Limpopo River Basin Digital Twin Open Data Cube Catalog

Abdul Afham [1], Paulo Silva [1], Surajit Ghosh [1], Zolo Kiala [1], Hugo Retief [2], Chris Dickens [1], Mariangel Garcia Andarcia [1]

[1] International Water Management Institute, [2] Association for Water and Rural Development

INFORMATION

Flagship	Digital Twin
Work package	Real time monitoring
Partners	IWMI, Leona M. and Harry B. Helmsley Charitable Trust, AWARD, AWS, Digital Earth Africa

Citation

Afham, A.; Silva, P.; Ghosh, S.; Kiala, Z.; Retief, H.; Dickens, C.; Garcia Andarcia, M. 2024. *Limpopo River Basin Digital Twin Open Data Cube Catalog*. Colombo, Sri Lanka: International Water Management Institute (IWMI). CGIAR Initiative on Digital Innovation. 22p.

ABSTRACT

A Limpopo River Basin (LRB) Digital Twin, a significant technological innovation, is being developed to assist water managers to assess and manage the water resources of the basin in sustainable ways. At the core of this innovation, the LRB Digital Twin compiles and serves a variety of raster datasets with unique spatial and temporal resolutions tailored to the basin's characteristics. These datasets include hydrological variables (e.g., precipitation), climate indicators (e.g., temperature), and land surface characteristics (e.g., vegetation cover, land use). Each data set compromises with advanced preprocessing, fusion, and calibration to ensure consistency, accuracy, and reliability, making it suitable for long-term analysis and immediate decision-making.

The Open data cube (ODC) framework was used to organize, visualize and analyze the datasets for different stakeholders of LRB effectively. Automations were built on top of the ODC stack for a smooth workflow, which include steps like creating and indexing the metadata into a database. The present report describes the structure ODC follows in cataloging raster datasets, automations in place for handling ODC tasks and implementation steps taken to get ODC and its components up and running.

The available products for the Limpopo region include comprehensive data on irrigated areas and drought index. Additionally, there are environmental flow (E-flow) warnings that provide more insights. These resources collectively support thorough understanding of the region's water and land management dynamics.

This helps policymakers and water managers make informed decisions about resource management and agricultural planning, enabling them to respond more effectively to water-related challenges.

INTRODUCTION

The Limpopo River Basin (LRB) is characterized by relatively arid landscapes at elevation leading to highly variable river flows and a vast coastal plain before entering the sea. The prototype Digital Twin (Garcia Andarcia et al., 2024) aims to assist stakeholders make informed decisions that positively impact the water resource management in the basin. This is done by adding different spatiotemporal layers of earth observation datasets on top of the LRB region for seamless access, visualization and analysis, thus providing a unique analysis of the region.

With the growing amount of available raster data, better management of the data is crucial as it must be organized according to its collection, spatial and temporal fields, bands, projection etc. ODC (Open Data Cube) was used as a tool to organize all the raster datasets for the Digital Twin. ODC is an open-source exploitation tool that enables harnessing the power of geospatial data and cloud computing technology. It uses a set of Python libraries and a PostgreSQL database to manage geospatial raster

data. It is a tool that can catalog raster data so that it can be queried in an efficient manner (Leith, 2018).

The origins of the ODC were in 2011 when Geoscience Australia worked with other organizations on a project to unlock Landsat data stored on tapes into spinning disks. Sometime later, with an upgrade in technology, the Australian Geoscience Data Cube (AGDC) was formed (Leith, 2018).

AGDC was then rewritten as AGDCv2, improving features of the system and in 2017, AGDCv2 was renamed the Open Data Cube (Leith, 2018).

Below are some implementations of ODC that have taken place over the years,

- Digital Earth Australia – Continental scale
- Digital Earth Africa – Continental scale
- Swill Data Cube - Earth Observations Analysis Ready Data for Switzerland
- Brazil Data Cube – Entire Brazilian territory

The Brazil data cube uses AWS as their cloud platform for data storage and processing. They make use of services like AWS Lambda, S3, and Simple Queue Service (SQS) to efficiently handle the process of converting remote sensing data into data cubes.

The concept of a "product", a dataset collection in ODC terms, was introduced into the Digital Twin (see Table 1). Each product displayed on the Digital Twin helps to identify unique geospatial information about the region, leading to support data driven informed decisions to be undertaken. In the prototype of the Digital Twin, seven products were identified for the LRB. The below table contains these seven products along with their descriptions.

The products scale based on how often their datasets are updated. For example, the vegetation condition index product, which looks at vegetation indices of the LRB region, is to be updated every month as new data becomes available.

Each raster layer of a product is managed by creating and storing a metadata document. The PostgreSQL database managed by ODC does the heavy lifting in securely performing database operations to manage all these documents seamlessly. ODC offers command line interactions to assist with these operations. To visualize the raster layers indexed in the database, ODC offers Open Web Services (OWS), which is a spatial web service that uses different web map protocols.

Another major attribute of the ODC is that it can index data directly from AWS Simple Storage Service (s3) since it uses Rasterio and GDAL libraries to read data, hence there is no need to transfer vast amounts of data locally for processing, unless specifically required, depending on the workflow.

ODC also provides a Spatiotemporal Asset Catalogs (STAC) API endpoint to easily access and process the data stored in Cloud platforms like AWS and Azure. STAC is the standard metadata format for geospatial data stored on a cloud service. It provides details about the datasets and links to work with the data without downloading the whole file locally. Figure 1 summarizes how ODC integrates with all these systems.

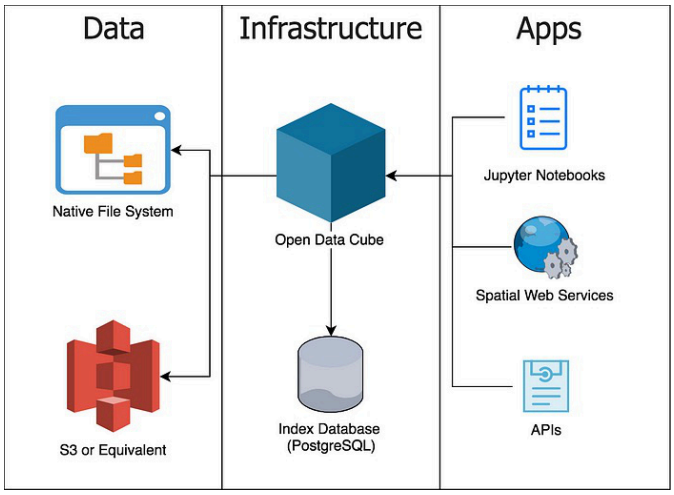

Fig 1: ODC services [source: medium.com]

Table 1. Current products in our ODC index

Product	Description
vegetation_condition_index_limpopo	Monthly vegetation condition index (VCI) calculated using MODIS data to visualize drought extent map produced by IWMI under the Digital Twin project (Ghosh et al., 2024)
eflow_warnings_limpopo	Rasterized average monthly e-flow warnings, derived from SWAT-simulated discharges, implemented by IWMI as part of the LIMCOM Digital Twin project.
incremental_channel_contributions_limpopo	Rasterized average monthly incremental channel contributions, simulated by SWAT and implemented by IWMI under the LIMCOM Digital Twin project.
irrigated_areas_limpopo	Monthly irrigated areas extent map produced by IWMI under the Digital Twin project (Kiala and Karthikeyan, 2024).
land_use_limpopo	Land_use_limpopo
soil_map_limpopo	Soil_map_limpopo
dem_limpopo	DEM (Digital Elevation Mode) limpopo

METHODOLOGY

Open Data Cube

ODC offers a way to index and serve raster data based on the spatial, temporal, and other criteria like measurements, also known as bands or assets in STAC terms.

ODC uses a PostgreSQL database in the backend to handle all the raster datasets. This database can only be interacted with the datacube command. This command is used throughout ODC to index products, metadata, and even in other services that ODC provides such as their Web Map Service (WMS). More about the usage of these commands can be found in the implementation chapter.

Below is the high-level architectural diagram depicting the ODC catalog structure (Fig 2).

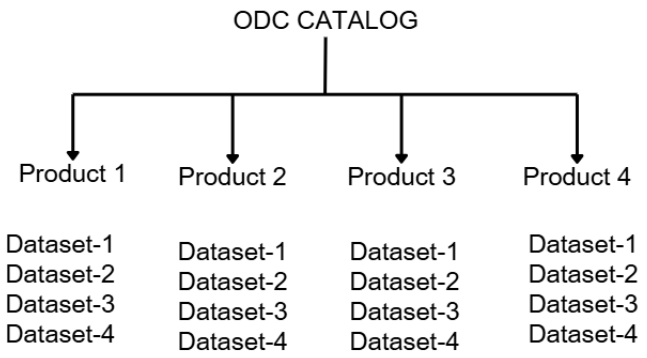

Fig 2: ODC high-level catalog architecture [source: IWMI]

The datasets mentioned here are the metadata of each dataset. Each product (dataset collection) has specific criteria specifying the kind of dataset it can hold. These criteria are specified in a YAML, (a human-readable data serialization language) file called the 'Product definition document'. The structure of this document is shown below, and an example use case can be found in the implementation chapter.

```
name: <product_name>

description: <product_description>
metadata_type: eo3

license: CC-BY-4.0

metadata:
  product:
  name: <product_name>

storage:
  crs: EPSG:<value>
  resolution:
  latitude: <value>
  longitude: <value>

measurements:
  - name: '<measurement_1>'
  aliases: ['<alias>']
  dtype: <data_type>
  nodata: <Nodata value>
  units: '<unit>'

  - name: '<measurement_2>'
  aliases: ['<alias>']
  dtype: <data_type>
  nodata: <Nodata value>
  units: '<unit>'
```

Each dataset's metadata that comes under a specific product must adhere to its geospatial characteristics. These include projection, resolution, and relevant measurements. The irrigated areas product, for instance, contains three measurements i.e., map, filtered and probability, hence the metadata documents that come under this product must contain datasets with one of these measurements.

Indexing datasets into ODC

ODC also has a unique approach to cataloging datasets for a particular product. A dataset for a particular product would have a unique set of spatial (latitude and longitude), temporal (time) and measurement (band) values (presence of irrigated land, drought condition, etc.).

To index this raster dataset into ODC, a metadata document should be created. The format of this document is shown below.

```
id: <id>

label: <label>

product:
  name: <product_name>

location: {s3 URI}

crs: <EPSG: value>

geometry:
  type: <geometry_type>
  coordinates: <coordinates>

grids:
  default:
  shape: <shape>
  transform: <transformation>

properties:
  datetime: <datetime>
  dataset_maturity: <dataset_maturity>
  dtr:end_datetime: <end_date>
```

```
  dtr:start_datetime: <start_date>
  odc:file_format:<file_format>
  odc:processing_datetime: <processing_time>
  odc:producer: <producer>
  odc:product_family: <product_family>
  odc:region_code: <region_code>

measurements:
  <measurement_1>:
    path:<path_to_measurement_1.tif>
  <measurement_2>:
    path:<path_to_measurement_2.tif>

accessories:
  checksum:sha1:
  path: <path_to_sha.sha1>
  metadata:processor:
  path: <path_to_.proc-info.yaml>

lineage: {}
...
```

NOTE: One or more raster datasets may point to the same metadata document if they have the same spatial and temporal fields but record different measurements.

Example: for a particular region and date dataset "region/date/band_1.tif" and "region/date/band_2.tif" are mentioned in the same metadata document but have unique measurements/bands.

The path to the dataset is mentioned in the metadata document using the dataset's URL in the measurements section. This URL can be used to download that dataset, and it is also used by ODC to serve it to the frontend for visualization. Below is the metadata catalog architecture for a particular product dataset. (Fig 3)

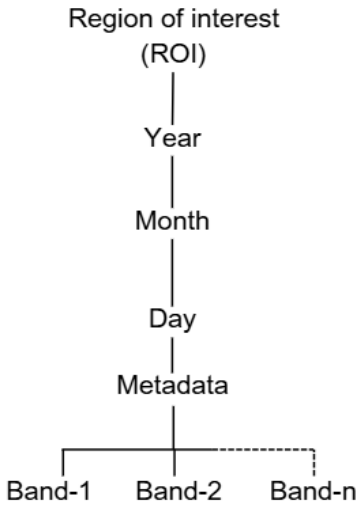

Fig 3: ODC metadata catalog architecture [source: IWMI]

It is not necessary for a product to start cataloging datasets from 'region' if all the regions for the datasets collection (product) are the same. i.e., for some products like irrigated areas, each dataset was grided into chunks of the LRB region during the metadata creation process, for easier analysis. Hence, the products catalog would have a unique code for each region (region code format- x207y023). The rest of the products, however, shared the full Limpopo region as their region geometry, resulting in a catalog structure starting from the 'year'. Figures 4 contains two examples of product geometries, one which is gridded (top) and one that is not (bottom).

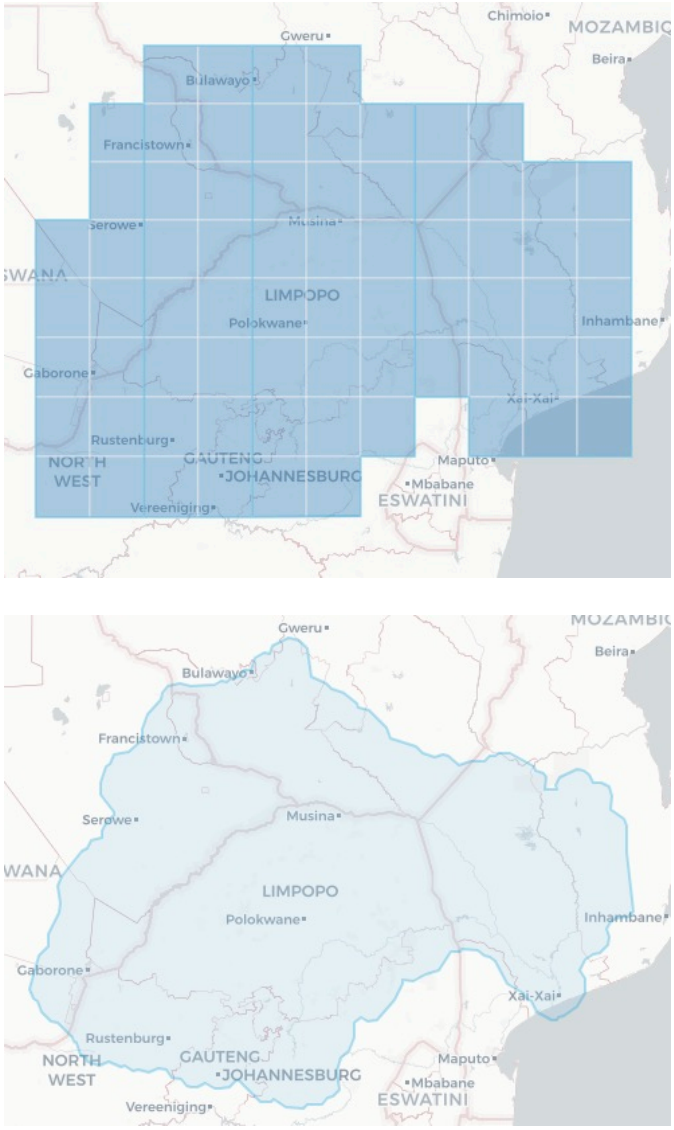

Fig 4: Grid and normal extensions of the data products [source: IWMI]

Hence the final architecture of how ODC catalogs datasets is Region of interest (ROI), time range and finally, bands. This process is shown in Figure 5.

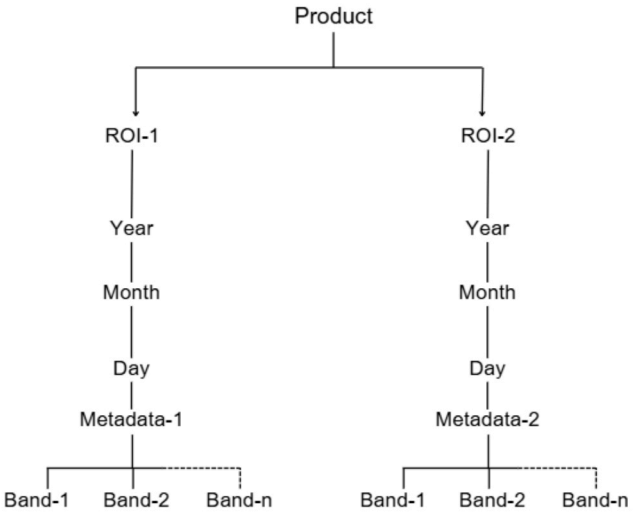

Fig 5: ODC dataset catalog architecture [source: IWMI]

Once the product definition document has been created and added into ODC, the metadata documents can be indexed. This is the underlying methodology used to index datasets into ODC.

Creating products

As mentioned in the introduction, the datacube for the prototype Digital Twin holds seven products.

Products such as land use, soil map and DEM are input layers to the Soil and Water Assessment Tool (SWAT) model while eflow warnings, and incremental channel contributions are products derived from the SWAT model (outputs). (Chambel-Leitão et al., 2024).

Data acquisition for all these products came from remote sensing satellites. The data acquisition and processing methodologies for the SWAT products as well as the other products developed can be found in their respective reports. In simple terms, data was gathered from remote sensing satellites and converted to the preferred format depending on the model that it is to be run on. The output files from the processing were converted to raster's and sent to the developer to be catalogued in a private AWS s3 bucket.

A diagram depicting the process of creating product datasets from these remote sensing satellite data is shown in Figure 6.

If the datasets are to be cataloged for a new product, the developer would need information about this new product to create the product definition document and the metadata document. If all the fields required to fill the product definition document are provided, then only the product_family and the products grided status (should this product be grided or not) must be known to the developer to create the metadata documents.

The private s3 bucket is where all the datasets of each product are sent to be indexed into ODC. This bucket is the landing zone for all the product datasets and is not meant for public use. Figure 7 illustrates the product catalog structure in this bucket.

It follows the same architecture as the front end 'Terria' which is a Catalog based web geospatial visualization platform. Terria

Once the data is stored in the s3 bucket, the indexing process can commence.

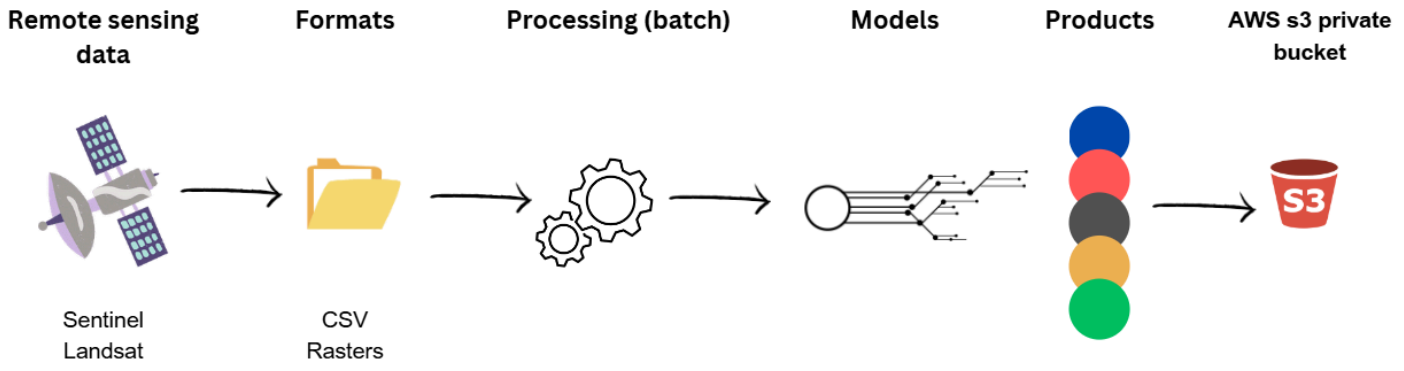

Fig 6: Pipeline: Creating product datasets [source: IWMI]

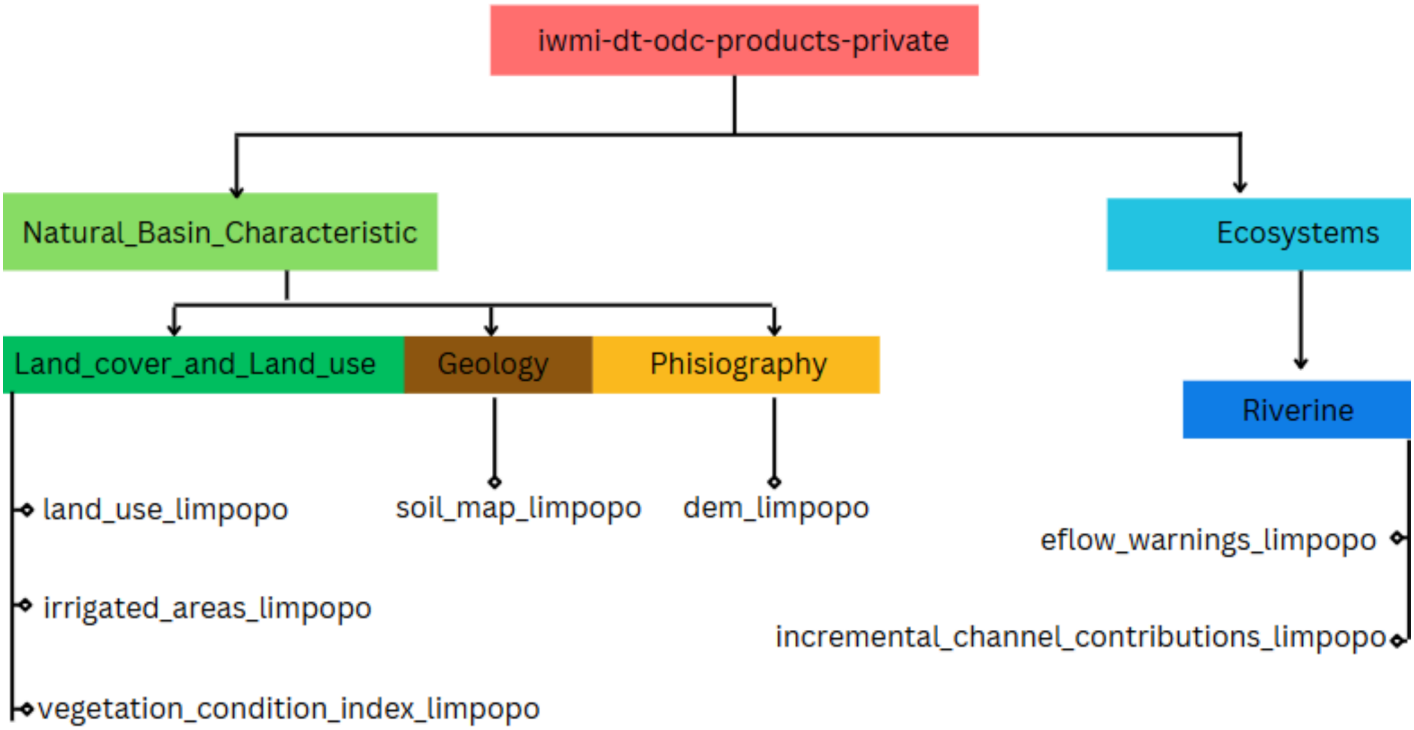

Fig 7: ODC products datasets catalog structure [source: IWMI]

Datacube OWS (Open Web Services)

Datacube OWS is used to serve data indexed in ODC as visualizations, through open web services (OGC WMS, WMTS and WCS). The ODC schema is created and handled by the 'datacube' command, OWS only needs read access to this schema. OWS will create materialized views which are used for intermediate calculation and datasets queries. As the ODC database keeps getting updated with new datasets these materialized views need to be refreshed with the 'update --view' command. OWS Database Documentation — datacube-ows 1.8.42? documentation

With the creation of materialized views, OWS also creates range tables used for caching full-layer spatial temporal extents. More about this architecture can be seen in the OWS documentation provided by ODC. OWS Database Documentation — datacube-ows 1.8.42? documentation. Figure 8 depicts the Entity Relationship Diagram (ERD) of the OWS managed database schemas.

The visualization parameters for each product are mentioned in the OWS configuration document, details can be found in the implementation chapter.

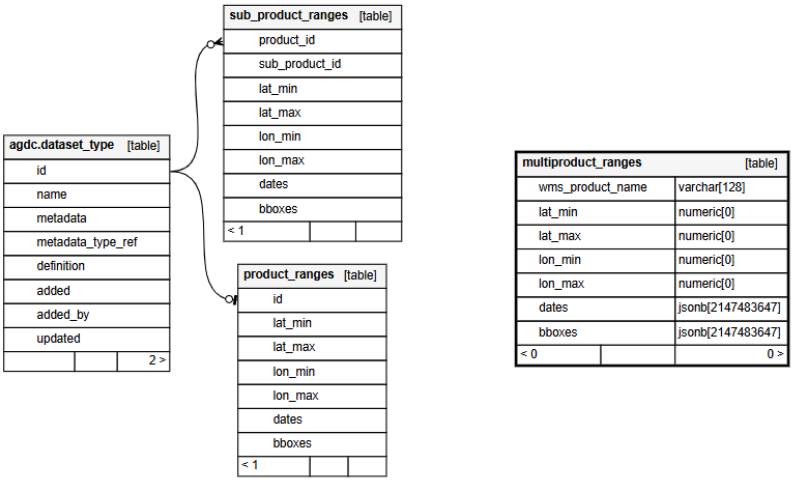

Fig 8: OWS database schemas. [source: OWS documentation]

Datacube explorer

Datacube Explorer is an application that allows users to explore the products in our datacube. Product details can be viewed from the command line by running certain 'cubedash' commands or a flask application could be spun to visualize the products in a more detailed manner. It also provides STAC endpoints to download and analyze datasets in a programming environment. Example provided in the Results chapter, section 5.3.

Deployment of OWS and Datacube explorer

Both the OWS and Explorer application were deployed following the same architecture. i.e., built using the Flask web framework, Gunicorn as the web service gateway interface and Caddy as the proxy web server. Figure 9 illustrates the architecture describing this process.

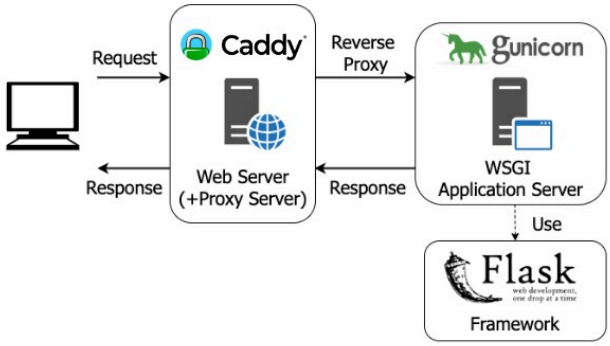

Fig 9: OWS and Explorer deployment architecture [source: IWMI]

IMPLEMENTATION

For the implementation of the ODC software stack and automations, two AWS machines running Ubuntu were used, one with more compute ability, which is only run to create metadata, and another ec2 which holds the components of ODC, i.e., Explorer, OWS, ODC database. All the ODC components were installed in the ec2 with the name iwmi-dt-v1-odc while the ec2 used for metadata-creation, was called iwmi-dt-v1-metadata-creation. Figure 10 shows the AWS architecture diagram.

A lambda function was not used for the creation of metadata due to some of its limitations which include, not able to install packages due to file size limitation even after they were zipped, runtime of the automation went over an hour for certain products and memory constraints during processing. Also, the private bucket is hosted in Mumbai (ap-south-1) since was encountering Rasterio errors while processing datasets, when it was placed in the Cape Town region (af-south-1).

The ODC software stack is composed of datacube-core, datacube-explorer and datacube-OWS. Following are the procedures undertaken to install and configure them in the "iwmi-dt-v1-odc" ec2

Installation and Configuration

ODC installation

An Aws EC2 running Ubuntu was selected (iwmi-dt-v1-odc) to run the ODC stack. The installation process for installing ODC on ubuntu was provided in the Open data cube installation

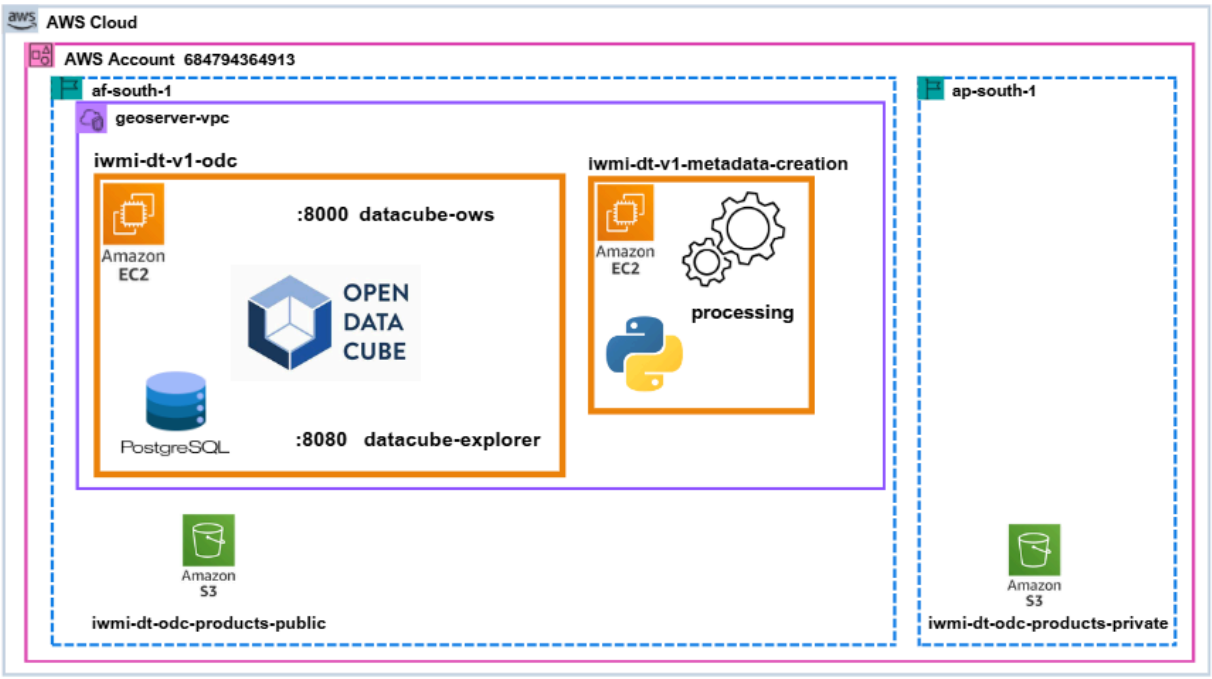

Fig 10: AWS architecture diagram [source: IWMI]

guide. Ubuntu Developer Setup — Open Data Cube 1.8 documentation. Below are the steps we took for the installation and configuration of ODC and Figure 11 summarizes these steps taken.

```
sudo apt-get install libgdal-dev libhdf5-serial-dev libnetcdf-dev
sudo apt-get install postgresql-14
sudo apt-get install postgresql-14-postgis-3

wget https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge-pypy3-Linux-x86_64.sh
git clone https://github.com/opendatacube/datacube-core
cd datacube-core

mamba env create -f conda-environment.yml
conda activate cubeenv
pip install --upgrade -e .
set password for user postgres
create <db name>
edit datacube config file ~/.datacube.conf
[datacube]
index_driver: default
db_database: datacube
db_hostname:
[test]
index_driver: default
db_database: datacube_test
[null]
index_driver: null
[localmemory]
index_driver: memory
initialize database
datacube -v system init
```

As listed above, datacube-core, PostgreSQL, Postgis, and some additional packages for handling geospatial data were installed in the ec2. The datacube configuration file was created with the PostgreSQL database credentials and the ODC database was initialized by running the command 'datacube -v system init' command. At this stage ODC is set up and ready to be indexed with datasets. The 'cubeenv' conda envioroment was the primary python environment used in the 'iwmi-dt-v1-odc' ec2 to install all ODC python packages.

Datacube-explorer installation

Once ODC is installed into the machine datacube-explorer was setup. To do this, the git repository was cloned into the ec2, and the following commands were run.

```
git clone https://github.com/opendatacube/datacube-explorer.git
pip install datacube-explorer
conda install fiona shapely
pip install gunicorn
```

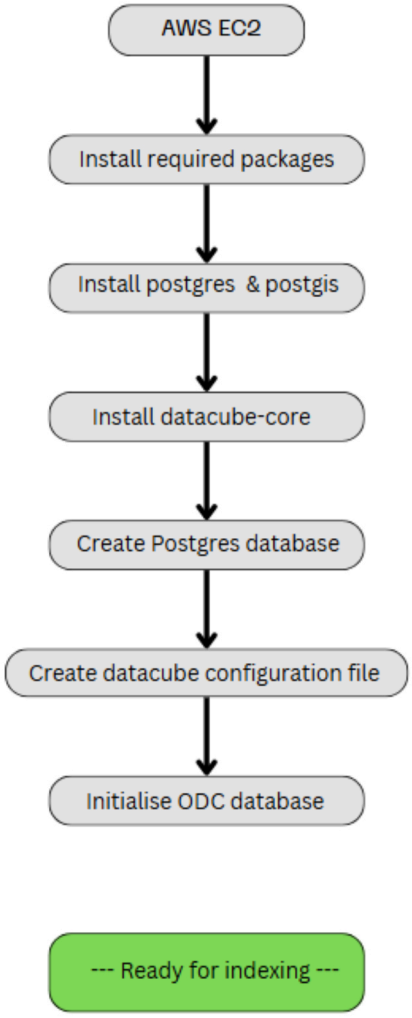

Fig 11: Installation and Configuration of ODC [source: IWMI]

A settings.env.py file was created into the cloned repo to add custom configuration options like changing the AWS region and THEME. Configuration Handling — datacube-explorer Unknown/Not Installed documentation.

Datacube OWS installation

Docker was the recommended approach by ODC for installing OWS but was met with several errors, therefore it was not used. Another approach ODC recommended was using Conda Datacube Open Web Services — datacube-ows 1.8.42? documentation, which after fixing some errors that were caused due to a mismatch in python versions, was proven successful. Hence, this was the path taken in installing OWS.

To install OWS using Conda,

```
pip install datacube-ows[all]
```

datacube-ows can be found in the following path:

```
cd miniforge3/envs/cubeenv/lib/python3.12/site-packages/datacube_ows
```

To connect OWS to ODC, an OWS configuration file was created inside the above-mentioned path. Run the following commands to let OWS have read access to the ODC database:

```
export DATACUBE_OWS_CFG=datacube_ows.ows_cfg_example.ows_cfg
datacube-ows-update --role postgres –schema
```

Whenever changes are made to the ODC database, run the below commands.

```
datacube-ows-update –views
datacube-ows-update
```

AWS command line interface (CLI) setup

Use AWS CLI to handle access privileges to the data stored on s3. This is done by setting up AWS access and secret keys for relevant users on the AWS console. To install AWS CLI for linux run,

```
sudo apt install unzip
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

then run, aws configure to set up the AWS access credentials

Indexing product datasets

As mentioned in the methodology to index a dataset into ODC, two documents are required i.e., a product definition document and a metadata document.

Product definition document

For each product, a product definition document is created manually with unique measurements/bands, according to the format mentioned in the methodology. An example product definition document for the irrigated areas product is shown below.

```
---
name: irrigated_areas_limpopo
description: Monthly irrigated areas extent map produced by IWMI under the digital twin project.
metadata_type: eo3

license: CC-BY-4.0

metadata:
  product:
    name: irrigated_areas_limpopo

storage:
  crs: EPSG:4326
  resolution:
    latitude: -0.0002777777777777778
    longitude: 0.0002777777777777778

measurements:
  - name: 'map'
    aliases: ['Map', 'MAP']
    dtype: uint8
    nodata: 0
    units: '1'

  - name: 'prob'
    aliases: ['irrigated_prob', 'PROBABILITY']
    dtype: float32
    nodata: -32768
    units: '1'

  - name: 'filtered'
    aliases: ['irrigated_filtered', 'FILTERED']
    dtype: uint8
    nodata: 0
    units: '1'
```

The product name must be mentioned in lower case according to ODC standards. Once the product definition documents are created, they can be indexed into the ODC database with the following command

```
datacube product add <product-name.odc-product.yaml>
```

To generate summaries of all the products in the ODC database, datacube-explorer provides the ‘cubedash’ command,

```
cubedash-gen –init all
```

It will output the products in the database and the number of datasets indexed for each.

Once the product definitions are defined the datasets can be indexed from s3. As mentioned in the methodology, this is done by creating metadata documents.

Note: products indexed into ODC cannot be deleted. They, however, can be updated with the following command. The ‘–allow-unsafe’ flag must be mentioned if ODC does not allow the update to go through.

```
datacube product update <product name> --allow-unsafe
```

Metadata documents

Creating metadata documents

As mentioned, the metadata document which contains the spatial and temporal fields, measurements, and projection for each raster dataset was created in a separate ec2, having more compute power. Increasing the compute on the main ec2 was not done due to the cost of having it run all the time, hence the metadata creation ec2 was kept separate and only turned on while processing datasets.

Due to the substantial number of datasets that need to be indexed, unlike the product definition documents, this process requires automation.

To create the metadata document ODC recommends using the eo3 format (shown in the methodology). The Eo-dataset library by ODC is used in creating the metadata document. opendatacube/eo-datasets: Easily write, validate and convert EO datasets and metadata. (github.com) . It can be installed with the following command: pip install eodatasets3 pip install eodatasets3

The basic process of how to create a metadata document using the Eo-dataset library is shown below. (Eo-dataset, EO Datasets 3 — eodatasets3 documentation)

```
from eodatasets3 import DatasetAssembler

with DatasetAssembler(collection, naming_
conventions='default') as p:
  p.product_family = "blues"

  # Date of acquisition (UTC if no timezone).
  p.datetime = datetime(2019, 7, 4, 13, 7, 5)
  # When the data was processed/created.
  p.processed_now() # Right now!
    # (If not newly created, set the date on the field:
`p.processed = ...`)

   # Write our measurement from the given path, calling it
'blue'.
  p.write_measurement("blue", blue_geotiff_path)

  # Complete the dataset.
  p.done()
```

This outputs the following files,

- metadata.yaml
- Cloud optimized geo tiff (COG) (for each measurement written)
- interim.proc-info.yaml
- interim.sha1

The metadata.yaml file is the metadata document and contains the paths to all the other files along with the processed datasets metadata, the COG file is similar to a GeoTiff but aims at being hosted on a HTTP server to enable more efficient workflows on the cloud, the interim.proc-info.yaml file holds the software version of the system, and interim.sha1 file is a hash file for cryptography.

For the product 'irrigated areas' three COG files were produced, one for each of its processed measurement.

Automation: Metadata creation

The 'iwmi-dt-v1-metadata-creation' EC2 was used for this automation. The automation consists of four steps as shown in Figure 12.

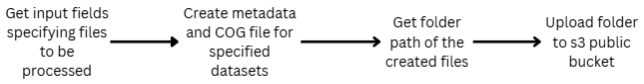

Fig 12: Automation: Metadata creation [source: IWMI]

Four files were used for this automation process,

- inputs.py: file with utility functions
- inputs.txt: file which takes inputs
- main.py: file that is to be run
- processer.py: file that creates the metadata

To start off the metadata creation process, the user must fill the fields in the input file, they are as follows:

```
start_date: <start date>
end_date: <end date>
date_list: <set of specific dates. Would overwrite start date
and end date if filled>
product_name: <product name>
grided: <True or False: based on if the dataset should be
divided into chunks for the Limpopo region>
```

If the date_list field is filled it will not consider the start and end date, only processing for the dates mentioned in the date list. This function handles the creation of the dates to be processed. (Detailed view of all source code can be found on github. iwmihq/dt-odc)

```
def get_date_list(inputs):
    date_list = inputs['date_list']
    if not date_list:
        date_list = []
        start_date = inputs['start_date']
        date_list.append(start_date)
        end_date = inputs['end_date']
        month = date_extract(start_date)['month']
        year = date_extract(start_date)['year']
        while start_date != end_date:
            if month ==12:
                month =0
                year +=1
            month +=1
                        start_date = f'{year}-{month}-{calendar.
monthrange(year, month)[1]}'
            print(start_date)
            date_list.append(start_date)
    else:
        date_list = date_list.strip().split(',')
    return date_list
```

The gridded field notes if the dataset must be chunked and indexed. It would take a considerably longer time to create the metadata this way, due to the gdal.Warp function that need to be used. The grid is extracted from a shapefile, that holds 71 equally sized chunks of the Limpopo region.

There are two processing functions: one for gridded datasets and another for datasets whose geometry is the full LRB region. Their processing functions are as follows:

```
def process_grided(pname):
  shapefile = '/home/ubuntu/Lim_Basin/Lim_Basin.shp'
  fnm = get_m_list(pname)

  for date in date_list:
    date_dict = date_extract(date)
    year = date_dict['year']
    month = date_dict['month']
    day = date_dict['day']
    with fiona.open(shapefile, 'r') as f:
      x=203
      y=23
      n=0
      for row in f:
        with DatasetAssembler(
          Path(''), naming_conventions="default") as p: #
          print(f"Writing for dataset number: {n}")
          p.product_family = fnm[0] #
          p.names.product_name = pname #

          p.datetime = datetime(year, month, day, 0, 0, 0) #
              p.properties["dtr:start_datetime"] = datetime(year,
month, 1, 0, 0, 0)
              p.properties["dtr:end_datetime"] = datetime(year,
month, day, 23, 59, 59)
          p.processed_now()

          p.producer = "https://www.iwmi.cgiar.org/"
          p.maturity = 'interim'

          region_code = f'x{x:03d}y{y:03d}'
          p.region_code = region_code

          polygon = shape(row['geometry'])
          p.geometry = polygon
          bbox = polygon.bounds

          for ms in fnm[1:]:
            my_file = Path(f"{year}-{month}-{ms}.tif")
            if my_file.is_file():
              print(f'{ms}-local')
              src_ds = gdal.Open(f"{year}-{month}-{ms}.tif")
              gdal.Warp(f'Warp.tif', src_ds, outputBounds=bbox)
              if x == 273:
                  time.sleep(1)
                print(f'last grid x{x}, deleting {year}-{month}-{ms}.
tif')
                  os.system(f"rm -rf {year}-{month}-{ms}.tif")
            else:
              def gdalWarp(src_ds,bbox):
                                  gdal.Warp(f'Warp.tif',  src_ds,
outputBounds=bbox)
              uri = get_uri(pname,ms,year, month, day)
              src_ds = gdal.Open(uri)
              try:
                print(f"{ms}-s3")
                func_timeout.func_timeout(27, gdalWarp, args =
(src_ds, bbox))
              except func_timeout.FunctionTimedOut:
                print(f"Function timed out-- downloading dataset
{year}-{month}-{ms}.tif locally")
                s3_client = boto3.client('s3')
                  s3_client.download_file('iwmi-dt-odc-products-
private', uri[36:], f'{year}-{month}-{ms}.tif')
                print('Download complete')
                time.sleep(1)
                src_ds = gdal.Open(f"{year}-{month}-{ms}.tif")
                                  gdal.Warp(f'Warp.tif',  src_ds,
outputBounds=bbox)
                        gdal.Translate(f'Translate.tif',  f'Warp.tif',
creationOptions=["COMPRESS=LZW", "TILED=YES"])
            p.write_measurement(ms, f'Translate.tif')
          x+=1
          y+=1
          n+=1
          p.done()
```

The code iterates over each of the 71 grids in the grided shapefile. Each grid is given a unique region code, ranges from x203y023 to x273y093.

```
def process_ungrided(pname):
  shapefile = 'shapefiles/Lim_Basin/Lim_Basin.shp'
  fnm = get_m_list(pname)

  for date in date_list:
    print(f'Writing {date}')
    date_dict = date_extract(date)
    year = date_dict['year']
    month = date_dict['month']
    day = date_dict['day']
    with DatasetAssembler(
      Path(''), naming_conventions="default") as p: #
      p.product_family = fnm[0] #
      p.names.product_name = pname #
      p.datetime = datetime(year, month, day, 0, 0, 0) #
          p.properties["dtr:start_datetime"] = datetime(year,
month, 1, 0, 0, 0)
      p.properties["dtr:end_datetime"] = datetime(year, month,
day, 23, 59, 59)
      p.processed_now()
      p.producer = "https://www.iwmi.cgiar.org/"
```

```
    p.maturity = 'interim'

    with fiona.open(shapefile, 'r') as f:
      p.geometry = shape(f[0]['geometry'])
    for ms in fnm[1:]:
      print(ms)
      uri = get_uri(pname,ms,year, month, day)
      src_ds = gdal.Open(uri)
                    gdal.Translate(f'Translate.tif',  src_ds,
creationOptions=["COMPRESS=LZW", "TILED=YES"])
      p.write_measurement(ms, f'Translate.tif')
    p.done()
```

Example scenarios: Creating metadata

1) Datasets have been created for the product vegetation condition index from the dates 2022-06-30 to 2024-06-30. Datasets should not be grided. The inputs.txt file should be filled as follows

```
start_date: 2022-06-30
end_date: 2024-06-30
date_list:
product_name: vegetation_condition_index_limpopo
grided: False
```

The dates have monthly increments from the start date to end date since all the product datasets in our ODC are recorded in monthly time periods.

Currently, only the "irrigated areas " product is gridded. This means that for each dataset, a metadata document should be created for each grid. The rest of the products use a shapefile of the LRB which is not gridded.

2) Datasets have been created for the product irrigated areas for the dates 2022-06-30, 2022-07-31 and 2022-8-31. Datasets should be grided. The inputs.txt file should be filled as follows

```
start_date: 2022-06-30
end_date: 2024-06-30
date_list: 2022-06-30, 2022-7-31, 2022-8-31
product_name: irrigated_areas_limpopo
grided: True
```

In this case, it does not matter if the start and end dates are filled since the date list field contains dates to be processed. Also, here gridded is set to True, since it was decided to have the irrigated areas datasets grided into regions, which makes it easy for analysis.

The process.py file contains the processing functions based on whether the datasets should be processed, gridded or not. It uses the eo-dataset library as mentioned above.

To process the datasets from s3, the s3 Uniform Resource Identifier (URI) of the file is used. The s3 URI is created using the dataset catalog structure, mentioned in the methodology.

Eg: s3://iwmi-dt-odc-products-private/Natural_Basin_Characteristic/Land_cover_and_Land_use/vegetation_condition_index_limpopo/VCI3_2022.tif

```
def get_uri(pn, ms, y, m , d):
  if pn == 'irrigated_areas_limpopo':
    if m != 10 and m != 11 and m != 12:
          m =  f'0{m}'
    uri = f'/vsis3/iwmi-dt-odc-products-private/Natural_Basin_
Characteristic/Land_cover_and_Land_use/irrigated_areas_
limpopo/{y}/June/{y}-{m}-{ms}.tif'

  if pn == 'vegetation_condition_index_limpopo':
    uri = f'/vsis3/iwmi-dt-odc-products-private/Natural_Basin_
Characteristic/Land_cover_and_Land_use/vegetation_
condition_index_limpopo/{ms}{m}_{y}.tif'

  if pn == 'incremental_channel_contributions_limpopo' or pn
== 'eflow_warnings_limpopo':
    if m != 10 and m != 11 and m != 12:
          m =  f'0{m}'
      uri  =  f'/vsis3/iwmi-dt-odc-products-private/Ecosystems/
Riverine/{pn}/{y}-{m}-{d}.tif'
  return uri
```

This function returns the URI, depending on the product mentioned. The parameter ‘m’ in the above function stands for the products measurements, which is retrieved from this function

```
def get_m_list(pname):
    m = []
    if pname == 'irrigated_areas_limpopo':
        m.append('Agriculture')
        m.append('map')
        m.append('filtered')
        m.append('prob')
    if pname == 'vegetation_condition_index_limpopo':
        m.append('Agriculture')
        m.append('VCI')
    if pname =='incremental_channel_contributions_limpopo'
orpname == 'eflow_warningslimpopo':
        m.append('Hydrology')
        m.append('LegendIndex')
    return m
```

Hence, every time a new product is indexed, these functions would have to be altered slightly to cater to the new product.

For the product mentioned in the input.txt file, a folder with the product name will be created locally. This folder is cataloged automatically by the eo-dataset library according to the metadata catalog structure mentioned in the methodology. i.e., Region/Year/Month/Day. (Fig 5)

Once all the metadata documents and the COG files are written into this folder, the folder is uploaded to a s3 public bucket.

The s3 public bucket has the same folder structure as the ODC

products catalog in the private bucket. When uploading the product folder, s3 will create relevant folders to mimic the same structure of the uploaded folder, hence the s3 public bucket is cataloged automatically. Sample URI of a metadata document in the public bucket is shown.

Eg: s3://iwmi-dt-odc-products-public/Natural_Basin_Characteristic/Land_cover_and_Land_use/ vegetation_condition_index_limpopo/2022/03/31_interim/vegetation_condition_index_limpopo_2022-03-31_interim.odc-metadata.yaml

This bucket is open to the public to view and download COG files. When a dataset is downloaded from the datacube-explorer application, its URL points to this public bucket.

After all the uploading is complete, this folder is removed from the local machine using the Linux command "rm -rf <product name>".

```
def upload_and_delete(pname):
    time.sleep(1)
    s3C = boto3.client('s3', region_name = 'af-south-1')
     folder_path = f'/home/ubuntu/dt-odc/metadata/{pname}/'
#############
    prefix = get_upload_prefix(pname)
    print(prefix)
    bucketname = 'd-e-testing'
    for root,dirs,files in os.walk(folder_path):
        for file in files:
            folder_name = root[29:]
             s3C.upload_file(os.path.join(root,file),bucketname,
f'{prefix}/{folder_name}/{file}')
    time.sleep(1)
    print("Uploading Complete")
    os.system(f'rm -rf Warp.tif Translate.tif {pname}')
```

To obtain the prefix name of the s3 path, the below function in used

```
def get_upload_prefix(pname):
        if pname =='irrigated_areas_limpopo' or pname ==
'vegetation_condition_index_limpopo':
            prefix = "Natural_Basin_Characteristic/Land_cover_
and_Land_use"
        elif pname =='eflow_warnings_limpopo' or pname
=='incremental_channel_contributions_limpopo':
        prefix = "Ecosystems/Riverine"
    return prefix
```

To run the pipeline, fill the inputs.txt file and run the main.py file using the command, "python main.py".

This is the full automation process currently in place for creating and uploading metadata to s3. Figure 13 illustrates the process flow of this automation.

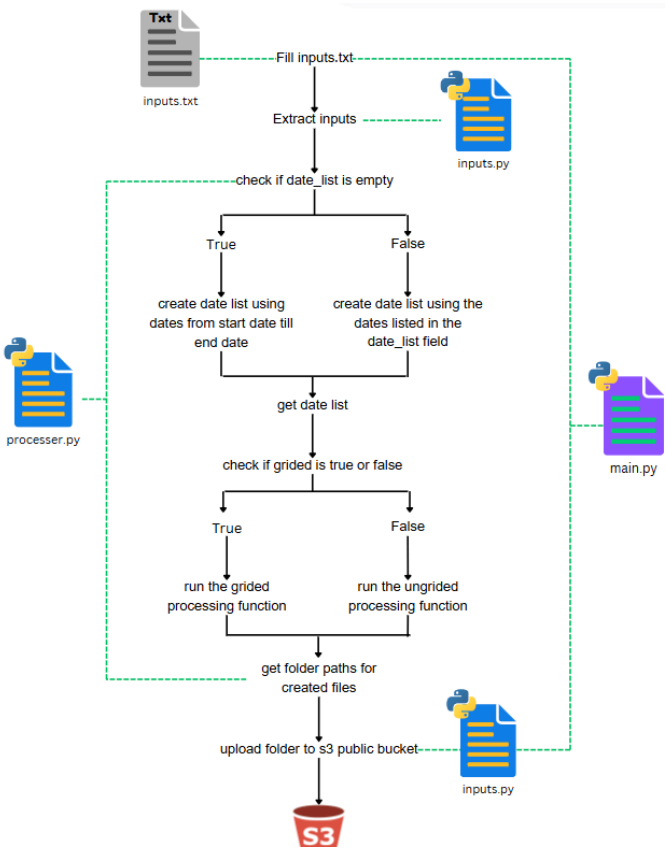

Fig 13: Metadata creation process flow [source: IWMI]

Note: While creating metadata, pay attention to the logs. If encountered with errors related to a particular dataset, it is best to process that dataset locally in the ec2 rather than from s3. This is now done automatically, using a timeout function. The error is commonly seen in the gdal.Warp function during the grided processing. If this function times out, the dataset will be downloaded and processed locally. And after processing it would be removed from the system.

Indexing metadata documents

Once the metadata documents have been uploaded to s3 and cataloged, they are ready to be indexed into the ODC database.

To index a metadata document locally run,

```
datacube dataset add <metadata_doc.yaml file path>
```

To validate the product definition document and the metadata document, eo-datasets provide a validate command.

```
eo3-validate my-product.odc-product.yaml /tmp/path/to/
dataset.odc-metadata.yaml
```

Since our metadata documents are stored in s3, the following procedure was adopted.

To index a metadata document into ODC from s3, ODC provides some tools.

```
pip install --extra-index-url="https://packages.dea.ga.gov.au" odc_apps_dc_tools
```

Once installed to index metadata from s3 run the following command:

```
s3-to-dc “<metadata document s3 URI>”
```

At the beginning, metadata was indexed manually. But as the number of datasets grew an automation for the indexing process was created. It is similar to the metadata creation pipeline, but instead of processing datasets, an indexing script was built.

Automation: Indexing metadata

This automation too consists of mainly four steps as shown in Figure 14.

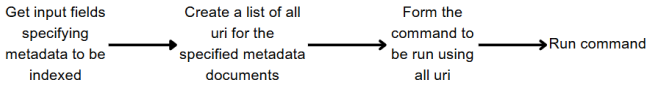

Fig 14: Automation: Indexing metadata [source: IWMI]

Four files are used for this automation,

- inputs.py: file with utility functions
- inputs.txt: file which takes inputs
- main.py: file that is to be run
- processer.py: file forms the command to be executed

Like the metadata creation pipeline, this automation too starts with the population of the index.txt file. The fields it contains serve the same purpose as well.

```
start_date: <start date>
end_date: <end date>
date_list: <set of specific dates. Would overwrite start date and end date if filled>
product_name: <product name>
grided: <True or False: based on if the dataset was divided into chunks for the Limpopo region>
```

It uses the same function to obtain the dates for processing.

The main task is to form the correct command to be run, to index all the datasets for a specific date range and specific product. The URIs can be concatenated to form a command as follows,

```
s3-to-dc “uri_metadata_1” “uri_metadata_2” “uri_metadata_3” “uri_metadata_4”
```

Hence all that the index.py file does is pass every URI that is to be concatenated to form the final command.

The gridded field checks if the datasets to be indexed were gridded into sub regions of the LRB. If true, then the s3 URI catalog structure would start from the region instead of the year. Below is the URI for the "irrigated_areas_limpopo" product date 2019/07/31 region code, x207y027.

```
s3://iwmi-dt-odc-products-public/Natural_Basin_Characteristic/Land_cover_and_Land_use/irrigated_areas_limpopo/x207/y027/2019/07/31_interim/irrigated_areas_limpopo_x207y027_2019-07-31_interim.odc-metadata.yaml
```

There are two functions again to index the datasets, one for datasets that were grided and another for datasets that were not gridded when creating their metadata.

```
def index_grided(pname):
    nonlocal command
    for date in date_list:
        x = 203
        y = 23
        date = date_extract(date)
        year = date["year"]
        month = date['month']
        day = date['day']
        for i in range(71):
            uri = get_uri(pname, year, month, day, x, y)
            x += 1
            y += 1
            command += uri

def index_ungrided(pname):
    nonlocal command
    for date in date_list:
        date = date_extract(date)
        year = date["year"]
        month = date['month']
        day = date['day']

        uri = get_uri(pname, year, month, day)
        command += uri

the get_uri function mentioned above is as follows,
def get_uri(pn, year, m, d, x=None, y=None):
    if m != 10 and m != 11 and m != 12:
        m =  f'0{m}'
    if x is None:
        uri_suffix = f"{year}/{m}/{d}_interim/{pn}_{year}-{m}-{d}_interim.odc-metadata.yaml"
    elif x is not None:
        uri_suffix = f"x{x}/y0{y}/{year}/{m}/{d}_interim/{pn}_x{x}y0{y}_{year}-{m}-{d}_interim.odc-metadata.yaml"
    if pn == 'irrigated_areas_limpopo':
        uri = f"'s3://iwmi-dt-odc-products-public/Natural_Basin_Characteristic/Land_cover_and_Land_use/irrigated_areas_limpopo/{uri_suffix}" '

    if pn == 'vegetation_condition_index_limpopo':
        uri = f"'s3://iwmi-dt-odc-products-public/Natural_Basin_
```

```
Characteristic/Land_cover_and_Land_use/vegetation_condition_index_limpopo/{uri_suffix}" '
    if pn == 'incremental_channel_contributions_limpopo' or pn == 'eflow_warnings_limpopo':
        uri = f"s3://iwmi-dt-odc-products-public/Ecosystems/Riverine/{pn}/{uri_suffix}' "
    return uri
```

'm' here stands for month. This function forms the URI based on the product name, date and grided status.

Example scenarios: Indexing metadata

1) Metadata has been created for the product vegetation condition index from the dates 2022-06-30 to 2024-06-30. Datasets were not gridded. The inputs.txt file should be filled as follows

```
start_date: 2022-06-30
end_date: 2024-06-30
date_list:
product_name: vegetation_condition_index_limpopo
gridded: False
```

This will create the appropriate URI for each metadata document for the specified date range. As mentioned, since grided is set to false, one such URI will have the below structure due to the s3 catalog, which is excluding the region code, starting from 'year'

```
s3://iwmi-dt-odc-products-public/Natural_Basin_Characteristic/Land_cover_and_Land_use/ vegetation_condition_index_limpopo/2022/06/30_interim/vegetation_condition_index_limpopo_2022-06-30_interim.odc-metadata.yaml
```

This will output the command and wait for verification. Once the command is verified by the user it will run, indexing all the metadata documents specified.

```
input = input('\nCorrect command?, enter Yes or No: ')

if input == 'Yes':
    print('Running command')
elif input =='No':
    print('Not running command')
else:
    print("did not run command")
```

2) Metadata has been created for the product irrigated areas for the dates 2022-06-30, 2022-07-31 and 2022-8-31. Datasets were grided. The inputs.txt file should be filled as follows

start_date: 2022-06-30

```
end_date: 2024-06-30
date_list: 2022-06-30, 2022-7-31, 2022-8-31
product_name: irrigated_areas_limpopo
grided: True
```

it is a similar process to the example before, except that the grided datasets would have a region code in their Uri

These are the two automations in place for creating and indexing metadata into ODC.

Note: Once new datasets are indexed into ODC, before running the datacube explorer application to view the product datasets, run the below command to refresh the explorer schema.

```
cubedash-gen --init —all
```

Datacube Explorer

The only implementation steps taken to run a custom version of datacube-explorer were changing the theme and the s3-region config variables by creating a settings.env.py file inside the already cloned datacube-explorer repo (see datacube-explorer installation section). A custom theme can be created inside the cubedash/themes path in the cloned datacube-explorer folder. Configuration Handling — datacube-explorer Unknown/Not Installed documentation.

Datacube OWS

To serve the data indexed in ODC 'Datacube-OWS' is used. It provides a way to serve data indexed in an Open Data Cube as visualizations, through open web services (OGC WMS, WMTS and WCS).

The configuration files, which govern OWS, can be found inside the datacube_ows directory mentioned in the installation section of this chapter.

To organize the configs for each product, the same product catalog structure was adopted. Figure 15 shows the folder structure of the product configs.

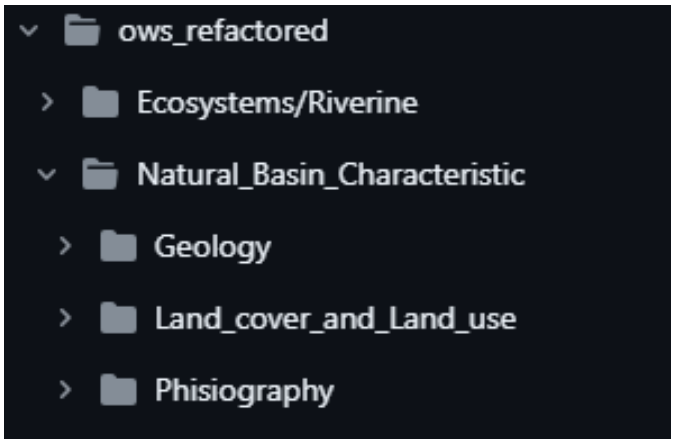

Fig 15: Datacube OWS folder structure [source: IWMI]

Inside each of these folders lies the configuration for each product.

OWS Root config

The root config for OWS is at the root of the ows_refactored directory. This is the main config file for OWS. It holds the

paths to all the other product configs inside the 'layer' section. The crux of the root config is as follows:

```
ows_cfg = {
  "global": { "response_headers": {
       "Access-Control-Allow-Origin": "*",
   },
   "services": {
     "wms": True,
     "wmts":True,
     "wcs": True
   },
   "title": "IWMI - OGC Web Services",
   "abstract": "OGC Web Services-root",

   "allowed_urls": [
        "http://13.247.134.30:8080/",
        "https://odc.digitaltwins.demos-only.iwmi.org/"
        ],
   "info_url": "http://opendatacube.org",

   "published_CRSs": { },
  },
  "wms": {},
  "wmts": {},
  "wcs": {},
  "layers": [{
           "include":   "datacube_ows.dt_odc.ows_refactored.
Natural_Basin_Characteristic.Land_cover_and_Land_use.
ows_irrigated_areas_limpopo_cfg.layer",
   "type": "python",
   },]
}
```

The global section contains headers, titles, and the published URLs. As mentioned, the layer section shown above holds the path to the layer object config for a particular product. As an example, one such layer path is shown. This layer path follows the catalog structure of the respective product.

OWS product layer config

Below is the structure of the layer config file for each product

```
layer = {
   "title": "irrigated_areas_limpopo",
   "abstract": "irrigated areas extent map",
   "name": "irrigated_areas",
   "product_name": "irrigated_areas_limpopo",
   "bands": {"map": [], "prob": [], "filtered": []},
   "native_crs": "EPSG:4326",
   "native_resolution": [10, -10],
   "resource_limits": reslim_continental,
   "image_processing": {
        "extent_mask_func": "datacube_ows.ogc_utils.mask_
by_val",
       "fuse_func": "datacube_ows.wms_utils.wofls_fuser",
        },
   "styling": {
           "styles": [
           style_prob,
           style_map,
           style_filtered
           ]
       }
}
```

One key factor to note down is that the product_name field must map to a product already indexed in the ODC database or else the layer would not be created. If the native_crs and resolution differ from the ones mentioned in the product definition, it will be overwritten by those values.

The next crucial step is the styling section. As shown above, the "irrigated areas" product has three styles, one for each of its measurements. The style structure of the measurement 'map' for this product is shown below.

```
style_map = {
   "name": "map",
   "title": "map",
   "abstract": "Probability of cropping",
   "needed_bands": ["map"],
   "index_function": {
      "function": "datacube_ows.band_utils.single_band",
      "mapped_bands": True,
      "kwargs": {
         "band": "map",
      },
   },
   "color_ramp": [
      {"value":1, "color": "#00ff00"}
    ],
   "range": [0, 100],
   "legend": {
      "begin": "0",
      "end": "1",
   },
}
```

Vital in the style section is the band mentioned. It should map to a band of the product mentioned in its product definition document. If there is a mismatch, OWS would fail to load the products layer prompting errors related to an unknown band measurement.

The color ramp takes two parameters as shown above, 'value' and 'color'. This specifies the raster value of the mentioned band and its corresponding color. Regarding the above style, the 'map' measurement for the irrigated areas product should show the color 'green' for the value '1'.

Note: when the developer receives data to be indexed for a new product, along with the details for the product definition document and the metadata document he/she would also

require the color ramps for the specific products measurements, to create its layer config.

If all product layer configs are prepared correctly as mentioned, OWS can load each of the product layers making them available for visualization using WMS protocols

Deployment

As mentioned in the methodology, two Flask applications had to be deployed (Explorer and OWS). They were run on the same machine on different terminals using the screen command.

Datacube explorer

```
cd datcube-explorer
screen
conda activate cubenenv
gunicorn --bind 0.0.0.0:8000 -w 4 cubedash:create_app\(\)
```

Datacube ows

```
cd miniforge3/envs/cubeenv/lib/python3.12/site-packages/datacube_ows
export DATACUBE_OWS_CFG = datacube_ows.dt_odc.ows_refactored.ows_root_cfg.ows_cfg
screen
conda activate cubeenv
gunicorn --bind 0.0.0.0:8080 wsgi:app
```

The explorer application runs on port 8000, while OWS runs on port 8080. Caddy web server was chosen to reverse proxy the two applications to their relevant domains, due to its simplicity and ease of use in setting up. The current Caddy configuration file is as follows:

```
odc.digitaltwins.demos-only.iwmi.org {
  reverse_proxy 127.0.0.1:8080
  }
odc-explorer.digitaltwins.demos-only.iwmi.org {
  reverse_proxy 127.0.0.1:8000
  }
```

RESULTS

Once ODC and its components were set up and deployed the following services were made available.

ODC database

The ODC database contains seven products and their indexed metadata documents. Table 2 contains the current products and their indexed metadata timestamps in the ODC database.

Table 2: Current status of the ODC database.

Product	Number of metadata documents
vegetation_condition_index_limpopo	8 datasets. 2022 – March, April, May 2023 – March, April, May 2024 – March, May Time period- Monthly
eflow_warnings_limpopo	288 datasets. 2001 January – 2024 December Time period- Monthly
incremental_channel_contributions_limpopo	287 datasets. 2001 January – 2024 November Time period- Monthly
irrigated_areas_limpopo	1704 datasets. (gridded 71 x 24) 2019 – June, July, August, September 2020 – June, July, August, September 2021 – June, July, August, September 2022 – June, July, August, September 2023 – June, July, August, September 2024 – June, July, August, September Time period- Monthly
land_use_limpopo	1 dataset 2024 - December
soil_map_limpopo	1 dataset 2024 - December
dem_limpopo	1 dataset 2024 - December

Datacube Explorer

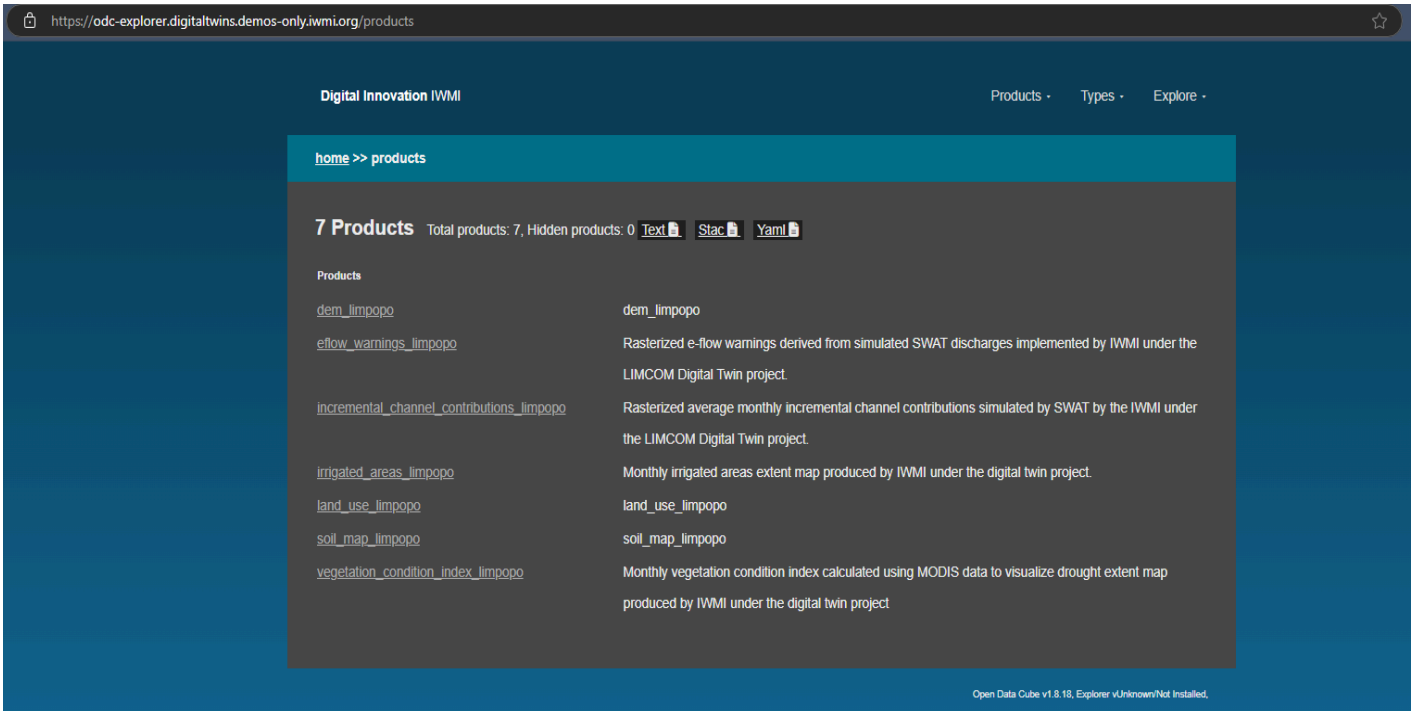

Fig 16: Datacube-explore products page ODC Products. [source: IWMI]

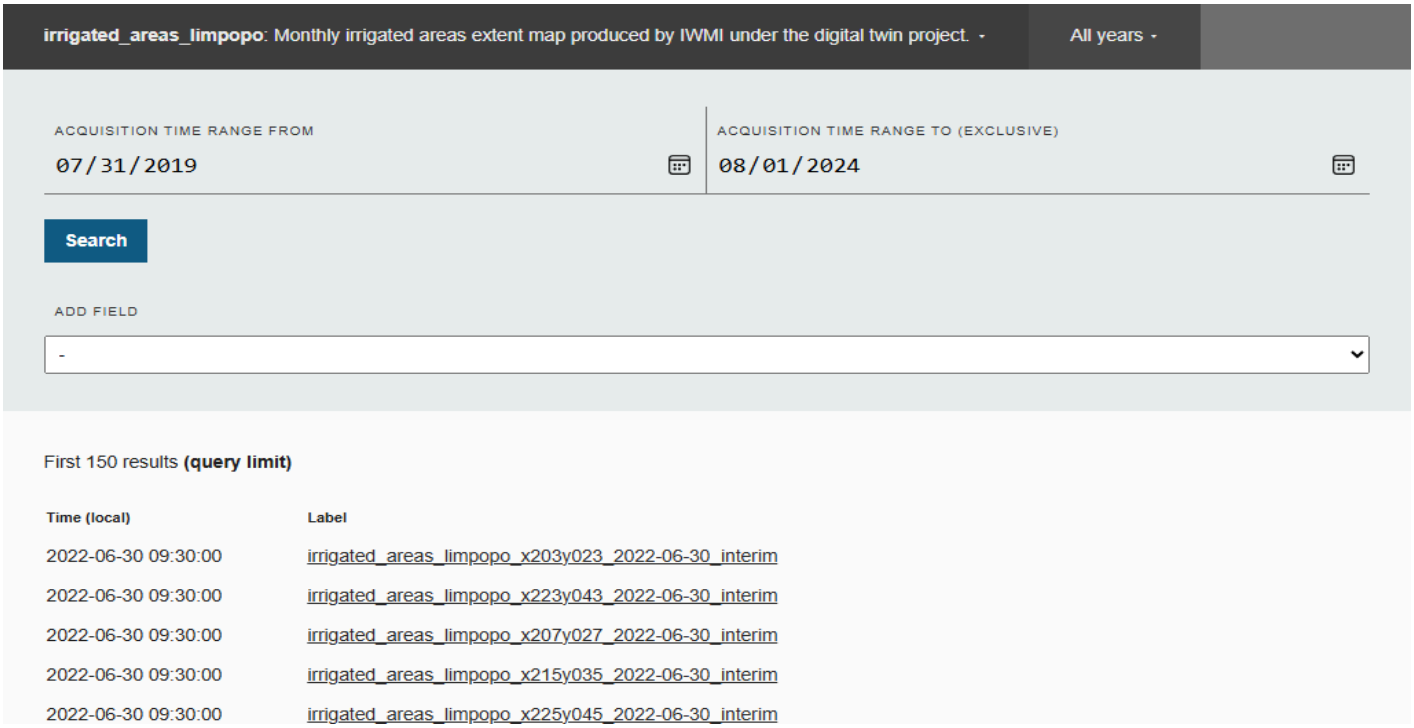

Fig 17: View all the metadata indexed for the product. [source: IWMI]

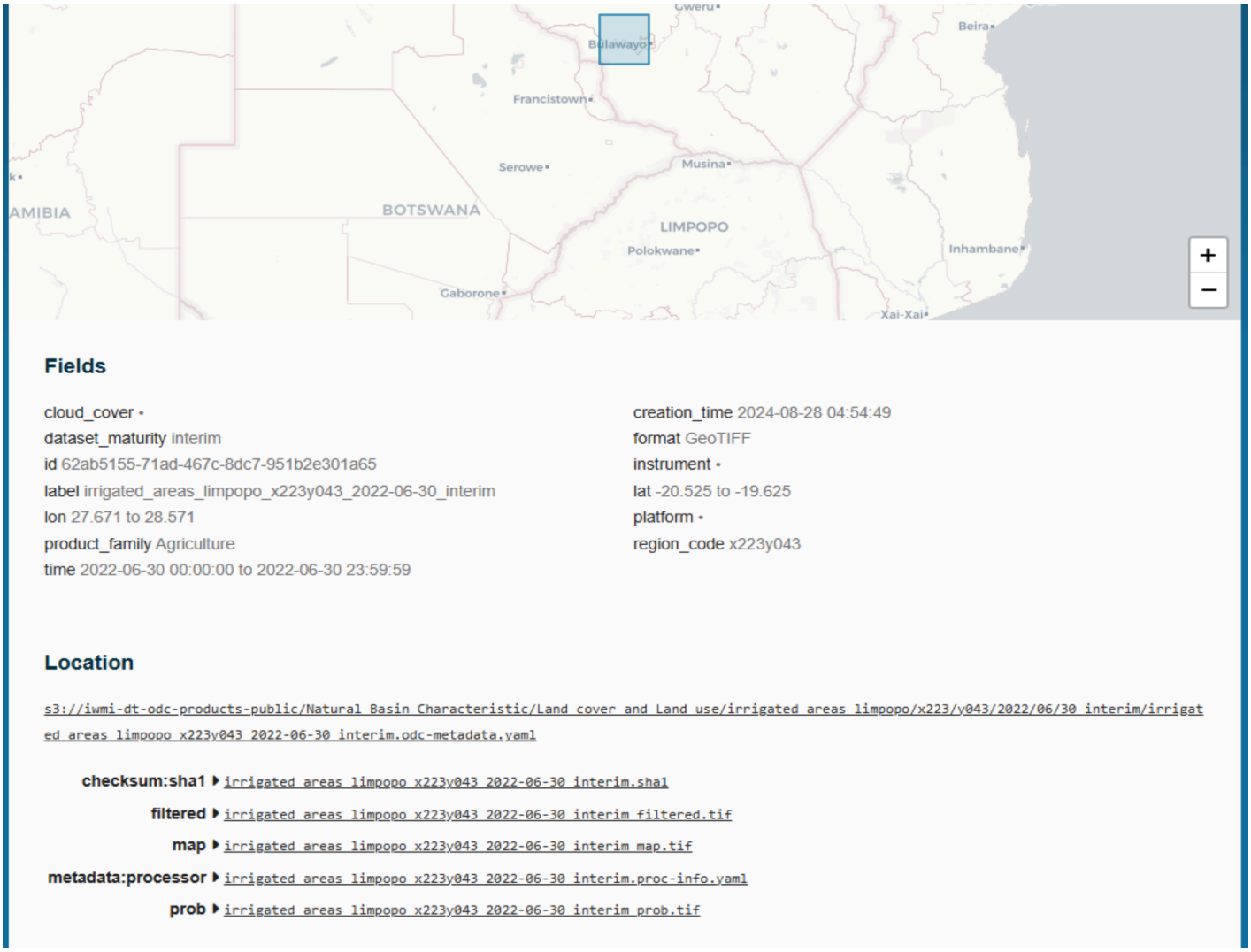

Fig 18: Download the dataset included in the respective metadata document. [source: IWMI]

STAC endpoint to the dataset.

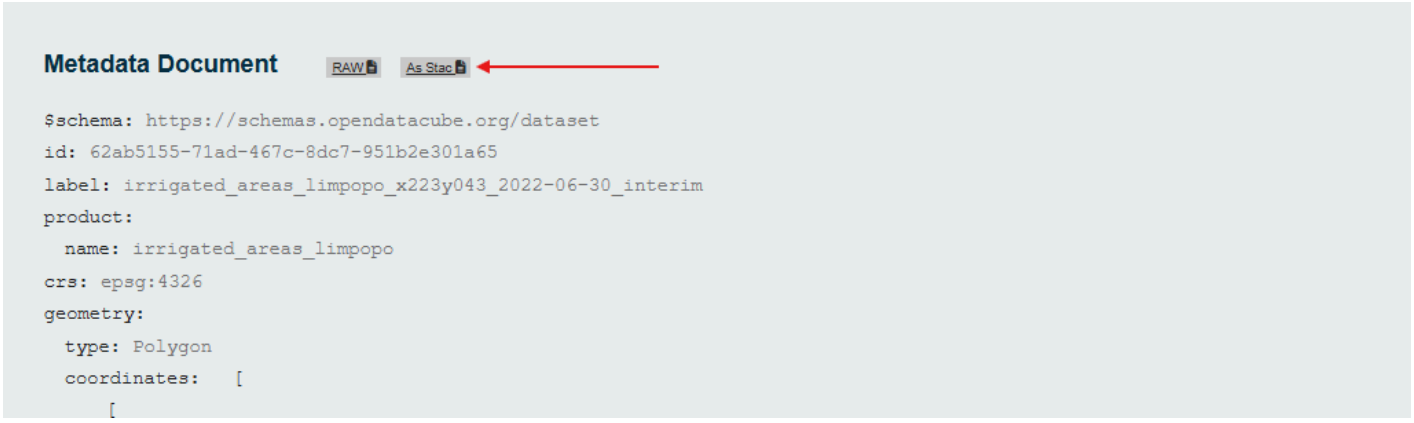

Fig 19: STAC endpoint for a dataset. [source: IWMI]

The STAC endpoint to the datasets stored in our public bucket is, odc-explorer.digitaltwins.demos-only.iwmi.org/stac/

STAC data extraction

Explore the collections (products), and work with datasets indexed inside our ODC.

```
from pystac_client import Client
from odc.stac import configure_rio, stac_load
import xarray as xr
import matplotlib.pyplot as plt

catalog = Client.open("https://odc-explorer.digitaltwins.demos-only.iwmi.org/stac/")
query = catalog.search(
    collections = ["vegetation_condition_index_limpopo"],
)
items = list(query.items())
print(f"Found: {len(items):d} datasets")
```

```
Found: 8 datasets
```

```
import os
os.environ["AWS_NO_SIGN_REQUEST"] = "YES"
os.environ["AWS_REGION"] = "af-south-1"
ds = stac_load(
    items,
    #bands=("VCI"),
)
ds
```

xarray.Dataset

Dimensions: (latitude: 3009, longitude: 4141, time: 8)

Coordinates:

latitude	(latitude)	float64	-19.77 -19.77 ... -26.52 -26.52
longitude	(longitude)	float64	25.14 25.14 25.14 ... 34.43 34.44
spatial_ref	()	int32	4326
time	(time)	datetime64[ns]	2022-03-31 ... 2024-05-31

Data variables:

VCI	(time, latitude, longitude)	float32	nan nan nan nan ... nan nan nan nan

Indexes: (3)

Attributes: (0)

```
ds['VCI'].plot(col="time", col_wrap=6, vmin=0, vmax=100)
```

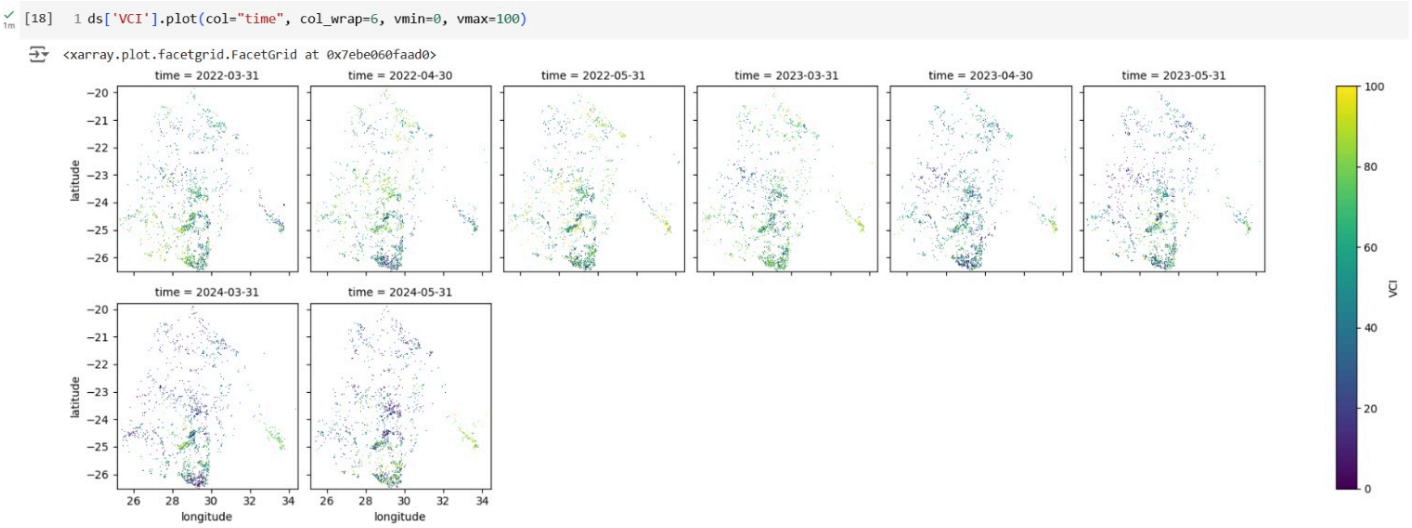

Fig 20: Vegetation Condition Index plots for the extracted dates. [source: IWMI]

```
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import numpy as np

cmap = mcolors.ListedColormap(['#fe3c19', '#ffac18', '#f2fe2a', '#7cb815', '#285605'])
bounds = [0, 20, 40, 60, 80, 100]
norm = mcolors.BoundaryNorm(bounds, cmap.N)

ds['VCI'].isel(time=0).plot(cmap=cmap, norm=norm)
```

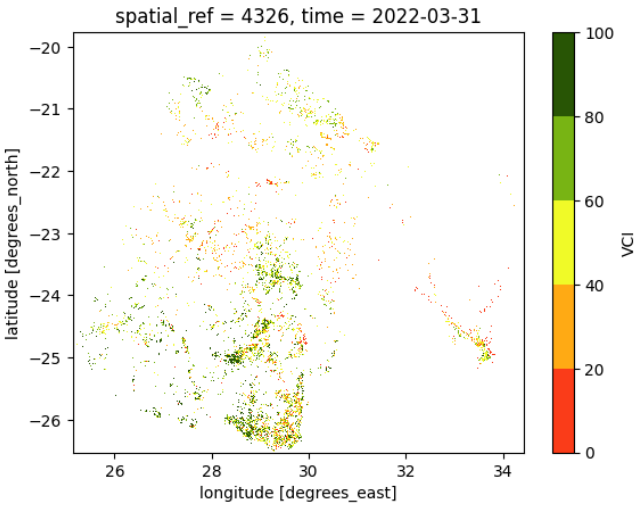

Fig 21: Vegetation index plot – 2021-March. [source: IWMI]

Datacube OWS

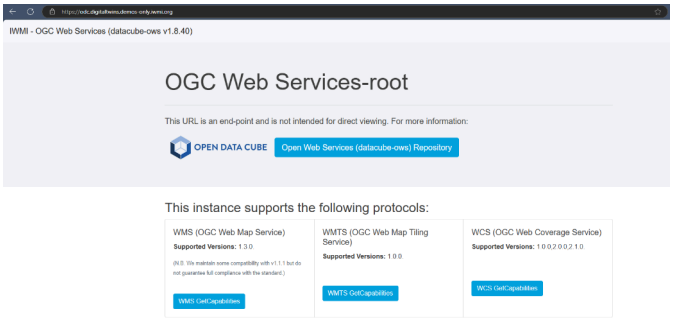

Fig 22: Datacube OWS IWMI - OGC Web Services (datacube-ows). [source: IWMI]

NOTE

This system can be replicated using the installation guide to the ODC stack mentioned in the implementation chapter and the git hub repo which holds the source code. iwmihq/dt-odc . The automations that include accessing data from the iwmi-private s3 bucket would require certain AWS access and secret keys in order to function.

ACKNOWLEDGEMENTS:

The authors would also like to thank Dr. Lisa Rebelo and the Digital Earth Africa (DEA) team for their timely support in resolving challenges related to setting up the Open Data Cube (ODC) for the Digital Twin project.

The development of the ODC was made possible through the generous support of The Leona M. and Harry B. Helmsley Charitable Trust as part of the DIWASA project, the CGIAR Initiative on Digital Innovation, and AWS, which provided technical support and guidance during the project's implementation phase.

We thank all funders who supported this research through their contributions to the CGIAR Trust Fund.

REFERENCES

Papers and Articles:

Leith, A.; 2018, What is the Open Data Cube? https://medium.com/opendatacube/what-is-open-data-cube-805af60820d7

Garcia Andarcia, M.; Dickens, C; Silva, P; Matheswaran, K; Koo, J. 2024. Digital Twin for management of water resources in the Limpopo River Basin: a concept. Colombo, Sri Lanka: International Water Management Institute (IWMI). CGIAR Initiative on Digital Innovation. 4p. https://hdl.handle.net/10568/151898.

Ghosh, S.; Vigneswaran, K.; Dickens, C.; Retief, H.; & Garcia Andarcia, M.; 2024. A description of recent drought prevalence in the Limpopo River Basin. Colombo, Sri Lanka: International Water Management Institute (IWMI). CGIAR Initiative on Digital Innovation. 13p

Kiala, Z.; Karthikeyan, M. 2024. Status of irrigated area in the Limpopo River Basin. https://hdl.handle.net/10568/155296

Chambel-Leitão, P.; Santos, F.; Barreiros, D.; Santos, H.; Silva, P.; Madushanka, T.; Matheswaran, K.; Muthuwatta, L.; Vickneswaran, K.; Retief, H.; Dickens, C.; Garcia Andarcia, M. (2024). Operational SWAT+ model: advancing seasonal forecasting in the Limpopo River Basin. Colombo, Sri Lanka: International Water Management Institute (IWMI). CGIAR Initiative on Digital Innovation. 97p. https://hdl.handle.net/10568/155533

Software:

AWS boto3 documentation https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

AWS CLI configuration https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

Datacube explorer documentation https://datacube-explorer.readthedocs.io/en/latest/

EO-datasets 3 documentation https://eodatasets.readthedocs.io/en/latest/

Open data cube documentation, https://opendatacube.readthedocs.io/en/latest/

OWS documentation https://datacube-ows.readthedocs.io/en/latest/

Terria frontend can be accessed at the following: https://terria.io/