restore active source
365     SetActiveSource(plotOverLine1)
366     # -----
367     EOF
368     displayName: Create a visualisation file
369 output:
370 - file: /image_1000000.0.vti
371 - file: /image_439397.0560760795.vti
372 - file: /image_193069.77288832495.vti
373 - file: /image_84834.28982440726.vti
374 - file: /image_37275.93720314938.vti
375 - file: /image_16378.937069540647.vti
376 - file: /image_7196.856730011521.vti
377 - file: /image_3162.2776601683795.vti
378 - file: /image_1389.4954943731375.vti
379 - file: /image_610.5402296585327.vti
380 - file: /image_268.2695795279727.vti
381 - file: /image_117.87686347935866.vti
382 - file: /image_51.794746792312125.vti
383 - file: /image_22.758459260747887.vti
384 - file: /image_10.0.vti
385 - file: /bhmz_3D.xdmf
386 - file: /bhmz_visualisation.py
387 - file: /tunnel_mesh.stl

```

Listing A.5.: Definition of the visualise BHMZ construction step

B. Provided manual for the Kohlberg dataset

B.1. 3D-Modell Kohlberg

B.1.1. Eingangsdaten

- Bohrungsdaten in DHDN3_GK5 EPSG 31469: Stammdaten, Schichtdaten
- Kartendaten in ETRS1989 UTM33N EPSG 25833: aus Blatt Pirna: L5148 Quartär, L5148 Präquartär
- DGM in ETRS1989 UTM33N EPSG 25833:
- Profilschnitt: Längsschnitt aus Bodengutachten Baugrund Dresden

B.1.2. Bohrungsdaten vorbereiten

B.1.2.1. Mit ArcGIS

- Koordinatentransformation der Bohrungsdaten aus GK5 in UTM 33
- Koordinatensystem des Datenrahmens auf DHDN3_GK5 EPSG 31469
- Tabellen als XY-Daten einlesen Stammdaten.xlsx, Schichtdaten.xlsx
- Tabellen-Daten exportieren -> Schichtdaten_GK5.shp, Stammdaten_GK5.shp
- Als Layer einfügen
- Datenrahmen umprojizieren auf ETRS1989 UTM33N EPSG 25833
- Daten exportieren, KS des Datenrahmens verwenden! -> Schichtdaten_UTM33.shp, Stammdaten_UTM33.shp
- Als Layer einfügen ->ja
- ASCII-Export der transformierten Daten
- ArcToolbox -> Spatial Statistic Tools -> Dienstprogramme ->Feature Attribut nach ASCII exportieren->Alle auswählen -> Trennzeichen Semikolon -> Feldnamen hinzufügen -> Schichtdaten_UTM33.txt; Stammdaten_UTM33.txt

B.1.2.2. Mit EXCEL

- Evtl. IDNr vergeben (für doppelte Bohrungsnummern)
- Koordinaten kontrollieren (Prüfung der Stellenanzahl)
- Leerzeilen entfernen
- Als CSV oder Tab-Text speichern
- Schichtdaten_UTM33_korr.txt; Stammdaten_UTM33_korr.txt

B.1.2.3. Mit Editor

Kommas durch Punkt ersetzen Schichtdaten_UTM33_korr_punkt.txt; Stammdaten_UTM33_korr_punkt.txt

B.1.3. Gocad

- Import der Stammdaten und Schichtdaten:
- File -> Import -> Well data -> Well locations -> column-based file
- File -> Import -> Well data -> Well markers -> column-based file

B.1.4. DGM-Daten vorbereiten:

- Webbrowser
- geodaten.sachsen.de -> Höhen- und Stadtmodelle -> DGM2 -> Daten herunterladen
- Kachel 4205640 (Pirna) -> Entpacken
- Von den entpackten XYZ-Dateien folgende behalten:
- 334245642dgm2.xyz; 33426542dgm2.xyz; 334245644dgm2.xyz; 334565644dgm2.xyz

B.1.5. Gocad

- Import der Punktdaten:
- Import -> Horizon Interpretations -> PointsSets -> XYZ
 - File name: 334245642dgm2.xyz | 33426542dgm2.xyz | 334245644dgm2.xyz | 334565644dgm2.xyz
- PointsSets zusammenführen:
- PointsSet -> New -> From PointsSet
 - Name: DGM2_Kohlberg
 - PointsSet: 334245642dgm2 | 33426542dgm2 | 334245644dgm2 | 334565644dgm2
- PointsSet -> Tools -> Parts -> Merge All
- Surface berechnen:
- Surface -> New -> From Points -> Object Points
 - PointsSet: DGM2_Kohlberg
 - Name: ts_DGM2_Kohlberg
- Triangulation dezimieren:
- Surface -> Tools -> Decimate
 - Tolerance: 0.5
 - Convergence: 0.5
 - Keep borders

B.2. Kartendaten vorbereiten

B.2.1. Mit ArcGIS

- L5148 Quartär, L5148 Präquartär, Gebietsgrenze_2021.shp importieren
- Zuschneiden der Kartendaten auf das Untersuchungsgebiet:
- ArcToolbox-> Analysis Tools-> Überlagerung -> Intersect -> Quartaer_zugeschnitten, Praequartaer_zugeschnitten
- Export der zugeschnittenen Kartendaten in Shapefile:
- Toolbox -> Conversion Tools -> In Geodatabase -> Feature Class in Feature Class
 - Eingabe: L5148_Pirna.gdb/Praequartaer_zugeschnitten (L5148_Pirna.gdb/Quartaer_zugeschnitten)
 - Ausgabe: lin_Praequartaer (lin_Quartaer)

B.2.2. Gocad

- Kartendaten importieren:
- Import -> Cultural Data -> ArcView Shape File
 - lin_Praequartaer.shp
 - lin_Quartaer.shp
- Curve -> Tools -> Densify
 - Curve: lin_Praequartaer lin_Quartaer

- Maximum Length: 15
- Curve -> Tools -> Project -> Vertically on Surface
 - Source objects: lin_Praequartaer lin_Quartaer
 - Target objects: ts_DGM2_Kohlberg

B.3. Schnittdaten importieren:

B.3.1. Gocad

- Shapefile importieren
- Import -> Cultural Data -> ArcView Shape File
 - Streckenachse_UTM.shp
 - Linien_Laengsschnitt.shp
 - Gebietsgrenze.shp
- Curve -> Property -> Set constant -
 - Object: Streckenachse_UTM
 - Property: Z
 - Value: 210
- Surface -> New -> Curves and Expansion Vector (Tube)
 - Name: ts_Laengsschnitt
 - Curves: Streckenachse_UTM
 - Dir: X:0 Y:0 Z:-90
 - Advanced - Select number of levels: 1
- Curve -> New -> from Intersections -> With Surfaces
 - Name: Laengsschnitt_GOK
 - Surfaces: ts_DGM2_Kohlberg ts_Laengsschnitt
- Curve -> Tools -> Move or Rotate
 - Objects: Linien_Laengsschnitt
 - o Rotate
 - Origin: X:424987.7 Y:5643829.7 Z:0
 - Dir Axis: X:366.328 Y:-102.97 Z:0
 - Angle: 90
 - o Translate
 - Dir Translation: X:19.86 Y:68.29 Z:131
- Curve -> Tools -> Densify
 - Curve: Linien_Laengsschnitt
 - Maximum Length: 15
- Curve -> Tools -> Move or Rotate
 - Objects: Linien_Laengsschnitt ts_Laengsschnitt
 - o Rotate
 - Origin: X:425374.5 Y:5643803.7 Z:0
 - Dir Axis: X:-367.85 Y:95.20 Z:0

- Angle: -90
- Curve -> Tools ->- Project -> Vertically on Surface
 - Source objects: Linien_Laengsschnitt
 - Target objects: ts_Laengsschnitt
- Curve -> Tools -> Move or Rotate
 - Objects: Linien_Laengsschnitt ts_Laengsschnitt
 - o Rotate
 - Origin: X:425374.5 Y:5643803.7 Z:0
 - Dir Axis: X:-367.85 Y:95.20 Z:0
 - Angle: 90

C. Construction Steps for the Kohlberg model

C.1. `xlsx -> csv`

The `xlsx -> csv` operation transforms a borehole export given as Microsoft Excel files into CSV files used for later processing. It expects a `Stammdaten_GK5.xlsx` and a `Schichtdaten_GK5.xlsx` file as input and produces two CSV files (`Stammdaten_GK5.csv` and `Schichtdaten_GK5.csv`). Listing C.6 contains the construction step definition as used by the case study presented in section 7.3. The construction step uses the `xlsx2csv` [165] to perform the format conversation. The script performed as part of this construction step consists of the following operations:

1. Convert the `Stammdaten_GK5.xlsx` file into CSV format (Line 5)
2. Convert the `Schichtdaten_GK5.xlsx` file into CSV format (Line 6)

```

1  image: local-registry:5000/qgis-processing-image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        xlsx2csv -i /tmp/input/Stammdaten_GK5.xlsx /tmp/input/Stammdaten_GK5.csv
6        xlsx2csv -i /tmp/input/Schichtdaten_GK5.xlsx /tmp/input/Schichtdaten_GK5.csv
7  output:
8    - file:
9        /tmp/input/Stammdaten_GK5.csv
10   - file:
11       /tmp/input/Schichtdaten_GK5.csv

```

Listing C.6.: Definition of the `xlsx -> csv` construction step definition

C.2. `reproject to utm33`

The `reproject to utm33` operation transforms a set of borehole data given as CSV files from the Gauss-Krüger 5 (EPSG:31469) projection into the UTM33 projection (EPSG:25833). It expects two input files (`Schichtdaten_GK5.csv` and `Stammdaten_GK5.csv`) and produces two output files (`Schichtdaten_UTM33.csv` and `Stammdaten_UTM33.csv`). The coordinate system transformation is applied via the `ogr2ogr` tool provided by Q-GIS [158]. Listing C.7

contains the construction step definition as used by the case study presented in section 7.3. The script performed as part of this construction step consists of the following operations:

1. Reproject `Stammdaten_GK5.csv` from Gaus-Krüger 5 to UTM33 (Line 5 - 8)
2. Reproject `Schichtdaten_GK5.csv` from Gaus-Krüger 5 to UTM33 (Line 9 - 12)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       ogr2ogr -s_srs EPSG:31469 -t_srs EPSG:25833 -oo X_POSSIBLE_NAMES=RECHTS \
6           -oo Y_POSSIBLE_NAMES=HOCH -f "CSV" /tmp/input/Stammdaten_UTM33.csv \
7           -lco GEOMETRY=AS_XY -oo KEEP_GEOM_COLUMNS=NO \
8           /tmp/input/Stammdaten_GK5.csv
9       ogr2ogr -s_srs EPSG:31469 -t_srs EPSG:25833 -oo X_POSSIBLE_NAMES=RECHTS \
10          -oo Y_POSSIBLE_NAMES=HOCH -f "CSV" /tmp/input/Schichtdaten_UTM33.csv \
11          -lco GEOMETRY=AS_XY -oo KEEP_GEOM_COLUMNS=NO \
12          /tmp/input/Schichtdaten_GK5.csv
13 output:
14   - file:
15       /tmp/input/Stammdaten_UTM33.csv
16   - file:
17       /tmp/input/Schichtdaten_UTM33.csv

```

Listing C.7.: Definition of the reproject to utm33 construction step

C.3. extract (Quartär)

The extract (Quartär) operation extracts a specific layer from a geological map and clips it to the study area. It expects a zipped geodatabase [166] (`L5148_Pirna.gdb.zip`) and a shape file defining the extends of the study area (`Gebietsgrenze_2021.shp`) as input and produces a shape file as output (`L5148_lin_Quartaer.shp`) as output. For the data extraction the `ogr2ogr` tool provided by Q-GIS [158] is used. Listing C.8 contains the construction step definition as used by the case study presented in section 7.3. The script performed as part of this construction step consists of the following operations:

1. Unpack the zipped geodatabase (Line 5)
2. Use `ogr2ogr` to clip and extract the necessary layer provided by the geodatabase archive. (Line 6 - 7)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       unzip /tmp/input/L5148_Pirna.gdb.zip -d /tmp/input
6       ogr2ogr -f "ESRI Shapefile" -clipdst /tmp/input/Gebietsgrenze_2021.shp \
7           /tmp/input/L5148_Pirna.shp /tmp/input/L5148_Pirna.gdb
8 output:
9   - file:
10       /tmp/input/L5148_Pirna.shp/L5148_lin_Quartaer.shp

```

```

11 - file:
12     /tmp/input/L5148_Pirna.shp/L5148_lin_Quartaer.prj
13 - file:
14     /tmp/input/L5148_Pirna.shp/L5148_lin_Quartaer.dbf
15 - file:
16     /tmp/input/L5148_Pirna.shp/L5148_lin_Quartaer.shx

```

Listing C.8.: Definition of the extract (Quartär) construction step

C.4. extract (Prequartär)

The extract (Prequartär) operation extracts a specific layer from a geological map and clips it to the study area. This construction step is almost identical with the construction step described in the previous section. The main difference is that it exports a different layer from the geological map. It expects a zipped geodatabase [166] (L5148_Pirna.gdb.zip) and a shape file defining the extends of the study area (Gebietsgrenze_2021.shp) as input and produces a shape file as output (L5148_lin_Quartaer.shp) as output. For the data extraction the ogr2ogr tool provided by Q-GIS [158] is used. Listing C.9 contains the construction step definition as used by the case study presented in section 7.3. The script performed as part of this construction step consists of the following operations:

1. Unpack the zipped geodatabase (Line 5)
2. Use ogr2ogr to clip and extract the necessary layer provided by the geodatabase archive. (Line 6 - 7)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       unzip /tmp/input/L5148_Pirna.gdb.zip -d /tmp/input
6       ogr2ogr -f "ESRI Shapefile" -clipdst /tmp/input/Gebietsgrenze_2021.shp \
7         /tmp/input/L5148_Pirna.shp /tmp/input/L5148_Pirna.gdb
8 output:
9   - file:
10       /tmp/input/L5148_Pirna.shp/L5148_lin_Praequartaer.shp
11   - file:
12       /tmp/input/L5148_Pirna.shp/L5148_lin_Praequartaer.prj
13   - file:
14       /tmp/input/L5148_Pirna.shp/L5148_lin_Praequartaer.dbf
15   - file:
16       /tmp/input/L5148_Pirna.shp/L5148_lin_Praequartaer.shx

```

Listing C.9.: Definition of the extract (Quartär) construction step

C.5. triangulate

The triangulate operation triangulates a given pointset to produce a triangulate surface. This operation expects 4 xyz files (334265644_dgm2.xyz, 334265642_dgm2.xyz, 334245642_dgm2.xyz and 334245644_dgm2.xyz) and an zipped empty gOcad project as input. It produces a triangulated surface as gOcad ASCII file [42]. Listing C.10 contains the construction

step definition as used by the case study presented in section 7.3. The script performed as part of this construction step consists of the following operations:

1. Definition of the gOcad Macro to perform the actual construction step (Line 5 - 65)
2. Starting a X-Server in background (required to execute gOcad) (Line 67)
3. Wait 5 seconds for the X-Server to start (Line 68)
4. Unpack the zipped empty gOcad project for later use (Line 69)
5. Execute the gOcad macro defined earlier via gOcad's command line interface (Line 70 - 71)

The gOcad macro defined in Line 5 - 65 performs the following operations:

1. Import the digital elevation model from the provided XYZ files (334265644_dgm2.xyz, 334265642_dgm2.xyz, 334245642_dgm2.xyz and 334245644_dgm2.xyz) (Line 7 - 18)
2. Create a merged pointset from the four imported files (Line 19 - 34)
3. Create a triangulate surface from the pointset (Line 35 - 42)
4. "Decimate" (Simplify) the triangulation of the created triangulated surface (Line 43 - 49)
5. Export the triangulated surface as GoCad ASCII file (Line 50 - 55)
6. Export the merged pointset as GoCad ASCII file (Line 56 - 61)
7. "Close" GoCad at this point by sending a SIG_KILL to the corresponding process (Line 63 - 64)

```

1  image: local-registry:5000/gocad-docker-image
2  input_path: "/tmp/gocad"
3  operation:
4    - command: |
5        cat > /tmp/gocad/mesh_dgm2.js << EOF
6        var skua = PDGM.require('skua');
7        skua.run('ImportXYZFile', {
8            'File_name':[
9                "/tmp/gocad/334265644_dgm2.xyz",
10               "/tmp/gocad/334265642_dgm2.xyz",
11               "/tmp/gocad/334245644_dgm2.xyz",
12               "/tmp/gocad/334245642_dgm2.xyz"
13            ],
14            'category':"Horizons",
15            'cvn':"Domain=Default_depth",
16            'ignore_points_with_no_data_value':"false",
17            'no_data_value':-9999
18        }, { blocking:false, typed:true } );
19        skua.run('VSetCreateFromAtomicGroup', {
20            'copy_properties':"false",
21            'dissociate_vertices':"true",
22            'merge_parts':"false",
23            'name':"DGM2_Kohlberg",
24            'points':[
25                "/gobj:334245642_dgm2",
26                "/gobj:334245644_dgm2",
27                "/gobj:334265642_dgm2",
28                "/gobj:334265644_dgm2"
29            ],

```

```

30     'region':"everywhere"
31   },{ blocking:false, typed:true } );
32   skua.run('VSetMergeAllSubVSets', {
33     'on':"/gobj:DGM2_Kohlberg"
34   },{ blocking:false, typed:true } );
35   skua.run('TSurfCreateFromAtomicGroup', {
36     'copy_properties':"true",
37     'dissociate_vertices':"true",
38     'name':"ts_DGM2_Kohlberg",
39     'normal':[0.,0.,1.],
40     'points':"/gobj:DGM2_Kohlberg",
41     'use_normal':"false"
42   },{ blocking:false, typed:true } );
43   skua.run('TSurfDecimate',{
44     'convergence':0.5,
45     'keep_borders':"true",
46     'length_unit':"Z unit",
47     'on':"/gobj:ts_DGM2_Kohlberg",
48     'tolerance':0.5
49   },{ blocking:false, typed:true } );
50   skua.run('GObjSaveAs',{
51     'file':"/tmp/ts_DGM2_Kohlberg.ts",
52     'filter':"none",
53     'on':"/gobj:ts_DGM2_Kohlberg",
54     'pattern':"*"
55   },{ blocking:false, typed:true } );
56   skua.run('GObjSaveAs',{
57     'file':"/tmp/DGM2_Kohlberg.ps",
58     'filter':"none",
59     'on':"/gobj:DGM2_Kohlberg",
60     'pattern':"*"
61   },{ blocking:false, typed:true } );
62
63   var kill_gocad = "kill -9 $PPID";
64   var ok = skua.System.command(kill_gocad);
65   EOF
66 - command: |
67   screen -d -m X -config /opt/dummy.conf
68   sleep 5
69   unzip /tmp/gocad/empty_project.sprj.zip -d /tmp/gocad/
70   /opt/Paradigm/SKUA-GOCAD-19/bin/SKUA -run-js-script /tmp/gocad/mesh_dgm2.js \
71     /tmp/gocad/empty_project.sprj
72 output:
73 - file:
74   /tmp/ts_DGM2_Kohlberg.ts
75 - file:
76   /tmp/DGM2_Kohlberg.ps

```

Listing C.10.: Definition of the triangulate construction step

C.6. extrude

The extrude operation imports and extrudes several shape files into corresponding geometries. It requires 4 different datasets as input:

- A shape file for the extends of the study area (`Gebietsgrenze_2021.shp`)
- A shape file representing a geological cross section (`Linien_Laengsschnitt.shp`)
- A shape file representing a potential tunnel route (`Streckenachse_UTM.shp`)
- A zipped empty gOcad project to carry over the initial gOcad project setup (`empty_project.sprj.zip`)

This operation produces several files as output:

- A triangulated surface surrounding the plane of the cross section in gOcad's ASCII format (`ts_Laengsschnitt.ts`)
- A multiline representing the different layers of the geological cross section in gOcad's ASCII format (`pl_Linien_Laengsschnitt.pl`)
- A multiline representing the potential tunnel route in gOcad's ASCII format (`pl_Steckachse_UTM.pl`)
- A multiline representing the extends of the study area in gOcad's ASCII format (`pl_Gebietsgrenze_2021.pl`)

Listing C.11 contains the construction step definition as used by the case study presented in section 7.3. The script performed as part of this construction step consists of the following operations:

1. Definition of the gOcad Macro (Line 5 - 62)
2. Starting a X-Server in background (required to execute gOcad) (Line 64)
3. Wait 5 seconds for the X-Server to start (Line 65)
4. Unpack the zipped empty gOcad project for later use (Line 66)
5. Execute the gOcad macro defined earlier via gOcad's command line interface (Line 67 - 69)

The GoCad macro defined in Line 5 - 63 performs the following operations:

1. Import the geological cross section and the potential rail way tunnel route from the provided shape files (Line 8 - 14)
2. Import the extends of the study area from the corresponding shape file (Line 15 - 18)
3. Move the rail way tunnel route to a constant depth of 210 m (Line 19 - 24)
4. Create a triangulated surface by extending the rail way tunnel by 90 m in z direction (Line 25 - 34)
5. Save the tube geometry of the railway tunnel created in the previous step as gOcad ASCII file (Line 35 - 40)
6. Save the planed road as polyline as gOcad ASCII file (Line 41 - 46)
7. Save the geological cross section as gOcad ASCII file (Line 47 - 52)
8. Save the extends of the study area as gOcad ASCII file (Line 53 - 58)
9. "Close" gOcad at this point by sending a `SIG_KILL` to the corresponding process (Line 60 - 61)

```

1  image: local-registry:5000/gocad-docker-image
2  input_path: "/tmp/gocad"
3  operation:
4    - command: |
5        cat > /tmp/gocad/extrude_streckenachse.js << EOF
6        var skua = PDGM.require('skua');
7
8        skua.run('ShapeImport', {
9            'File_name': [
10                "/tmp/gocad/Linien_Laengsschnitt.shp",
11                "/tmp/gocad/Streckenachse_UTM.shp"
12            ],
13            'cvn': "Domain=Default_depth"
14        }, { blocking: false, typed: true } );
15        skua.run('ShapeImport', {
16            'File_name': "/tmp/gocad/Gebietsgrenze_2021.shp",
17            'cvn': "Domain=Default_depth"
18        }, { blocking: false, typed: true } );
19        skua.run('PropertySetValue', {
20            'on': "/gobj:Streckenachse_UTM",
21            'property': "/Z",
22            'region': "everywhere",
23            'value': "value=210&unit=m&kind=Depth"
24        }, { blocking: false, typed: true } );
25        skua.run('TSurfCreateFromTube', {
26            'curves': "/gobj:Streckenachse_UTM",
27            'dissociate_vertices': "true",
28            'expansion': [0., 0., -90],
29            'name': "ts_Laengsschnitt",
30            'number_of_levels': 1,
31            'seal_ends': "false",
32            'select_number_of_levels': "true",
33            'two_ways': "false"
34        }, { blocking: false, typed: true } );
35        skua.run('GObjSaveAs', {
36            'file': "/tmp/ts_Laengsschnitt.ts",
37            'filter': "none",
38            'on': "/gobj:ts_Laengsschnitt",
39            'pattern': "*"
40        }, { blocking: false, typed: true } );
41        skua.run('GObjSaveAs', {
42            'file': "/tmp/pl_Streckenachse_UTM.pl",
43            'filter': "none",
44            'on': "/gobj:Streckenachse_UTM",
45            'pattern': "*"
46        }, { blocking: false, typed: true } );
47        skua.run('GObjSaveAs', {
48            'file': "/tmp/pl_Linien_Laengsschnitt.pl",
49            'filter': "none",

```



```

50         'on': "/gobj:Linien_Laengsschnitt",
51         'pattern': "*"
52     }, { blocking: false, typed: true } );
53     skua.run('GObjSaveAs', {
54         'file': "/tmp/pl_Gebietsgrenze_2021.pl",
55         'filter': "none",
56         'on': "/gobj:Gebietsgrenze_2021",
57         'pattern': "*"
58     }, { blocking: false, typed: true } );
59
60     var kill_gocad = "kill -9 $PPID";
61     var ok = skua.System.command(kill_gocad);
62     EOF
63 - command: |
64     screen -d -m X -config /opt/dummy.conf
65     sleep 5
66     unzip /tmp/gocad/empty_project.sprj.zip -d /tmp/gocad/
67     /opt/Paradigm/SKUA-GOCAD-19/bin/SKUA \
68         -run-js-script /tmp/gocad/extrude_streckenachse.js \
69         /tmp/gocad/empty_project.sprj
70 output:
71 - file:
72     /tmp/ts_Laengsschnitt.ts
73 - file:
74     /tmp/pl_Streckenachse_UTM.pl
75 - file:
76     /tmp/pl_Gebietsgrenze_2021.pl
77 - file:
78     /tmp/pl_Linien_Laengsschnitt.pl

```

Listing C.11.: Definition of the extrude construction step

C.7. project

The project operation projects the geological maps on top of the triangulated surface representing the digital elevation model. It requires 4 input datasets:

- A zipped empty gOcad project to carry over the initial gOcad project setup (`empty_project.sprj.zip`)
- The digital elevation model as triangulated surface in gOcad's ASCII format (`ts_DGM2_Kohlberg.ts`)
- The Quartär and Prequartär layer of the geological map as shape file (`L5148_lin-Quartaer.shp` and `L5148_lin-Praequartaer.shp`)

This operation produces two gOcad ASCII files as output (`pl_lin_Praequartaer.pl` and `pl_lin_Quartaer.pl`), which contain the corresponding projected geological map. Listing C.12 contains the construction step definition as used by the case study presented in section 7.3. The script performed as part of this construction step consists of the following operations:

1. Definition of the gOcad Macro (Line 5 - 45)

2. Starting a X-Server in background (required to execute gOcad) (Line 47)
3. Wait 5 seconds for the X-Server to start (Line 48)
4. Unpack the zipped empty gOcad project for later use (Line 49)
5. Execute the gOcad macro defined earlier via gOcad's command line interface (Line 50 - 52)

The gOcad macro defined in Line 5 - 63 performs the following operations:

1. Import the digital elevation model from the provided gOcad ASCII file (Line 7 - 10)
2. Import the geological map as multiline geometry from the provided shape files (Line 11 - 17)
3. Project both multiline geometries onto the digital elevation model (Line 18 - 27)
4. Export both projected multiline geometries as gOcad ASCII file (Line 30 - 41)
5. "Close" gOcad at this point by sending a SIG_KILL to the corresponding process (Line 43 - 44)

```

1  image: local-registry:5000/gocad-docker-image
2  input_path: "/tmp/gocad"
3  operation:
4    - command: |
5        cat > /tmp/gocad/project_map_to_dem.js << EOF
6        var skua = PDGM.require('skua');
7        skua.run('NewGObjLoad',{
8            'File_names':"/tmp/gocad/ts_DGM2_Kohlberg.ts",
9            'coordinate_system_name':"Default_depth"
10           },{ blocking:false, typed:true } );
11        skua.run('ShapeImport',{
12            'File_name':[
13                "/tmp/gocad/L5148_lin_Quartaer.shp",
14                "/tmp/gocad/L5148_lin_Praequartaer.shp"
15            ],
16            'cvn':"Domain=Default_depth"
17           },{ blocking:false, typed:true } );
18        skua.run('MapPointsOnPoints', {
19            'distance_max':100.0,
20            'length_unit':"Z unit",
21            'map_control_nodes':"true",
22            'targets':"/gobj:ts_DGM2_Kohlberg",
23            'to_map':[
24                "/gobj:L5148_lin_Praequartaer",
25                "/gobj:L5148_lin_Quartaer"
26            ],
27            'use_distance_max':"false"
28           },{ blocking:false, typed:true } );
29
30        skua.run('GObjSaveAs',{
31            'file':"/tmp/pl_lin_Praequartaer.pl",
32            'filter':"none",
33            'on':"/gobj:L5148_lin_Praequartaer",
34            'pattern':"*"
35           },{ blocking:false, typed:true } );

```

```

36     skua.run('GObjSaveAs',{
37         'file':"/tmp/pl_lin_Quartaer.pl",
38         'filter':"none",
39         'on':"/gobj:L5148_lin_Quartaer",
40         'pattern':"*"
41     },{ blocking:false, typed:true } );
42
43     var kill_gocad = "kill -9 $PPID";
44     var ok = skua.System.command(kill_gocad);
45     EOF
46 - command: |
47     screen -d -m X -config /opt/dummy.conf
48     sleep 5
49     unzip /tmp/gocad/empty_project.sprj.zip -d /tmp/gocad/
50     /opt/Paradigm/SKUA-GOCAD-19/bin/SKUA \
51         -run-js-script /tmp/gocad/project_map_to_dem.js \
52         /tmp/gocad/empty_project.sprj
53 output:
54 - file:
55     /tmp/pl_lin_Praequartaer.pl
56 - file:
57     /tmp/pl_lin_Quartaer.pl

```

Listing C.12.: Definition of the project construction step.

D. Construction step definitions for the Hydrologic balance model

D.1. Calculate bounding box

The calculate bounding box construction step calculates the bounding box of a given hydrotop shape file. This construction step expects a hydrotop shape file (**Hydrotope.shp**) as single input dataset and produces the outline of the bounding box represented as shape file (**bbox.shp**). It uses the **qgis_process** tool provided by Q-GIS [158] to calculate the bounding box for the provided hydrotop shape file. Listing D.13 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call **qgis_process** to extract the bounding box of the given hydrotop shape file (Line 5 - 6)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4 - command: |
5     qgis_process run qgis:polygonfromlayerextent \
6     --INPUT=/tmp/input/Hydrotope.shp --OUTPUT=/tmp/bbox.shp
7 output:
8 - file:
9     /tmp/bbox.shp

```

```

10 - file:
11     /tmp/bbox.cpg
12 - file:
13     /tmp/bbox.dbf
14 - file:
15     /tmp/bbox.prj
16 - file:
17     /tmp/bbox.shx

```

Listing D.13.: Definition of the calculate bounding box construction step

D.2. Clip DEM

The clip DEM construction step clips a provided raster dataset, representing a digital elevation grid, to the extends of a given bounding box. This construction step expects as input datasets a georeferenced raster in HGT format (`N50E013.hgt`), representing a digital elevation grid and a shape file representing a bounding box (`bbox.shp`). It produces a clipped version of the input raster grid `dem.tif` stored as TIF. The processing is done via `gdalwarp`, which is provided by Q-GIS [158]. Listing D.14 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `gdalwarp` to clip the provided raster dataset to the outline provided by the bounding box shapefile. (Line 5 - 6)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       gdalwarp -cutline /tmp/input/bbox.shp -t_srs EPSG:32633 \
6         /tmp/input/*.hgt /tmp/dem.tif
7 output:
8   - file:
9       /tmp/dem.tif

```

Listing D.14.: Definition of the clip DEM construction step

D.3. Clip BK50

The clip BK50 construction step clips a provided shape file, representing a soil map, to the extends of a given bounding box. This construction step expects two shape files as input datasets. One represents the soil map (`BK50.shp`) which is clipped by the outline provided by the other one (`bbox.shp`). The construction step produces a clipped version of the input raster grid `dem.tif` stored as TIF. The processing is done via `ogr2ogr`, which is provided by Q-GIS [158]. Listing D.15 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `ogr2ogr` to clip the provided vector layer to the outline provided by the bounding box shapefile. (Line 5 - 6)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       ogr2ogr -clipsrc /tmp/input/bbox.shp /tmp/bk50_clip.shp /tmp/input/BK50.shp
6 output:
7   - file:
8       /tmp/bk50_clip.shp
9   - file:
10      /tmp/bk50_clip.shx
11  - file:
12      /tmp/bk50_clip.prj
13  - file:
14      /tmp/bk50_clip.dbf

```

Listing D.15.: Definition of the clip BK50 construction step

D.4. Calculate aspect

The calculate aspect construction step calculates the slope aspect in degree based given digital elevation raster grid. This construction step expects a digital elevation raster grid as TIF (`dem.tif`) as input dataset and produces aspect raster grid as TIF (`aspect.tif` as output. The processing is done via `gdaldem`, which is provided by Q-GIS [158]. Listing D.16 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `gdaldem` to calculate the slope aspect based on a given digital elevation grid. (Line 5)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       gdaldem aspect /tmp/input/dem.tif /tmp/aspect.tif -of GTiff -b 1
6 output:
7   - file:
8       /tmp/aspect.tif

```

Listing D.16.: Definition of the calculate aspect construction step

D.5. Calculate slope

The calculate slope construction step calculates the slope in percent based on given digital elevation raster grid. This construction step expects a digital elevation raster grid as TIF (`dem.tif`) as input dataset and produces slope raster grid as TIF (`slope.tif` as output. The processing is done via `gdaldem`, which is provided by Q-GIS [158]. Listing D.17 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `gdaldem` to calculate the slope based on a given digital elevation grid. (Line 5)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       gdaldem slope /tmp/input/dem.tif /tmp/slope.tif -of GTiff -b 1 -s 1.0 -p
6 output:
7   - file:
8       /tmp/slope.tif

```

Listing D.17.: Definition of the calculate slope construction step

D.6. Calculate slope length

The calculate slope length construction step calculates the length of a slope in meter based on given digital elevation raster grid for each raster grid point. This construction step expects a digital elevation raster grid as TIF (`dem.tif`) as input dataset and produces slope raster grid as SAGA grid (`slope_length.sgrd`) as output. The processing is done via `qgis_process`, which is provided by Q-GIS [158]. Listing D.18 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `qgis_process` to calculate the slope length based on a given digital elevation grid. (Line 5 - 6)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       qgis_process run saga:slopelength --DEM=/tmp/input/dem.tif \
6         --LENGTH=/tmp/slope_length
7 output:
8   - file:
9       /tmp/slope_length.mgrd
10  - file:
11      /tmp/slope_length.prj
12  - file:
13      /tmp/slope_length.sdat
14  - file:
15      /tmp/slope_length.sdat.aux.xml
16  - file:
17      /tmp/slope_length.sgrd

```

Listing D.18.: Definition of the calculate slope construction step

D.7. Calculate avg height

The calculate avg height construction step calculates the average height per hydrotop given in a shape file. This construction step expects a digital elevation grid as TIF (`dem.tif`) and a hydrotop shape file (`Hydrotope.shp`) as input dataset. It produces a shape file (`hydrotope_with_height.shp`) containing any information already present in the hydrotop shape file with an additional height property containing the average height per shape file geometry. The processing is done via the `qgis_process`, which is provided by Q-GIS [158]. Listing D.19 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `qgis_process` to calculate the average height per hydrotop (Line 5 - 10)

```

1  image: local-registry:5000/qgis-processing-image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        qgis_process run native:zonalstatisticsfb \
6            --INPUT=/tmp/input/Hydrotope.shp \
7            --INPUT_RASTER=/tmp/input/dem.tif \
8            --OUTPUT=/tmp/hydrotope_with_height.shp \
9            --RASTER_BAND=1 --STATISTICS=2 \
10           --COLUMN_PREFIX=height_
11  output:
12    - file:
13        /tmp/hydrotope_with_height.shp
14    - file:
15        /tmp/hydrotope_with_height.shx
16    - file:
17        /tmp/hydrotope_with_height.prj
18    - file:
19        /tmp/hydrotope_with_height.dbf
20    - file:
21        /tmp/hydrotope_with_height.cpg

```

Listing D.19.: Definition of the calculate avg height construction step

D.8. Calculate slope per hydrotope

The calculate slope per hydrotope construction step calculates the average slope per hydrotop given in a shape file. This construction step expects a slope grid as TIF (`dem.tif`) and a hydrotop shape file (`hydrotope_with_height.shp`) as input dataset. It produces a shape file (`hydrotope_with_slope.shp`) containing any information already present in the hydrotop shape file with an additional slope property containing the average slope per shape file geometry. The processing is done via the `qgis_process`, which is provided by Q-GIS [158]. Listing D.20 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `qgis_process` to calculate the average slope per hydrotop (Line 5 - 10)

```

1  image: local-registry:5000/qgis-processing-image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        qgis_process run native:zonalstatisticsfb \
6            --INPUT=/tmp/input/hydrotope_with_height.shp \
7            --INPUT_RASTER=/tmp/input/slope.tif \
8            --OUTPUT=/tmp/hydrotope_with_slope.shp \
9            --RASTER_BAND=1 --STATISTICS=2 \
10           --COLUMN_PREFIX=slope_
11 output:
12   - file:
13       /tmp/hydrotope_with_slope.shp
14   - file:
15       /tmp/hydrotope_with_slope.shx
16   - file:
17       /tmp/hydrotope_with_slope.prj
18   - file:
19       /tmp/hydrotope_with_slope.dbf
20   - file:
21       /tmp/hydrotope_with_slope.cpg

```

Listing D.20.: Definition of the calculate slop per hydrotop construction step

D.9. Calculate aspect per hydrotope

The calculate aspect per hydrotope construction step calculates the average slope aspect per hydrotop given in a shape file. This construction step expects a slope grid as TIF (**dem.tif**) and a hydrotop shape file (**hydrotope_with_slope.shp**) as input dataset. It produces a shape file (**hydrotope_with_aspect.shp**) containing any information already present in the hydrotop shape file with an additional aspect property containing the average slope aspect per shape file geometry. The processing is done via the **qgis_process**, which is provided by Q-GIS [158]. Listing D.21 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call **qgis_process** to calculate the average slope aspect per hydrotop (Line 5 - 10)

```

1  image: local-registry:5000/qgis-processing-image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        qgis_process run native:zonalstatisticsfb \
6            --INPUT=/tmp/input/hydrotope_with_slope.shp \
7            --INPUT_RASTER=/tmp/input/aspect.tif \
8            --OUTPUT=/tmp/hydrotope_with_aspect.shp \
9            --RASTER_BAND=1 --STATISTICS=2 \
10           --COLUMN_PREFIX=aspect_
11 output:

```



```

12 - file:
13     /tmp/hydrotope_with_aspect.shp
14 - file:
15     /tmp/hydrotope_with_aspect.shx
16 - file:
17     /tmp/hydrotope_with_aspect.prj
18 - file:
19     /tmp/hydrotope_with_aspect.dbf
20 - file:
21     /tmp/hydrotope_with_aspect.cpg

```

Listing D.21.: Definition of the calculate aspect per hydrotop construction step

D.10. Calculate maximal slope length per hydrotop

The calculate maximal slope length per hydrotop construction step calculates the maximal slope length occurring per hydrotop given in a shape file. This construction step expects a slope grid as TIF (dem.tif) and a hydrotop shape file (hydrotope_with_aspect.shp) as input dataset. It produces a shape file (hydrotope_with_slope_length.shp) containing any information already present in the hydrotop shape file with an additional slope_length property containing the maximal slope length per shape file geometry. The processing is done via the `qgis_process`, which is provided by Q-GIS [158]. Listing D.22 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `qgis_process` to calculate the maximal slope length per hydrotop (Line 5 - 10)

```

1 image: local-registry:5000/qgis-processing-image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       qgis_process run native:zonalstatisticsfb \
6         --INPUT=/tmp/input/hydrotope_with_aspect.shp \
7         --INPUT_RASTER=/tmp/input/slope_length.sdat \
8         --OUTPUT=/tmp/hydrotop_with_slope_length.shp \
9         --RASTER_BAND=1 --STATISTICS=6 \
10        --COLUMN_PREFIX=slope_length_
11 output:
12   - file:
13       /tmp/hydrotop_with_slope_length.shp
14   - file:
15       /tmp/hydrotop_with_slope_length.shx
16   - file:
17       /tmp/hydrotop_with_slope_length.prj
18   - file:
19       /tmp/hydrotop_with_slope_length.dbf
20   - file:
21       /tmp/hydrotop_with_slope_length.cpg

```

Listing D.22.: Definition of the calculate maximal slope length per hydrotop construction step

D.11. Calculate hydrotop area

The calculate hydrotop area construction step calculates the area of any hydrotop provided in the input dataset in hectare. It expects a hydrotop shape file (`hydrotope_with_slope_length.shp`) as single input dataset and produces a shape file (`hydrotope_with_area.shp`) containing any information already present in the input dataset with an additional property area containing the hydrotop size in hectare added. The processing is done via `qgis_process`, which is provided by Q-GIS [158]. Listing D.23 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `qgis_process` to calculate the hydrotop area (Line 5 - 10)

```

1  image: local-registry:5000/qgis-processing-image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        qgis_process run native:fieldcalculator \
6            --FIELD_NAME='area' --FIELD_PRECISION='4' \
7            --FORMULA='$area / (100.0*100.0)' \
8            --INPUT=/tmp/input/hydrotop_with_slope_length.shp \
9            --OUTPUT=/tmp/hydrotope_with_area.shp \
10           --FIELD_TYPE='0' --FIELD_LENGTH='5'
11 output:
12   - file:
13       /tmp/hydrotope_with_area.shp
14   - file:
15       /tmp/hydrotope_with_area.shx
16   - file:
17       /tmp/hydrotope_with_area.prj
18   - file:
19       /tmp/hydrotope_with_area.dbf
20   - file:
21       /tmp/hydrotope_with_area.cpg

```

Listing D.23.: Definition of the calculate hydrotop area construction step

D.12. Lookup soil type

The lookup soil type construction step determines the prevailing soil kind per hydrotop, by doing a spatial join between the different input datasets. This construction step expects two input datasets, one shape file that contains a soil type map (`bk50_clip.shp`) and one that contains information about the hydrotopes (`hydrotope_with_area.shp`). It produces a shape file (`hydrotopes_joined_with_bk50.shp`) containing all information present in the input hydrotop dataset with additional properties coming from the soil map. The processing is done via `qgis_process`, which is provided by Q-GIS [158]. Listing D.24 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Call `qgis_process` to perform a spatial join to lookup the prevailing soil type per hydrotop (Line 5 - 9)

```

1  image: local-registry:5000/qgis-processing-image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        qgis_process run native:joinattributesbylocation \
6            --INPUT=/tmp/input/hydrotope_with_area.shp \
7            --JOIN=/tmp/input/bk50_clip.shp --METHOD=2 \
8            --OUTPUT=/tmp/hydrotopes_joined_with_bk50.shp \
9            --PREDICATE=0 --JOIN_FIELDS='LEG_NR_12' --PREFIX='BK50_'
10 output:
11   - file:
12       /tmp/hydrotopes_joined_with_bk50.shp
13   - file:
14       /tmp/hydrotopes_joined_with_bk50.shx
15   - file:
16       /tmp/hydrotopes_joined_with_bk50.prj
17   - file:
18       /tmp/hydrotopes_joined_with_bk50.dbf
19   - file:
20       /tmp/hydrotopes_joined_with_bk50.cpg

```

Listing D.24.: Definition of the lookup soil type construction step

D.13. Preprocess climate data

The preprocess climate data construction step converts climate data as provided by the German meteorological survey (DWD) into the format expected by BoWaHald [30]. The construction step expects a variable number of CSV files as input and will produce a single Microsoft Excel file as output dataset (`metdata.xls`). The construction step extracts all required parameters from the provided CSV files by looking at the different files in their provided order. This means the construction step will first try to gather all parameters from the first file (according to an order based on the file name). Missing values will be filled with information from the second file and so on. The script will extract any required parameter for the time period between 1.11.1990 and 31.10.2020. The processing is done by a self written R-Script. Listing D.25 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Definition of the R script used to perform the processing (Line 5 - 77)
2. Execute the R script defined in step 1 (Line 79)

```

1  image: local-registry:5000/wasserhaushalt_r_processing_image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5      cat > /tmp/preprocess_metdata.R << 'EOF'
6      library(dplyr)
7      library(zoo)
8      library(xlsx)
9
10     path.to.csv <- "/tmp/input"
11     files<-list.files(path.to.csv, pattern = "*.csv")
12     print(files)  ## list all files in path
13
14     result = data.frame(
15       MESS_DATUM = integer(),
16       RS = numeric(),
17       TMK = numeric(),
18       UPM = numeric(),
19       FM = numeric(),
20       SDK = numeric()
21     )
22
23
24     for(i in 1:length(files)) {
25       print(files[i])
26       df <- read.csv(paste0(path.to.csv,"/",files[i]), sep = ";")
27
28       if ('RS' %in% names(df)) {
29         result <- result %>% full_join(df, by = "MESS_DATUM") %>%
30           mutate(RS = coalesce(RS.x, RS.y)) %>%
31           select(MESS_DATUM, RS, TMK, UPM, FM, SDK)
32
33       } else {
34         result <- result %>% full_join(df, by = "MESS_DATUM") %>%
35           mutate(
36             RS = coalesce(RS, RSK),
37             TMK = coalesce(TMK.x, na_if(TMK.y, -999)),
38             UPM = coalesce(UPM.x, na_if(UPM.y, -999)),
39             FM = coalesce(FM.x, na_if(FM.y, -999)),
40             SDK = coalesce(SDK.x, na_if(SDK.y, -999))
41           ) %>%
42           select(MESS_DATUM, RS, TMK, UPM, FM, SDK)
43       }
44     }
45
46     # cleanup data
47
48     result <- result %>% arrange(MESS_DATUM) %>%
49     # Verbleibende Lücken per Interpolation schließen

```

```

50     # (Maximale per Interpolation zu schließende Lückengröße: 20 Tage)
51     mutate(
52       MESS_DATUM = as.Date(strptime(MESS_DATUM, "%Y%m%d")),
53       RS = na.approx(RS, maxgap = 20, rule = 2),
54       TMK = na.approx(TMK, maxgap = 20, rule = 2),
55       UPM = na.approx(UPM, maxgap = 20, rule = 2),
56       FM = na.approx(FM, maxgap = 20, rule = 2),
57       SDK = na.approx(SDK, maxgap = 20, rule = 2)
58     ) %>%
59     # Kein Sonnenschein wenn Sonnenschein NA
60     mutate(SDK = replace(SDK, is.na(SDK), 0)) %>%
61     # Nur die für den Untersuchungszeitraum notwendigen Daten exportieren
62     filter(MESS_DATUM > "1990-11-01" & MESS_DATUM <= "2020-10-31") %>%
63     mutate(
64       DAT = MESS_DATUM,
65       T = TMK,
66       TF = ifelse(lead(TMK) >= 0 | is.na(lead(TMK)), 1, -1),
67       RLF = UPM,
68       RG = 0.0,
69       SSD = SDK,
70       VW = FM,
71       P = RS
72     ) %>%
73     select(DAT, T, TF, RLF, RG, SSD, VW, P)
74
75     write.xlsx(result, '/tmp/metdata.xls', row.names = FALSE)
76
77     EOF
78 - command: |
79     R --no-save < /tmp/preprocess_metdata.R
80 output:
81 - file:
82     /tmp/metdata.xls

```

Listing D.25.: Definition of the preprocess climate data construction step

D.14. Generate bwmhydro files

The generate bwmhydro files construction step generates for a bwmhydro file as expected by BoWaHald [30] for each hydrotop present in the provided hydrotop shape file. This construction step expects the aggregated hydrotop shape file as single input dataset (`hydrotopes_joined_with_bk50.shp`) and produces a TAR archive containing a compatible bwmhydro file for each hydrotop present in the hydrotop shapefile. The processing is done based on the attributes present in the hydrotop shape file with the help of a self written python script. Listing D.26 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Create the output directory (Line 5)
2. Definition of the python script used to perform the processing (Line 7 - 91)

3. Execute the python script defined in step 2 (Line 94)
4. Create a TAR archive containing all created bwmhydro files. It is important here to explicitly set a sorting order and a timestamp because otherwise the `tar` command may introduce non-reproducible behaviour. (Line 95)

```

1  image: local-registry:5000/wasserhaushalt_python_processing_image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5      mkdir /tmp/out
6    - command: |
7      cat > /tmp/convert_to_bwmhydro.py << EOF
8      #!/usr/bin/env python3
9
10     import geopandas as gpd
11     import pandas as pd
12     from math import ceil, isnan
13
14     def bowa_hydrotop_template(layer):
15
16         height = round(layer[1]['height_mea'], 1)
17         aspect = layer[1]['aspect_mea']
18         slope_length = round(layer[1]['slope_leng'], -1)
19         center = layer[1]['geometry'].centroid
20         deg = int(center.y)
21         mins = int(60 * (center.y - deg))
22         use = layer[1]['Nutzung']
23
24
25         exposure_target = aspect / 22.5
26         if exposure_target < 1:
27             exposure = "Nord"
28         elif exposure_target < 3:
29             exposure = "Nordost"
30         elif exposure_target < 5:
31             exposure = "Ost"
32         elif exposure_target < 7:
33             exposure = "Südost"
34         elif exposure_target < 9:
35             exposure = "Süd"
36         elif exposure_target < 11:
37             exposure = "Südwest"
38         elif exposure_target < 13:
39             exposure = "West"
40         elif exposure_target < 15:
41             exposure = "Nordwest"
42         else:
43             exposure = "Nord"
44
45         if use == "Landwirtschaftlich-Ackerbaulich":
46             land_use = "Ackerland (Getreide, Hackfrüchte...)"

```

```

47         elif use == "Wald (ggf. mit Unterholz und Gras)" or use == "Sträucher":
48             land_use = "Baum- und Buschbewuchs ggf. mit Gras"
49         elif use == "Dauergrünland (Wiese, Weide)":
50             land_use = "Gras-/Krautbewuchs (Wiese, Weide)"
51         elif use == "Gewässer":
52             land_use = "Gewässer"
53         elif use == "Ortschaft, versiegelte Fläche":
54             land_use = "Bebauung (teil- bzw. vollversiegelt)"
55         else:
56             land_use = "ohne Bewuchs"
57
58         template = """<?xml version="1.0" encoding="UTF-8"?>
59 <hydroipitation>
60 <version>0.2</version>
61 <data>
62 <deg>{deg}</deg>
63 <min>{min}</min>
64 <height>{height}</height>
65 <length>{slope_length}</length>
66 <exposure>{exposure}</exposure>
67 <use>{land_use}</use>
68 <eva>2500</eva>
69 </data>
70 </hydroipitation>""".format(
71     height = height,
72     slope_length = slope_length,
73     exposure = exposure,
74     deg = deg,
75     min = mins,
76     land_use = land_use
77 )
78     return template
79
80 shapefile = gpd.read_file("/tmp/input/hydrotopes_joined_with_bk50.shp")
81 shapefile = shapefile.to_crs('EPSG:4326') # to wgs 84
82
83 for row in shapefile.iterrows():
84     id = row[0]
85     name = row[1]['Name']
86     bwmhydro_file_name = f'/tmp/out/{id}_{name}.bwmhydro'
87     with open(bwmhydro_file_name, 'w') as bwmhydro_file:
88         bwmhydro = bowa_hydrotop_template(row)
89         bwmhydro_file.write(bwmhydro)
90
91 EOF
92
93 - command: |
94     python /tmp/convert_to_bwmhydro.py
95     tar --sort=name --mtime='UTC 2021-12-31' -cf /tmp/bwmhydro.tar /tmp/out/*
96 output:
97 - file:

```

98 /tmp/bwmhydro.tar

Listing D.26.: Definition of the generate bwmhydro files construction step

D.15. Generate bwmuse files

The generate bwmhydro files construction step generates for a bwmuse file as expected by BoWaHald [30] for each hydrotop present in the provided hydrotop shape file. This construction step expects the aggregated hydrotop shape file as single input dataset (`hydrotopes_joined_with_bk50.shp`) and produces a TAR archive containing a compatible bwmuse file for each hydrotop present in the hydrotop shapefile. The processing is done based on the attributes present in the hydrotop shape file with the help of a self written python script. Listing D.27 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Create the output directory (Line 5)
2. Definition of the python script used to perform the processing (Line 7 - 91)
3. Execute the python script defined in step 2 (Line 94)
4. Create a TAR archive containing all created bwmuse files. It is important here to explicitly set a sorting order and a timestamp because otherwise the `tar` command may introduce non-reproducible behaviour. (Line 95)

```

1  image: local-registry:5000/wasserhaushalt_python_processing_image
2  input_path: "/tmp/input"
3  operation:
4      - command: |
5          mkdir /tmp/out
6      - command: |
7          cat > /tmp/convert_to_bwmuse.py << EOF
8          #!/usr/bin/env python3
9
10         import geopandas as gpd
11         import pandas as pd
12         from math import ceil, isnan
13
14         def bowa_landuse_template(layer):
15             use = layer[1]['Nutzung']
16             sub_use = layer[1].get('Nutzung2') or 'unbekannt'
17             kc = layer[1].get('Bestandskoeffizient') or 0
18             coverage = layer[1].get('Deckungsgrad') or 0
19             rootdepth = layer[1].get('Wurzeltiefe') or 0
20             rootthickness = layer[1].get('Wurzeldichte') or 0
21             dev = layer[1].get('Entwicklung') or 'unbekannt'
22             form = layer[1].get('Wuchsform') or 'unbekannt'
23             structure = layer[1].get('Structure') or 'unbekannt'
24             damage = layer[1].get('Schadklasse') or 'unbekannt'
25             sealing = layer[1].get('Versieglun') or 0
26             if isnan(sealing):
27                 sealing = 0

```



```

28
29
30     template = """<?xml version="1.0" encoding="UTF-8"?>
31 <landuse>
32     <version>0.2</version>
33     <data>
34         <use>{use}</use>
35         <use2>{sub_use}</use2>
36         <kc>{kc}</kc>
37         <coverage>{coverage}</coverage>
38         <rootdepth>{rootdepth}</rootdepth>
39         <rootthickness>{rootthickness}</rootthickness>
40         <dev>{dev}</dev>
41         <form>{form}</form>
42         <structure>{structure}</structure>
43         <damage>{damage}</damage>
44         <sealing>{sealing}</sealing>
45     </data>
46 </landuse>""".format(
47     use = use,
48     sub_use = sub_use,
49     kc = kc, coverage = coverage,
50     rootdepth = rootdepth,
51     rootthickness = rootthickness,
52     dev = dev,
53     form = form,
54     structure = structure,
55     damage = damage,
56     sealing = sealing
57 )
58
59     return template
60
61
62 shapefile = gpd.read_file("/tmp/input/hydrotopes_joined_with_bk50.shp")
63 shapefile = shapefile.to_crs('EPSG:4326') # to wgs 84
64
65 for row in shapefile.iterrows():
66     id = row[0]
67     name = row[1]['Name']
68     bwmuse_file_name = f'/tmp/out/{id}_{name}.bwmuse'
69     with open(bwmuse_file_name, 'w') as bwmuse_file:
70         bwmuse = bowa_landuse_template(row)
71         bwmuse_file.write(bwmuse)
72
73 EOF
74
75 - command: |
76     python /tmp/convert_to_bwmuse.py
77     tar --sort=name --mtime='UTC 2021-12-31' -cf /tmp/bwmuse.tar /tmp/out/*
78 output:

```

```

79 - file:
80     /tmp/bwmuse.tar

```

Listing D.27.: Definition of the generate bwmuse files construction step

D.16. Generate bwmlayer files

The generate bwmlayer files construction step generates for a bwmlayer file as expected by BoWaHald [30] for each hydrotop present in the provided hydrotop shape file. This construction step expects the aggregated hydrotop shape file (`hydrotopes_joined_with_bk50.shp`) and the corresponding legend for the soil map as Microsoft Excel file (`BK50_legend.xlsx`) as input. It produces a TAR archive containing a compatible bwmlayer file for each hydrotop present in the hydrotop shapefile. The processing is done based on the attributes present in the hydrotop shape file and the soil map legend by a self written python script. Listing D.28 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Create the output directory (Line 5)
2. Definition of the python script used to perform the processing (Line 7 - 91)
3. Execute the python script defined in step 2 (Line 94)
4. Create a TAR archive containing all created bwmlayer files. It is important here to explicitly set a sorting order and a timestamp because otherwise the `tar` command may introduce non-reproducible behaviour. (Line 95)

```

1  image: local-registry:5000/wasserhaushalt_python_processing_image
2  input_path: "/tmp/input"
3  operation:
4      - command: |
5          mkdir /tmp/out
6      - command: |
7          cat > /tmp/convert_to_bwmlayers.py << EOF
8          #!/usr/bin/env python3
9
10         import geopandas as gpd
11         import pandas as pd
12         from math import ceil, isnan
13
14         KA5_LOOKUP = {
15             "Ss": {
16                 "kftemp": [4.3403E-05,3.9352E-05,2.662E-05],
17                 "tab70": [36,32,27,9,7,7,14,11,10,5,4,3],
18                 "tab72": [0,0,-1,-2,-3,0,1,3,4,5,0,3,6,9,12]
19             },
20             "S12": {
21                 "kftemp": [1.8634E-05,1.1343E-05,6.0185E-06],
22                 "tab70": [23,18,13,20,18,17,28,25,23,8,7,6],
23                 "tab72": [0,0,1,2,3,0,2,3,4,6,0,3,6,9,13],
24             },
25             "S13": {

```

```

26         "kftemp": [1.1343E-05,7.5231E-06,3.3565E-06],
27         "tab72": [0,1,2,3,4,0,1,3,4,6,0,3,5,9,12],
28         "tab70": [18,15,10,22,18,17,34,27,25,12,9,8]
29     },
30     "S14": {
31         "kftemp": [1.2269E-05,4.8611E-06,2.4306E-06],
32         "tab70": [18,12,8,22,18,15,36,30,26,14,12,11],
33         "tab72": [0,2,2,3,4,0,2,4,5,6,0,3,7,11,14]
34     },
35     "Slu": {
36         "kftemp": [6.9444E-06,3.2407E-06,1.5046E-06],
37         "tab70": [14,10,7,23,21,19,38,33,30,15,12,11],
38         "tab72": [0,2,3,4,6,0,1,2,4,6,0,2,5,8,11],
39     },
40     "St2": {
41         "kftemp": [2.0718E-05,1.3657E-05,7.8704E-06],
42         "tab70": [24,20,15,18,16,13,26,22,18,8,6,5],
43         "tab72": [0,0,0,1,1,0,3,4,5,7,0,5,7,11,15],
44     },
45     "St3": {
46         "kftemp": [1.3194E-05,4.8611E-06,2.7778E-06],
47         "tab70": [18,14,9,18,15,12,35,30,26,17,15,14],
48         "tab72": [0,1,2,3,4,0,2,4,6,9,0,2,5,10,14],
49     },
50     "Su2": {
51         "kftemp": [2.0139E-05,1.4699E-05,7.6389E-06],
52         "tab70": [24,21,15,20,18,17,26,23,21,6,5,4],
53         "tab72": [0,0,0,-1,-2,0,2,3,4,6,0,3,6,9,13],
54     },
55     "Su3": {
56         "kftemp": [1.0185E-05,6.8287E-06,3.588E-06],
57         "tab70": [17,14,10,25,21,20,35,29,26,10,8,6],
58         "tab72": [0,1,1,2,2,0,1,3,3,4,0,2,6,8,11],
59     },
60     "Su4": {
61         "kftemp": [6.713E-06,4.3981E-06,1.9676E-06],
62         "tab70": [14,11,8,27,23,21,39,32,28,12,9,7],
63         "tab72": [0,2,3,4,6,0,1,2,3,4,0,2,4,8,11],
64     },
65     "Ls2": {
66         "kftemp": [6.1343E-06,2.662E-06,1.1574E-06],
67         "tab70": [13,9,6,21,16,14,40,34,31,19,18,17],
68         "tab72": [0,2,3,4,5,0,1,3,5,8,0,3,6,11,14],
69     },
70     "Ls3": {
71         "kftemp": [8.5648E-06,2.662E-06,1.2731E-06],
72         "tab70": [15,9,6,21,16,14,39,33,30,18,17,16],
73         "tab72": [0,1,2,3,4,0,1,3,5,8,0,3,6,11,14],
74     },
75     "Ls4": {
76         "kftemp": [7.8704E-06,4.1667E-06,1.2731E-06],

```

```

77         "tab70": [15,11,7,20,16,13,39,32,28,19,16,15],
78         "tab72": [0,1,2,3,3,0,2,4,6,8,0,4,6,12,15],
79     },
80     "Lt2": {
81         "kftemp": [3.8194E-06,1.5046E-06,6.9444E-07],
82         "tab70": [11,7,5,18,14,11,42,36,32,24,22,21],
83         "tab72": [0,2,3,5,6,0,3,5,8,10,0,5,8,13,15],
84     },
85     "Lt3": {
86         "kftemp": [2.3148E-06,8.1019E-07,3.4722E-07],
87         "tab70": [8,5,3,17,12,10,45,39,35,28,27,25],
88         "tab72": [0,1,2,4,7,0,2,4,8,11,0,5,6,12,15],
89     },
90     "Lts": {
91         "kftemp": [3.588E-06,1.1574E-06,8.1019E-07],
92         "tab70": [10,6,5,17,14,11,44,37,31,27,23,20],
93         "tab72": [0,1,2,5,6,0,3,5,7,9,0,3,7,13,15],
94     },
95     "Lu": {
96         "kftemp": [5.2083E-06,1.8519E-06,6.9444E-07],
97         "tab70": [12,7,4,21,17,15,41,36,33,20,19,18],
98         "tab72": [0,2,3,6,7,0,3,5,7,8,0,6,7,13,14],
99     },
100    "Uu": {
101        "kftemp": [3.7037E-06,1.5046E-06,2.3148E-07],
102        "tab70": [10,7,3,30,26,23,43,38,35,13,12,12],
103        "tab72": [0,2,3,5,9,0,1,2,3,4,0,2,4,8,11]
104    },
105    "Uls": {
106        "kftemp": [5.6713E-06,2.3148E-06,8.1019E-07],
107        "tab70": [13,8,5,24,22,21,39,35,33,15,13,12],
108        "tab72": [0,2,3,4,8,0,3,4,4,7,0,4,7,10,15],
109    },
110    "Us": {
111        "kftemp": [4.2824E-06,2.5463E-06,5.787E-07],
112        "tab70": [11,9,4,28,25,22,41,35,32,13,10,10],
113        "tab72": [0,2,3,5,8,0,1,2,3,4,0,2,4,7,10],
114    },
115    "Ut2": {
116        "kftemp": [3.7037E-06,1.3889E-06,2.3148E-07],
117        "tab70": [10,6,3,28,26,23,40,37,35,12,11,12],
118        "tab72": [0,2,4,6,8,0,1,1,2,4,0,2,4,7,12]
119    },
120    "Ut3": {
121        "kftemp": [4.7454E-06,1.3889E-06,3.4722E-07],
122        "tab70": [11,6,3,26,25,23,39,37,35,13,12,12],
123        "tab72": [0,2,4,6,8,0,1,1,2,4,0,2,3,8,12],
124    },
125    "Ut4": {
126        "kftemp": [5.2083E-06,1.5046E-06,3.4722E-07],
127        "tab70": [12,7,3,23,21,19,39,37,35,16,16,16],

```

```

128         "tab72": [0,2,4,6,7,0,2,3,4,6,0,4,6,9,13],
129     },
130     "Tt": {
131         "kftemp": [4.6296E-07,3.4722E-07,2.3148E-07],
132         "tab70": [4,3,2,15,13,12,51,43,35,36,30,23],
133         "tab72": [0,1,2,4,8,0,2,4,5,7,0,5,6,9,11],
134     },
135     "Tl": {
136         "kftemp": [9.2593E-07,6.9444E-07,2.3148E-07],
137         "tab70": [5,4,3,15,13,11,48,41,35,33,28,24],
138         "tab72": [0,1,2,3,7,0,2,4,6,8,0,5,6,11,13],
139     },
140     "Tu2": {
141         "kftemp": [9.2593E-07,3.4722E-07,2.3148E-07],
142         "tab70": [5,4,3,16,12,10,47,42,36,31,30,26],
143         "tab72": [0,1,2,3,7,0,1,3,5,8,0,5,6,10,13]
144     },
145     "Tu3": {
146         "kftemp": [2.0833E-06,1.0417E-06,3.4722E-07],
147         "tab70": [8,6,3,17,13,10,45,38,35,28,25,25],
148         "tab72": [0,2,2,3,6,0,2,4,7,9,0,6,8,12,14],
149     },
150     "Tu4": {
151         "kftemp": [3.8194E-06,1.3889E-06,3.4722E-07],
152         "tab70": [10,6,3,19,17,16,41,37,35,22,20,19],
153         "tab72": [0,1,3,4,6,0,3,5,6,8,0,5,8,11,15],
154     },
155     "Ts2": {
156         "kftemp": [9.2593E-07,5.787E-07,3.4722E-07],
157         "tab70": [5,4,3,16,13,12,47,39,34,31,26,22],
158         "tab72": [0,1,2,3,7,0,2,4,6,8,0,6,7,12,14]
159     },
160     "Ts3": {
161         "kftemp": [1.7361E-06,1.2731E-06,9.2593E-07],
162         "tab70": [7,6,5,16,13,11,45,37,32,29,24,21],
163         "tab72": [0,2,3,4,5,0,2,5,7,9,0,5,6,12,14],
164     },
165     "Ts4": {
166         "kftemp": [5.9028E-06,4.3981E-06,9.2593E-07],
167         "tab70": [13,10,6,17,14,11,43,32,30,26,18,19],
168         "tab72": [0,2,3,4,5,0,2,4,7,9,0,4,6,11,14],
169     },
170     "fS": {
171         "kftemp": [4.7454E-05,3.4722E-05,2.3148E-05],
172         "tab70": [34,31,23,10,9,8,16,14,12,6,5,4],
173         "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
174     },
175     "fSms": {
176         "kftemp": [4.7454E-05,3.4722E-05,2.3148E-05],
177         "tab70": [34,31,23,10,9,8,16,14,12,6,5,4],
178         "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

```

```

179         },
180         "fSgs": {
181             "kftemp": [4.7454E-05,3.4722E-05,2.3148E-05],
182             "tab70": [34,31,23,10,9,8,16,14,12,6,5,4],
183             "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
184         },
185         "mS": {
186             "kftemp": [7.8125E-05,5.6713E-05,2.8935E-05],
187             "tab70": [36,32,26,9,6,5,14,10,8,5,4,3],
188             "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
189         },
190         "mSfs": {
191             "kftemp": [7.8125E-05,5.6713E-05,2.8935E-05],
192             "tab70": [36,32,26,9,6,5,14,10,8,5,4,3],
193             "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
194         },
195         "mSgs": {
196             "kftemp": [7.8125E-05,5.6713E-05,2.8935E-05],
197             "tab70": [36,32,26,9,6,5,14,10,8,5,4,3],
198             "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
199         },
200         "gS": {
201             "kftemp": [2.4306E-04,9.6644E-05,3.8773E-05],
202             "tab70": [38,33,29,8,5,4,12,8,6,4,3,2],
203             "tab72": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
204         },
205     }
206
207     def bowa_layer_template(id, size, grad, soil_kind, humus):
208         size = int(round(size * 1000, 0)) # size is in mm
209         id += 1 # bowa starts at 1
210         ts = ceil(size / 100) # we use 1 "layer" per 100mm
211
212         h = 1
213         if humus == "h0" or humus == "h1":
214             h = 1
215         elif humus == "h2":
216             h = 2
217         elif humus == "h3":
218             h = 3
219         elif humus == "h4":
220             h = 4
221         else:
222             h = 5
223
224         n = 2 # mittel
225
226         ka5 = KA5_LOOKUP[soil_kind]
227
228         # calculation copied from bowa
229         t0lki = n-1

```

```
230         t2lki = h-1
231         t0nfki = 3+n-1
232         t2nfki = 5+h-1
233         t0fki = 6+n-1
234         t2fki = 10+h-1
235         t0twi = 9+n-1
236
237         kf = ka5["kftemp"][n-1]
238
239         fk1 = ka5["tab70"][t0fki] + ka5["tab72"][t2fki]
240         lk1 = ka5["tab70"][t0lki] + ka5["tab72"][t2lki]
241         nfk = ka5["tab70"][t0nfki] + ka5["tab72"][t2nfki]
242         pwp = fk1 - nfk
243         sat = fk1 + lk1
244         ksh = 0
245
246         if (kf <= 2.5E-10):
247             ksh = 2200
248         if (kf > 2.5E-10):
249             ksh = 2100
250         if (kf > 7.5E-10):
251             ksh = 1800
252         if (kf > 2.5E-09):
253             ksh = 1500
254         if (kf > 7.5E-09):
255             ksh = 1300
256         if (kf > 2.5E-08):
257             ksh = 1100
258         if (kf > 7.5E-08):
259             ksh = 1000
260         if (kf > 2.5E-07):
261             ksh = 900
262         if (kf > 7.5E-07):
263             ksh = 800
264         if (kf > 2.5E-06):
265             ksh = 700
266         if (kf > 7.5E-06):
267             ksh = 600
268         if (kf > 2.5E-05):
269             ksh = 500
270         if (kf > 7.5E-05):
271             ksh = 400
272         if (kf > 2.5E-04):
273             ksh = 300
274         if (kf > 7.5E-04):
275             ksh = 200
276
277
278         template = "" <layer>
279         <id>{id}</id>
280         <bez>{layer_kind}</bez>
```

```

281         <dicke>{size}</dicke>
282         <ts>{ts}</ts>
283         <kf>{kf}</kf>
284         <swg>{swg}</swg>
285         <fk>{fk}</fk>
286         <pwp>{pwp}</pwp>
287         <awg>{awg}</awg>
288         <ksh>{ksh}</ksh>
289         <grad>{grad}</grad>
290         <drain>false</drain>
291     </layer>"".format(
292         id = id,
293         layer_kind = soil_kind,
294         size = size,
295         grad = grad,
296         ts = ts,
297         kf=kf,
298         swg = sat,
299         fk = fk1,
300         pwp = pwp,
301         awg = fk1,
302         ksh = ksh
303     )
304     return template
305
306 def process_bowa_soil_file(layer, bk50_legend):
307     bk50_entry = layer[1]['BK50_LEG_N']
308     slope = layer[1]['slope_mean']
309
310     layers = bk50_legend[bk50_legend['LEG_NR'] == bk50_entry]
311     out = "<?xml version='1.0' encoding='UTF-8'?>"
312     <soil>
313         <version>0.4</version>""
314
315         for id, layer in enumerate(layers.iterrows()):
316             name = layer[1]["BOTYP"]
317             size = layer[1]["UTIEF"]- layer[1]["OTIEF"]
318             soil_kind = layer[1]["BOART"]
319             humus = layer[1]["Humus"]
320             out += "\n" + bowa_layer_template(id, size, slope, soil_kind, humus)
321
322         out += "\n</soil>"
323     return out
324
325 shapefile = gpd.read_file("/tmp/input/hydrotopes_joined_with_bk50.shp")
326 shapefile = shapefile.to_crs('EPSG:4326') # to wgs 84
327 bk50_legend = pd.read_excel(
328     "/tmp/input/BK50_LBF_202005225.xlsx",
329     sheet_name="Horizont"
330 )
331

```



```

332     for row in shapefile.iterrows():
333         id = row[0]
334         name = row[1]['Name']
335         area = row[1]['area']
336         bwmlayers_file_name = f'/tmp/out/{id}_{name}.bwmlayers'
337         with open(bwmlayers_file_name, 'w') as bwmlayers_file:
338             bwmlayers = process_bowa_soil_file(row, bk50_legend)
339             bwmlayers_file.write(bwmlayers)
340
341     EOF
342
343 - command: |
344     python /tmp/convert_to_bwmlayers.py
345     tar --sort=name --mtime='UTC 2021-12-31' -cf /tmp/bwmlayers.tar /tmp/out/*
346 output:
347 - file:
348     /tmp/bwmlayers.tar

```

Listing D.28.: Definition of the generate bwmlayer files construction step

D.17. Run Bowahald

The run bowahald construction step calculates the hydrologic balance models for each hydrotop based on the provided inputs. This construction step expects 5 different input datasets:

- A bwmcor file `korr.bwmcor` containing information about how the precipitation data should be corrected
- A Microsoft Excel file `metdata.xls` containing aggregated climate data for the model time period in a BoWaHald compatible format
- A TAR archive containing a bwmhydro file per hydrotop (`bwmhydro.tar`)
- A TAR archive containing a bwmuse file per hydrotop (`bwmuse.tar`)
- A TAR archive containing a bwmlayers file per hydrotop (`bwmlayers`)

This construction step produces a TAR archive, that contains the BoWaHald simulation results as Microsoft Excel file per hydrotop. The data processing performed by this construction step is done by BoWaHald [30]. Listing D.29 contains the construction step definition as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Unpack the provided TAR archives (Line 5 - 7)
2. Create a output directory (Line 8)
3. Loop over the hydrotopes and run BoWaHald for each hydrotop (Line 9 - 21)
4. Create a TAR archive based on the simulation output of all hydrotops. It is important here to explicitly set a sorting order and a timestamp because otherwise the `tar` command may introduce non-reproducible behaviour. (Line 22)

```

1  image: local-registry:5000/bowahald
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        (cd /tmp/input && tar -xf bwmuse.tar)
6        (cd /tmp/input && tar -xf bwmlayers.tar)
7        (cd /tmp/input && tar -xf bwmhydro.tar)
8        mkdir /tmp/out
9        for f in /tmp/input/tmp/out/*.bwmhydro
10       do
11           filename="${f%.*}"
12           echo "($filename.bwmhydro, $filename.bwmlayers, $filename.bwmuse)"
13
14           java -jar /bowahald.jar \
15               --hydro-file "$filename.bwmhydro" \
16               --met-file /tmp/input/metdata.xls \
17               --corr-file /tmp/input/korr.bwmcrr \
18               --use-file "$filename.bwmuse" \
19               --soil-file "$filename.bwmlayers" \
20               --itter true --output /tmp/out
21       done
22       tar --sort=name --mtime='UTC 2021-12-31' -cf /tmp/results.tar /tmp/out/*
23  output:
24    - file:
25        /tmp/results.tar

```

Listing D.29.: Definition of the run bowahald construction step

D.18. Plotting results

The plotting results construction step post-processes the BoWaHald simulation results to create some summary plots. It expects a TAR archive containing BoWaHald simulation results as input and produces several plots as PNG images as output. The plot creating is done by a self written R script. Listing D.30 contains the construction step definition as used by section 7.4. The script performed as part of this construction step consists of the following operations:

1. Unpack the TAR archive, that contains a Microsoft Excel file per hydrotop (Line 5)
2. Create a output directory (Line 6)
3. Define the R scripted used to create the plots (Line 8 - 109)
4. Execute the previously defined R script (Line 111)

```

1 image: local-registry:5000/wasserhaushalt_r_processing_image
2 input_path: "/tmp/input"
3 operation:
4   - command: |
5       (cd /tmp/input && tar -xf results.tar)
6       mkdir /tmp/out
7   - command: |
8       cat > /tmp/generate_summary_plots.R << 'EOF'
9
10      options(java.parameters = "-Xmx8g")
11      library(xlsx)
12      library(dplyr)
13      library(ggplot2)
14
15      input_path <- "/tmp/input/tmp/out"
16      output_path <- "/tmp/out"
17      all_result_files <- list.files(path = input_path, pattern = "*.xls")
18
19      results <- lapply(all_result_files, function(i) {
20          r <- strsplit(i, split = "_")
21          r <- r[[1]]
22          r <- tail(head(r, -2), -3)
23
24          WHB_Monat <- read.xlsx(
25              paste(input_path, i, sep = "/"), "WHB (komplett) Monat"
26          )
27          WHB_Monat <- WHB_Monat[, c(1, 2, 3, 5, 10, 13)]
28          WHB_Monat$hydrotop <- paste(r, collapse = "_")
29          WHB_Monat
30      })
31
32      results <- do.call("rbind.data.frame", results)
33
34      order_index <- function(i) {
35          r <- strsplit(i, split = "_")
36          return(as.integer(r[[1]][1]))
37      }
38      hydrotopes <- unique(results$hydrotop)
39      order <- order(sapply(hydrotopes, order_index))
40      hydrotopes <- hydrotopes[order]
41
42      results$hydrotop <- factor(results$hydrotop, levels = hydrotopes)
43      results$month <- as.integer(format(results$DAT, "%m"))
44      names(results)[2] <- "P"
45      names(results)[3] <- "ETR"
46      names(results)[4] <- "ETP"
47      names(results)[5] <- "RO"
48      names(results)[6] <- "RU"
49

```

```

50 generate_median_plot <- function(data, name, plot_title, ylab_title, var) {
51   ggplot(
52     data, aes(x = month, y = {{ var }}, group = hydrotop, color = hydrotop)
53   ) +
54     xlab("Month") +
55     ylab(ylab_title) +
56     scale_x_continuous(
57       labels = c(
58         "JAN", "FEB", "MAR", "APR", "MAY", "JUN",
59         "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"
60       ),
61       breaks = c(1:12)
62     ) +
63     theme_bw() +
64     theme(legend.position = "bottom", legend.box = "vertical") +
65     theme(legend.margin = margin()) +
66     theme(plot.title = element_text(size = 14, hjust = 0.5)) +
67     ggtitle(plot_title) +
68     stat_summary(fun = median, geom = "line") +
69     stat_summary(fun = median, geom = "point") +
70     guides(color = guide_legend(ncol = 1)) +
71     labs(color = "Hydrotop")
72
73     ggsave(paste(output_path, name, sep = "/"),
74           height = 7 + length(unique(results$hydrotop)) * 0.2)
75   }
76
77 generate_median_plot(results, "etp.png",
78   "Median Potential Evaporation", "ETP [mm/month]", ETP)
79 generate_median_plot(results, "etr.png",
80   "Median Real Evaporation", "ETR [mm/month]", ETR)
81 generate_median_plot(results, "ro.png",
82   "Median Run Off", "RO [mm/month]", RO)
83 generate_median_plot(results, "ru.png",
84   "Median Underground Drain", "RU [mm/month]", RU)
85
86
87 results %>%
88   group_by(month) %>%
89   summarise(P = median(P)) %>%
90   ggplot(aes(x = month, y = P)) +
91   xlab("Month") +
92   ylab("P [mm/month]") +
93   scale_x_continuous(
94     labels = c(
95       "JAN", "FEB", "MAR", "APR", "MAY", "JUN",
96       "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"
97     ),
98     breaks = c(1:12)
99   ) +
100   theme_bw() +

```

```

101     theme(legend.position = "bottom", legend.box = "vertical") +
102     theme(legend.margin = margin()) +
103     theme(plot.title = element_text(size = 14, hjust = 0.5)) +
104     ggtitle("Median Percipitation") +
105     geom_line() +
106     geom_point()
107
108     ggsave(paste(output_path, "percipation.png", sep = "/"))
109 EOF
110 - command: |
111     R --no-save < /tmp/generate_summary_plots.R
112 output:
113 - file:
114     /tmp/out/etp.png
115 - file:
116     /tmp/out/etr.png
117 - file:
118     /tmp/out/ro.png
119 - file:
120     /tmp/out/ru.png
121 - file:
122     /tmp/out/percipation.png

```

Listing D.30.: Definition of the plotting results construction step

D.19. Summarise

The summarise construction step generate a short summary based on the BoWaHald simulation results. It expects a TAR archive containing a Microsoft Excel file per hydrotop with the corresponding simulation results as input. The construction step produces a CSV file containing some summary statistic as output. The processing is performed by a self written R script. Listing D.31 contains the definition of the construction step as used by the case study presented in section 7.4. The script performed as part of this construction step consists of the following operations:

1. Unpack the TAR archive, that contains a Microsoft Excel file per hydrotop (Line 5)
2. Create a output directory (Line 6)
3. Define the R scripted used to create the summary statistics (Line 8 - 46)
4. Execute the previously defined R script (Line 48)

```

1  image: local-registry:5000/wasserhaushalt_r_processing_image
2  input_path: "/tmp/input"
3  operation:
4    - command: |
5        (cd /tmp/input && tar -xf results.tar)
6        mkdir /tmp/out
7    - command: |
8        cat > /tmp/generate_summary.R << 'EOF'
9        options(java.parameters = "-Xmx8g")
10       library(xlsx)
11       library(dplyr)
12
13       input_path = "/tmp/input/tmp/out"
14       all_result_files <- list.files(path = input_path, pattern = "*.xls")
15
16       results = lapply(all_result_files, function(i) {
17         r = strsplit(i, split = "_")
18         r = r[[1]]
19         r = tail(head(r,-2), -3)
20
21         WHB_Jahr = read.xlsx(paste(input_path, i, sep="/"), "WHB (komplett) Jahr" )
22         WHB_Jahr = WHB_Jahr[,c(1,2,3,5,10,13)]
23         WHB_Jahr$hydrotop = paste(r, collapse = "_")
24         WHB_Jahr
25       })
26
27       results <- do.call("rbind.data.frame", results)
28       results$hydrotop <- as.factor(results$hydrotop)
29       results$year <- as.integer(format(results$DAT, "%Y"))
30       names(results)[2] <- "P"
31       names(results)[3] <- "ETR"
32       names(results)[4] <- "ETP"
33       names(results)[5] <- "RO"
34       names(results)[6] <- "RU"
35
36       summary_results <- results %>% group_by(hydrotop) %>% summarise(
37         min_P = min(P), median_P = median(P), max_P = max(P),
38         min_ETR = min(ETR), median_ETR = median(ETR), max_ETR = max(ETR),
39         min_ETP = min(ETP), median_ETP = median(ETP), max_ETP = max(ETP),
40         min_RO = min(RO), median_RO = median(RO), max_RO = max(RO),
41         min_RU = min(RU), median_RU = median(RU), max_RU = max(RU),
42       )
43
44       write.csv(summary_results, "/tmp/out/summary.csv")
45
46       EOF
47    - command: |
48        R --no-save < /tmp/generate_summary.R
49  output:

```

```
50 - file:
51     /tmp/out/summary.csv
```

Listing D.31.: Definition of the summarise construction step

E. GeoHub User Manual

This guide aims to give a short overview over all functionality provided by GeoHub

E.1. Overview

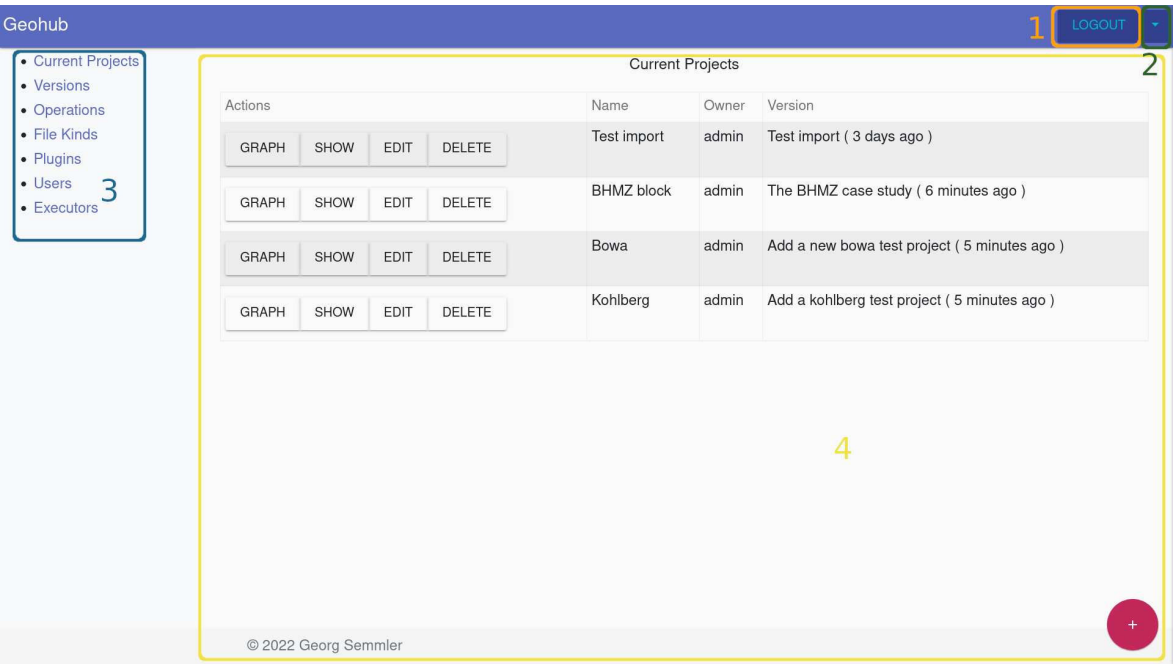


Figure E.2.: Main View

Figure E.2 shows the application main screen after a successful login.

There are main 4 components of the application visible here:

- 1. Logout button, which logs out the current user
- 2. Settings drop down menu. See section E.1.1 for details
- 3. Side panel to choose different main views. See section E.1.2 for details.
- 4. Main view, dependent on side panel option chosen. See section E.2 for details.

E.1.1. Settings drop down menu



Figure E.3.: Settings drop down menu

The setting drop down menu offers several options:

1. A logout button to logout the current user
2. An option to change the password of the current user. This opens the dialog shown in figure E.4
3. An option to show information about the current software version. This opens the dialog shown in figure E.5

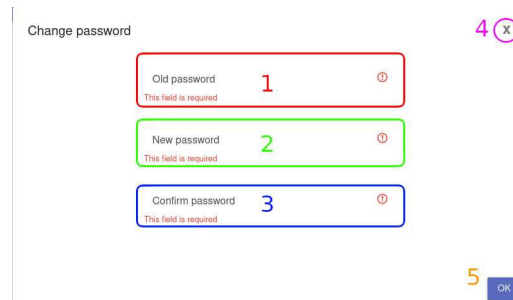


Figure E.4.: Change password dialog

Figure E.4 show the dialog used to change a user password. The dialog has 3 mandatory fields numbered with 1-3 in the picture. Text field 1 needs to be filled with the current password, while text field 2 and 3 should contain the new password. The Close Button (4) allows to cancel the password change, while the OK button (5) submits the change.

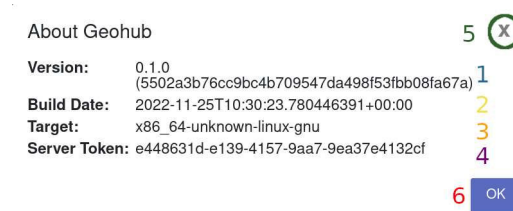


Figure E.5.: About GeoHub dialog

Figure E.5 show the “About GeoHub” dialog which displays information about the currently used GeoHub version. Field 1 contains information about the used software version and from which git revision the software was build. Field 2 contains the build date, field 3 contains the target platform. Field 4 contains the server token, which can be used to register additional executors. See section E.5 for details. Button 5 and 6 can be used to close the dialog.

E.1.2. Side panel

Figure E.6 shows the GeoHub side panel. This control panel allows you to switch between the different main views. There are the following options available:

1. Project view, see section E.2 for details
2. Version view, see section E.3 for details
3. Operation view, see section E.4 for details
4. File Kind view, see section E.7 for details
5. Plugin view, see section E.8 for details
6. User view, see section E.9 for details. This view is only available for administrative users.
7. Executor view, see section E.5 for details



Figure E.6.: Side panel

E.2. Project View

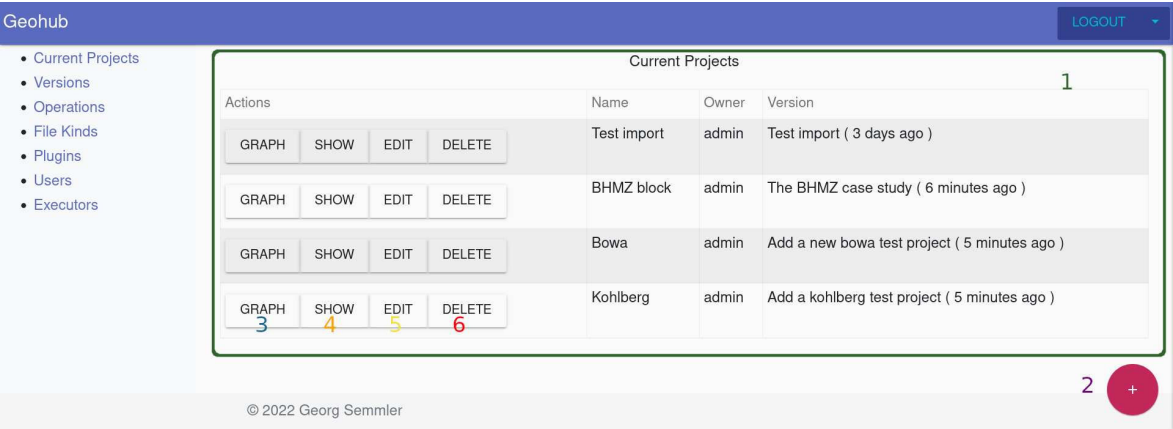



Figure E.7.: Project View

Figure E.7 gives you an overview over the project view page. A project is GeoHubs basic unit to address a complete construction workflow. This view gives you the ability to manage existing projects or create new projects. Table 1 in that figure lists all projects currently visible for the logged in user. The **Name** column of the table contains the project name, the **Owner** column the name of the user that created this project and the **Version** column lists subject and time of the last change to this project. Button 2 allows you to create new projects. This will open the “Create Project” dialog shown in figure E.8. Button 3 will open the corresponding construction graph for the given project. This view is explained in detail in section E.10. Button 4 will open the project information dialog shown in figure E.9. Button 5 will open the modify project dialog shown in figure E.10. Button 6 will open the delete project dialog shown in figure E.11.

Figure E.8 shows the dialog used to create new projects. The text field 1 allows you to specify the name of the newly created project. This field is required. Text field 2 allows you to specify an optional description. The content of this text field is interpreted as markdown. Text field 3 allows you to specify a set of required metadata keys. Metadata values for keys specified here are required for any dataset uploaded later into this project. Keys are separated by ,. Text field 4 allows you to set the subject of the version used to create the new project. This field is required. Text field 5 allows you to specify an optional description for this version. The content of this text field is interpreted as markdown. Button 6 allows you to cancel this operation, while button 7 creates the new project.

Figure E.9 shows the project info dialog that shows some details about a specific project.


Create Project

6 

1 
Name
This field is required

2
Description

3
Enforced Node Attributes

4 
Version Subject
This field is required

5
Version Description

7

SAVE CHANGES

Figure E.8.: Create project

BHMZ

4 

1
Name: BHMZ

2
Description: This is a BHMZ test project
• Something
• Some other thing

3
Enforced attributes: author location

5
OK

Figure E.9.: Project Info Dialog

Field 1 contains the name of the project, Field 2 the corresponding description interpreted as markdown. Field 3 shows the list of required metadata keys. Button 4 and 5 are used to close the dialog.

6

1

2

3

4

5

7

SAVE

Figure E.10.: Project Edit Dialog

Figure E.10 shows the edit project dialog. This dialog allows to modify existing projects. It contains the same fields and controls as the create project dialog shown in figure E.8. These fields contain the current value as initial value.

3

1

2

4

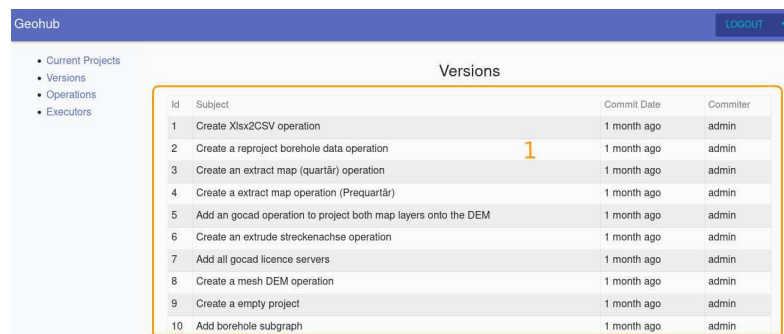
DELETE

Figure E.11.: Delete project Dialog

Figure E.11 shows the delete project dialog used to delete an existing project. Text field 1 allows you to set the subject of the version used to delete the project. This field is required. Text field 2 allows you to specify an optional description for this version. Button 3 allows you to cancel this operation, while button 4 submits the delete request.

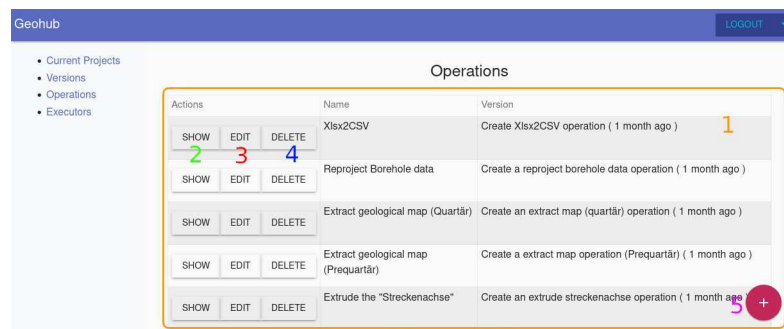
E.3. Version View

Figure E.12 shows the version view page. A version is the central unit of editor time in GeoHub and tracks how stored entities evolve over time. Table 1 just listed all versions stored in the database to give users an overview. The **Subject** column contains the subject of the corresponding version, the **Commit Date** column contains information about when a version was applied to the stored data and the **Committer** column specifies who has committed the version.



Id	Subject	Commit Date	Committer
1	Create Xlsx2CSV operation	1 month ago	admin
2	Create a reproject borehole data operation	1 month ago	admin
3	Create an extract map (quartär) operation	1 month ago	admin
4	Create a extract map operation (Prequartär)	1 month ago	admin
5	Add an gocad operation to project both map layers onto the DEM	1 month ago	admin
6	Create an extrude streckenachse operation	1 month ago	admin
7	Add all gocad licence servers	1 month ago	admin
8	Create a mesh DEM operation	1 month ago	admin
9	Create a empty project	1 month ago	admin
10	Add borehole subgraph	1 month ago	admin

Figure E.12.: Version view

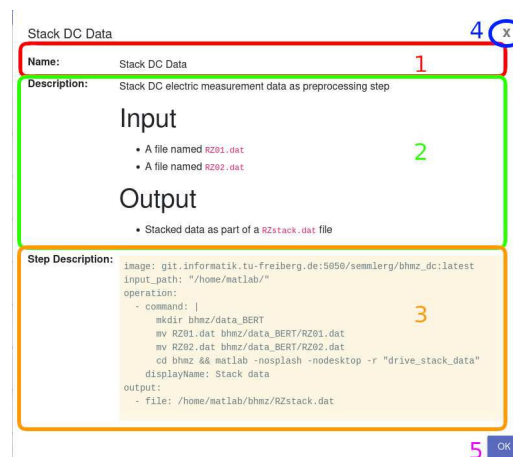


Actions	Name	Version
SHOW EDIT DELETE	Xlsx2CSV	Create Xlsx2CSV operation (1 month ago)
SHOW EDIT DELETE	Reproject Borehole data	Create a reproject borehole data operation (1 month ago)
SHOW EDIT DELETE	Extract geological map (Quartär)	Create an extract map (quartär) operation (1 month ago)
SHOW EDIT DELETE	Extract geological map (Prequartär)	Create a extract map operation (Prequartär) (1 month ago)
SHOW EDIT DELETE	Extrude the "Streckenachse"	Create an extrude streckenachse operation (1 month ago)

Figure E.13.: Operations View

E.4. Operations View

Figure E.13 shows the operation view page. An operation in GeoHub is defined as reusable construction step. Operations can be shared between different construction graphs and projects. Table 1 lists all available operations. The **Name** column contains the operation name, the **Version** column contains subject and date of the last change to this operation. Button 2 opens the operation information dialog shown in figure E.14. Button 3 allows you to edit an existing operation via the operations edit dialog shown in figure E.15. Button 4 allows you to delete an existing operation by opening the dialog shown in figure E.16. Button 5 allows you to create new operations using the “Create Operations” dialog shown in figure E.17.



Name: Stack DC Data

Description: Stack DC electric measurement data as preprocessing step

Input

- A file named R291.dat
- A file named R292.dat

Output

- Stacked data as part of a R2stack.dat file

Step Description:

```
image: git.informatik.tu-freiberg.de:5950/semmlerg/bhmz_dc:latest
input_path: "/home/matlab/"
operation:
  - command: |
    mkdir bhmz/data_BERT
    mv R291.dat bhmz/data_BERT/R291.dat
    mv R292.dat bhmz/data_BERT/R292.dat
    cd bhmz && matlab -nosplash -nodesktop -r "drive_stack_data"
  displayName: Stack data
output:
  - file: /home/matlab/bhmz/R2stack.dat
```

OK

Figure E.14.: Operations info dialog

Figure E.14 shows the Operations info dialog that contains a summary of information about a

given operation. Field 1 contains the name of the current view operation. Field 2 contains an optional description interpreted as markdown. Field 3 contains the operation step description in the yaml format described in section E.4.1. Button 4 and 5 can be used to close the dialog.

Figure E.15 shows the 'Operations edit dialog' for editing an existing operation named 'Xlsx2CSV'. The dialog contains the following fields and controls:

- Field 1 (Name):** A text input field containing 'Xlsx2CSV'.
- Field 2 (Description):** A text input field containing '# Converts a Excel File into a CSV file'.
- Field 3 (Step YAML):** A text input field containing a YAML snippet:


```
Step YAML:
image: git:informatik.tu-freiberg.de:5050/seminlrg/qgis-processing-
image
input_path: "/tmp/input"
operation:
- command: |
```
- Field 4 (Version Subject):** A text input field that is empty, with a red error message 'This field is required' below it.
- Field 5 (Version Description):** A text input field that is empty.
- Buttons:** A 'SAVE' button at the bottom right and a close button (X) at the top right.

Figure E.15.: Operations edit dialog

Figure E.15 show the Operations edit dialog that allows to edit an existing operation. Text field 1 allows to change the name of the operation. This text field must not be empty. Text field 2 allows to change the description for the current operation. The text is interpreted as markdown. Text field 3 contains the operation step specification in the yaml format described in section E.4.1. This field must not be empty and the content must follow the described format. Text field 4 allows to specify the version subject associated with the corresponding change. This field is required. Text field 5 allows to provide an optional version description. Button 6 allows to cancel the edit operation, while button 7 saves the changes.

Figure E.16 shows the 'Operations delete dialog' for deleting an existing operation named 'Xlsx2CSV'. The dialog contains the following fields and controls:

- Field 1 (Version Subject):** A text input field that is empty, with a red error message 'This field is required' below it.
- Field 2 (Version Description):** A text input field that is empty.
- Buttons:** A 'DELETE' button at the bottom right and a close button (X) at the top right.

Figure E.16.: Operations delete dialog

Figure E.16 shows the delete operation dialog used to delete an existing operation. Text field 1 allows you to set the subject of the version used to delete the operation. This field is required. Text field 2 allows you to specify an optional description for this version. Button 3 allows you to cancel this operation, while button 4 submits the delete request.

Figure E.17 shows the create operation dialog used to create a new operation. Text field 1 allows to specify the name of the new operation. This field is required. Text field 2 allows to provide an optional description. The text is interpreted as markdown. Text field 3 allows to specify the operation step specification in the yaml format described in section E.4.1. This field must not be empty and the content must follow the described format. Text field 4 allows to specify the subject of the version used to create this operation. This field is required. Text field 5 allow to provide an optional version description. Button 6 allows to close the dialog without creating a new operation. Button 7 creates a new operation using the details specified in text field 1-5.

Create Operation

6

1 Name This field is required

2 Description

3 Step YAML

4 Version Subject This field is required

5 Version Description

7

Figure E.17.: Operations create dialog

E.4.1. Construction step specification yaml format

Construction steps are specified by a yaml document. Yaml documents consist of key-value pairs of various types.

```

1 image: python:3.9
2 input_path: /path/to/write/input
3 operation:
4   - command: |
5       echo "Foo"
6       /my/super/command
7     displayName: Some step description
8     enviroment:
9       FOO: BAR
10  - command: |
11      echo "Bar"
12      /another/command
13 output:
14   - file: /path/to/file1
15   - file: /path/to/file2
16   - stdout: SomeName.ext
17   - metadata: metadata_file.json

```

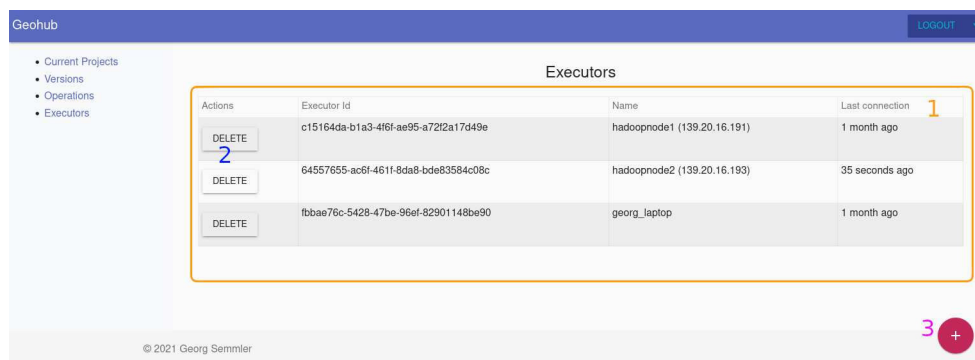
Listing E.32.: Listing: Exemplary construction step definition

Listing E.32 contains a complete exemplary construction step specification. Each specification consists of 4 required keys:

- **image** which specifies which docker image is used by this operation step. The value needs to be a string following dockers image syntax. This field is required
- **input_path** specifies where the operation step executor will place input datasets inside the image provided to the operation. This field is required and the value is required to be a string.
- **operation** specifies which commands should be executed as part of this operation. This field is required. It accepts a list of nested key value structure to describe the actual commands:

- **command** specifies the actual commands that should be executed as part of this construction step. This field is required and accepts a string value. The value will be passed as it is to the **sh** binary available inside of the docker binary, so that it's executed as shell script there.
 - **displayName** is an optional identifier for the given command. It's intended to be used to group parts of the operation log later on. This field is optional.
 - **environment** allows to specify a list of optional environment variables that should be set during executing inside of the container. Variables are defined as key value list, where each key corresponds to the variable name and each value corresponds to the variable value.
- **output** specifies which files are used as output of the current operation. It accepts a nested key value list as argument:
 - **file** specifies that the file located at the location specified by the string value is treated as operation output. The executor will attempt to download the corresponding file after the command execution is finished. If the file does not exist the operation fails. **output** is allowed to have multiple **file** entries.
 - **stdout** specifies that the command line output of all commands listed above are interpreted as operation result. The specified string value defines a file name that should be used to store the corresponding file later on.
 - **metadata** specifies which file contains additional metadata for the generated dataset. This file is assumed to contain a single JSON object. Any top level key is treated as metadata key, while values are treated as metadata values.

E.5. Executor View



Actions	Executor Id	Name	Last connection
DELETE	c15164da-b1a3-416f-ae95-a7212a17d49e	hadoopnode1 (139.20.16.191)	1 month ago
DELETE	64557655-ac6f-461f-8da8-bde83584c08c	hadoopnode2 (139.20.16.193)	35 seconds ago
DELETE	fbbae76c-5428-47be-96ef-82901148be90	georg_laptop	1 month ago

Figure E.18.: Executor view

Figure E.18 shows the executor view page. Executors are GeoHubs basic unit to actually run operation steps on. They can be distributed over multiple different systems as long it's possible to send HTTP requests to the main server system from there. This page can be used to register and manage operation step executors. Table 1 lists all registered executors. The **Executor ID** column lists the corresponding executor id, the **Name** column the name of the executor as specified on registering the corresponding executor. **Last connection** lists the time the corresponding executor last reach out to the server. Button 2 allows to remove existing executors. Button 3 allows to register new executors by opening the corresponding dialog shown in figure E.19.

Figure E.19 shows the register executor dialog that gives the necessary instruction to register a new executor. The following steps are required:

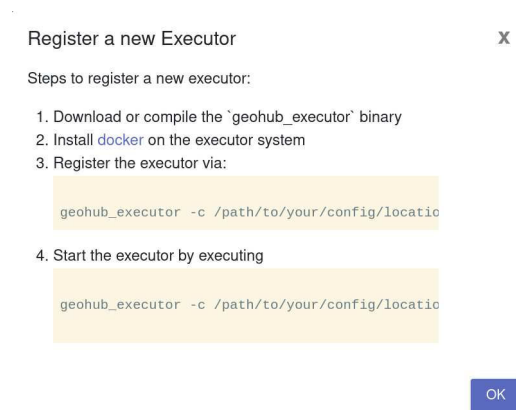


Figure E.19.: Register executor dialog

1. Get a copy of `geohub_executor` compiled for your operating system or compile it on your own based on the source code in `geohub_distributed_executor`.
2. You need to install docker onto the system that you want to use for running `geohub_executor`. `geohub_executor` must be executed with user rights that allow to start and manage docker container instances.
3. Register a new executor by running `geohub_executor -c /path/to/your/config/location.toml register " $URL_TO_GEOHUB "your-executor-name" $SERVER_TOKEN` on the system you want to run geohub executor. There are a few configurable values here:
 - `/path/to/your/config/location.toml` specifies the path to a location that can be used from `geohub_executor` to store some basic configuration. See section E.6 for details about this configuration file.
 - `$URL_TO_GEOHUB` needs to be replaced by the an URL pointing to the corresponding GeoHub server installation. If you copy the command from the register executor dialog this will automatically be filled with the correct URL for that sever.
 - `your-executor-name` allows you to specify a name for the given executor. This can later be used to identify the executor in Table 1 of the executor view page.
 - `$SERVER_TOKEN` specifies a “secret” server token used to initially authenticate the new executor against the server. This token can currently only be obtained by accessing the register executor dialog. If you copy the command from there the token will already be pre-filled.
4. Start the executor by running `geohub_executor -c /path/to/your/config/location.toml run` where `/path/to/your/config/location.toml` specifies the location of the config file as used by step 3. After this the executor will reach out for the GeoHub server periodically and ask for new operations to execute. Optional debugging output can be accessed by setting the `RUST_LOG` environment variable according to the syntax specified here before starting the executor.

E.6. Executor configuration

```
1  [[remotes]]
2  server_connection = "http://geohub.something.org"
3  executor_name = "my fancy runner"
4  executor_token = "13d70b3e-200b-4f9b-b816-7980b251120f"
5  executor_id = "f6af6023-882a-409f-bcc4-cc968438a1e6"
6
7  [remotes.executor_config.credentials."dockerregistry.something.org:5050"]
8  username = "john"
9  password = "my-secret"
10
11 [remotes.timeout]
12 secs = 14400
13 nanos = 0
14
15 [poll_time]
16 secs = 10
17 nanos = 0
```

Listing E.33.: Exemplary executor configuration

Listing E.33 contains an exemplary executor configuration. This configuration file format is based on the toml format, which allows you to specify nested key value pairs in a structured manner.

A single executor can execute operations from multiple GeoHub instances. Each configuration for a GeoHub instance is grouped under a `remotes` entry. `remotes.server_connection` specifies the GeoHub remote url. `remotes.executor_name` specifies the name of the executor as listed in Table 1 of the corresponding executor view. `remotes.executor_token` contains the secret executor token used to authenticate the executor against the corresponding server. This token should be kept secret as it allows to access possibly confidential information on the corresponding GeoHub server. `remotes.executor_id` contains the corresponding executor id as also shown in Table 1 of the executor view. These 4 configurations are generated by registering a new executor using the workflow detailed under the register executor dialog. All 4 entries are required. Additionally it's possible to store several information to authenticate the runner against different docker repositories. This can be done by creating a corresponding `remotes.executor_config.credentials."dockerregistry.something.org:5050"` entry, where `dockerregistry.something.org:5050` specifies the URL of the corresponding docker registry. The `username` field allows you to specify the corresponding user name used to log into that registry, while `password` contains the corresponding password. These information are optional. Their usage depends on what docker images are used as part of different operations. The corresponding credentials should be kept secret. `remotes.timeout` allows you to specify the maximal amount of time a whole operation can use before the executor considers the operation stuck and stops the job. If this amount of time is exceeded for a specific job the container will be shut down and the operation will be marked as failed. This configuration is optional, by default operations are allowed to run up to 20 minutes.

`poll_time` allows you to specify the time interval used to contact the server if no operation was found previously. Smaller times lead to faster execution of scheduled operations but will increase the load on the corresponding server.

E.7. File Kind View

Actions	Name	Extensions	Equality check plugin	Metadata extraction plugin
DELETE	default		No Plugin Set	No Plugin Set
DELETE	STL Geometry	stl	No Plugin Set	No Plugin Set
DELETE	Seismic Picks	txt	No Plugin Set	No Plugin Set
DELETE	Matlab DAT	dat	No Plugin Set	No Plugin Set
DELETE	Gmsh mesh	geo, msh	No Plugin Set	No Plugin Set
DELETE	Paraview model	vti, xdmf, py, stl	No Plugin Set	No Plugin Set
DELETE	Paraview Grid	vti	No Plugin Set	No Plugin Set
DELETE	XDMF	xdmf, vtu, 2, 3	No Plugin Set	No Plugin Set
DELETE	Geotiff	tif	No Plugin Set	No Plugin Set

Figure E.20.: File Kind View

Figure E.20 shows the file kind view. This view allows to manage the different kind of files accepted by GeoHub. Table 1 lists the currently registered file kinds. Button 2 allows to remove the corresponding file kind by using the dialog shown in figure E.21. Button 3 allows to register now file kinds by using the dialog shown in figure E.22.

Delete "Paraview model"

Version Subject 1 ⓘ
This field is required

Version Description 2

3 X

4 DELETE

Figure E.21.: Delete File Kind Dialog

Figure E.21 shows the delete file kind dialog. Text field 1 allows you to set the subject of the version used to delete the project. This field is required. Text field 2 allows you to specify an optional description for this version. Button 3 allows you to cancel this operation, while button 4 submits the delete request.

Figure E.22 shows the create file kind dialog. This dialog allows to register new file kinds. Text field 1 allows to set the name of the new file kind. This field is required. Text field 2 allows to specify an optional description. Text field 3 allows you to list accepted file extensions. These are separated by ,. Drop down menu 4 and 5 allow you to specify which file kind plugin should be used for comparing datasets (drop down menu 4) and extracting metadata (drop down menu 5). Text field 6 allows you to provide the required version subject. Text field 7 accepts an optional version description. Button 8 cancels the file kind creation. Button 9 creates a new file kind based on the provided input.

Create File Kind

8

Name

1

This field is required

Description

2

Extensions

3

Equality plugin

No Plugin configured

4

Metadata plugin

No Plugin configured

5

Version Subject

6

This field is required

Version Description

7

9

SAVE CHANGES

Figure E.22.: Create File Kind Dialog

E.8. Plugin View

Figure E.23 shows the plugin view, which is used to manage file type specific extensions. These extensions are provided as WASM binaries. Table 1 contains an overview of currently registered plugins. Button 2 allows to remove the corresponding plugin, if it is not used by any file kind. Button 3 displays additional information about the corresponding plugin by showing the plugin info dialog shown in figure E.24. Button 4 allows to add a new plugin via the add plugin dialog shown in figure E.25.

Figure E.24 shows additional information about specific plugins. Field 1 contains the name of the plugin. Field 2 contains an optional description rendered as markdown. Field 3 specifies which users created this plugin. Field 4 contains the creation data. Button 5 allows to download the WASM binary for this plugin. Button 6 and Button 7 close this dialog.

Figure E.25 shows the dialog used to add new plugins. Field 1 allows to specify the name of the plugin. Field 2 allows to specify an optional description. The content of this field is rendered as markdown by other dialogues. Button 3 allows to select a plugin WASM binary. A provided plugin must implement either the metadata extraction API (Listing E.34 lines 111-119) or the equality check API (Listing E.34 lines 100-109). Plugins implementing both APIs are accepted as well. Button 3 aborts the current plugin creation. Button 5 uploads the plugin to the server.

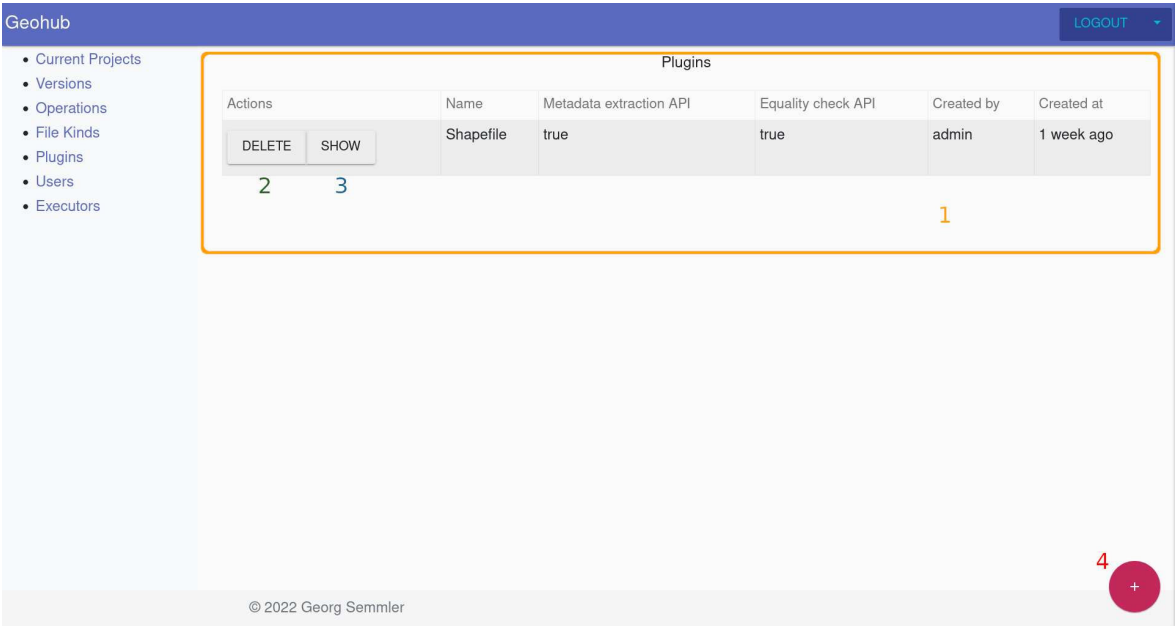


Figure E.23.: Plugin View

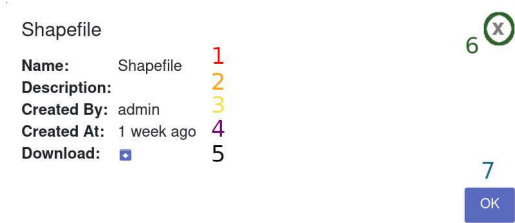


Figure E.24.: Plugin Info Dialog

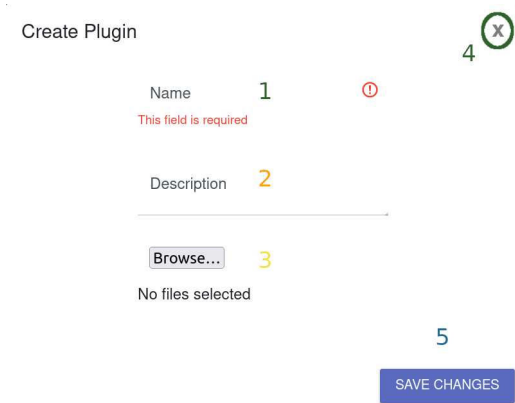


Figure E.25.: Plugin Add Dialog

```
1  struct filehandler_t {
2      int id;
3  };
4
5  // String struct
6  struct string_t {
7      char *text;
8      unsigned int len;
9  };
10
11 // Bytes for the bytestream of our file
12 struct bytes_t {
13     unsigned char *bytes;
14     unsigned int len;
15 };
16
17 // Data-Struct for the metadata we want to extract
18 struct data_t {
19     string_t key;
20     string_t value;
21 };
22
23 // File-Struct containing the name of the file and the content
24 struct file_t {
25     string_t name;
26     bytes_t content;
27 };
28
29 // Struct for the file slice
30 struct wasm_file_slice {
31     file_t *file;
32     unsigned int len;
33 };
34
35 // Struct for handling multiple files
36 struct files_t {
37     wasm_file_slice files;
38 };
39
40 // Struct fo the data slice
41 struct data_slice_t {
42     data_t *data;
43     unsigned int len;
44 };
45
46 // Metadata struct
47 struct metadata_t {
48     data_slice_t data_slice;
49 };
```

```

50
51 // Our parse_files function has two possible return-values:
52 // An Error or a positive file return result
53 // The "positive" response containing the files and the metadata
54 struct parse_file_response_t {
55     metadata_t metadata;
56 };
57
58 // The "negative" result containing an error message
59 struct parse_file_error_t {
60     string_t msg;
61 };
62
63 // The load_files function has two possible return-Values:
64 // An Error or a positive file return result
65 // The "negative" result containing an error message
66 struct compare_files_error_t {
67     string_t msg;
68 };
69
70 // The "positive" result containing the files
71 struct load_file_response_t {
72     files_t files;
73 };
74
75 // Result-Tag as enum
76 enum result_tag : uint8_t { Ok = 1, Err = 0 };
77
78 // Struct that gets returned from the parse_files function
79 struct __attribute__((packed)) parse_files_result_t {
80     int free;
81     int layout;
82     result_tag tag;
83     union {
84         struct parse_file_error_t *err_ptr;
85         struct parse_file_response_t *ok_ptr;
86     } data;
87 };
88
89 // Struct that gets returned from the check_quality function
90 struct __attribute__((packed)) compare_files_result_t {
91     int free;
92     int layout;
93     result_tag tag;
94     union {
95         bool *ok_ptr;
96         struct compare_files_error_t *err_ptr;
97     } data;
98 };
99
100 // Method to create a new handler

```

```

101 extern "C" filehandler_t *__geohub_new_equality_check_plugin(void);
102
103 // Method to compare two datasets
104 extern "C" compare_files_result_t *
105 __geohub_check_equality(filehandler_t *handler, files_t *files_a,
106                        files_t *files_b);
107
108 // Method to destroy the handler
109 extern "C" void __geohub_destroy_equality_check_plugin(filehandler_t *handler);
110
111 // Method to create a new handler
112 extern "C" filehandler_t *__geohub_new_file_handler(void);
113
114 // Method to destroy the handler
115 extern "C" void __geohub_destroy_file_handler(filehandler_t *handler);
116
117 // Method for the parsing of the files = read out metadata
118 extern "C" parse_files_result_t *__geohub_parse_files(filehandler_t *handler,
119                                                    files_t *files);

```

Listing E.34.: API definition for the Plugin interfaces

E.9. User View

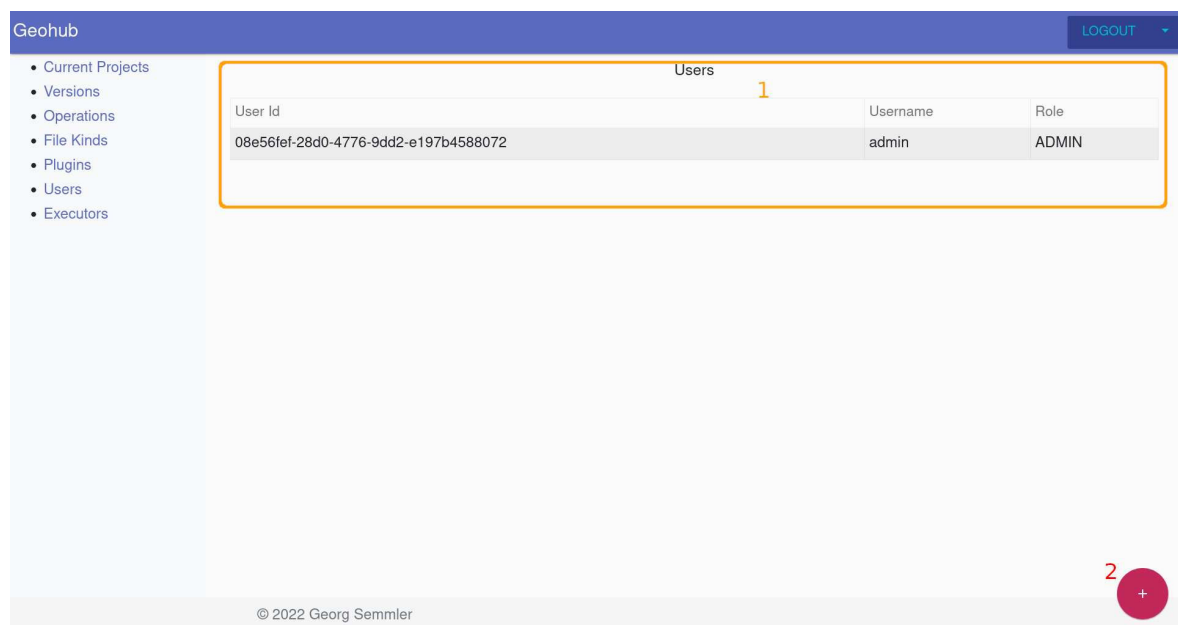


Figure E.26.: User View

Figure E.26 shows the user view. This view allows to manage currently registered users. It is only shown for users with the role ADMIN. Table 1 contains a list of current users. Button 2 allows you to add new users via the add user dialog shown in figure E.27.

Figure E.27 shows the add user dialog. This dialog allows to add new users. Text field 1 allows to set the name of the user, Text field 2 allows to specify an initial password. Text

Create User

1 Name
This field is required

2 Password
This field is required

3 Role
This field is required

5 SAVE CHANGES

Figure E.27.: Add user dialog

field 3 allows to specify the user role. This needs to be one of **ADMIN** (to specify that this user is an administrator), **WRITE** (to specify that this user is allowed to create and modify construction graphs) or **READ**. Note these roles are not strictly enforced everywhere.

E.10. Graph View

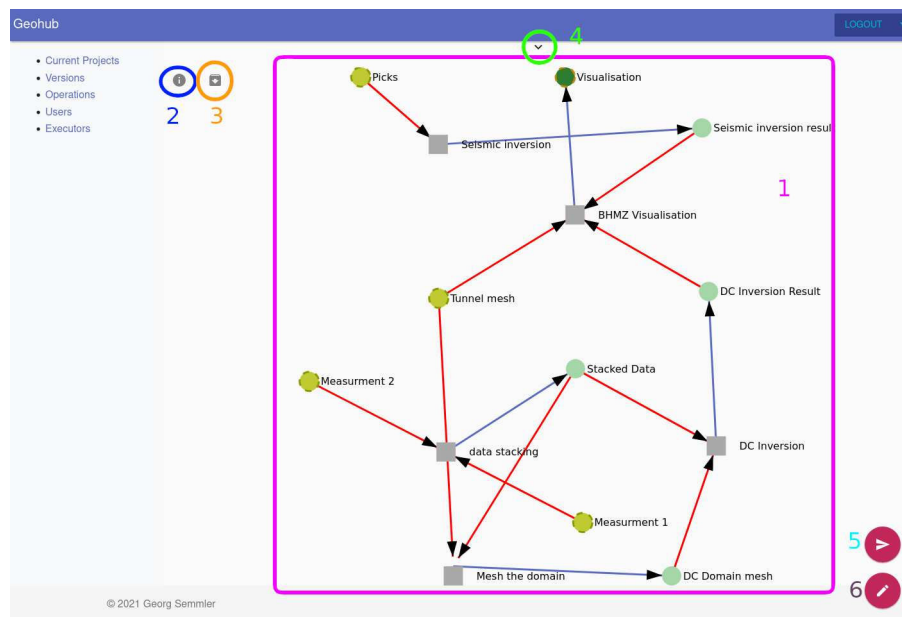


Figure E.28.: Graph view (Overview)

Figure E.28 contains a overview over the graph view. This view is one of the main views of the GeoHub frontend. It is used to construct, manipulate and interact with construction graphs. This image gives an overview over the different functionality available here. The main part of the page is occupied by the construction graph itself (marked by number 1 in the picture). This part will be explained in detail in section E.10.1. Button 2 allows you to open the legend shown in figure E.29 for the current construction graph. Button 2 allows you to download the current version of the construction graph. Button 4 allows you to expand the version slider shown in figure E.31. Button 5 switches to the Graph schedule view, which allows you to trigger actual runs of the construction graph. This view is explained in detail in section E.11. Button 6 allows you to switch in edit mode. See section E.10.2 for details.

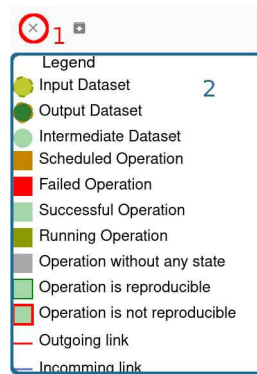


Figure E.29.: Graph legend

Figure E.29 shows the graph legend, which tries to provide some information about the colours and shapes used by the graph view. Button 1 allows you to close the legend. The block 2 contains information about the various entities used in the graph view.



Figure E.30.: Graph Export Panel

Figure E.30 shows the graph export panel. It allows you to export the currently shown graph version. Button 1 allows you to close the panel. Link 2 allows you to download the currently shown graph version as JSON. This format is designed to be used to move existing construction graph definitions to other instances. Link 3 allows you to download the currently shown graph version as PlantUML file. This can be used to generate a printable version of the graph by using PlantUML.

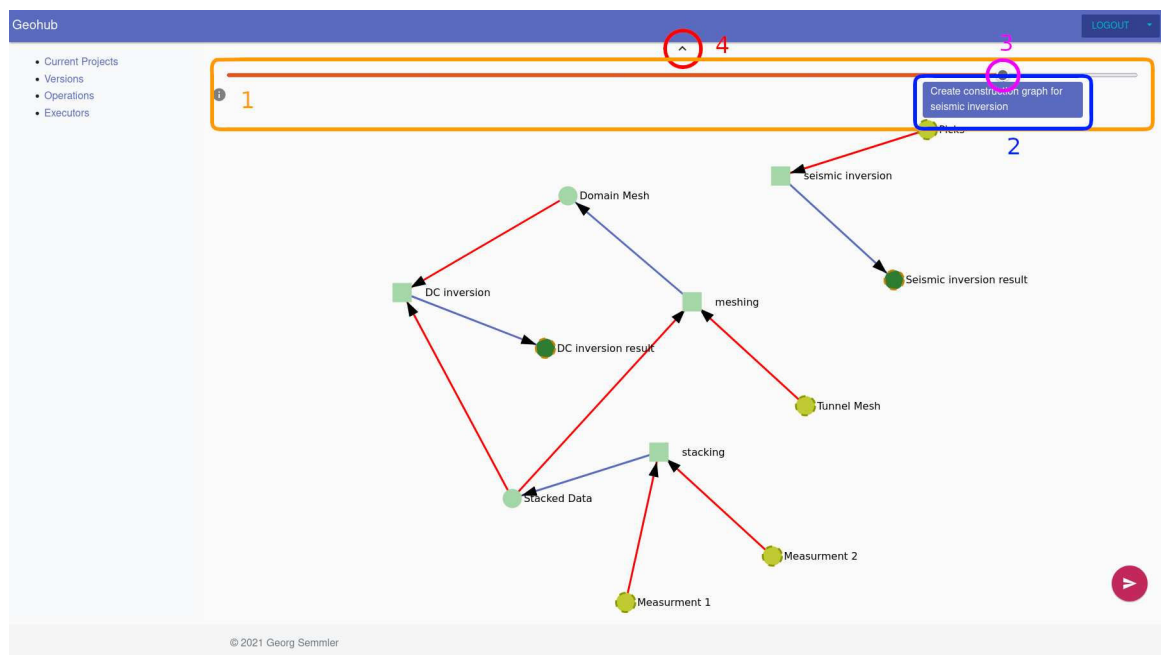


Figure E.31.: Graph view with version slider

Figure E.31 shows the graph view with expanded version slider. The version slider (1) can

be used to manipulate at which version a construction graph should be viewed. The floating label 2 shows the version subject of the currently displayed version. The slider 3 allows you to manipulate the currently shown version by moving the slider left (into the past) or right (newer versions). Button 4 allows you to collapse the version slider. Generally it is only possible to edit the last version of a construction graph, therefore the edit button (Button 5 in figure E.28) will only show up if the slider is moved to the position on the right side.

E.10.1. View mode

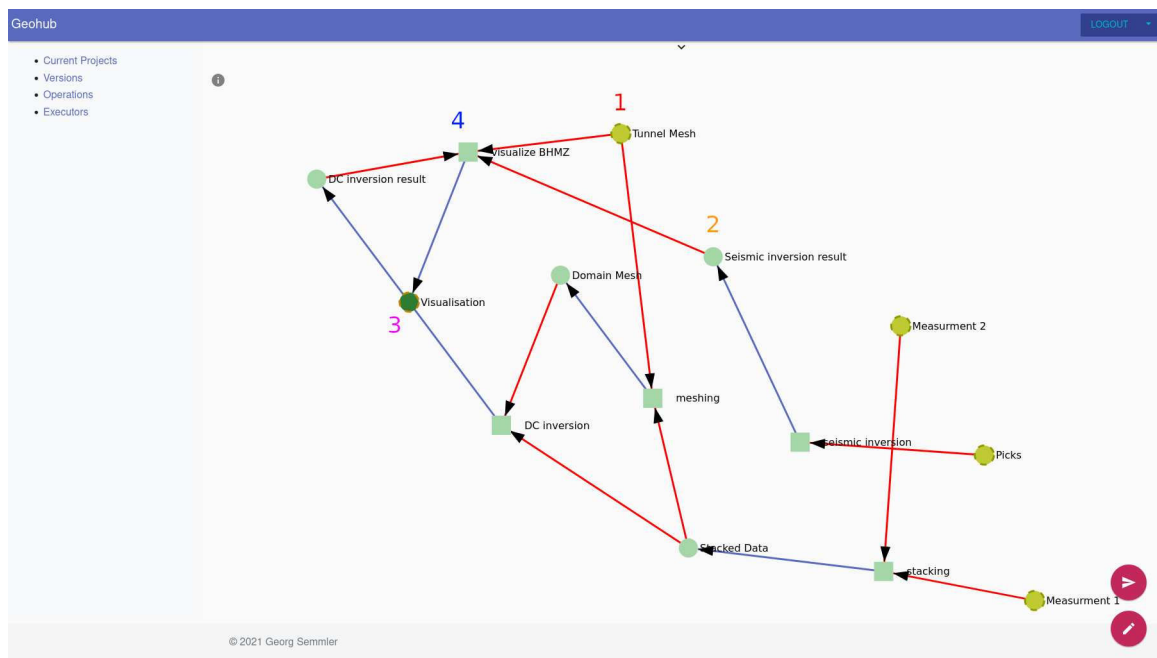


Figure E.32.: Graph view (View mode)

Figure E.32 gives you an overview over the view mode of the construction graph. This mode can be used to get an overview over the current construction graph. It cannot be used to manipulate the graph in any way. For this you need to enable the edit mode by using the edit Button (Button 5 in figure E.28). The graph view displays the construction hypergraph using different kind of edges and nodes. Each circle corresponds to a dataset (or node) in the construction hypergraph. There are different kind of nodes. Yellow nodes with dashed boarder like Node 1 represent input datasets. These are only consumed by operations. Light green nodes without boarder like node 2 represent intermediate datasets. These are produced by operations and consumed by operations. Dark green nodes with dashed boarder like node 3 represent output datasets. Those are only produced by operations. Clicking on any of those nodes will open the Node info dialog as shown in figure E.33. The text displayed next to each dataset shows the name of the corresponding dataset.

Operations (or hyperedges) of the construction hypergraph are displayed by squares like square 4. Clicking on any of those will open the Edge info dialog as shown in figure E.34. Datasets and operations are connected by arrows, which indicate their dependencies. Red arrows always point from a consumed dataset to an operation, while blue arrows point always from a operation to an produced dataset. Each operation can have multiple red arrows pointing to itself, but only one blue arrow starting at this operation. The text displayed next to each hyperedge shows the name of the corresponding hyperedge.

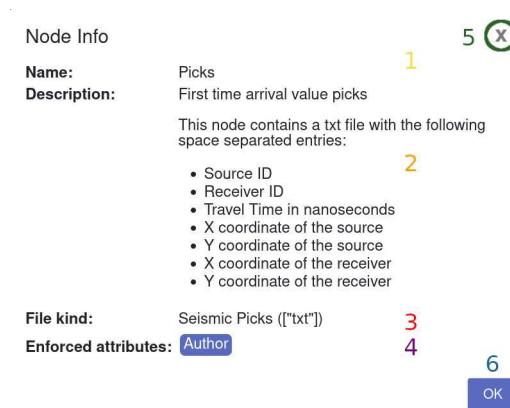


Figure E.33.: Node info dialog

Figure E.33 shows the node info dialog that displays some information about nodes in construction hypergraphs. Field 1 displays the name of the corresponding node, field 2 displays an optional description interpreted as markdown. Field 3 contains the file kind set for this specific node, field 4 contains a list of node specific required metadata keys. Button 5 and 6 allow to close the dialog.

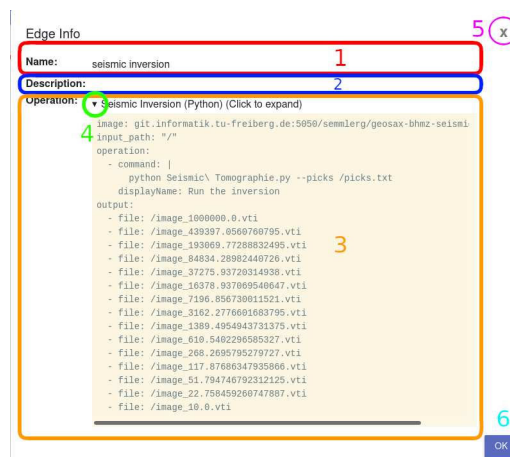


Figure E.34.: Edge info dialog

Figure E.34 shows the edge info dialog that displays some information about hyperedges in the construction hypergraph. Field 1 displays the name of the corresponding hyperedge, field 2 displays an optional description interpreted as markdown. Field 3 displays the operation step specification of the associated operation. Button 4 allows to expand and collapse the corresponding specification. Button 5 and 6 allow to close the dialog.

E.10.2. Edit mode

Figure E.35 shows the graph view in edit mode. You can recognise the edit mode by the edit bar (1). This toolbar is shown in detail in figure E.36. Button 2 allows you to exist edit mode and switch back to view mode. If there are any unsaved changes the discard changes dialog shown in figure E.43 is shown. Similarly to view mode clicking on any dataset (3) or operation (4) will open the corresponding edit dialog.

Figure E.36 shows the graph edit bar, that allows you to edit the construction graph. Button 1 allows you to add new datasets (nodes) to the construction graph by opening the add node

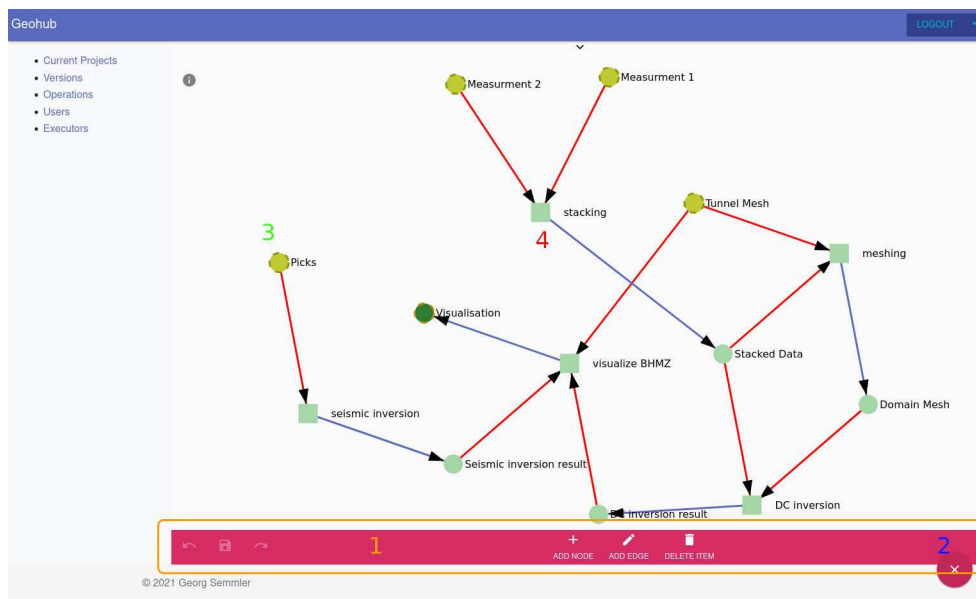


Figure E.35.: Graph view (Edit mode)



Figure E.36.: Graph edit bar

dialog shown in figure E.38.

Button 2 allows you to add new operations or add another input dataset to existing operations. By clicking this button a mode is enabled that allows you to start dragging between different nodes or operations displayed in the construction graph. If the drag starts at a dataset (node) and is released above another dataset (node) a new hyperedge is created by opening the create edge dialog shown in figure E.39. If the drag starts at a dataset (node) and ends above an existing hyperedge the dataset is added as additional input dataset for this hyperedge. Any other input is dismissed. The add edge mode can be exited by either successfully completing an operation as specified above or by using the cancel button that appears in place of button 1-3.

Button 3 allows you to remove datasets (nodes) and hyperedges by just clicking on the corresponding element. This mode can be exited by either clicking on the element you want to remove or by using the cancel button appearing in place of button 1-3.

Button 4 allows to upload an existing construction graph in ZIP format, as it can be downloaded via the graph export panel. This option is only available for otherwise empty construction graphs. It will open the dialog shown in figure E.37.

Button 5 allows you to save all changes that you've done so far (since the last save or since you started editing). It will open the save changes dialog shown in figure E.42. If this button is greyed out there are no changes to save yet.

Button 6 allows you to undo the last change you made to the construction graph. This option is available till you save the changes. If the button is greyed out there are no changes to undo yet.

Button 7 allows you to redo the last change you've undone. If this button is greyed out there is no change that could be redone yet.

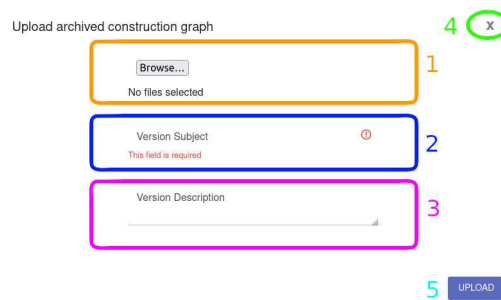


Figure E.37.: Upload archive dialog

Figure E.37 shows the upload archive dialog. It can be used to upload archived construction graphs in JSON format to an empty project. These files can be exported via the Graph Export panel. The file picker 1 can be used to select an existing JSON archive. This field is required. Text field 2 allows to specify the version subject used to import the archived crate. This field is required. Text field 3 allows to specify an optional description. Button 4 allows to close the dialog without performing any action. Button 5 submits the archive and triggers the import.

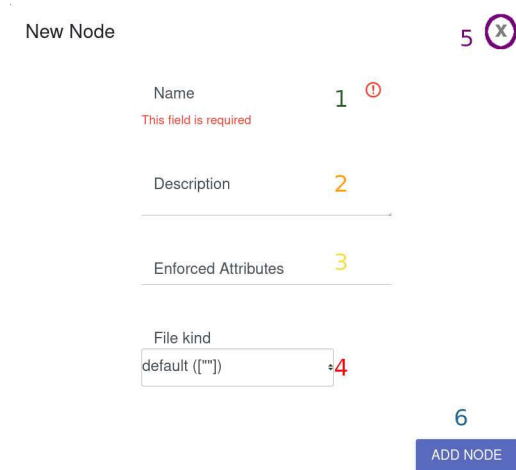


Figure E.38.: Add node dialog

Figure E.38 shows the add node dialog, which allows to add new datasets (nodes) to a construction graph. Text field 1 specifies the name of the dataset. This field is required. Text field 2 allows to provide an optional description. The content of this field is interpreted as markdown. Field 3 allows to add a dataset specific list of required metadata keys. The keys are separated by ,. Dropdown menu 4 allows to select the file kind for this dataset. Button 5 allows to close the dialog without adding a new dataset. Button 6 adds a new dataset based on the information provided in text field 1 and 2.

Figure E.39 shows the add edge dialog, which allows to add a new hyperedge to a construction graph. A hyperedge consumes a set of input datasets to produce a output dataset by using a specified operation. Text field 1 allows you to specify the name of this hyperedge. This field is required. Text field 2 allows you to provide an optional description. The content of this field is interpreted as markdown. Dropdown menu 3 allows you to select a operation, which should be associated with this hyperedge. Button 4 allows to close the dialog discarding any input, so that no new hyperedge is added. Button 5 adds a new hyperedge based on the input in text field 1 and 2 and drop down menu 3 to the construction graph.

New Edge

1 Name This field is required

2 Description

3 Operation
Xlsx2CSV

4 X

5 ADD EDGE

Figure E.39.: Add edge dialog

Edit Node

1 Name
Picks

2 Description
First time arrival value picks

3 Enforced Attributes
Author x

4 File kind
Seismic Picks (["txt"])

5 X

6 SUBMIT

Figure E.40.: Edit node dialog

Figure E.40 shows to edit node dialog, which allows you to change existing datasets (nodes). Text field 1 allows to change the name. This field is required not to be empty. Text field 2 allows you to change the description of an node. This field is optional. The content of this text field is interpreted as markdown. Field 3 allows to add a dataset specific list of required metadata keys. The keys are separated by ,. Dropdown menu 4 allows to select the file kind for this dataset. Button 5 allows you to close the dialog discarding any change. Button 6 closes the dialog and applies the change to the displayed construction graph.

Edit Edge

1 Name
stacking

2 Description

3 Operation
Stack DC Data

4 X

5 SUBMIT

Figure E.41.: Edit edge dialog

Figure E.41 shows the edit edge dialog, which allows you to change existing hyperedges. Text field 1 allows to change the name of the hyperedge. This field is required not to be empty. Text field 2 allows you to change the description of the hyperedge. This field is optional and the content is interpreted as markdown. Drop down list 3 allows you to change the associated operation. Button 4 closes the dialog discarding any changed value. Button 5

closes the dialog and applies the changes to the displayed construction graph.

Figure E.42.: Save changes dialog

Figure E.42 shows the save changes dialog. This dialog is used to upload a set of changes for a specific construction graph to the GeoHub server and associate the changeset with a given version. Text field 1 allows you to specify the subject of the associated version. This field is required. Text field 2 allows you to provide an optional description. The content of this field is interpreted as markdown. Button 3 closes the dialog without saving any changes. This brings you back into edit mode without losing any change, but also without uploading the changes to the server. Button 4 submits the changeset to the server.

Figure E.43.: Discard changes dialog

Figure E.43 shows the discard changes dialog. This dialog is shown every time you try to exist edit mode while there are unsaved changes. The dialog itself displays the number of unsaved changes. Button 1 allows you to close the dialog without discarding any changes and without leaving the edit mode. This gives you the possibility to save those changes. Button 2 allows you to discard any existing changes.

E.11. Graph Schedule View

Figure E.44 shows an overview over the graph schedule view. This view is used to manage executions of construction graphs. Similarly to the Graph view the main component of this view is occupied by the construction graph itself (1). This view is described in detail in section E.12. Button 2 again allows you to expand the version slider described in section E.10 figure E.31. Button 3 expands the revision panel showing different realisations of the construction graph. Figure E.45 shows this panel in detail. Button 4 opens the legend described in section E.10 figure E.29. Button 5 opens the Graph Export view as described in figure E.30. This allows to download the shown revision of the construction graph, including all attached datasets. Button 6 allows you to switch back to the graph view, to edit the structure of the construction graph. Button 7 switches to the edit mode described in section E.13.

Figure E.45 shows the revision panel. It contains a list of revisions (1) that are available for the current version of the construction graph. An revisions represents an execution of the construction hypergraph with a specific set of input data. The blue revision (3) is the currently selected revision, while the grey ones (like 4) are other available revisions. Clicking on any revision will switch the displayed revision. For each revision the corresponding subject, the user who scheduled the graph execution and the time of creation is listed. Button 2 allows to collapse the revision panel.

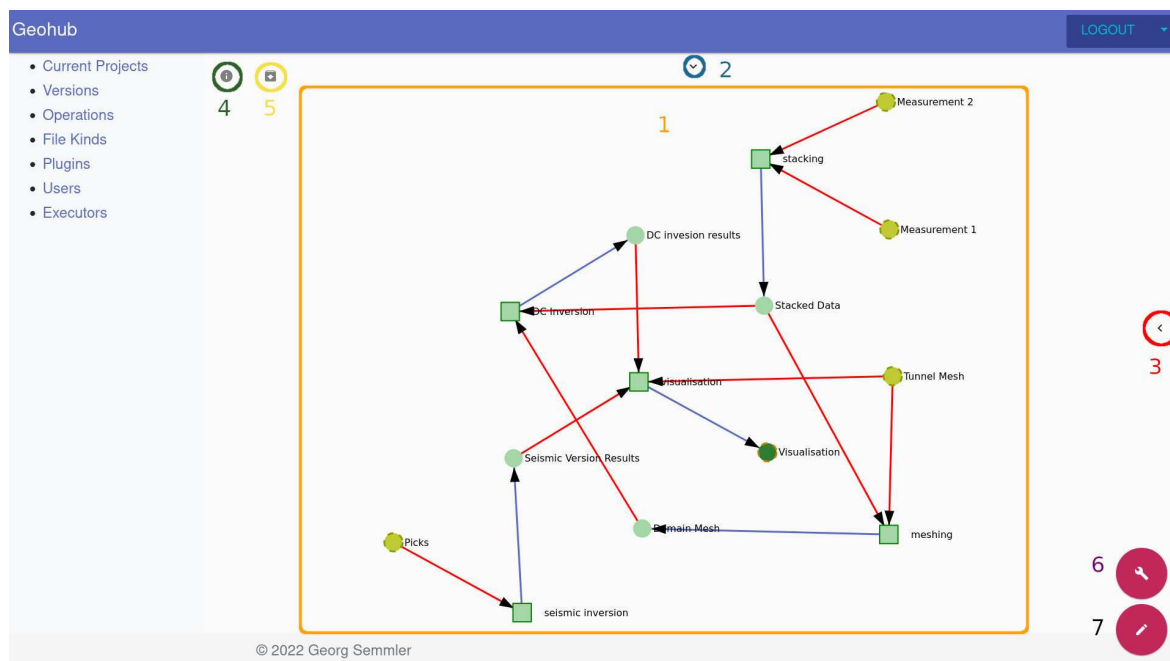


Figure E.44.: Graph Schedule View (Overview)

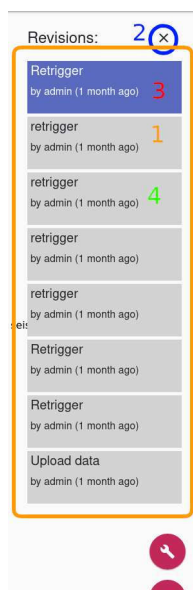


Figure E.45.: Revision panel

E.12. View mode

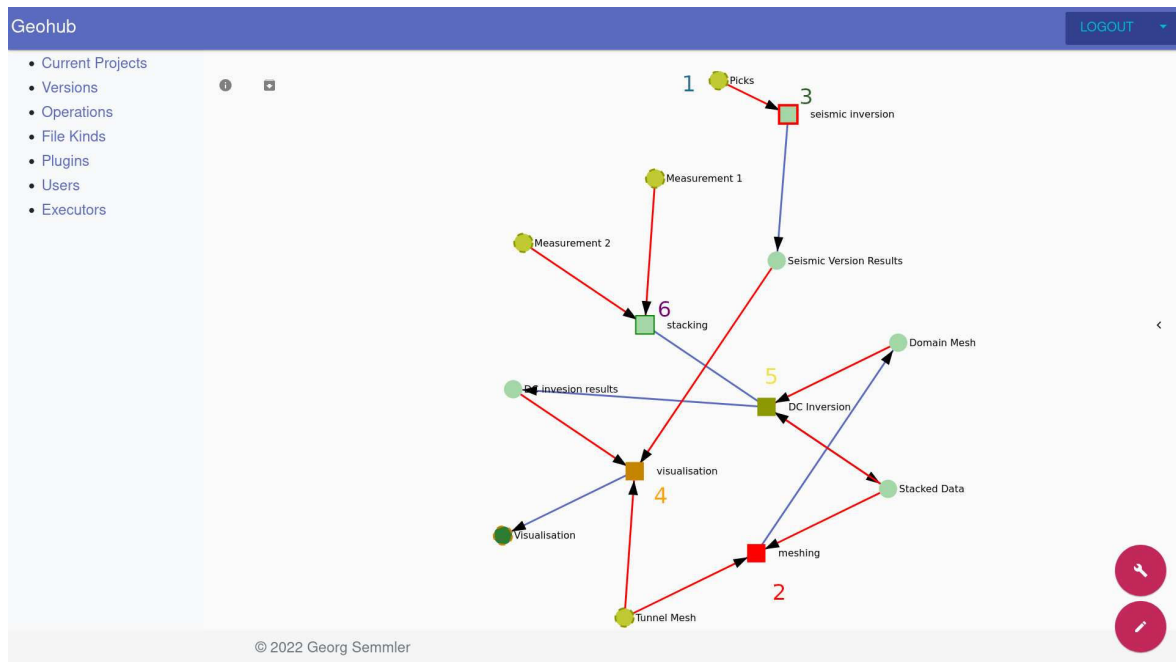


Figure E.46.: Graph Schedule View (View Mode)

Figure E.46 shows the view mode of the graph schedule view. This mode allows you to inspect existing revisions of a construction hypergraph. Similarly to the graph mode clicking on any dataset (node) (1) or hyperedge (2 - 6) will open the corresponding info dialog, which contains more detailed information about the specific entity. The colour of the hyperedges represent the state of the underlying operation. Light green (3,6) indicates that the operation was successful. Red (6) indicates a failure to execute the construction step. Checkout the execution log shown in the corresponding info dialog for more details about this failure. Orange (4) indicates that the operation is waiting to be executed. Olive (5) indicates that the operation is currently executed by one of the executors. A red outline (3) indicates that GeoHub failed to reproduce this operation. Checkout the difference between the output datasets to understand why this happened. GeoHub offers a download of the original generated datasets and the reproducing realisation for such cases. A green outline (6) indicates that a operation was reproduced successfully. See figure E.29 contains an summary of this colours and their meaning. Text near to a dataset or hyperedge represent the name of the corresponding entity.

Figure E.47 shows the node info dialog. This dialog displays details about datasets for a concrete realisation of the construction graph. The information in area 1 correspond to what is shown in the ordinary node info dialog described in figure E.33. Field 2 lists all attached files for this dataset. Each file can be downloaded by clicking the corresponding link. For images a preview is shown. The table 3 contains a list of metadata keys and their corresponding values. Button 4 and 5 closes the dialog.

Figure E.48 shows the edge info dialog. This dialog displays details about hyperedges for a concrete realisation of the construction graph. Field 1 contains the name of the hyperedge, Field 2 contains an optional description interpreted as markdown. Field 3 contains the name and the step specification of the associated operation. The step specification can be expanded/collapse using button 7. Field 4 contain the execution state for the hyperedge. This field indicates if the underlying operation was already executed by an operation executor and

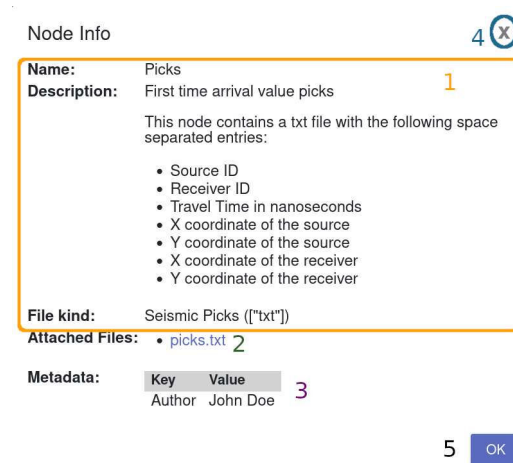


Figure E.47.: Node Info Dialog

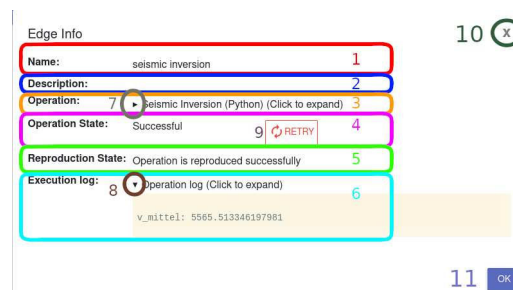


Figure E.48.: Edge Info Dialog

if this execution was successful. Field 5 contains the reproduction state of the underlying operation. This field indicates the underlying operation was already executed twice and both runs yield the same result or not. Field 6 contains the execution log, which contains any output emitted by the commands specified in the operation step specification. Button 8 expands/collapses the log view. Button 9 allows to rerun this specific operation (and any depend part of the construction graph) by scheduling a new revision using the same data as the currently viewed revision. Button 10 and 11 closes the dialog.

E.13. Edit mode

Figure E.49 shows the edit mode of the graph schedule view. This mode allows you to trigger new realisations of the construction graph by attaching concrete data to each input dataset. Similarly to the graph edit mode describe in section E.10.2 a bottom bar (1) contains a number of available operations. Clicking on any input dataset, recognisable by the circular nodes with dashed boarder (like 2), opens the manage file dialog shown in figure E.50. Clicking on any other dataset (node) or hyperedge opens the corresponding info dialog as described in section E.12.

Button 3 closes the edit mode and switches back to the view mode. Similarly to the graph edit mode described in section E.10.2 this will open the discard changes dialog shown in figure E.43 if there are any unsaved changes left.

Button 4 allows to upload all changes to the server. This will show the save revision dialog shown in figure E.51 to confirm uploading. If the button is greyed out there are no changes that can be uploaded.

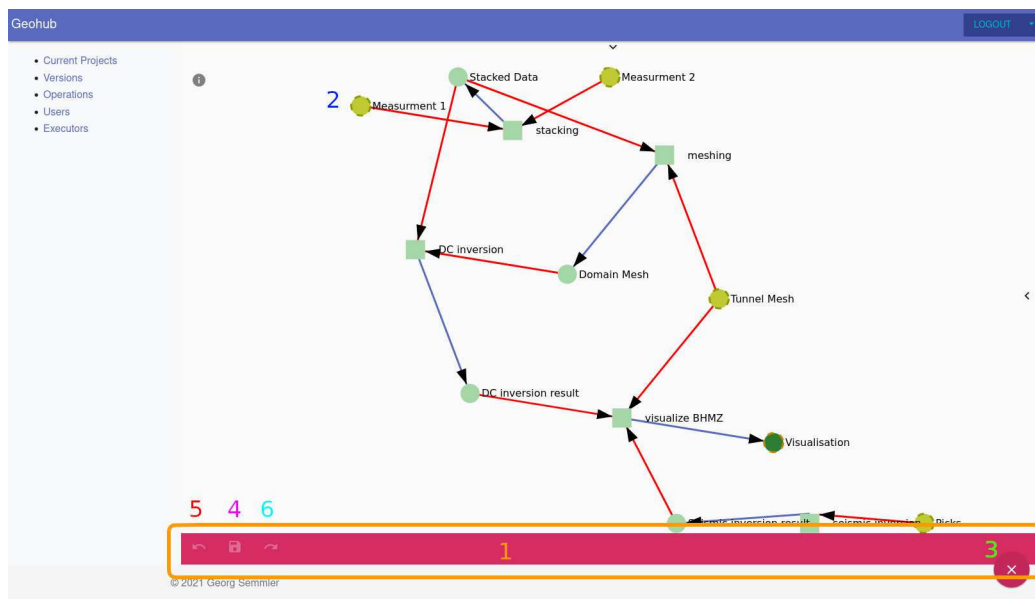


Figure E.49.: Graph Schedule View (Edit Mode)


Button 5 allows to undo an edit to the construction graph, while button 6 allows to redo the last previously undone change. Greyed out buttons indicate that no change is available that can be undone or redone.

Figure E.50 shows the manage file dialog. This dialog can be used to control which data are attached to a input node at a specific revision. The fields in the area 1 match the corresponding fields in the node info dialog described in figure E.33. The manage file element allows you to attach new files via the **Browse** button (2) or to remove individual files via the trash bin button (3). The browse file dialog automatically restricts the selection to files matching the given file kind. Text field 4 allows you to set the corresponding metadata value for a required metadata key. The dialog automatically contains a separate field for each required key. Button 5 allows you to add other metadata keys based on the input in the text field left of the button. This will create a new text field similar to text field 4 to input the corresponding value. Button 6 allows to discard all changes and close the dialog afterwards. Button 7 applies the changes to the displayed construction graph and closes the dialog afterwards.

Figure E.51 shows the save revision dialog. This dialog is used to upload a set of changes as new revision to the server so that a new realisation of the construction graph is created from the changed input datasets. Text field 1 allows you to specify a revision subject. This field is required. Text field 2 allows you to specify an optional description. The content of this text field is interpreted as markdown. Button 3 allows you to cancel the submission. This brings you back to the edit mode without losing any data, but also without uploading any data. Button 4 submits a new revision based on the changes specified earlier in edit mode and the revision information entered into the corresponding text fields. Depending on the size of data which needs to be uploaded a dialog showing the upload process may show up.

E.14. gOcad History Export tool

The gOcad history export tool is a small command line program to generate construction hypergraphs based on existing gOcad/Skua sessions. It provides the following feature sets:

6 

Edit input files

Name: Picks


Description: First time arrival value picks

This node contains a txt file with the following space separated entries:


- Source ID
- Receiver ID
- Travel Time in nanoseconds
- X coordinate of the source
- Y coordinate of the source
- X coordinate of the receiver
- Y coordinate of the receiver

File kind: Seismic Picks (["txt"])


Enforced attributes: **Author**

 2


Selected 1 files:

3  picks.txt

Metadata

Author 4 

This field is required

New metadata key  5




7 

Figure E.50.: Manage File dialog

3 

Schedule Construction

Revision Subject: 1 

This field is required

Revision Description: 2


4 

Figure E.51.: Save revision dialog

- List existing sessions in a given gOcad project
- Export a given session a graphical construction hypergraph representation via Plantuml
- Export a given session as GeoHub compatible construction hypergraph. This generates a ZIP archive in the format accepted by the the Upload functionality in the Graph Edit view. See figure E.37 for details on how to import the generated archive.

Checkout the command line help of the gOcad history export tool for details on how to perform each of these operations.