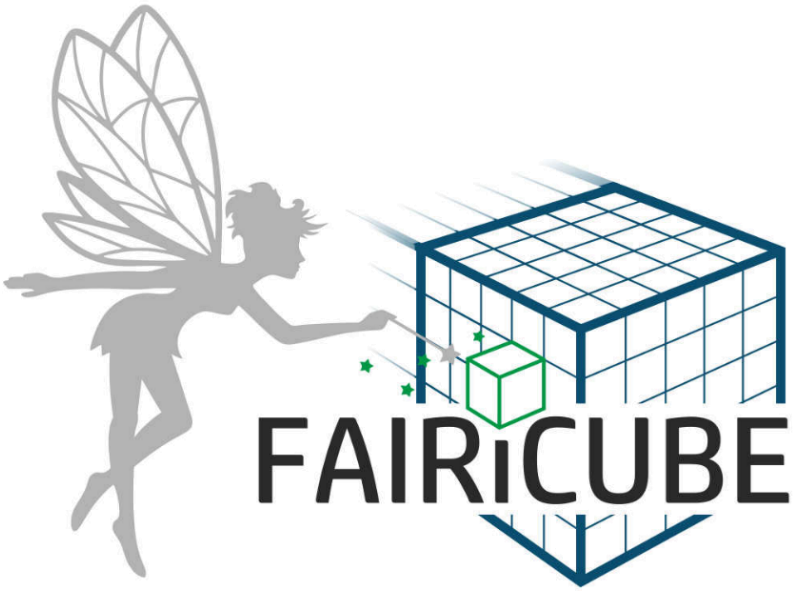

FAIRICUBE – F.A.I.R. INFORMATION CUBES

Project Number: 101059238

WP 3 Process

D3.3 Processing and ML applications

Deliverable Lead: NIL
Deliverable due date: 30/06/2024

Version: 3.2
2024-05-13

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	30/06/2024
Type	Text, Data
Language	EN-GB
Rights	Copyright "FAIRICUBE – F.A.I.R. information cubes"
Audience	☒ Public ☐ Confidential ☐ Classified
Status	☐ In Progress ☐ For Review ☒ For Approval ☐ 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

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.

Table of Contents

List of Figures

List of Tables

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 wants to also 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*.

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 is the optimal solution the most accurate one but a well-selected compromise of resources and sufficient accuracy.

In this chapter, we will begin by providing a brief overview of the monitoring methods that we applied (chapter 2.1). Subsequently, we will present a detailed account of the computational tasks/jobs performed for each use case, along with hardware/software resources consumed during the execution (chapter 3.1 - 3.4.). As use case 2 (UC2) has spent significant resources on the development of user-defined functions (UDFs) which are executed close to the Rasdaman database engine and lay out the foundation of machine learning applications, there will be a main focus on the UDF development (chapter 3.2). All other use cases did not significantly design and/or deliver processing steps and will focus on the listing of the processing resources.

2.1 Monitoring methods

Monitoring of computational resources and demands have long been driven by limitations of the availability of 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 parameters and methods to measure these parameters that appear useful to us 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 (laptop) or pseudo-local (single virtual machines) hardware where direct access to build-in monitoring tools are available (see chapter 2.1.1) or where we can execute automatic resource monitoring scripts (see chapter 2.1.2).

2.1.1 Manual

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	
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

Monitoring on EOX Hub requires an account associated with the hub in question. Firstly, a GitHub account is required with access to the FAIRiCUBE project. Secondly, you communicate your GitHub to EOX to access the urban-climate hub in our case (i.e., UC1 hub). Finally, you connect to https://eoxhub.fairicube.eu using your GitHub credentials. Once connected, the user can create, run, and share Jupyter notebooks using **AWS (**Amazon Web Services) **resources** configured by EOxHub.

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. In addition, one can specify a configuration that is planned to be used (memory, clock speed,…). The exact AWS allocated CPU changes with time, and it is hard to identify the exact used type of CPU. However, using the billing report (see Figure 1), the CPU that was most likely used during UC1 processing is **"c5.4xlarge"**[1] (with memory limited to 7GB and clock speed to 1.8 GHz following the configuration of the profile). Interestingly, the memory usage is provided in real time in the EOXhub app, and JupyterHub interface, so we can check the exact memory usage of a given process.

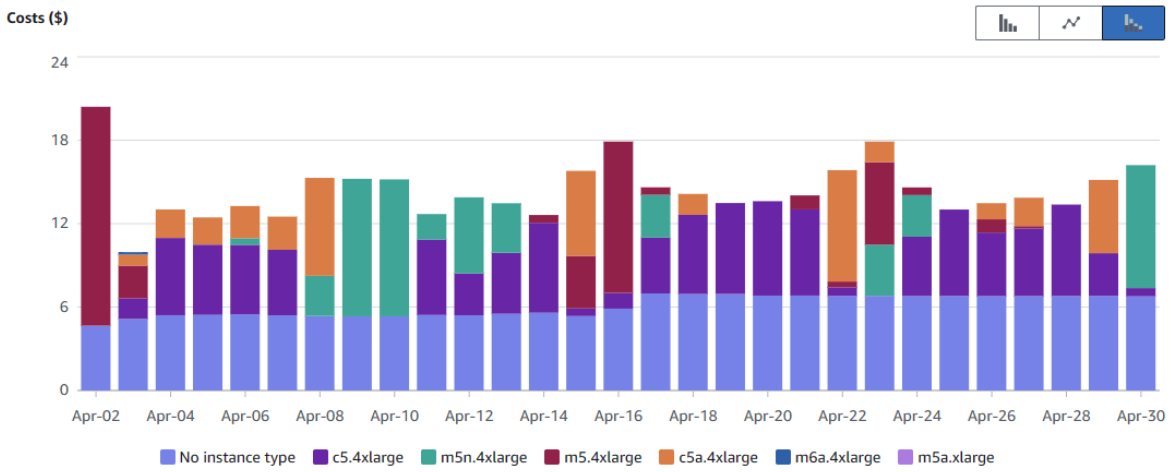

Figure 1 : AWS billing report for April 2023 (where most of the work has been run).

2.1.2 Automatic

To automatize resources monitoring we have implemented a python library called Measurer. Measurer can be simply called from any python source code to compute parameter values.[2] In Table 2, we report

[1] Check the complete list here: https://aws.amazon.com/ec2/instance-types/

[2] https://github.com/FAIRiCUBE/common-code/tree/main/record-computational-demands-automatically

each measure and how it is calculated in our library. Note that at this level our library works only on python scripts.

Table 2 : How does Measurer compute the metrics values?

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_2eq] (kg)	Use the EmissionsTracker function from the 'codecarbon' library
Programming language	Returns e.g. 'python'
Essential libraries	Read lines in the code that contain 'import' and return the list of libraries

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 running on EOXHub.

	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)	0.0 MB	100%
Main memory			
Available on machine/node	32 GB	61.46 GB	92%
Consumed on machine/node	520.5 MB	14.1 MB	97%
Compute resources			

Description of CPU/GPU	C5.4xlarge with 1.8 GHz CPU	x86_64 with 8 physical cores and 16 logical cores 0.0 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 (including elbow method) 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 an interesting initiative to automatize the documentation of computational resources. However, at this level, it warrants further improvement to be considered as the main tool. As a result, as for now, we consider manual reporting for the rest of the report.

2.2 Provisioning of resource monitoring data

Our resource monitoring will be ingested and stored on the project's Knowledge Base (KB) using an online form (see Figure 2). The online form for the ingestion of the monitoring resource metadata allows the upload of the CSV file resulting from the measurer's calculations. After the online form submission (i.e., after the user clicks on the submit button) and after having done all the necessary checks on the info provided, the application:

- Creates the related STAC-JSON file (so that the metadata resource can follow the pipeline for publication into the STAC catalog).
- Ingests needed information into the KB Postgres DB tables. In particular, the database contains two tables, one with information on the resources and the other with information on the measurements (linked to related resources by the "resource_id" foreign key). In this way, queries can be made on the measurements related to consumed compute resources using the query tool.

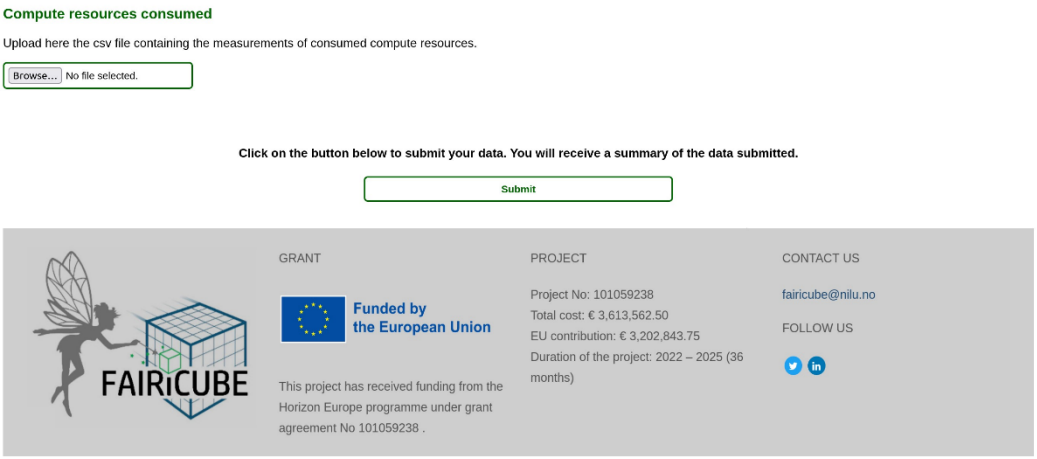

Figure 2 : Knowledge base form for ingestion of the computational resources.

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. Note that UC5 started delayed and is not yet at the data processing stage. 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

Together with use case owners, we share a bucket (called S3 bucket) through the EOxHub profile to ingest, process and share data. We have used the currently configured EOxHub profile with an AWS resource (most likely c5.4xlarge CPU) of 7 GB of RAM and 1.8 GHz CPU. Using Jupyter notebooks, we have succeeded in running our Machine Learning tests (three algorithms using the sklearn.cluster library).

On the ML part, and because the algorithms were not memory/time consuming on this specific study, we did not have any problem running it within the currently configured EOxHub profile. All the code was implemented in python using the library *sklearn* for clustering and seaborn for visualization. For reading the data (that is in CSV format), we have used the Pandas library. The data size is less than 110 KB (only a matrix of size 1,000 * 6, i.e., 1,000 cities and 6 features). Hence, the clustering algorithms were not time/memory demanding. However, generating the csv file from the original data (gridded data) was time-consuming, but still possible (see

Table 5).

Table 1 provides an overview of the resource usage for each main process, starting from the calculation of ratios 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$, calculating level 1 land classification ratios and the clustering processes cost between 20$ and 25$ each for computing (i.e., using AWS resources).

Table 5: UC1 Processing resources usage overview

	Calculating level 1 land classification ratios	Upload the ratios data (csv)	k-means clustering with elbow method	k-means with selected k = 4	Mean Shift (window size = 0.25)	Bidirectional LSTM running (100 epochs)	Gradient Boosting Regressor
Compute platform	AWS configured by EOxHub	AWS configured by EOxHub	AWS configured by EOxHub	AWS configured by EOxHub	AWS configured by EOxHub	AWS configured by EOxHub	AWS configured by EOxHub
Storage							
Data size in grid points	827,993x 402,621					51 x 34	1673 x 5
Data size in MB/GB	6.77 GB	1.1 MB				0.01 MB (input) 0.03 MB (output)	0.05 MB (input) 0.27 MB (output)
Main memory							
Available on machine/node		7 GB		30 GB		30 GB	
Consumed on machine/node	900MB	1.3MB	9.5MB	1.2MB	2.7MB	968.3 MB	520.5 MB
Compute resources							
Description of CPU/GPU		C5.4xlarge with 1.8 GHz CPU		C5.4xlarge with 1.8 GHz CPU		C5.4xlarge with 1.8 GHz CPU	
Compute wall time	2 h 15 min	0.01 s	1.43 s	0.08 s	7.75 s	11.93 s	6.45 s
Max. energy consumed							
CO2 consumed							
Network							
Network traffic in MB/GB							
Cost							
Storage							
Compute	21.59$	24.41$	-	-			
Network							
Software environment							
Programming language	Python	Python	Python	Python	Python	Python	Python
Essential libraries	sentinel hub, shapely, geopandas	Pandas	sklearn.cluster	sklearn.cluster	sklearn.cluster	Numpy, pandas, math, matplotlib.pyplot, time, os, sklearn, keras	numpy, time, os, pandas, math, joblib, matplotlib.pyplot, seaborn, psutil, sklearn

3.2 UC2 Agriculture and Biodiversity Nexus

Design and delivery of processing steps

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 3) 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 3). The closest one to the server (*rasserver*) uses C++ 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.

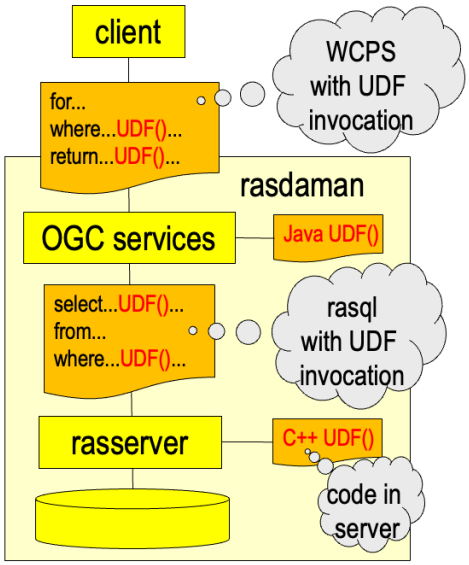

Figure 3 : User-Defined Functions in rasdaman (from Rasdaman documentation)

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 will be added for sure).

To implement a first proof-of-concept of DL integration into rasdaman we selected to 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, while the core rasdaman server is programmed in C++ and has no direct support for Python code. Hence a bridge needed to be established between these two worlds, 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 4). All code and further information are published as part of the FAIRiCUBE project on GitHub[1].

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

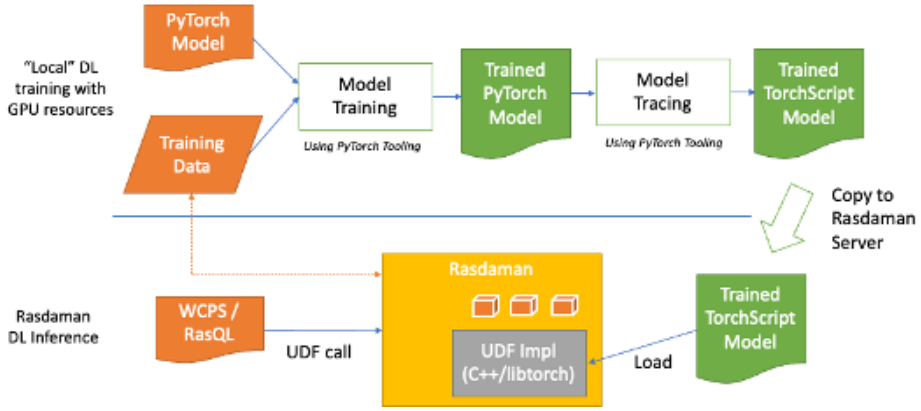

Figure 4 : Integration of DL model inference in rasdaman

This approach allows using the model for *inference* on data in data cubes (supporting model training is a next step). Since the UDF has been added at the core level, a further bridging UDF (in Java) has been added so that WCPS requests can be used to start model inference and return the results as a spatial data cube.

Extensive support has been provided by rasdaman to achieve this implementation. As one of the next developments it is now being investigated if a more direct use of Python is possible for executing ML code as a UDF.

Resource metadata[1] for the example crop classification model used has been entered using the GitHub form as well, so that it can be included in the FAIRiCUBE catalogue.

Processing and ML execution

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 5.

```
you@wcps>: 14. Executing WCPS query. 92 seconds passed. Done.
 for $sentinel2 in ( sentinel2_2018_flevopolder_10m_7x4bands ),
      $maxes in ( maxes_sentinel2_2018_flevopolder_10m_7x4bands )
return
      encode( faricube.predictCropClass($sentinel2[E(674900:729800),N(5832260:5853960)],
            $maxes),
            "tiff" )
```

Figure 5 : Example WCPS query for applying the crop classification model

A result produced by the model can be seen in Figure 6. 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.

[1] https://github.com/FAIRiCUBE/resource-metadata/issues/7

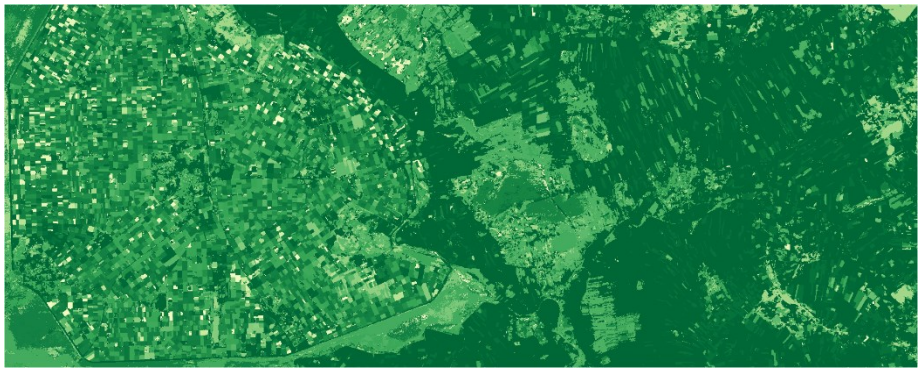

Figure 6 : Model inference result, showing crop classes by green to yellow colour ramp

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 5, 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 6 with the listing of computational resources in Table 7.

Table 6 : Rasdaman crop classification inference - WCPS wall times

Input portion	Width (pixels)	Height (pixels)	Total Pixels	Size	Wall 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 7), 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).

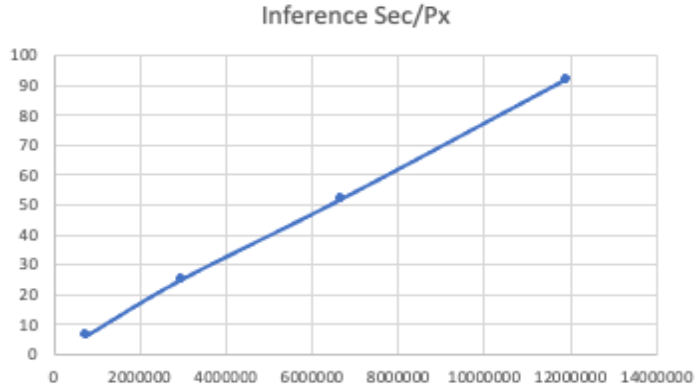

Figure 7 : Initial inference scalability testing

Table 7 : 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 on machine/node	32 GiB
Consumed on machine/node	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

Future work
With this proof-of-concept, the integration of machine learning inference in rasdaman via UDFs has been demonstrated and tested. Rasdaman will further develop and generalize the approach internally and meanwhile has also enabled the use of Python code for writing UDFs. This should allow for easier integration with the Python-oriented machine learning community and prevalent ML frameworks such as TensorFlow, PyTorch, and SciKit-Learn. In case project time and resources allow, use case 2 can perform additional testing of new facilities.

3.3 UC3 Biodiversity occurrence cubes – *Drosophila* landscape genomics

According to the machine learning strategy as described in D3.2 we have developed a gap filling method based on k-means cluster denoted as ML baseline which will first be in focus to list the required processing resources (Table 8). 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 8 : 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 (oputput)	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 (

Table 9) 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.

Table 9 : 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

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 10 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.

Table 10 : UC4 Processing resources usage overview

	Building height estimation			
	Factor x Number of levels	**Geoclimate-Random Forest**	**DTM - DSM**	**Energy demand calculation (Oslo)**
Compute platform	Laptop	Laptop	Laptop	Laptop
Storage				
Data size in grid points (GeoTiff)	1218x1126	1526x1367	1203x1022	-
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
Main memory				
Available on machine	32GB	32GB	32GB	32GB
Consumed on machine	8GB	6GB	17GB	530MB
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
Compute wall time	2min	10min	5min	56min
Max. energy consumed				
CO2 consumed				
Network				
Network traffic in MB/GB				
Cost				
Storage	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)
Network	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
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

3.5 UC5 Validation of Phytosociological Methods through Occurrence Cubes

The use case on the *validation of phytosociological methods through occurrence cubes* (UC5) started with a significant delay compared to the other use cases due to staffing issues and exploitation of synergies with a sister project funding under the same call. UC5 aims to validate the traditional methods

applied in phytosociology to characterize and classify plant communities and to develop a new phytosociological approach to characterize and predict the presence of plant communities for yet unknown localities. This will be approached by linking distribution data of plant taxa and vegetation communities based on habitat types with EO environmental data.

Currently, UC5 is in the data exploration phase which is prerequisite for entering the machine learning phase. No significant data processing or machine learning algorithms has been applied within the UC5 work and therefore no consumption of resources can be reported and documented yet. A future scheduled update of this deliverable will cover the progress from UC5 as well.

4 Summary and conclusion

All use cases (except UC5) have now started to execute data-driven processing techniques to answer their respective scientific questions and we have only started to document the computational resources that have been used. This collection will grow over time and will be a valuable catalogue to estimate the demands for similar tasks, give insights for further optimization and is an essential input to weight computational costs with gained improvements.

Currently, we have started the standardization of collecting numerical parameters as tables, and we will assess the need for any additional parameters in the future. Furthermore, we have implemented a library to compute these parameters more systematically, automatically, and transparently. This library warrants further development to consider other parameters and improve the calculation of the current ones. In addition, we employed a form to ingest the parameter values to the project's knowledge base to keep track of all use case experiences.