

FAIRiCUBE – F.A.I.R. INFORMATION CUBES

Project Number: 101059238

WP 3 Process

D3.3 Processing and ML applications

Deliverable Lead: NIL
Deliverable due date: 31/03/2025

Version: 4.4
2025-04-14

Document Control Page

Document Control Page	
Title	D3.3 Processing and ML applications
Creator	NIL
Description	D3.3 Processing and ML applications
Publisher	"FAIRICUBE – F.A.I.R. information cubes" Consortium
Contributors	NIL, WER, NHM, S4E, EPS
Date of delivery	31/03/2025
Type	Text, Data
Language	EN-GB
Rights	Copyright "FAIRICUBE – F.A.I.R. information cubes"
Audience	<input checked="" type="checkbox"/> Public <input type="checkbox"/> Confidential <input type="checkbox"/> Classified
Status	<input type="checkbox"/> In Progress <input type="checkbox"/> For Review <input checked="" type="checkbox"/> For Approval <input type="checkbox"/> Approved

Revision History			
Version	Date	Modified by	Comments
0.1	16/05/2023	Stefan Jetschny, NIL	Draft setup, headings, and partner / contributor assignments
	26/05/2023	Rob Knapen, WER	Use Case 2 contribution
0.2	11/06/2023	Stefan Jetschny	Ready for partial review, only UC4 contribution missing
1.0	21/06/2023	Stefan Jetschny	Ready for review, minor comments still open
1.1	28/06/2023	Jaume Targa and Stefan Jetschny	Review and minor modifications according to review comments.
2.0	07/11/2023	Stefan Jetschny	Planned update according to project proposal to reflect the progress of the work, read-through version for assigning writing-tasks
2.1	14/12/2023	Mohamed-Bachir Belaid	Updating structure, adding automatic monitoring, validation, updating UC contribution
2.2	10/01/2024	Jaume Targa	Review
3.0	15/04/2024	Stefan Jetschny	Preparation for scheduled M24 update
3.1	23/04/2024	Stefan Jetschny	Review with feedback from General Project Review Consolidated Report, UC5 chapter has been added as a placeholder for future progress
3.2	12/05/2024	Mirko Gregor (S4E)	Internal review
3.3	10/08/2024	Stefan Jetschny	Adding new chapter "Common processing and ML Applications"

4.0	16/10/2024	Stefan Jetschny	Reopening report and preparing for the final update
4.1	20/03/2025	Dimitar Misev	Add Python UDF support to the "Rasdaman UDF developments", section, and add new "WCS/WCPS Python libraries" and "WCPS ChatCUBE" sections
4.2	24/03/2025	UC leads, Stefan Jetschny	Updated UC sections
4.3	26/03/2025	Stefan Jetschny, Mona Waagsbø	Internal review and format checking
4.4	27/03/2025	Jaume Targa & María Colina	Full review



Disclaimer

This document is issued within the frame and for the purpose of the FAIRiCUBE project. This project has received funding from the European Union's Horizon research and innovation programme under grant agreement No. 101059238. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

This document and its content are the property of the FAIRiCUBE Consortium. All rights relevant to this document are determined by the applicable laws. Access to this document does not grant any right or license on the document or its contents. This document or its contents are not to be used or treated in any manner inconsistent with the rights or interests of the FAIRiCUBE Consortium or the Partners' detriment and are not to be disclosed externally without prior written consent from the FAIRiCUBE Partners. Each FAIRiCUBE Partner may use this document in conformity with the FAIRiCUBE Consortium Grant Agreement provisions.



Table of Contents

Document Control Page	2
Disclaimer	4
Table of Contents	5
1 Introduction	8
2 Processing and ML applications	9
2.1 Monitoring methods	9
2.1.1 Manual	10
2.1.2 Automatic	11
2.1.3 Validation of the automatic resource monitoring	12
2.2 Provisioning of resource monitoring data	14
2.3 Financial Monitoring of AWS resources	15
3 Examples of resource monitoring data by UC	18
3.1 UC1 Urban adaptation to climate change	18
3.2 UC2 Agriculture and Biodiversity Nexus	20
3.3 UC3 Biodiversity occurrence cubes – <i>Drosophila</i> landscape genomics	24
3.4 UC4 Spatial and temporal assessment of neighbourhood building stock	27
3.5 UC5 Validation of Phytosociological Methods through Occurrence Cubes	28
4 Common processing and ML applications	31
4.1 Common code repository	31
4.2 QueryCube	32
4.3 WCS/WCPS Python libraries	33
4.4 Rasdaman UDF developments	33
4.5 FAIRiCUBE ChatBots	38
4.5.1 WCPS ChatCube	38
4.5.2 Knowledge Base ChatBot	39
5 Summary and conclusion	41



List of Figures

Figure 1 : Knowledge base form for ingestion of the computational resources. _____	14
Figure 2 : Screenshot of Query tool results. _____	15
Figure 3 : Monthly AWS costs for 6 months in 2024 _____	16
Figure 4 : Use Case 1 monthly AWS costs for storage (S3) and processing (EC2-Instances) _____	16
Figure 5 : Daily AWS costs of month 02/2025 split by service categories _____	17
Figure 6 : Daily AWS costs of month 02/2025 split by consumption according to UC server profiles _____	17
Figure 7 : Illustration of QueryCube functionality taken from the GitHub documentation ¹ _____	32
Figure 8 : User-Defined Functions in rasdaman (from Rasdaman documentation) _____	33
Figure 10 : Integration of DL model inference in rasdaman _____	34
Figure 10 : Example WCPS query for applying the crop classification model _____	35
Figure 11 : Model inference result, showing crop classes by green to yellow colour ramp _____	35
Figure 13 : Initial inference scalability testing _____	36
Figure 13 : Screenshot of the FAIRiCUBE WCPS ChatBot _____	38
Figure 14 : Screenshot of the FAIRiCUBE KB ChatBot _____	39



List of Tables

Table 1 : Monitoring methods and tools	10
Table 2 : Resource monitoring metrics by the <i>Measurer</i> script	11
Table 3 : Manual vs Automatic resource computation on Gradient Boosting Regressor.	12
Table 4 : Manual vs Automatic resource computation of k-means clustering on local resources.	13
Table 5 : UC1 Processing resources usage overview	19
Table 6: Species abundance calculation	20
Table 7 : SDM Model training resource usage for "Skylark" in arable land zone of the study region.	21
Table 8 : SDM Model inference resource usage for "Skylark" in arable land zone of the study region.	21
Table 9 : Resource usage of causal graph analysis <i>estimand</i> step.	22
Table 10 : Resource usage of causal graph analysis <i>estimate</i> step.	23
Table 11 : Resource usage of causal graph analysis <i>refuta</i> step.	23
Table 12 : UC3 Processing resources usage overview, ML baseline (k-means)	24
Table 13 : UC3 Processing resources usage overview, advanced ML methods	25
Table 14 : UC3 Resources usage overview on Environmental Association Analysis	26
Table 15 : UC4 Processing resources usage overview	27
Table 16 : UC5 Processing resources usage overview	29
Table 17 : Overview of common code in the respective FAIRiCUBE repository	31
Table 18 : Rasdaman crop classification inference - WCPS wall-clock times	35
Table 19 : UC2 Processing resources usage overview	36



1 Introduction

WP3 aims to provide guidance, recommendations, technical expertise, and implementation support expertise to all use case efforts in terms of data analysis and processing. While the tasks will be executed by the use case developers, support will be given to assist in all data handling steps after ingestion and provision on both the Rasdaman- and EOxHub services as part of FAIRiCUBE's overall data and model services. Special emphasis is given to the data-driven machine learning (ML) model generation.

This deliverable needs to be seen as one item of a classical and logical execution of a machine learning application. Given the availability/ingestion of data, we first perform an exploratory data analysis to get familiar with the data, analyse statistical parameters and distribution, and check for completeness, outliers and other characteristics which could be relevant to the choice of the machine learning. This in-depth data analysis is covered by the deliverable *D3.1 UC exploratory data analysis*.

Subsequently, the raw data might require conversion into features through a data engineering process. This could imply a combination of several input data sources or applying simple mathematical operations to enhance the meaningfulness of the raw data given the relationships that are to be revealed. The more a priori information is available, the better the feature engineering process can be performed. Based on the findings from the exploratory data analysis, the formulation of the research question, and the relationship between raw data sources/features, machine learning algorithms can be recommended to establish a baseline model if this is not provided by use case owners. Starting from the most efficient machine learning algorithm, more advanced ML methods can be identified to form a machine learning strategy. Several different methods might also be tested to recommend a method based on computational demands and accuracy of the ML output. Typically, the testing of ML algorithms is performed on a subset of the original input data or selected cases. The feature engineering process, testing of ML algorithms and the recommendation of a cascade to ML algorithms, as well as analysing the output of ML methods is covered by deliverable *D3.2 Machine learning strategy specific for each use case*.

As the FAIRiCUBE Hub ultimately aims to provide resource estimations and guidance for ML applications, we want to collect and share computational parameters, timings, and requirements and give an outlook on the expected scalability of the ML problems defined by the use cases. For each ML algorithm identified and executed as described in D3.2 we collect information on e.g., disk storage, CPU runtime, main memory consumption, describe the hardware and environment where the ML algorithm is executed on and list essential libraries that are needed to exactly replicate the ML application. This technical documentation of the ML execution is covered in this deliverable *D3.3 Processing and ML applications*. Chapter 2 covers the technical description of how we measure, monitor and report the resource estimates while chapter 3 shows some monitoring examples representative for each FAIRiCUBE use case (UC). Furthermore, any software or tool development that have surfaced as part of the UC execution and that can have a wider use for other UCs or beyond the FAIRiCUBE project are briefly described in chapter 4.

In summary, the exploratory data analysis (D3.1) can be seen as essential input to the development of a UC-specific machine learning strategy (D3.2) whereas the technical description in D3.3 (this document) acts as a reference to follow up on the execution and serves as valuable input to estimate the demands for other ML applications.



2 Processing and ML applications

Monitoring computational resources can provide three significant advantages. Firstly, it allows us to provide insights and estimations to other users who may be running similar jobs on similar hardware. This information can serve as a starting point to scale it for different hardware configurations. This will help the project planning of especially computational heavy tasks. Usually, regular processing and several basic machine learning methods scale linearly.

Secondly, information on hardware requirements and resource usage can directly translate into costs of cloud resources which is also frequently unknown during the project planning. Finally, collecting information on the actual performance and demands of computational tasks can be the starting point for numerical optimization especially when expectations are not met by the measures. Optimization can of course include the careful balance of computational efforts/needs with the output metrics as well. Not in all cases the optimal solution is the most accurate one but a well-selected compromise of resources and sufficient accuracy.

2.1 Monitoring methods

Monitoring of computational resources and demands have long been driven by limitations of the availability of local computational resources. Nowadays, there seems to be no limit concerning the availability but more on the financial aspect of securing [cloud] resources and the environmental implications of executing computational jobs. In the following, we have listed the parameters and measurement methods for those parameters that we considered useful to estimate costs and also prepare environmental impact assessments as a result of the energy consumption. The monitoring methods focus on the execution on local (e.g. laptop or workstation pc) or cloud (e.g. cluster pc, virtual machines) hardware where direct access to built-in monitoring tools is available (see chapter 0) or where we can execute automatic resource monitoring scripts (see chapter 2.1.2). Note that we mostly focus on the dominant data science programming languages Python and the operation system Linux first and do not cover all variants of Use Case work (e.g. R, Java or JavaScript). The automatic monitoring concept in chapter 2.1.2 can however be ported to other scripting languages and platforms easily.

We then compared the results and concluded that none of the methods can currently be considered 100% reliable for the following reasons:

- manual measurements, by their very nature, cannot be considered extremely accurate. This is because some of the needed metrics cannot be calculated, while others only allow the output to be calculated visually, without having precise control over the running processes and what is happening inside the machine.
- more iterations would have been required to fine-tune the automated process.



2.1.1 Manual

Traditionally, computational resources are collected manually using system tools provided by the operation system and require user interaction. Depending on the chosen toolsets and their given accuracy, monitoring results can be obtained relatively easily but may not be comparable across platforms or applications. For a default operation system, we have compiled some desired measures in Table 1 and indicated how to collect these.

Table 1 : Monitoring methods and tools

	How to monitor
Storage	
Data size in grid points	Variable allocation, variable monitoring in IDE, only the main variables need to be listed
Data size in MB/GB	Allocation on disk
Main memory	
Available on machine/node	Linux: Settings / About (View information about your system)
Consumed on machine/node	Top / Htop in %
Compute resources	
Description of CPU/GPU	Linux: Settings / About (View information about your system)
Compute wall time	Either count with clock or include time measures in script
Max. energy consumed	Linux: powertop (Power est.), MacOS: power metrics command
CO2 consumed	CO2 conversion factor available?
Network	
Network traffic in MB/GB	n/a
Cost	
Storage	not applicable directly if executed on local resources, for AWS we can pull out numbers based on the information provided above
Compute	
Network	
Software environment	
Programming language	self-explanatory
Essential libraries	main libraries/dependencies that are used in execution

Given installation status on a local computer/server resource, monitoring can be obtained by a user directly. For monitoring on EOx Hub, a user account is required. Firstly, a GitHub account needs to be set up with access to the FAIRiCUBE project. Secondly, the GitHub handle needs to be configured and assigned to a use case specific EOx Hub server option. This is currently a manual process which can be executed by any member of the FAIRiCUBE GitHub organisation. Finally, a user can connect to <https://eoxhub.fairicube.eu> using its GitHub credentials. Once connected, the user can create, run, and share Jupyter notebooks using **AWS (Amazon Web Services) resources** configured by EOxHub. Similar to local compute infrastructure, basic resource monitoring can be performed in the FAIRiCUBE EOx Hub as well.



2.1.2 Automatic

To automate resource monitoring, a tool called 'Measurer' has been developed and tested that can collect information about the computational resources used and the characteristics of the executing machine at the end of the execution of an a/p resource. The information is stored in a csv file that can be uploaded through the metadata form during the a/p resource metadata editing/input process. As more than one csv file can be generated for each a/p resource (depending on the execution configuration), multiple csv files may be loaded.

The Measurer is a Python library that can be simply called from any Python source code to compute parameter values. It is worth noting here that our library works only on Python scripts. More details together with source code and examples of use can be found in the related folder ¹ of the GitHub common-code² repository. The Table 2 below shows the resource consumption metrics and how they are calculated by the Measurer.

Table 2 : Resource monitoring metrics by the *Measurer* script

Metric	How is it computed in Measurer?
Data size (MB)	Compute the difference between data added to disk and data removed from disk.
Data size in grid points	Return the 'shape array' of the input data
Largest allocated array in grid points	Return 'shape array' of the largest array in the code
Main memory available (GB)	Return the system's virtual memory available variable using 'psutil' library
Main memory consumed (GB)	Return the memory consumed between two lines of code using the 'tracemalloc' library
The sum of allocated variable sizes (GB)	Return the size of the total allocated variables in the code
Description of CPU/GPU	Use the 'platform' library to return machine and processor details
Wall time in seconds	Return total time using the 'time' library
Energy consumed (kW)	Use the EmissionsTracker function from the 'codecarbon' library
Network traffic (MB)	Use the 'net_io_counters' function from the 'psutil' library
CO ₂ -equivalents [CO ₂ eq] (kg)	Use the EmissionsTracker function from the 'codecarbon' library ³
Programming language	Returns e.g. 'Python'
Essential libraries	Read lines in the code that contain 'import' and return the list of libraries

¹<https://github.com/FAIRiCUBE/common-code/tree/main/record-computational-demands-automatically>

² <https://github.com/FAIRiCUBE/common-code/tree/main>

³ <https://doi.org/10.5281/zenodo.14518377>

2.1.3 Validation of the automatic resource monitoring

In this subsection, we evaluate the measurer library compared to the manual monitoring of two main tasks from UC1 and UC3. The first task (in Table 3) corresponds to a simple regression model that was run on a server platform (AWS configured by EOxHub). The second task corresponds to an ML-assisted gap filling (in Table 4) that was run on a local machine.

Table 3 : Manual vs Automatic resource computation on Gradient Boosting Regressor.

	Gradient Boosting Regressor (Manual)	Gradient Boosting Regressor (Automatic)	Deviation Abs(manual- automatic)/manual)
Compute platform	AWS configured by EOxHub	AWS configured by EOxHub	~
Storage			
Data size in grid points	1673 x 5	1673 x 5	~
Data size in MB/GB	0.05 MB (input) 0.27 MB (output)	0.05 MB	0%
Main memory			
Available on machine/node	32 GB	61.46 GB	92%
Consumed on machine/node	520.5 MB	514.1 MB	1.2%
Compute resources			
Description of CPU/GPU	C5.4xlarge with 1.8 GHz CPU	x86_64 with 8 physical cores and 16 logical cores 1.8 GHz CPU	☒
Compute wall time	6.45 s	4.10 s	36%
Max. energy consumed	-	36.7 W	-
CO2 consumed	-	13.g	-
Network			
Network traffic in MB/GB	-	0.01611 MB	-
Cost			
Storage			
Compute			
Network			
Software environment			
Programming language	Python	Python	~
Essential libraries	NumPy, time, os, pandas, math, joblib, matplotlib.pyplot, seaborn, psutil, sklearn	numpy, time, os, pandas, math, joblib, matplotlib.pyplot, seaborn, psutil, sklearn	~

Table 4 : Manual vs Automatic resource computation of k-means clustering on local resources.

	k-means clustering including elbow method (Manual)	k-means clustering including elbow method (Automatic)	Deviation Abs(manual- automatic)/manual)
Compute platform	Laptop	Laptop	~
Storage			
Data size in grid points	50024 x 122 (input), 50024x x1 (output)	50024 x 122 (largest array)	~
Data size in MB/GB	30 MB (input), 1 MB (output)	1 MB (output file size added to storage)	~
Main memory			
Available on machine/node	32 GB	31 GB	~
Consumed on machine/node	640 MB (2%)	206 MB (>1%)	67 %
Compute resources			
Description of CPU/GPU	Intel Core i7- 1085H@2,7 GHz x 12	Machine type: x86_64 Processor type: x86_64 Number of physical cores: 6 Number of logical cores: 12 Min CPU frequency: 800.0 GHz Max CPU frequency: 5100.0 GHz No GPU available	~
Compute wall time	30 s	40 s	33 %
Max. energy consumed	1 W	0.4 W	60 %
CO2 consumed		0.01 g	
Network			
Network traffic in MB/GB	no	no	
Cost			
Storage	No direct costs (local laptop)	No direct costs (local laptop)	
Compute	No direct costs (local laptop)	No direct costs (local laptop)	
Network	No direct costs (local laptop)	No direct costs (local laptop)	
Software environment			
Programming language	Python	Python	~
Essential libraries	Pandas, NumPy, Sklearn	Pandas, NumPy, Sklearn	~

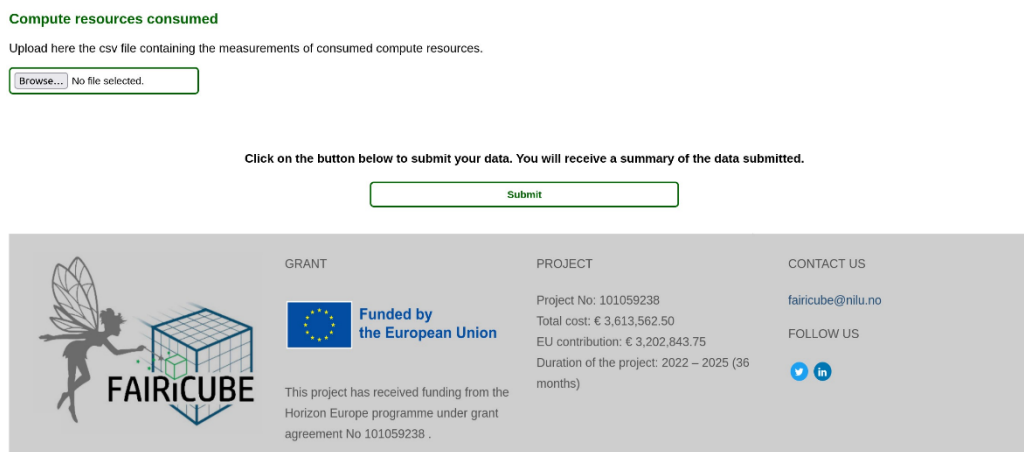
Obviously, according to the current results, the automatic computation of resources requires further tuning to be considered as the main tool for documenting the computational resources. For instance, at this level, 'total consumed memory on machine' computes what a program allocates between two lines of code. However, a program requires more than only the memory used by a few lines of code (e.g., importing libraries, etc). For example, we have noticed that running the regressor requires more than 520 MB according to the task manager but only 14 MB was reported (see Table 3).

Also, the measurer library fails to return the exact frequency of the processor when a program is run on a server (e.g., on AWS configured by EOxHub). From Table 3, we notice that 'Available on machine' is almost doubled compared to the manual reporting. This is because, even though we allocate a memory of only 32GB from AWS, some more resources are allocated to us randomly when free.

Overall, the measurer library is a relevant initiative to automatize the documentation of computational resources. It has matured enough to be included as a reference monitoring tool for each processing job to automatically. As a result, as for now, we consider manual reporting for the rest of the report.

2.2 Provisioning of resource monitoring data

Once uploaded during the a/p resource metadata editing/input process, the resource consumption information in the csv file(s) is stored in a database in order to be made available in the Query tool¹, the interactive tool that allows easy discovery and analysis of the FAIRiCUBE data a/p resources. Specifically, the online metadata form allows the upload of the CSV file(s) resulting from the Measurer's calculations (see Figure 1).



Compute resources consumed


Upload here the csv file containing the measurements of consumed compute resources.

[Browse...](#) No file selected.

Click on the button below to submit your data. You will receive a summary of the data submitted.

[Submit](#)

GRANT

 **Funded by the European Union**

This project has received funding from the Horizon Europe programme under grant agreement No 101059238.

PROJECT

Project No: 101059238
 Total cost: € 3,613,562.50
 EU contribution: € 3,202,843.75
 Duration of the project: 2022 – 2025 (36 months)

CONTACT US

fairicube@nilu.no

FOLLOW US

[Twitter](#) [LinkedIn](#)

Figure 1 : Knowledge base form for ingestion of the computational resources.

After the metadata information of the online form has been submitted (i.e. after the user has clicked on the submit button) and after all necessary checks have been performed on the information provided, the metadata ingestion application:

- creates the related STAC-JSON metadata file (so that the metadata resource can follow the pipeline for publication into the STAC catalog).
- ingests needed information into the Knowledge Base Postgres DB tables. In particular, the database contains two tables, one with information on the a/p resources and the other with information on the consumed resources measurements (linked to related resources by the "resource_id" foreign key). In this way, measurements of resource consumption for a particular resource are available when this is displayed in the Query Tool (see Figure 2 below).

¹ <https://fairicube-kb.epsilon-italia.it/>



FAIRiCUBE Query Tool

Home

Query Tool Results

Selected Parameters


Name	Description	Link	Use Case	Consumed Resources
LeNet Classifier	Multi-layer CNN for image classification	https://catalog.eoxhub.fairicube.eu/collections/no-ML%20collection/items/WNBP1DMM99	common	

Figure 2 : Screenshot of Query tool results.

2.3 Financial Monitoring of AWS resources

AWS is a service that provides cloud-computing resources. The payment model for AWS is based on the pay-as-you-go principle, which means that you are only charged for the services you use when running computations using AWS resources. Despite the overwhelming complexity of the AWS console and the need to get some basic training in order to find service and resources there, it provides detailed and comprehensive resources information that helps to monitor costs of running cloud-based storage and compute infrastructure accessed through the FAIRiCUBE EOX Lab. Together with the technical description of the actual data science work executed under the FAIRiCUBE Hub services, this financial resource overview can be a very good starting point for managing and planning of projects and proper allocation of funds. As a starting point, monthly AWS costs can be visualized through the Billing and Cost Management service¹ as shown in Figure 3 where the total costs of 6 subsequent months is shown as line charts. Most of the different service costs are stable only the S3 related costs are increasing from month 06/2024 to 10/2024 and the associated tax, respectively.

As each EOX Lab server options² is tagged, these tags can be used to get use case specific input into consumption of resources as shown in Figure 4 where we can see the evolution of e.g. the S3 storage related costs of UC1 over 6 subsequent months. The compute related EC2-Instances category further reflects computational workload of the UC. We can further drill into the cost overview by zooming into the daily consumption as shown in Figure 5 and Figure 6 where the costs are separated over the different AWS service categories and selected use case server profiles respectively. With the data reported in the AWS Billing and Cost Management service individual and aggregated [use case] activities and a general project projection can be obtained. Sub-billing of individual UC consumption is thereby possible but not foreseen in FAIRiCUBE. In any case, the cost details help to optimize cloud-based service costs by identifying drivers and evolution of the main AWS costs.

¹ <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/billing-what-is.html>

² https://fairicube.readthedocs.io/en/latest/user_guide/eox_lab/

Total cost
\$10,586.35

Average monthly cost
\$1,512.34

Service count
15

Costs (\$)

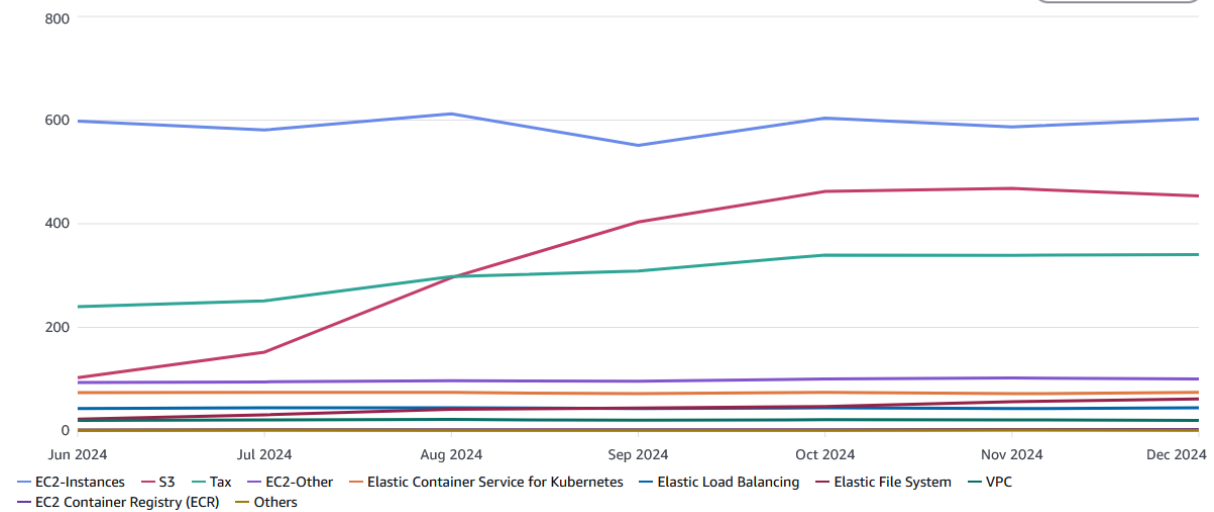


Figure 3 : Monthly AWS costs for 6 months in 2024

Cost and usage graph [Info](#)

Total cost
\$209.67

Average monthly cost
\$34.94

Service count
3

Costs (\$)

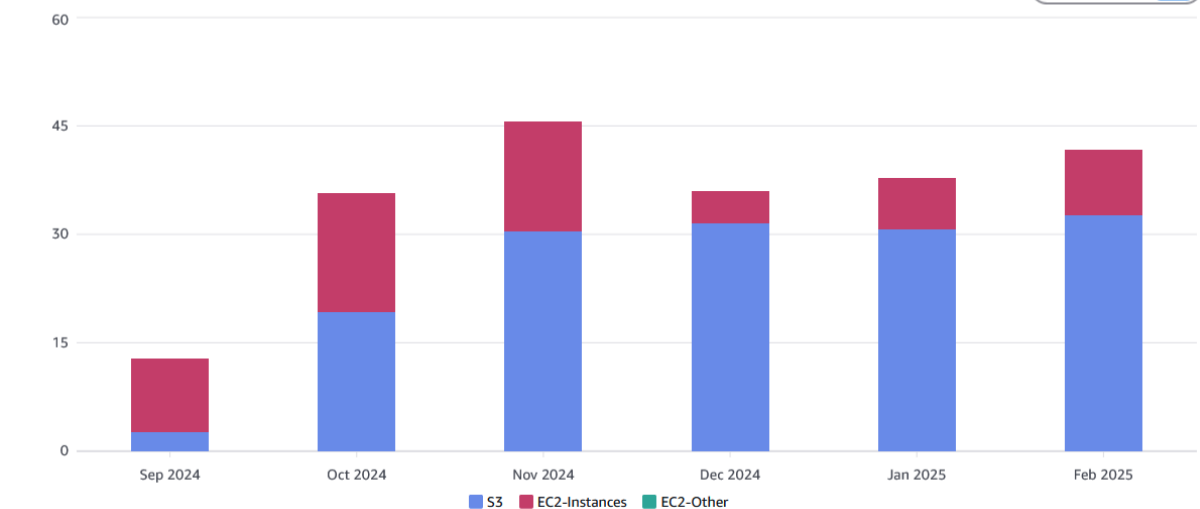


Figure 4 : Use Case 1 monthly AWS costs for storage (S3) and processing (EC2-Instances)

Cost and usage graph [Info](#)

Total cost
\$2,152.07

Average daily cost
\$79.71

Service count
14

Costs (\$)

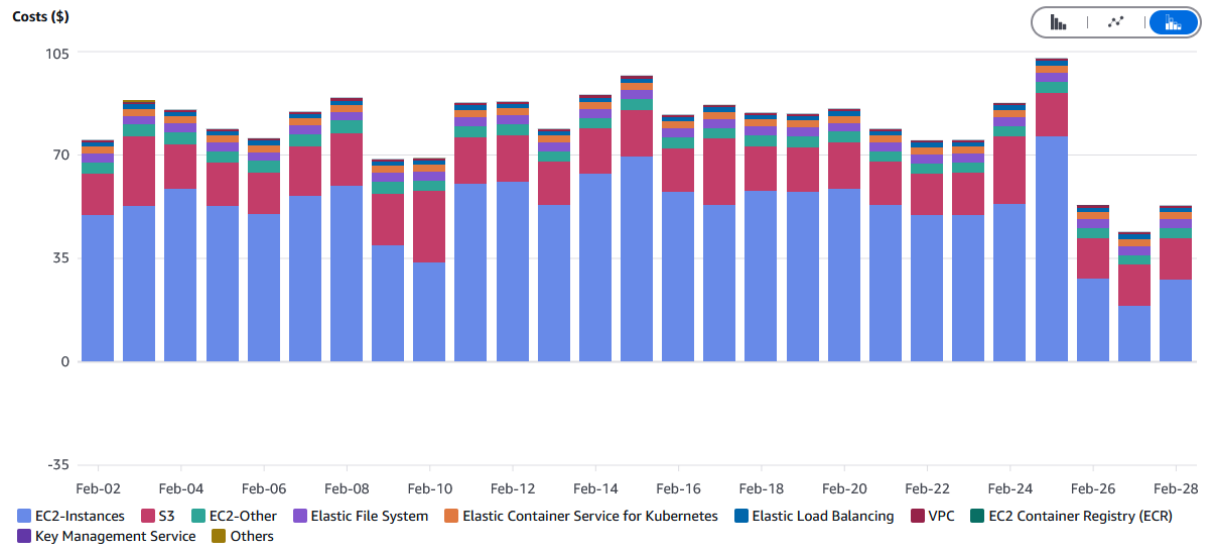


Figure 5 : Daily AWS costs of month 02/2025 split by service categories

Cost and usage graph [Info](#)

Total cost
\$2,152.07

Average daily cost
\$79.71

eks:nodegroup-name count
11

Costs (\$)

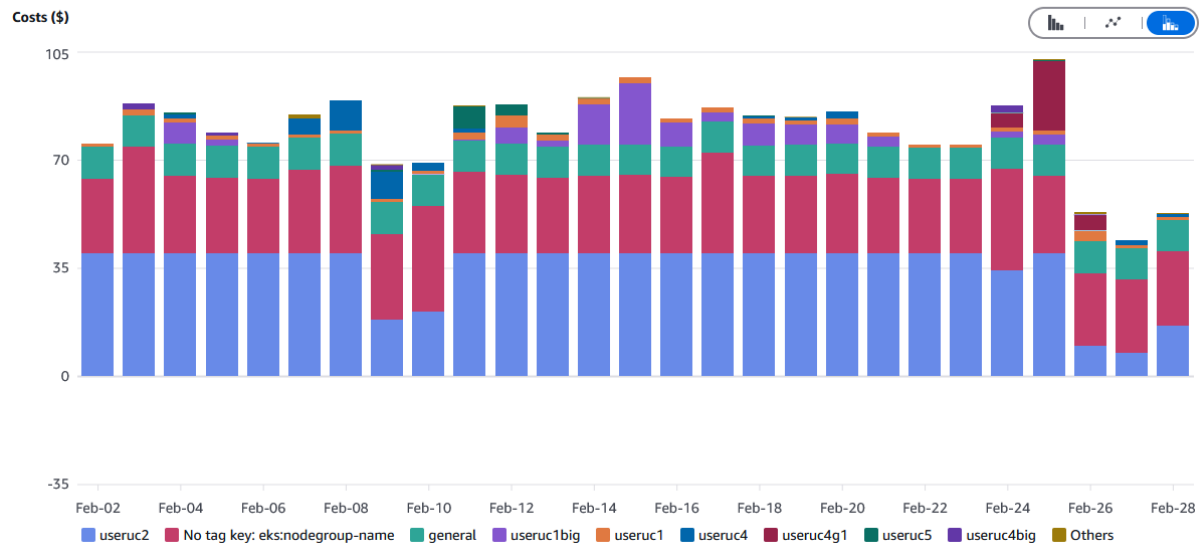


Figure 6 : Daily AWS costs of month 02/2025 split by consumption according to UC server profiles



3 Examples of resource monitoring data by UC

In the following, we will give examples from all use cases (UCs) of which processing resources are monitored. This is not a complete overview but should provide insights into typical compute-tasks which are covered in each UC. A complete overview will be given by including all the resource records in the Knowledge Base (KB) as described in chapter 2.2.

3.1 UC1 Urban adaptation to climate change

The UC1 team shares an AWS S3 bucket in the EOXHub to ingest, process and share data. We have used the currently configured EOXHub profile (c5.4xlarge CPU) with an AWS resource of 7 GB of RAM and 1.8 GHz CPU. We have implemented ML algorithms in Jupyter notebooks and Python scripts.

For the **European level application** (EU cities clustering exercise) the ML algorithms were not memory or time consuming, we did not have any problem running them within the currently configured EOXHub profile. The clustering itself was implemented using the Python library *scikit-learn*, and the library *seaborn* was used for results visualization. The pre-processed city data cube was stored in the S3 bucket in CSV format, and the I/O was handled with the Python data analysis library *pandas*. The data size is less than 110 KB (1,000 rows and 6 columns). Generating the city data cube from the original gridded was the most time-consuming part.

Table 5 provides an overview of the resource usage for each main process, starting from the calculation of features to the execution of ML algorithms. We can clearly see that the process that consumed resources the most is "Calculating level 1 land classification ratios" (e.g., more than 900 MB of memory usage and more than 2 hours). This is because it requires uploading and processing data from different cities. On the other hand, running the clustering algorithms required between 1 and 10 MB only. Calculating level 1 land classification ratios was very time-consuming (more than 2 hours). However, we can clearly see that the overall ML process was fast (around 10 seconds), with clustering algorithms running in less than 1 second, except Mean-Shift that required more than 7 seconds. In addition, running Bidirectional LSTM and Gradient boosting regressor for gap filling was very fast, however with a relatively high memory usage, 968 MB and 520, MB respectively. Cost-wise, in addition to the AWS fixed daily deduction of around 3\$, calculating level 1 land classification ratios and the clustering processes, for example, cost between 20\$ and 25\$ each for computing (i.e., using AWS resources).

For the **local level application** (invasive plant species distribution in urban context), we have used the species distribution model Maxent as implemented in the Python library *elapid* with default parameters. In addition, we have used the [SHAP \(SHapley Additive exPlanations\)](#) approach (and Python implementation) to explain and identify the most important features used for learning the species distribution. The model with explanatory analysis has consumed around 1.5 GB of memory and run in almost 45 minutes (with 1'000 samples for the SHAP analysis). The most time-consuming process by far is the explanatory analysis.



Table 5 : UC1 Processing resources usage overview

	Gap-filling: Bidirectional LSTM	Gap-filling: Gradient Boosting Regressor	MaxEnt + SHAP (4 species, 1000 samples)
Compute platform	EOX	EOX	EOX
Storage			
Data size in grid points	51 x 34	1673 x 5	2646 x 15
Data size in MB/GB	0.14 MB (input) 0.03 MB (output)	0.05 MB (input) 0.27 MB (output)	20 MB input, 3.3 MB output
Main memory			
Available on machine/node	30 GB	61 GB	62 GB
Consumed on machine/node	23.6 MB	13.86 MB	1449.7 MB
Compute resources			
Description of CPU/GPU	Machine type: x86_64, Processor type: x86_64, Number of physical cores: 8, Number of logical cores: 16 Min CPU frequency: 0.0 GHz, Max CPU frequency: 0.0 GHz, No GPU available		
Compute wall time	11.65 s	4.05 s	2599.58 s
Max. energy consumed	0.17 W	0.036 W	46.84 W
CO2 consumed	0.063 g	0.013 g	17.12 g
Network			
Network traffic in MB	0.41 MB	0.024 MB	125.7 MB
Cost			
Storage			
Compute			-
Network			
Software environment			
Programming language	Python	Python	Python
Essential libraries	Numpy, pandas, math, matplotlib.pyplot, time, os, sklearn, keras	numpy, time, os, pandas, math, joblib, matplotlib.pyplot, seaborn, psutil, sklearn	numpy, pandas, xarray, sqlalchemy, os, glob, matplotlib.pyplot, sklearn.metrics, sklearn.model_selection, rioarray, matplotlib.patches, elapid, shap

3.2 UC2 Agriculture and Biodiversity Nexus

Some of the early work in this use case has been focussed on the collaborative development of the existing Rasdaman UDF (User-Defined Functions) functionality toward its use for running machine learning models, and deep learning models in particular, for inference within the Rasdaman distributed compute infrastructure. This is further elaborated in section 0. Later, the focussed shifted to realisation of the research objectives of the use case. For that, there are three major steps in the processing and ML workflow:

- Species abundance per grid cell calculations
- Species distribution modelling and species richness calculation
- Causal machine learning

It should be noted that for some of these steps the research is only in its initial stages with experiments based on low-complexity algorithms and models, for relatively small study areas. Future work might increase the resource requirements significantly.

Species abundance calculation

Calculating species abundance is an important step toward spatial biodiversity estimation, providing essential input for species distribution modelling and biodiversity index calculation. In this step, farmland bird species observations from the NDFF dataset were used as the source data. These observations consist of vector points or polygons containing information on the number of individuals recorded. The vector layer was overlaid with a 100×100 m regular grid, and the proportional number of observations per grid cell was calculated to represent species abundance.

Table 6: Species abundance calculation

Metric	How is it computed in Measurer?
Data size (MB)	-0.27
Data size in grid points	N/A
Largest allocated array in grid points	[1632578, 28]
Main memory available (MB)	126177.4
Main memory consumed (MB)	1771.35
The sum of allocated variable sizes (MB)	1640.16
Description of CPU/GPU	Processor type: x86_64; Number of physical cores: 16; Number of logical cores: 32; CPU frequency: 1.5007 GHz; GPU monitoring disabled
Wall time in seconds	380.36
Energy consumed (W)	8.86
Network traffic (MB)	4.93
CO ₂ -equivalents [CO ₂ eq] (kg)	3.37
Programming language	Python
Essential libraries	gdal/ogr, shapely, numpy, pandas, geopandas

Species distribution modelling

For the species distribution modelling (SDM) initially the well-known Java MaxEnt desktop application was used, but this was later switched to the *elapid* Python package that has better compatibility with the EOX compute platform. The use case considers biodiversity species richness, derived from farmland bird species presence observations, in relation to agricultural activities at detailed level. To that end SDM is used to model the distribution of different farmland bird species. This is done using the MaxEnt model implemented in the *elapid* Python package.

Performance measurements are split into model training (in Table 7), and model inference (in Table 8), using as an example a model run for a single bird (Skylark, "veldleeuwerik" in Dutch) in an zone of the study region that has predominantly arable farming. All data was read from an AWS S3 bucket.

Table 7 : SDM Model training resource usage for "Skylark" in arable land zone of the study region.

Metric	How is it computed in Measurer?
Data size (MB)	-0.33
Data size in grid points	N/A
Largest allocated array in grid points	[2658345, 8]
Main memory available (GB)	123
Main memory consumed (GB)	11
The sum of allocated variable sizes (GB)	0.5
Description of CPU/GPU	Processor type: x86_64; Number of physical cores: 16; Number of logical cores: 32; CPU frequency: 1.5007 GHz; GPU monitoring disabled
Wall time in seconds	1058.58 (17.7min)
Energy consumed (kW)	26.01
Network traffic (MB)	17.62
CO ₂ -equivalents [CO ₂ eq] (kg)	9.22
Programming language	Python
Essential libraries	Elapid, numpy, pandas, geopandas, rasterio

For the MaxEnt model the resource usages mostly depends on the number of grid points, which in the use case are the results of the species abundance calculations. The model loads all data into memory first, so it is memory-bound. Compute is handled by only CPU in *elapid*, making it time consuming. Once the model is trained (fitted to the data), inference is rather quick.

Table 8 : SDM Model inference resource usage for "Skylark" in arable land zone of the study region.

Metric	How is it computed in Measurer?
Data size (MB)	0.0
Data size in grid points	N/A
Largest allocated array in grid points	[2658345, 8]
Main memory available (GB)	123
Main memory consumed (GB)	8
The sum of allocated variable sizes (GB)	0.5
Description of CPU/GPU	Processor type: x86_64; Number of physical cores: 16; Number of logical cores: 32; CPU frequency: 1.5007 GHz; GPU monitoring disabled
Wall time in seconds	5.84
Energy consumed (kW)	0.08
Network traffic (MB)	0.06
CO ₂ -equivalents [CO ₂ eq] (kg)	0.03
Programming language	Python
Essential libraries	Elapid, numpy, pandas, geopandas, rasterio

Causal machine learning

In the Causal Machine Learning (CML) task of the use case we explored the use of Causal Graph Analysis, to proof the expected causal correlations between agricultural activities and biodiversity, expressed as species richness related to selected farmland bird species. This biodiversity indicator is derived from the SDM results described in the previous step. For the farming activities we used proxy data, such as the crop rotation index and the grassland mowing intensity. The following performance and resource usage measurements are from an experiment using the data from the arable zone of our study region (farmland birds species richness, the crop rotation index, and various environmental covariates and confounding variables).

Briefly, there are three steps in causal graph analysis: (i) Deriving an *estimand* from a graph of the expected correlations. This estimand is the causal quantity we want to know about, e.g., "what is the average increase in farmland bird species richness if the crop rotation index is increased by one unit?"; (ii) Computation of the *estimate*-the numerical approximation of the estimand, derived from the input data; (iii) *Refutal* of the estimate by checking if the result holds up under possible violations, such as a placebo test, the adding of random confounders, or a data subset test. When such refutal fails, i.e. the estimate holds up, it is considered *robust*. Of these steps, the later in general is the most resource intensive. The following tables show the resource usage as reported by the Measurer for the three steps mentioned before. All data was read from an AWS S3 bucket.

Table 9 : Resource usage of causal graph analysis *estimand* step.

Metric	How is it computed in Measurer?
Data size (MB)	0.0
Data size in grid points	N/A
Largest allocated array in grid points	[5627244, 6]
Main memory available (GB)	123
Main memory consumed (GB)	0.002 (1.6 MB)
The sum of allocated variable sizes (GB)	0.236 (241.5 MB)
Description of CPU/GPU	Processor type: x86_64; Number of physical cores: 16; Number of logical cores: 32; CPU frequency: 1.5007 GHz; GPU monitoring disabled
Wall time in seconds	8.59
Energy consumed (kW)	0.15
Network traffic (MB)	0.12
CO ₂ -equivalents [CO ₂ eq] (kg)	0.06
Programming language	Python
Essential libraries	dowhy, numpy, pandas, geopandas, rasterio

Table 10 : Resource usage of causal graph analysis *estimate* step.

Metric	How is it computed in Measurer?
Data size (MB)	0.0
Data size in grid points	N/A
Largest allocated array in grid points	[5627244, 6]
Main memory available (GB)	123
Main memory consumed (GB)	1.36 (1394.2 MB)
The sum of allocated variable sizes (GB)	0.24 (241.5 MB)
Description of CPU/GPU	Processor type: x86_64; Number of physical cores: 16; Number of logical cores: 32; CPU frequency: 1.5007 GHz; GPU monitoring disabled
Wall time in seconds	13.71
Energy consumed (kW)	0.28
Network traffic (MB)	0.13
CO ₂ -equivalents [CO ₂ eq] (kg)	0.11
Programming language	Python
Essential libraries	dowhy, numpy, pandas, geopandas, rasterio

Table 11 : Resource usage of causal graph analysis *refutal* step.

Metric	How is it computed in Measurer?
Data size (MB)	0.2
Data size in grid points	N/A
Largest allocated array in grid points	[25627244, 6]
Main memory available (GB)	123
Main memory consumed (GB)	2.05 (2093.7 MB)
The sum of allocated variable sizes (GB)	0.24 (241.5 MB)
Description of CPU/GPU	Processor type: x86_64; Number of physical cores: 16; Number of logical cores: 32; CPU frequency: 1.5007 GHz; GPU monitoring disabled
Wall time in seconds	381.64 (6.4min)
Energy consumed (kW)	9.34
Network traffic (MB)	7.47
CO ₂ -equivalents [CO ₂ eq] (kg)	3.56
Programming language	Python
Essential libraries	dowhy, numpy, pandas, geopandas, rasterio

3.3 UC3 Biodiversity occurrence cubes – *Drosophila* landscape genomics

According to the machine learning strategy as described in D3.2, one dominant and computational demanding UC3 activity was to develop a gap filling method to improve the robustness of analysis of the genomic data. First, a *k*-means clustering method was performed to establish machine learning baseline which will first be in focus to list the required processing resources (Table 12). The split in columns follows the Python scripting tasks to first introduce gaps in the data, apply *k*-means with the elbow method to determine the optimal number of clusters and finally the *k*-means cluster labels to fill the gaps. For these tasks, it is expected that the computational demand scales linearly with size of the input data.

Table 12 : UC3 Processing resources usage overview, ML baseline (k-means)

	Introduce Gaps	k-means clustering (including elbow method)	Gap filling using k- means cluster labels
Compute platform	Laptop	Laptop	Laptop
Storage			
Data size in grid points	50024 x 122 (input), 50024x x122 (output)	50024 x 122 (input), 50024x x1 (output)	50024 x 122 (input), 50024x x122 (output)
Data size in MB/GB	30 MB (input), 30 MB (output)	30 MB (input), 1 MB (output)	30 MB (input), 30 MB (output)
Main memory			
Available on machine/node	32 GB	32 GB	32 GB
Consumed on machine/node	320 MB (1%)	640 MB (2%)	640 MB (2%)
Compute resources			
Description of CPU/GPU	Intel Core i7- 1085H@2,7 GHz x 12	Intel Core i7- 1085H@2,7 GHz x 12	Intel Core i7- 1085H@2,7 GHz x 12
Compute wall time	3 s	30 s	2 s
Max. energy consumed	1 W	1 W	1 W
CO2 consumed			
Network			
Network traffic in MB/GB	no	no	no
Cost			
Storage	No direct costs (local laptop)	No direct costs (local laptop)	No direct costs (local laptop)
Compute	No direct costs (local laptop)	No direct costs (local laptop)	No direct costs (local laptop)
Network	No direct costs (local laptop)	No direct costs (local laptop)	No direct costs (local laptop)
Software environment			
Programming language	Python	Python	Python
Essential libraries	Pandas, NumPy	Pandas, NumPy, Sklearn	Pandas, NumPy

More advanced and tuned machine learning methods are applied following the ML baseline which are usually more computationally demanding (Figure 8) but provide better accuracy. For these tasks, it is expected that the computational demand does not scale linearly with size of the input data. Finally, a trade-off between execution time and resources and gained accuracy in the prediction can be weighed against each other to form a decision basis for further recommendations and upscaling of the ML application. In Table 14, we further describe the in-kind computational resources, which were used to execute highly specialized population genomic analyses to intersect genomic and earth observation data based on custom bioinformatics pipelines. For further insights into the UC3 analysis pipeline, we refer to deliverable *D2.2 Report on UC Analysis Plans*.

Table 13 : UC3 Processing resources usage overview, advanced ML methods

	Variational Autoencoder (VAE)	Generative Adversarial Networks (GAN)
Compute platform	Laptop	Laptop
Storage		
Data size in grid points	50024 x 122 (input), 50024x x122 (output)	50024 x 122 (input), 50024x x122 (output)
Data size in MB/GB	30 MB (input), 30 MB (output)	
Main memory		
Available on machine/node	32 GB	32 GB
Consumed on machine/node	8 GB	10 GB
Compute resources		
Description of CPU/GPU	AMD Ryzen 9 / NVIDIA RTX 3070 (8GB)	AMD Ryzen 9 / NVIDIA RTX 3070 (8GB)
Compute wall time	30 s	5 min
Max. energy consumed		
CO2 consumed		
Network		
Network traffic in MB/GB	no	no
Cost		
Storage	No direct costs (local laptop)	No direct costs (local laptop)
Compute	No direct costs (local laptop)	No direct costs (local laptop)
Network	No direct costs (local laptop)	No direct costs (local laptop)
Software environment		
Programming language	Python	Python
Essential libraries	Pytorch, NumPy, Pandas	Pytorch, NumPy, Pandas

Table 14 : UC3 Resources usage overview on Environmental Association Analysis

	Genetic data	Environmental data	Association Analysis
Compute platform	NHM server	NHM server	NHM server
Storage			
Data size in MB/GB	DEST VCF (700 Populations x 1,4M SNPs) ~ 10 GB		Analysis Results (Intermediate & Final) ~ 70 GB
Intermediate data size in MB/GB	Sample Metadata ~ 3 GB	QueryCube Result (344 Samples x 605 Layers) ~ 450 KB	
Data formats	VCF, CSV, TXT	CSV, RDS	CSV, PDF, JPG, TXT, RDS, HTML, SH, PY
Main memory			
Available on machine/node	2 TB	2 TB	2 TB
Consumed on machine/node			
Compute resources			
Description of CPU/GPU	2 x AMD 7742 (2.25GHz; 64cores/128threads; 256MB cache)	2 x AMD 7742 (2.25GHz; 64cores/128threads; 256MB cache)	2 x AMD 7742 (2.25GHz; 64cores/128threads; 256MB cache)
Compute wall time			
Max. energy consumed	225W	225W	225W
CO2 consumed			
Network			
Network traffic in MB/GB			
Cost			
Storage	No direct costs (local)	No direct costs (local)	No direct costs (local)
Compute	No direct costs (local)	No direct costs (local)	No direct costs (local)
Network	No direct costs (local)	No direct costs (local)	No direct costs (local)
Software environment			
Programming language	Shell, Python	Shell, Python, R	R
Essential programs/libraries	VCTtools, snpEff	QueryCube	Vegan, Ggplot, LEA, Raster, FactorMineR

3.4 UC4 Spatial and temporal assessment of neighbourhood building stock

According to the machine learning strategy as described in D3.2 and the preceding exploratory data analysis, we have been testing several methods to estimate building heights as input for further calculations of the energy performance and the classification of building compositions. The three methods as described by their consumption of compute resources in Table 15 are very different in terms of their numerical background were

- *Factor x Number of levels* is a brute force minimization of the difference between a constant building story height multiplied by the number of stories and the ground truth building height data
- *Geoclimate-Random Forest* is the model inference of a published ML model to our test data
- *DTM – DSM* is a simple subtraction of data layers with a more demanding aggregation method to convert from DTM / DSM point data to an outline of buildings.

For height estimation, all computational efforts can be seen as moderate and pose no bottleneck in terms of runtime. Scaling up any of the methods to additional cities is feasible.

On the other hand, Energy demand calculation for buildings in Oslo requires almost one hour of running time, however, only around 500 MB of memory is used.

Finally, Rooftop height estimation from satellite images consumed an important amount of memory exceeding 43 GB. This is explained by the number of images we needed to upload (more than 26K images). On the other hand, the processing time was relatively quick (less than 4 minutes) thanks to the use of pretrained models and the available resources under the NILU server (e.g., 4 GeForce RTX 3090 GPUs).

Table 15 : UC4 Processing resources usage overview

Building height estimation					
	Factor x Number of levels	Geoclimate- Random Forest	DTM - DSM	Energy demand calculation (Oslo)	Rooftop height estimation from satellite images
Compute platform	Laptop	Laptop	Laptop	Laptop	NILU's server
Storage					
Data size in grid points (GeoTiff)	1218x1126	1526x1367	1203x1022	-	26798 x 224 x 224 x 3
Data size in MB/GB	11MB	18MB	22MB	20MB (input)	-
Intermediate data size in MB/GB	200MB	300MB	1.5GB		-
Data formats	Geojson and GeoTiff	Geojson and GeoTiff	Pointcloud (xyz) and GeoTiff	Shape file	Jpeg images
Main memory					
Available on machine	32GB	32GB	32GB	32GB	385 GB
Consumed on machine	8GB	6GB	17GB	530MB	43 GB
Compute resources					

Description of CPU/GPU	8-Core Processor, 1550 MHz/ nvidia geforce 1080 ti	8-Core Processor, 1550 MHz/ nvidia geforce 1080 ti	8-Core Processor, 1550 MHz/ nvidia geforce 1080 ti	13th Gen Intel(R) Core (TM) i7-1355U 1.70 GHz	Intel Xeon Silver 4215R @ 3.20 GHz (32 cores), GPUs: 4 × NVIDIA GeForce RTX 3090 (24 GB each), Total VRAM: 96 GB (4 × 24 GB)
Compute wall time	2min	10min	5min	56min	230.01 s
Max. energy consumed					23 W
CO2 consumed					0.7 g
Network					
Network traffic in MB/GB					60.92 MB
Cost					
Storage	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)
Compute	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)
Network	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)
Software environment					
Programming language	Python	Python	Python	Python	Python
Essential libraries	Geopandas, QGis, gdal, geocube, folium, numpy, osmnx, rioxarray	Geopandas, gdal, geocube, folium, numpy, osmnx, rioxarray	Geopandas, gdal, geocube, folium, numpy, osmnx, rioxarray	Geopandas, pandas, numpy, matplotlib.pyplot, Fiona, rasterio, os, shapely	Tensorflow (VGG16), xgboost, numpy, pandas, joblib

3.5 UC5 Validation of Phytosociological Methods through Occurrence Cubes

UC5 integrates vegetation occurrence data with EO data to investigate and predict species distribution patterns in relation to the EUNIS Habitat classification system, as illustrated in Deliverables 2.2 and 3.2. The workflow was first developed for a specific case study using the European Habitat S22 and its eight characteristic species. All processing was conducted in R, with datasets and outputs stored both locally and in the AWS S3 bucket within the EOxHub Lab. Table 16 provides an overview of the computational resources required for UC5.

Downloading and processing occurrence data from GBIF was a relatively lightweight computational task. In general, the processing time can vary depending on the number of occurrences requested, and for this case study, downloading and processing data for eight species took approximately 3 hours. Despite the large spatial and temporal extent of the datasets—totalling over 180,000 occurrence points—the memory requirements remained modest. The raw dataset had an initial size of 32.3 MB, which was reduced to 28.3 MB after data cleaning.

Since GBIF is an open-access resource, no costs were incurred during data acquisition. Additionally, occurrence data were transformed into a gridded format, a step that required minimal memory and processing time.

In contrast to occurrence data, processing environmental datasets was significantly more resource intensive. The datasets, stored in TIFF format, had a total size of approximately 441 MB, with intermediate data volumes remaining the same. However, memory consumption during data extraction and transformation reached approximately 150 GB. The main processing steps included extracting environmental variables at species occurrence points and transforming resolution to match modelling requirements. These operations were computationally demanding due to the high resolution and extensive coverage of EO data. The total compute wall time for processing environmental datasets was approximately 2 hours.

The most computationally intensive phase of UC5 was the modelling and prediction stage, which involved training multiple individual models, generating predictions, and evaluating results using TSS-weighted scores in a final ensemble model. The full modelling process had a compute wall time ranging from 4 to 10 hours, depending on the number of occurrence data for the species included. Memory consumption during this phase remained high, reaching up to 150 GB, particularly during model training and final prediction generation. The ensemble approach, while improving predictive accuracy, significantly increased computational costs due to the multiple iterations required.

Table 16 : UC5 Processing resources usage overview

	Occurrence datasets	Environmental datasets	Ensemble model
Compute platform	Laptop	Desktop	Desktop
Storage			
Data size in grid points (GeoTiff)	3444 × 5602	4600 × 6500	4600 × 6500
Data size in MB/GB	28.3 MB	441 MB	441 MB
Intermediate data size in MB/GB	32.3 MB	441 MB	441 MB
Data formats	CSV, GeoTiff	tif	tif
Main memory			
Available on machine	8 GB	190 GB	190 GB
Consumed on machine	0.28 GB	150 GB	150 GB
Compute resources			
Description of CPU/GPU	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx/AMD Radeon(TM) Vega 8 Graphics	AMD Ryzen 9 9950X, no GPU	AMD Ryzen 9 9950X, no GPU
Compute wall time	3 h	< 2 h	4 to 10 h
Max. energy consumed			
CO2 consumed			
Network			
Network traffic in MB/GB	Used, but not measured	Used, but not measured	
Cost			
Storage	No direct costs (local PC); S3 bucket	No direct costs (local PC); S3 bucket	No direct costs (local PC); S3 bucket (covered by the project)

	(covered by the project)	(covered by the project)	
Compute	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)
Network	No direct costs (local PC)	No direct costs (local PC)	No direct costs (local PC)
Software environment			
Programming language	R	R	R
Essential libraries	Rgbif, sf, lubridate, dplyr, tidyverse	Tidymodels, tidyverse, randomForest, terra, sf	Tidymodels, tidyverse, randomForest, terra, sf



4 Common processing and ML applications

While executing the diverse FAIRiCUBE use case work, several development activities arose that potentially have a wider application beyond the FAIRiCUBE project. Some are written as handy [python] scripts that can make EO data format handling, processing and output easier (e.g. code listed in the FAIRiCUBE common code repository), some developed from a script into a stand-alone application (QueryCube) or library (WCS/WCPS python library) and some can be seen as extension to a general product development framework (UDF for rasdaman software stack). Further, FAIRiCUBE has explored the query and search functionality provided by large language models (see chapter 0).

4.1 Common code repository

Python code and Jupyter Notebooks that have its origin in specific UC work but have been identified as potentially useful for other use case work were collected in a [common FAIRiCUBE code repository](#)¹. Code is there shall be seen as technical tools for data scientists, i.e. they are not fully streamlined and validated outside its UC domain. Some have however matured well, e.g. the Measurer scripts (as described in 2.1.2) or the command line script to convert geospatial tiff to cloud optimized geotiff files² which is an essential step for visualization of data both in the FAIRiCUBE data preview service vis.fairicube.eu and the scrollytelling stories hosted on the FAIRiCUBE website³. In Table 17, we have compiled a brief overview of existing code and notebooks.

Table 17 : Overview of common code in the respective FAIRiCUBE repository

Name of folder	Brief Description
GBIF occurence eea_grid	producing raster occurrence files from EEA-csv_table
QGIS-functionality	Creating a binary layer from data layers in QGIS
access_data apis	Accessing data cubes from various data sources using Python APIs.
fromTIFF to COG	transforming GeoTiff files into Cloud Optimized GeoTiff (COG)
from VECTOR to RASTER	Transforming vector data (polygons, line & point data) to raster files
geotiff-generation	Exporting python numpy arrays as GeoTiff files
pytorch-verification	creating a binary layer from data layers in QGIS.
quality check	Performing quality checks of raster data ingested into FAIRiCube / SentinelHub
record-computational-demands-automatically	Monitoring and provisioning library to assess the computational resourced during execution of python scripts
registerdata2sentinel	Registering of data on the EOX Lab stack to the sentinel data catalog

¹ <https://github.com/FAIRiCUBE/common-code>

² https://github.com/FAIRiCUBE/common-code/tree/main/fromTIFF_to_COG

³ <https://fairicube.nilu.no/use-cases-2/>

4.2 QueryCube

As part of UC3 workflow we developed a Python-based tool called “QueryCube” which allows users to dynamically access and interact with gridded data stored on the FAIRiCUBE Rasdaman instance. By utilizing the OGC Web Coverage Service (WCS), this tool enables seamless data retrieval and interaction, providing users with the flexibility to specify both spatial locations and the relevant resources for which they wish to obtain data. Users can input custom coordinates (Latitude, Longitude, Datetime) in the form of a csv file, with each coordinate combination carrying a specific ID and specify which layer available in the Rasdaman OGC Web Services to query. By querying the data with given coordinates, corresponding data values are retrieved and given back to the user, creating another tabular file carrying information for specified parameters. This approach eliminates the need for manual data extraction from downloaded layers or predefined areas, enabling a broad range of use cases to retrieve data directly by specifying point coordinates.

The tool exists as Python code and is accessible via the UC3 GitHub repository, in which the primary workflow was implemented through several custom defined python functions, which can either be executed manually or run as [standalone python script](#)¹. Dependencies are documented and can be created by installing the creating a specific Conda environment as provided in the repository. Additional functionalities, like accounting for missing data like approximating Datetime of queries, are available in this software. Another instance of our tool is a [Web Application](#)² hosted by NILU. Both implementations rely on the OGC WCS service provided by rasdaman³, with the FAIRiCUBE-specific WCS access⁴ and FAIRiCUBE credentials for Rasdaman services.

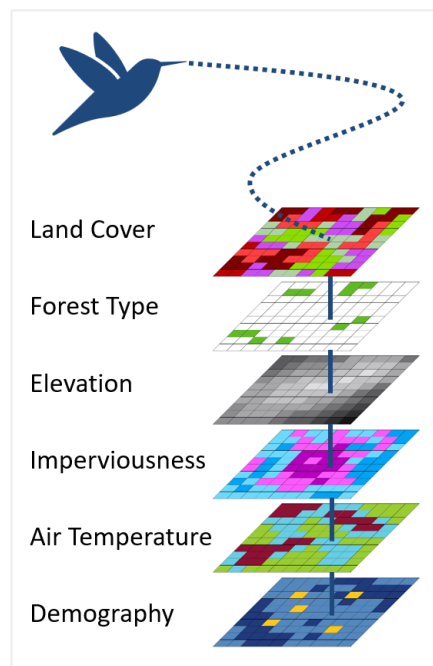


Figure 7 : Illustration of QueryCube functionality taken from the GitHub documentation¹

¹ <https://github.com/FAIRiCUBE/uc3-drosophola-genetics/tree/main/projects/QueryCube>

² <https://querycube.nilu.no/>

³ <https://ows.rasdaman.org/rasdaman/ows#/services>

⁴ <https://fairicube.rasdaman.com/rasdaman/ows#/services>

4.3 WCS/WCPS Python libraries

To facilitate easier interaction with the rasdaman WCS/WCPS services for Python developers, the rasdaman team developed two open-source Python libraries. The WCS python library ([GitHub repository](#), [documentation](#)) allows extracting information about datacubes, such as a list of available datacubes, their bounding box/axes, range type, metadata, etc. Downloading data is not directly supported, as this is already possible with the WCPS library. The WCPS python library ([GitHub repo](#), [documentation](#)) allows to dynamically build WCPS queries and execute on a WCPS server. A query usually selects a subset of the data and applies some processing, aggregation and general analytics on its, and finally downloads the results in a suitable format such as JSON, PNG, TIFF, JPEG, NetCDF, etc. The documentation provides many [examples](#) of the capabilities.

The implementation of these libraries was prompted by the observation that use cases (e.g. UC2 and UC3) needed to do a significant amount of implementation work such as parsing XML to access information about the available datacubes and send queries on them, which is completely unrelated to the goals of the use case itself. Hence, hiding these details behind convenient libraries that a use case can install and use is valuable and worth as soon as there is more than one use case accessing WCS/WCPS services from Python code.

4.4 Rasdaman UDF developments

While rasdaman provides an excellent environment for storing and handling multi-dimensional array data, supporting arithmetic computation, with spatio-temporal awareness, it has no specific support for machine learning at this moment. Neither for using ML models for inference nor for ML algorithms to train models on available data. Rasdaman however is extensible through the addition of User-Defined Functions (UDFs), which can be integrated into the core rasdaman server (see Figure 8) and thus operate directly on the data stored in the data cubes. Such *data locality* can result in performance benefits, which are always welcome in the usual time-consuming and compute-demanding machine learning. Particularly when applying deep learning (DL) models on complex multi-dimensional and large datasets.

Rasdaman supports the use of UDFs at two different levels (see Figure 8). The closest one to the server (*rasserver*) uses C++ or Python as programming language and works on the core data cubes that have no specific notion of spatio-temporal data dimensions. Those are only available at a higher layer that provides rasdaman's spatial capabilities. At this level, the UDFs use the Java programming language, and it is where *Web Coverage Processing Service* (WCPS) requests are processed first in rasdaman.

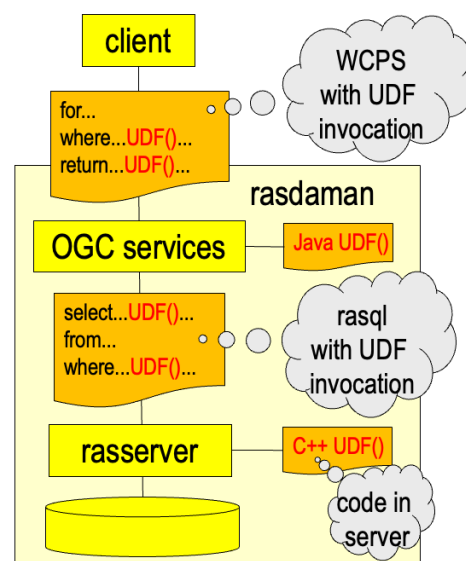


Figure 8 : User-Defined Functions in rasdaman (from Rasdaman documentation)

Since deep learning models are frequently used for processing Earth Observation (satellite) data, it seems particularly relevant to add DL capabilities to rasdaman, with the (reasonable) expectation that it will be needed to realize the use case. The known main applications of DL in EO are (1) image classification, (2) image (pixel) segmentation, (3) object detection, and (4) image generation (new types of applications are sure to be added in future).

To implement a first proof-of-concept of DL integration into rasdaman we selected work on image segmentation, which can be used for many purposes and could be based on an available convolutional neural network (CNN) model developed and trained in another project for the classification of Dutch crop types using Sentinel-2 data as input. This model has been developed using the Python PyTorch framework.

Initially, the core rasdaman server had no direct support for Python UDF code. Hence a bridge needed to be established between these two worlds of C++ and Python, which has been achieved by (1) the use of standard PyTorch functionality to 'trace' the existing model into TorchScript format (which has no Python dependencies), and (2) writing a rasdaman UDF in C++ using the libtorch implementation of PyTorch to load and run the TorchScript model (see Figure 9). All code and further information are published as part of the FAIRiCUBE project on [GitHub](https://github.com/FAIRiCUBE/rasdaman-ml-udf)¹.

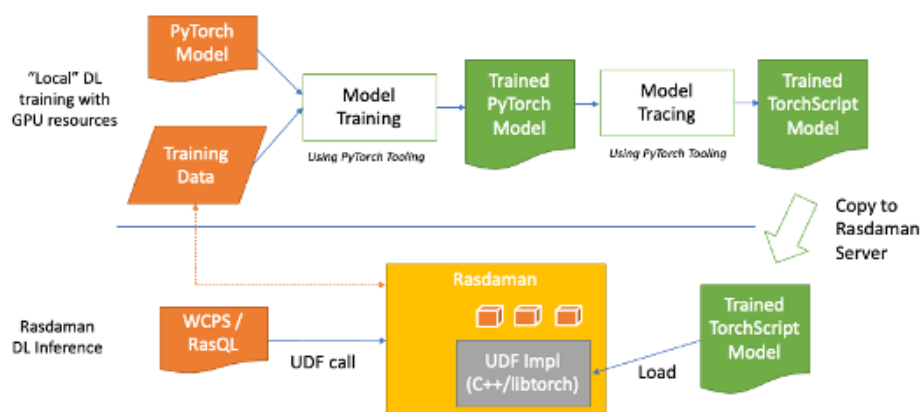


Figure 9 : Integration of DL model inference in rasdaman

This approach allows using a pretrained model for *inference* on data in data cubes. Since the UDF has been added at the core level, a further bridging UDF (in Java) was necessary so that WCPS requests can be used to start model inference and return the results as a geo-referenced data cube.

The UDF-based DL solution described before has been deployed on a Virtual Machine (VM) provided by rasdaman for use by the use cases. Some initial testing on this configuration has been performed, showing that the solution works and can produce the same model inference results as the original model when run on a local computer. An example of the WCPS query with which the model can be invoked is given in Figure 10.

¹ <https://github.com/FAIRiCUBE/uc2-agriculture-biodiversity-nexus/tree/main/rasdaman-ml-udf>

you@wcps> 14. Executing WCPS query. 92 seconds passed. Done.

```
for $sentinel2 in ( sentinel2_2018_flevopolder_10m_7x4bands ),
$maxes in ( maxes_sentinel2_2018_flevopolder_10m_7x4bands )
return
  encode( fairicube.predictCropClass($sentinel2[E(674900:729800),N(5832260:5853960)],
    $maxes),
    "tiff" )
```

Figure 10 : Example WCPS query for applying the crop classification model

A result produced by the model can be seen in Figure 11. For actual use however the 76 possible crop classes that the model can infer should not be visualized with a continuous colour ramp as used in this image. It is also noteworthy that for post-processing and interpretation of the output returning the per-class probabilities and not only the inferred class with the maximum probability is needed. However, the inferred crop classes in this test output exactly match those inferred by the original PyTorch model when used outside of rasdaman.



Figure 11 : Model inference result, showing crop classes by green to yellow colour ramp

On the provided rasdaman VM an initial test of this proof of concept has been carried out to get a sense of its scalability and compute requirements. The VM runs Ubuntu Linux 20.04.6 LTS on an x86_64 architecture, with 8 vCPUs (Intel Xeon) @ 2.3 GHz, and 32 GB memory. For the Python programming, Python 3.10 was used, and PyTorch 1.13.x (2.0.1 works as well). The C++ programming has been done in C++ 14, and a matching version of the libtorch library. All further needed C++ libraries for rasdaman were already provided on the server.

The tests were based on the WCPS request shown in Figure 10, by varying the selected portion of the input data (image) to use for model inference and measuring the wall time (wall-clock time, which is the elapsed real time, so different from only the CPU processing time). The results are presented in Table 18 with the listing of computational resources in Table 19.

Table 18 : Rasdaman crop classification inference - WCPS wall-clock times

Input portion	Width (pixels)	Height (pixels)	Total Pixels	Size	Wall-clock time (seconds)
25%	1373	542	744166	3.0 MiB	6
50%	2745	1085	2978325	11.4 MiB	25
75%	4118	1627	6699986	25.6 MiB	52
100%	5490	2170	11913300	45.5 MiB	92

These test results show that the current scaling is linear (see Figure 12), not only in processing time but in memory usage as well. An issue to be addressed then (typical for deep learning on satellite imagery) is that larger images quickly don't fit into the available memory (either CPU or GPU) anymore and some form of data partitioning (and distributed processing) must be implemented to be able to handle model (training and) inference for larger areas. Potentially some of Rasdaman's distributed architecture can be leveraged for this (to be further investigated).



Figure 12 : Initial inference scalability testing

Table 19 : UC2 Processing resources usage overview

Example crop classification inference	
Compute platform	Multi-user VM, shared rasdaman instance
Storage	
Data size in grid points	5490px x 2170px (x 28 channels) (input), 5490px x 2170px (x 1 channel) (output)
Data size in MB/GB	668 MiB (input TIFF), approx. 45 MiB (output TIFF)
Main memory	
Available machine/node	on 32 GiB
Consumed machine/node	on Not measured, estimated at least 22 GiB (69%) based on neural network architecture
Compute resources	
Description of CPU/GPU	8 vCPUs on Intel Xeon @ 2.3 GHz
Compute wall time	92 s (measured in Jupyter Notebook cell executing the WCPS request)
Max. energy consumed	Unknown – not measured on VM
CO2 consumed	Unknown (cannot be calculated)
Network	
Network traffic in MB/GB	Data already in data cube on a server, 45 MiB download of inference result
Cost	
Storage	Undetermined (included in the cost of VM provisioning by rasdaman)
Compute	Undetermined (included in the cost of VM provisioning by rasdaman)
Network	Undetermined (included in the cost of VM provisioning by rasdaman)
Software environment	
Programming language	Python, Java, C++
Essential libraries	rasdaman 10.x, PyTorch



Subsequently, rasdaman developed support for natively writing UDFs in Python code. Parameter passing between the rasdaman engine and the Python UDF is made as convenient as possible through intelligent transparent conversion: numerical and string values are passed as native Python types, while array values are passed as NumPy arrays in the Python code. This allows for a much easier integration with the Python-oriented machine learning community and prevalent ML frameworks such as TensorFlow, PyTorch, NumPy, and SciKit-Learn. As an example, the previously discussed C++ UDF was reimplemented as a much simpler Python UDF (code available on [GitHub](#)).

As of now, the UDF concept is mostly suited for the inference of pre-trained models and therefore relies on the training of ML models on other infrastructure and the compatibility of these models. Especially when transferring an ML model from a GPU infrastructure to a CPU environment, this interoperability might not be guaranteed if it is actually possible at all. For less demanding ML methods a similar Python environment can be constructed on both the training and inference infrastructure to allow for transferability of models.

Furthermore, the ML models that can be deployed as UDF only reflect the training aspect given the feature input data and the objective of the ML training. To make ML models available for wider applications, concepts like transfer learning and adaptation of parts of the model architecture in the last neuron-layer(s) is required and can currently only be applied outside of the rasdaman environment. In short, ML models deployed as UDF modules can and shall only be used for sufficiently similar format and dimensionality of data and ML-objective as they have been trained for, which restricts the wide-ranged reuse of the models (a typical concern for ML models). The most immediate issue is however posed by the operationalisation of ML models from training to inference infrastructures.

4.5 FAIRiCUBE ChatBots

4.5.1 WCPS ChatCube

To facilitate faster learning and writing of WCPS queries, the rasdaman team has implemented a chatbot like ChatGPT, but with improved knowledge and accuracy on questions about WCPS. The service is available on fairicube.rasdaman.com. An example conversation is shown in Figure 14.

ChatCUBE

Access & analyze datacubes through the powerful **OGC WCPS geo datacube language!**

New to WCPS? Then simply ask our chatbot any question about it:

Conversation started

Conversations are limited to 6 questions currently to ensure quality and accuracy. You can start a new conversation after, you're not limited in the number of conversations you can have.

👤: How can I subset a datacube corine_land_cover on Lat/Lon axes in WCPS?

🤖: To subset a datacube named `corine_land_cover` on Latitude (Lat) and Longitude (Lon) axes in WCPS, you can use the trimming operation. Here's an example that subsets the datacube over specific Lat and Lon ranges:

```
for $c in (corine_land_cover)
return
  encode($c[Lat(40:50), Lon(20:30)], "image/tiff")
```

COPY

This query selects a range of values along the Lat axis from 40 to 50 and along the Lon axis from 20 to 30. The result is encoded as a TIFF image.

👍 🏹

SEND **NEW CHAT**

Figure 13 : Screenshot of the FAIRiCUBE WCPS ChatBot

4.5.2 Knowledge Base ChatBot

The FAIRiCUBE chatbot ¹ is an intelligent assistant that acts as a guide to FAIRiCUBE resources. It is a useful tool for those wishing to interact with the project, thanks to its ability to provide contextually relevant answers, coupled with a strong retrieval mechanism. The chatbot support users to find, navigate and understand the FAIRiCUBE Digital Library² and GitHub artefacts through a friendly interface illustrated in Figure 14.

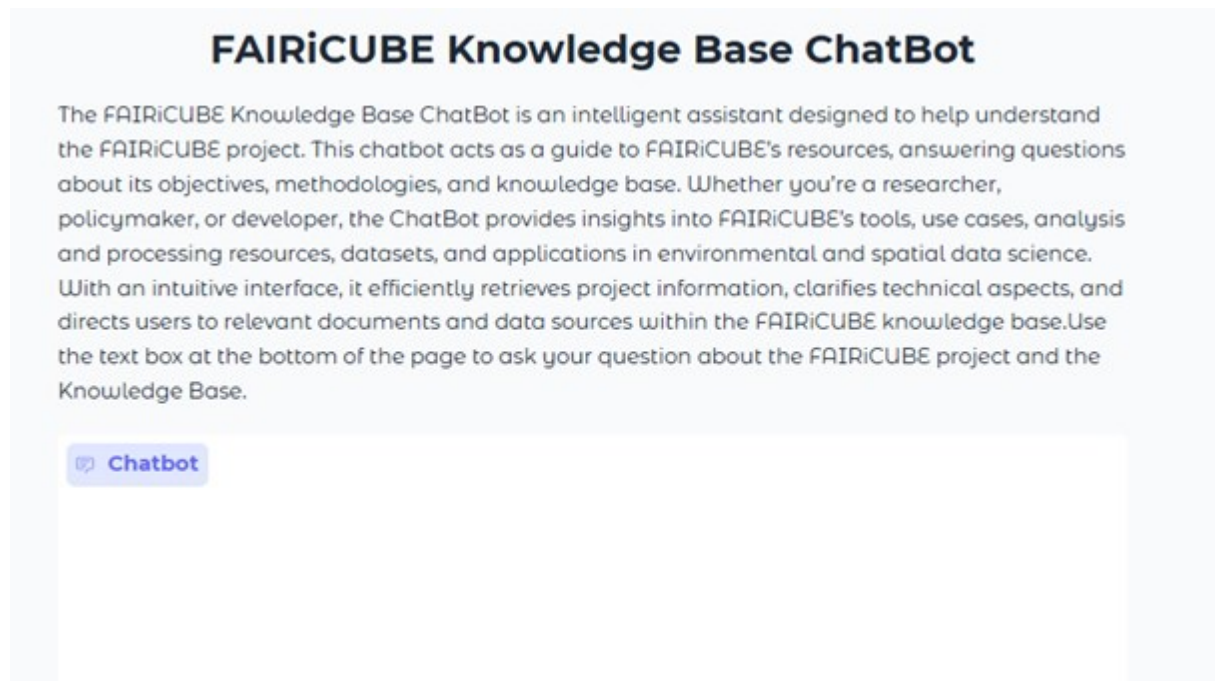


Figure 14 : Screenshot of the FAIRiCUBE KB ChatBot

The chatbot application integrates Retrieval-Augmented Generation (RAG) - a technique that extends the capabilities of generative AI models by allowing them to retrieve and use information from external sources - with state-of-the-art technology to ensure that answers are accurate, context-sensitive and reliable.

Specifically, the integration of retrieval-based search and generative AI allows the chatbot to seamlessly explore and work with both structured and unstructured information. The chatbot's RAG-based architecture integrates three key components - OpenAI's text-embedding-3-large model, chromaDB and GPT-4o - into a unified system. This combination enables the chatbot to provide real-time access to relevant information, while augmenting it with natural language generation capabilities. The use of chromaDB for information retrieval ensures that the system is scalable as new data is incorporated, allowing it to adapt to changes and enrich the FAIRiCUBE project information.

When a user submits a query through the interface of the chatbot, the input is processed by the OpenAI text-embedding-3-large model, which embeds the query in a high-dimensional vector space. This query vector is then passed on to ChromaDB, an open-source vector database which is the chatbot's retrieval engine. ChromaDB with pre-computed embeddings for relevant knowledge base documents identifies the most contextually appropriate content in the shortest time. The vectors in ChromaDB were obtained from the GitHub artifacts and all Knowledge Base Digital Library documents and information using

¹ <https://fic-chatbot-v2-1061322294515.us-central1.run.app/>

² <https://fairicube.readthedocs.io/en/latest/>



OpenAI's text-embedding-3-large model. The obtained information is combined with the user's query and sent to the large language model, GPT-4o to generate a response coherent, precise, and similar in tone and structure to human communication. By anchoring responses in the data provided through ChromaDB, GPT-4o minimizes the risk of hallucinated or factually incorrect responses. Combining active information retrieval and generative reasoning ensures that the chatbot provides answers that are both conversational and based on real data. More details about the FAIRiCUBE Chatbot can be found in the ultimate version of Deliverable D3.4 Processing knowledge base services.



5 Summary and conclusion

This deliverable report provides a technical overview of processing and machine learning applications within FAIRiCUBE, focusing on resource monitoring, computational requirements, and financial tracking. It documents methods for monitoring resource usage, including manual and automatic approaches, and evaluates their effectiveness. Key insights into financial tracking of AWS resources highlight cost management strategies for cloud-based ML applications.

The use case analysis covers diverse applications and demonstrates the computational demands of various (pre)processing steps as well as machine learning applications, including clustering, species distribution modeling, causal inference, and deep learning. They also highlight challenges in balancing accuracy with computational efficiency.

Additionally, the report presents shared tools and developments that have originally been developed as part of the specific use case but can have a wider use outside of the project and thereby contribute to open science principles. The webservice tool QueryCube can easily provide geospatial data access for point data locations, the Python libraries allow for easier WCPs service access, and AI-driven chatbots enhancing the user interaction with project resources. Several other scripts and notebooks are provided in a common-code repository that especially cover transformation and output of geospatial data formats.

Overall, this report provides valuable insights into optimizing ML applications, resource management, and computational efficiency of data science work executed on FAIRiCUBE Hub services.