COMPUTER RESEARCH
INSTITUTE
OF MONTREAL

FINAL REPORT

Project CCCOT03 – Technical Report

Samuel Foucher
Senior Researcher
Francis Charette-Migneault
Research Software Developer
David Landry
Data Scientist
December 14, 2020

Key financial partner:
Québec

101 – 405 Ogilvy Avenue, Montreal, Quebec H3N 1M3
info@crim.ca | 514 840-1234 | 1 877 840-2746 | F 514 840-1244

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Context

As the organization that plays the role of the national mapping agency, the Canada Centre for Mapping and Earth Observation (CCMEO) is, among other things, responsible for creating, managing, publishing and updating a vast array of geospatial data covering the Canadian continental mass. Through the GeoBase division, the CCMEO is responsible for updating fundamental geospatial data, such as the national river system. That data are of primordial importance to the country because they are used as the basis for creating a plethora of value-added geospatial data (i.e. the reference point for the country's statistical data is derived from fundamental data). To be relevant, that data must be constantly updated. The technological development of recent years has resulted in new approaches, such as deep learning (DL), which helps identify short-term solutions for the rapid and efficient update of geospatial data. Approximately three years ago, the GeoBase division formed a research and development team to develop tools based on artificial intelligence (AI) to extract mapping objects using remote sensing data (optical imaging, radar and elevation data).

The CCMEO is currently working on implementing an inference service to extract mapping elements using imaging and support several DL models. One of the components of that service is researching a trained model. To be able to use a model on new images, we need to be able to determine the context in which a model was trained (type of images and their pre-processing, spatial resolution, classes extracted, etc.). Hence, this component impacts two aspects, namely 1) the metadata of the model (describing its training context) and 2) the manner in which that information is stored and requested.

There is immense interest in AI and many efforts are being made to increase reuse of DL models and to facilitate their deployment in production. For the time being many solutions are offered and they are very specific to the area of application of the models. That is why the CCMEO wants to determine the metadata required to facilitate reuse of the models in its operational context. Moreover, given the emerging technologies to request that information, it is relevant to have a detailed picture of them to guide the CCMEO's technological choice.

1.2 Use case

- **Case 1:** As a technician, I want to search for models that are capable of extracting features (such as semantic segments) of satellite images:
    - For a specific class (type) of object (for example, buildings)
    - For multiple types of objects, for example, buildings, water, vegetation, etc.
- **Case 2:** As a technician or researcher, I want to be able to compare models on images (given images with corresponding labels):
    - On images provided by the user (the user may be a citizen, in this case)
    - On other groups of reference data
- **Case 3:** As a generic user (may be a technician, a scientific researcher with NRCan, etc.), I would like to find the best models for my images (sensors, time intervals), for feature extraction:
    - As a technician, I would like to find the highest performing model to extract a specific class of objects
    - I want to know what the model needs in terms of data and obtain general information on the model (for example, configuration options, limits) to be able to find the data myself and use the model later

Scenario 1 involves only viewing metadata on model inputs and outputs

Scenario 2 involves the possibility of instantiating a ML pipeline and testing it on new data.

Scenario 3 can initially be fulfilled solely using metadata on model inputs and outputs as well as its performance on a known testbed.

1.3 Working theories

1. The only task considered here is semantic segmentation.
2. Only feedforward architectures are covered (not recurrent or generational networks).
3. The data are very high resolution multispectral satellite imaging (for example, WorldView).
4. The deep learning framework considered here is PyTorch, but the discussion should be agnostic to the software framework, if possible.

5. The proposed solutions are considered sufficient for reproducing the inference of a pretrained model. The extension of those solutions to permit reproducibility of training is deferred to future work.

1.4 Definition of an inference pipeline in machine learning

Given a trained model, inference consists of applying that model to new data consistent with the model's specifications to produce a new prediction.

An inference pipeline contains at least the following components:

1. A `Dataset` object responsible for loading and pre-processing the data:
   a. A driver compatible with the format and organization of the original data
   b. A basic transformation for data pre-processing
2. An instantiation model using the following information:
   a. Implementation of the model architecture
   b. Weight resulting from the training
3. A post-processing function responsible for formatting the model's output:
   a. In the case of semantic segmentation, this function may be interpolation toward a fixed size, like the input layer.

1.5 Reproducibility and replicability in machine learning

Ideally, instantiation of a machine learning pipeline should at least focus on the reproducibility of the results on test data similar to that of the work at the origin of the model. According to the American Statistical Association (ASA):

Reproducibility: A study is reproducible if you can take the original data and computer code used to analyze the data and reproduce all of the study's numerical results. That may seem like a trivial task at first glance, but experience has shown that it is not always easy to achieve that apparently minimal standard.

Replicability: Is the act of reproducing an entire study, independently of the original investigator, without using the original data (but generally using the same methods).

References:

1. https://www.natur.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970
2. https://www.amstat.or/asa/files/pdfs/POL-ReproducibleResearchRecommendations.pdf
3. Reproducibility challenge: https://paperswithcode.com/rc2020
4. The Machine Learning Reproducibility Checklist (Version 1.2, Mar. 27, 2019): https://www.cs.mcgill.ca/~jpineau/ReproducibilityChecklist-v1.2.pdf

5. Brinckman, A., Chard, K., Gaffney, N., et al.: Computing environments for reproducibility: Capturing the "Whole Tale". Future Generation Comp. Syst. 94, 854-867 (2019) DOI: 10.1016/j.future.2017.12.029
6. Chapp, Dylan, Victoria Stodden, and Michela Taufer. "Building a Vision for Reproducibility in the Cyberinfrastructure Ecosystem: Leveraging Community Efforts." Supercomputing Frontiers and Innovations 7.1 (2020): 112-129

1.6 Defining requirements for a catalogue of models

One of the objectives of this study is to propose a metadata organization related to the models resulting from deep learning to 1) be able to conduct relevant research on those model catalogues and 2) be able to specify a test instance. By considering only the area of computer vision for which the data features considered are consistent (3-band RGB, 8 bit), there are still major issues regarding the reproducibility and replicability of the research work done. In the area of Earth observation (EO), there is a multitude of observation modes (radar, optical, hyperspectral, etc.) and associated acquisition parameters. We think there are four essential requirements to be met for a model catalogue:

R1: sufficient metadata to obtain relevant research results

R2: definition of input data consistent with the model and output data resulting from the model

R3: sufficient metadata to achieve good replication of the entire pipeline

R4: ability to validate proper operation of a model from the catalogue

2 METADATA SCHEMA PROPOSAL

The schema should cover at least the following elements:

1. Description of data consistent with the model:
    a. Sensor
    b. Product level (e.g.: L1, L2, etc.)
    c. Type of measurement (Digital count, reflectance values, etc.)
    d. Description of the whole training :
        i. Link to the training kit
2. Description of the transformation between the data and the network's input layer:
    a. Band selection
    b. Size of input layers
    c. Statistical normalization (mean and standard deviation)
3. Description of the model architecture:
    a. Number of parameters, layer description, etc.
4. Description of the runtime environment:
    a. Dependencies (packages) in the form of a requirements folder

   b. Runtime environment
   c. Dockerfile
5. Description of the model output:
   a. Semantic mapping (synset)
6. Model documentation:
   a. Relevant publications in the form of DOI links
   b. GitHub links
   c. Performance on public testkits
7. Test samples to validate the model's instantiation:
   a. An input tensor (NxCixHixWi)
   b. A target output tensor of the model (NxHoxWo)

The figure below provides a conceptual view of an inference pipeline with the information provided by the metadata considered and the interaction with the various external resources (catalogues, source code, etc.).

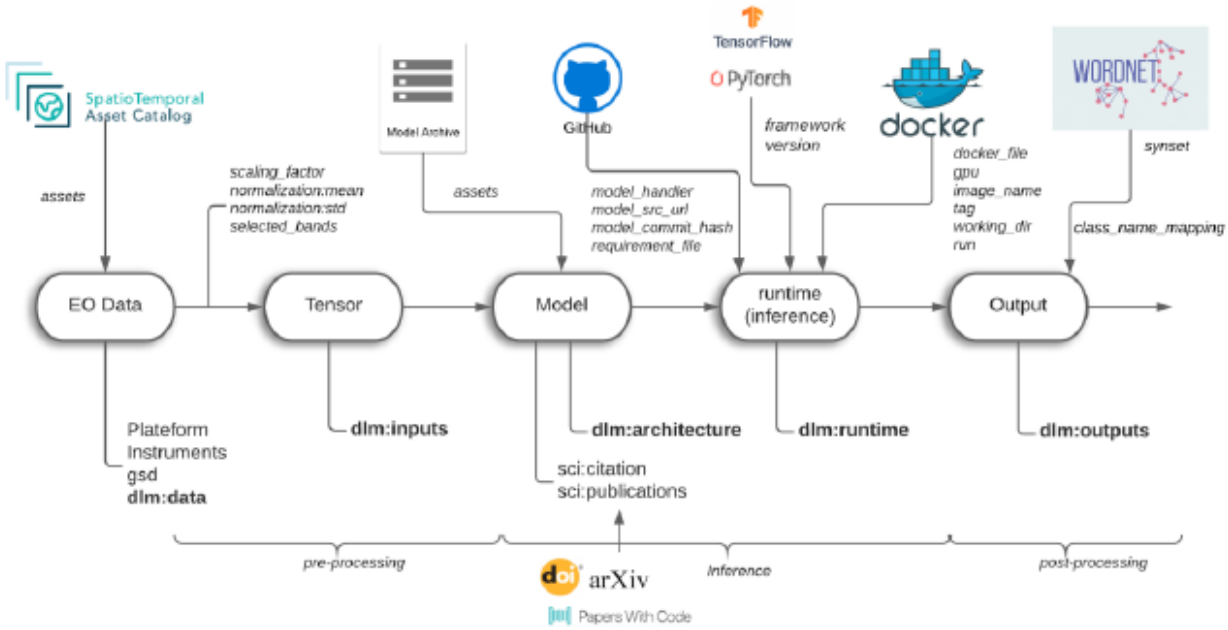


Figure 1: Conceptual view of an inference pipeline with a view of the metadata and resources.

2.1 Existing metadata schemas

A portion of the proposed nomenclature will be based on existing metadata schemas:

- OGC® Earth Observation Metadata profile of Observations & Measurements:
  - https://docs.opengeospatial.org/is/10-157r4/10-157r4.html
  - #definitions/ProcessingInformation/properties/processingLevel
- STAC items:
  - https://github.com/radiantearth/stac-spec/tree/master/item-spec

- STAC extensions:
  - https://github.com/radiantearth/stac-spec/tree/master/extensions

2.2 References and tools used for the JSON schemas

- Understanding JSON Schema: https://json-schema.org/understanding-json-schema/
- JSON to Schema transformers:
  - https://www.altova.com/xmlspy-xml-editor/json_schema_editor

In the following, to simplify the text, we apply the following notation to describe a JSON object:

JSON	NOTATION

'test_file': {
'filename' : 'test_file.tif'
'bands' : [0, 1, 2]
}
	test_file:
	filename: [str]
	bands: [int*n]

Notations and approach taken:

- The examples and the preliminary JSON schemas are available at GitHub: https://github.com/crim-ca/CCCOT03
- For the proposed schema, we make reference to the version tagged 2020.12.11[1].
- For the URIs, when we point to existing schemas, the URI is preceded by *stac*: for item-spec schemas or one of the extensions
- The new metadata proposed are sometimes preceded by the *dlm* prefix: (for deep-learning-model)
- When a STAC field of an extension is mentioned, it should be included in the final item level (there is no inclusion between extensions, i.e. other key extensions cannot be specified in the dlm space)
- The STAC style is followed for key format: lowercase text, use of _ for complex names
- In the following tables, when the key mentioned is already in STAC, it has a grey background and with a yellow background for a key in an extension

[1] https://github.com/crim-ca/CCCOT03/blob/2020.12.11/extension/dl-model/json-schema/schema.json

2.3 General organization

The figure below provides an overview of the JSON objects that characterize the models. There are six types of objects:

1. **dlm:architecture** contains general information on the architecture
2. **dlm:inputs** characterizes the expected input tensor and information on the transformations to be applied
3. **dlm:outputs** contains information on the model's output layer and its semantic interpretation
4. **dlm:runtime** provides minimal information to replicate the runtime environment in the form of a docker image and to start the run
5. **dlm:archive** describes the contents of the model's archive (.mar)

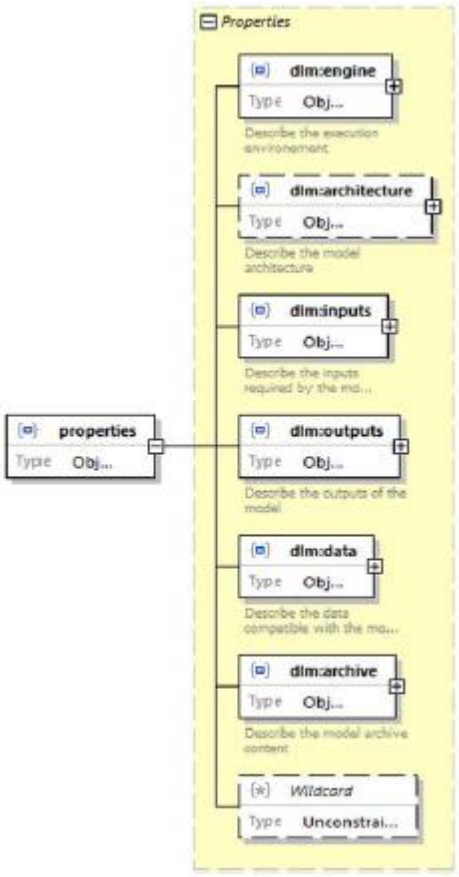


Figure 2: General view of proposed JSON objects

2.4 Model properties

2.4.1 General properties

The general properties cover information such as authors, license, scientific references, etc.

TERM	DEFINITION	REMARKS	URI
title: [str]	Model's short name		stac:basics.json#properties/name
description: [str]	Multiline description of the model		stac:basics.json#properties/descriptions
license: [str]	Information on the model's license	SPDX License identifier	stac:licensing.json#
providers: name: [str] roles: [enum] url: [uri]	An enum helps standardize roles, for example: "enum": ["producer", "licensor", "processor", "host"]	Probably optional, to be discussed, useful if several actors	stac:provider.json#
created	Item creation date	Defined in stc-item	stac:datetime.json#
sci:citation: [str]	Official reference for citing the implementation of the model here	Defined in STAC's scientific extension	scientific.json#properties/sci:citation
sci:publicatioins:citation: [str] doi:	List of applicable publications and references to describe the source of the model	Defined in STAC's scientific extension	scientific.json#properties/sci:citation
dlm:publisher: name: [str] entity: [str] email: [str]	Information on the organization that publishes the model	The object may contain the following fields: name, organization, name, email	schema.json#/dl-model/properties/publisher
dlm:version	Version of the model		dlm:schema.json#/dl-model/properties/dlm:version

2.4.2 Description of the transformation between the data and the network's input layer

This information describe the pre-processing between the raw data and the network's input layer. Additional pre-processing upstream may be specified using a Python function such as `pre_processing_function.py`. At the least, normalization of the dynamic and normalization of statistics should be specified (the equivalent of the NormalizeMinMax and NormalizeZeroMeanUnitVar functions).

This information will be stored in a JSON object under #/properties/dlm:inputs.

TERM	DEFINITION	REMARKS	URI
name [str]	Input variable name		schema.json#/dl-model/properties/inputs/name
input_tensors: [array * [batch: [int] dim: [int] height: [int] width: [int]]]	Size of the network's input tensors, for example 1x3x224x224, the input may be multi-tensor (tensor vector)		schema.json#/dl-model/properties/inputs/input_tensors
scaling_factor [number]	Scaling factor to be applied before input	For example, if the data are uint8 then scaling_factor=1/255	schema.json#/dl-model/properties/inputs/scaling_factor
normalization:mean: [array*number] normalization:std: [array*number]	Specifies the tensor's statistical normalization		schema.json#/dl-model/properties/inputs/normalization
selected_bands: [array*int]	Specifies the selected bands using the data described in dlm:data		schema.json#/dl-model/properties/inputs/selected_bands
pre_processing_function	Defines a complete pre-processing function	Helps define more complex transformations between the EO data and the input tensor	schema.json#/dl-model/properties/inputs/pre-_processing_function

2.4.3 Description of the model's architecture

Provides information on the architecture; at least the number of parameters should be specified. The pytorchsummary package helps extract a summary of the layers that might be useful to add in the summary field (note that this input could become large for complex networks). In PyTorch, once the model is created, it is possible to do a simple `print(model)`. This is an example of a partial output for a UNet (note that this output can be rather long). In addition, it may also be problematic for the formatting of this content to be compatible with the JSON format (for example, the presence of special characters).

```
ExternalModule(
  (model): EncoderDecoderNet(
    (encoder1): Sequential(
      (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    )
    (encoder2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
...
    (decoder1): DecoderUnetSCSE(
      (block): Sequential(
        (0): Conv2d(96, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): _ActivatedBatchNorm(
          (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (act): ReLU(inplace=True)
        )
        (2): SCSEBlock(
          (avg_pool): AdaptiveAvgPool2d(output_size=1)
          (channel_excitation): Sequential(
            (0): Linear(in_features=64, out_features=4, bias=True)
            (1): ReLU(inplace=True)
            (2): Linear(in_features=4, out_features=64, bias=True)
          )
          (spatial_se): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1), bias=False)
        )
        (3): ConvTranspose2d(64, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      )
    )
    (logits): Sequential(
      (0): Conv2d(496, 64, kernel_size=(1, 1), stride=(1, 1))
      (1): _ActivatedBatchNorm(
        (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (act): ReLU(inplace=True)
      )
      (2): Conv2d(64, 5, kernel_size=(1, 1), stride=(1, 1))
    )
  )
)
```

This is an example of output with torchsummary with the same model:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 32, 32]           9,408
       BatchNorm2d-2          [-1, 64, 32, 32]             128
              ReLU-3          [-1, 64, 32, 32]               0
         MaxPool2d-4          [-1, 64, 16, 16]               0
 ...
       SCSEBlock-137          [-1, 64, 32, 32]               0
 ConvTranspose2d-138          [-1, 16, 64, 64]          16,400
 DecoderUnetSCSE-139          [-1, 16, 64, 64]               0
          Conv2d-140          [-1, 64, 64, 64]          31,808
     BatchNorm2d-141          [-1, 64, 64, 64]             128
             ReLU-142          [-1, 64, 64, 64]               0
_ActivatedBatchNorm-143          [-1, 64, 64, 64]               0
          Conv2d-144           [-1, 5, 64, 64]             325
EncoderDecoderNet-145           [-1, 5, 64, 64]               0
================================================================
Total params: 42,813,873
Trainable params: 42,813,873
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 20.35
Params size (MB): 163.32
Estimated Total Size (MB): 183.72
----------------------------------------------------------------
```

This information will be stored in a JSON object under #properties/dlm:architecture.

TERM	DEFINITION	REMARKS	URI
architecture: total_nb_parameters: [int] estimated_total_size_mb: [float] type: [str] summary: [str] pretrained: [str]	Provides information on the architecture: • The total number of parameters • The equivalent size in memory in MB • The type of network (e.g.: ResNet) • A summary of the layers • Source of pretraining (e.g.: ImageNet)	There is certainly more information to add here, such as the number of layers, the backbone used	schema.json#/dl-model/properties/architecture

2.4.4 Description of the model's output

The model's outputs are generally class indices (discrete values from 0 to C-1), therefore there is no associated semantic information. Semantic mapping is therefore necessary, for example ImageNet uses a synset.txt file.

WordNet® is a large database of the English language. Nouns, verbs, adjectives and adverbs are grouped together in groups of cognitive synonyms (called synsets), each expressing a separate concept. The synsets are tied together through semantic and lexical conceptual relationships.

ImageNet is based on WordNet 3.0. To uniquely identify a synset, the "WordNetID" (wnid) is used, which is a concatenation of POS (i.e. a part of speech) and WordNet's SYNSET OFFSET. ImageNet currently only takes into account nouns, therefore each wnid starts with "n". For example, the name of the synset *"lake, a body of (usually fresh) water surrounded by land"* is "n09351810":

http://wordnetweb.princeton.edu/perl/webwn?c=8&sub=Change&o2=&o0=1&o8=1&o1=1&o7=&o5=&o9=&o6=&o3=&o4=&i=-1&h=000&s=lake

TERM	DEFINITION	REMARKS	URI
task: [enum]	Helps characterize the network's output, the task may be an enum {semantic classification, object detection,…}		schema.json#/dl-model/properties/outputs/task
number_of_classes: [int]	Number of classes in output		schema.json#/dl-model/properties/outputs/number_of_classes
final_layer_size: [n*int]	Size of tensor of the network's output	Size of the output image in the case of semantic segmentation	schema.json#/dl-model/properties/outputs/final_layer_size
class_name_mapping: [array: int: str]	Helps map the output index on class names	The equivalent of index_to_name.json used by TorchServe	schema.json#/dl-model/properties/outputs/class_name_mapping
dont_care_index: [int]	Helps specify a "dont care" class used by the network	The "dont care" value impacts mainly training data, but can identify output indices	schema.json#/dl-model/properties/outputs/dont_care_index
post_processing_function: [str]	Certain models need post-processing	Potentially helps specify "custom" transformations	schema.json#/dl-model/properties/outputs/post_processing_function

2.5 Description of satellite data compatible with the model

The description of the satellite images should specify the source of the products (assets) and the necessary and sufficient features compatible with the model. It is not necessary to have all of the metadata related to the acquisition (a reference to them should suffice).

Relevant schemas:

- STAC Common Metadata: Instrument:
  - https://github.com/radiantearth/stac-spec/blob/master/item-spec/common-metadata.md#instrument
- Relevant STAC extensions:
  - eo
  - sat
- OGC EO Dataset Metadata GeoJSON(-LD) Encoding Standard:
  - https://docs.opengeospatial.org/is/17-003r2/17-003r2.html
  - Schemas and examples: http://schemas.opengis.net/eo-geojson/12.0/

It is suggested that new fields be grouped together in a JSON object under #/properties/dlm:data.

TERM	DEFINITION	REMARKS	URI
platform: [str]	Platform or satellite name	Already defined in the instrument	stac:instrument.json#properties/plateform
instruments: [n * str]	Sensor name	Already defined in the instrument	stac:instrument.json#properties/instruments
gsd	Ground sampling distance	Already defined in the instrument – optional unless the model is specific to a particular resolution	stac:instrument.json#properties/gsd
process_level [str]	Level of data processing (L0= raw, L4=ARD). The levels are described by an enum.	Important parameter because can impact the apparent variability of the data	schema.json#dl-model/properties/data/process_level ogc:eo#definitions/ProcessingInformation/properties/processingLevel
dtype: [str]	Type of data storage (uint8, uint16, etc.) Enum based on the types of numpy bases[2]	Potentially important for data normalization and therefore on pre-processing	schema.json#dl-model/properties/data/dtype

[2] https://numpy.org/doc/stable/reference/arrays.scalars.html#arrays-scalars-build-in

TERM	DEFINITION	REMARKS	URI
number_of_bands	Number of bands used		schema.json#dl-model/properties/data/dtype
useful_bands: [array * [int,str]]	Describes the relevant bands for the model	Replicates the eo:bands object, but only indicates the relevant bands	schema.json#dl-model/properties/data/useful_bands
item_examples	Link to relevant catalogue items	Optional	schema.json#dl-model/properties/data/item_examples
test_file: filename: [str] output: [str]	Optional test file that would be available in the archive	Permits a sample of the expected data at time of input and the corresponding output	schema.json#dl-model/properties/data/test_file
catalog	Link to a catalogue of relevant data	Could also be specified in the assets or links	schema.json#dl-model/properties

2.6 Description of the runtime environment

This section describes the minimal runtime environment required. The best approach to guarantee a replicable environment is to specify an image docker via a link on a register (e.g.: dockerhub) or a docker file.

A basic image with installation of the framework (e.g.: PyTorch) and necessary dependencies (e.g.: gdal) is recommended. A target working directory (`working_dir`) will help share the data with the instance. The instance will be able to support gpus (`gpu: True`) or function in cpu mode.

This information will stored in a JSON objet under #properties/dlm:runtime.

TERM	DEFINITION	REMARKS	URI
framework: [str]	Framework used	Perhaps put an enum here: enum {pytorch, tensorflow, mxnet}	schema.json#dl-model/properties/runtime/framework
version: [str]	Version of the framework	Some models require a specific version of the framework	schema.json#dl-model/properties/runtime/version
model_handler: [str]	Test runtime function		schema.json#dl-model/properties/runtime/model_handler
Model_src_url: [str]	Source code URL	E.g.: GitHub repo	schema.json#dl-model/properties/runtime/model_src_url

TERM	DEFINITION	REMARKS	URI
docker: docker_file: [str] image_name: [str] tag: [str] working_dir: [str] run: [str] gpu: [boolean]	Information for deploying the model in an instance docker, specifying a dockerfile (url), an image name, a tag	The information contained here should help instantiate a complete test container	schema.json#dl-model/properties/engine/docker

2.7 Description of the archive content

In the case where an archive of the model is used, we think it would be interesting to have an overview of the content of the model's archive without having to download it. The archive is in the form of an MAR file (.mar), which contains all of the files necessary to instantiate a model (particularly the weights).

This information will be stored in a JSON object under #/properties/dlm:archive.

TERM	DEFINITION	REMARKS	URI
archive name: [str] role: [str]	Archive content	The role of the file could be an enum {pth, dependency, handler, etc.}	schema.json#dl-model/properties/dlm:archive

This is an example of the content for an .mar for Alexnet:

```
"/content/models/alexnet/alexnet-symbol.json",
"/content/models/alexnet/synset.txt",
"/content/models/alexnet/model_handler.py",
"/content/models/alexnet/signature.json",
"/content/models/alexnet/alexnet-0000.params",
"/content/models/alexnet/mxnet_model_service.py",
"/content/models/alexnet/mxnet_vision_service.py",
"/content/models/alexnet/MAR-INF/MANIFEST.json",
"/content/models/alexnet/mxnet_utils/__init__.py",
"/content/models/alexnet/mxnet_utils/nlp.py",
"/content/models/alexnet/mxnet_utils/ndarray.py",
"/content/models/alexnet/mxnet_utils/image.py"
```

The MANIFEST.json file also contains relevant information that can be recovered in the general metadata:


```
"MANIFEST.json": {
            "specificationVersion": "1.0",
            "implementationVersion": "1.0",
            "description": "squeezenet",
            "modelServerVersion": "1.0",
            "license": "Apache 2.0",
            "runtime": "python",
            "engine": {
              "engineName": "MXNet",
              "engineVersion": "^1.2"
            },
            "model": {
              "modelName": "squeezenet_v1.1",
              "description": "squeezenet",
              "modelVersion": "1.0",
              "handler": "mxnet_vision_service:handle"
            },
            "publisher": {
              "author": "MXNet SDK team",
              "email": "noreply@amazon.com"
            }
        }
```


2.8 Model artefacts (assets)

2.8.1 Model archive

The information saved by most frameworks is often incomplete (serialization with Python's pickle module) and does not allow re-instantiating of the model. The recommendation here would be to save a complete artefact in the form of an archive (.mar) containing all of the necessary information (class, weight, handling functions, dependencies, etc.). However, metadata should also be present describing the content of the .mar (see section 2.6).

TERM	DEFINITION	REMARKS	URI
assets: href: [url] type: [str] title: [str] role: [enum]	Points to an archive (.mar) containing the minimal information to instantiate the model in inference mode.	The .mar could contain at least the following files: • A .py file containing the model implementation • A test datum (sample) • The weight • An index_to_name.json file • A signature.json file describing the network's signature • A .yml? environment file	Stac:item.json#/properties/assets

3 COMPARISON OF OPEN SOLUTIONS FOR STORING AND REQUESTING METADATA

3.1 Solutions for the catalogue

3.1.1 Spatio Temporal Asset Catalog (STAC)

STAC allows catalogues to be organized in the form of sets and *items*. The items are in a GeoJSON format containing metadata and "assets" pointing to the data.

STAC Item Specification

The providers should include the metadata fields that are relevant to users (essentially instrument, product, geographic position and dates) in the catalogue, but it is recommended to not select those that are necessary to the research. Satellite imaging can generate hundreds of metadata, so it is recommended that a link be included in Link Object or an Asset.

The STAC specification (Spatio Temporal Asset Catalog) aims to normalize the way in which geospatial assets are set out online and queried. A "spatio-temporal asset" *(asset)* is a file that represents geospatial information, entered on a region of interest and at a certain time. The initial goal is mainly the remote sensing imaging (from satellites, but also from airports, drones, balloons, etc.), but the mode is designed to be able to extend to the SAR, moving video, point clouds, hyperspectral, LiDAR and derivative data, such as NDVI, digital models of elevation, mosaics, etc.

Certain extension proposals are relevant:

- eo extension
- View Geometry Extension Specification:
    - Viewing geometry can be specified:
    - https://github.com/radiantearth/stac-spec/blob/master/extensions/view/README.md
- In terms of annotations, an extension suggestion (Label Extension Specification) for STAC permits the addition of annotations for classification, object detection and segmentation tasks. The https://registry/mlhum.earth/ registry provides access to several training groups (BigEarthNet, SpaceNet, etc.).

The STAC sets point to the geospatial information cataloging[3]. Given that the STAC items describe a geometric entity, the *geometry* property is mandatory, which makes less sense for a model that does not have geometric properties per se (however, one could argue that a model could have been trained in a given geographic area).

STAC API

A STAC API is the dynamic version of a catalogue of spatio-temporal assets. It refers to a catalogue, a set, an article or a set of STAC API articles, based on the endpoint. A STAC API is defined in OpenAPI 3.0[4] supporting a specific version of STAC. For the research functionalities, an entry point for the research is defined:

https://stacspec.org/STAC-ext-api.html#operation/getSearchSTAC

Note that the API also supports extensions[5].

There is now a growing list of tools[6] of which we can mention a few:

- Intake-STAC (https://github.com/intake/intake-stac) is a Python API environment that permits STAC catalogues to be loaded directly into a Python environment.
- Cognition-datasource API (https://github.com/geospatial-jee/cognition-datasources).
- Sat-API (https://github.com/sat/utils/sat-api) is a Web API compatible with STAC and deployable in AWS[7]; uses ElasticSearch to do searches.

[3] https://github.com/radiantearth/stac-spec/blob/master/best-practices.md#data-that-is-not-spatial
[4] https://stacspec.org/STAC-ext-api.html
[5] https://github.com/radiantearth/stac-api-spec/blob/master/extensions.md
[6] https://stacspec.org/#tools
[7] https://aws.amazon.com/earth/

3.1.2 OGC API Records/Features

OGC's API standards (OGC API) define the basic elements of modular APIs to facilitate the distribution of geospatial data on the Web. OpenAPI is used to define the API's basic reusable elements. The collection of OGC's standards (WMS, WFS, WCS, WPS, etc.) will be refactored like this (see roadmap here: https://ogcapi.ogc.org/apiroadmap.html). The OpenAPI specification (formerly *"Swagger Specification"*) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API, including: the available *"endpoints"* (/users) and the operations on each of them (GET /users, POST /users), the functioning parameters, the inputs and outputs for each operation.

OGC API – Records is undergoing specification by the OGC (https://ogcapi.ogc.org/records/) and allows you to discover and access metadata on the geospatial data and services (equivalent to the CWS 3.0 standard). The API will be similar to the OGC – API Features API with a few extended search abilities (https:/www.ogc.org/standards/ogcapi-features):

1. The recommended coding for a catalogue notice will be GeoJSON to build on the tools available and facilitate STAC alignment.
2. There will be an endpoint to recover all of the searches (i.e. the properties that can be used to build a filter).
3. The content model of a recording will be comprised of a set of "basic" searches (i.e. a set of required properties that each recording must have; things such as identification, type, title, editing, rights, license, etc.).

Discussions are underway to align the STAC items/catalogues specifications with OGC API Features / Records[8]. The pygeoapi library, which proposes an initial implementation of OGC API Features, allows adding a STAC catalogue as a data provider (https://docs/pygeoapi.io/en/latest/data-publishing/stac.html). The GDAL library also has a driver for OGC API Records (https://gdal.org/drivers/vector/oapif.html#vector-oapif).

A demo with pygeoapi is available online:

https://demo.pygeoapi.io/stable

An OGC API Record catalogue gathers several sets in which one can find a list of "features" (see Figure 3 below). An OpenAPI interface is completely defined allowing other clients to connect:

[8] https://medium.com/radiant-earth-insights/ogc-api-features-stac-sprint-recap-6c876b44c9d2

https://demo.pygeoapi.io/stable/openapi?f=html

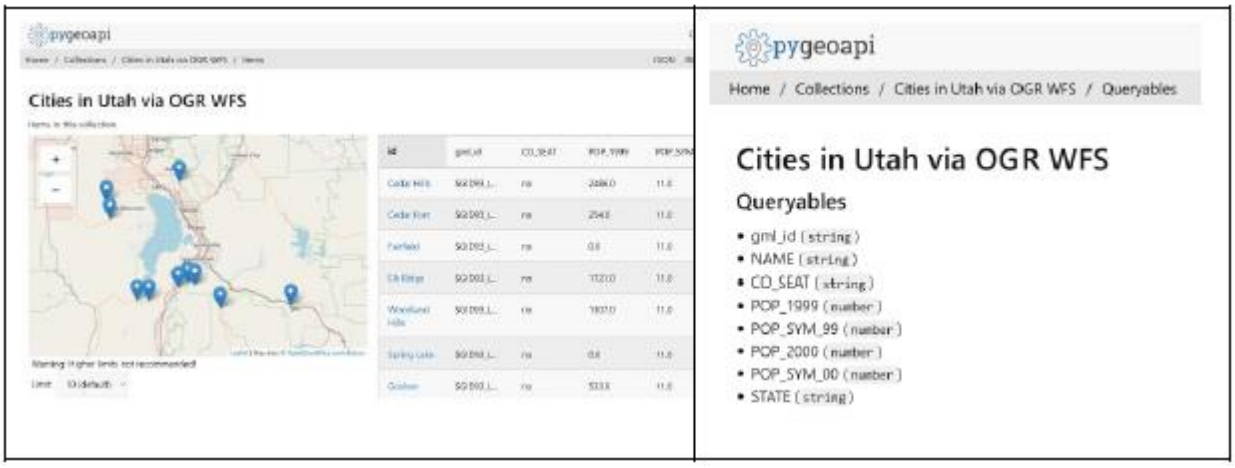


Figure 3: Example of a "Features" catalogue (left) and the properties that can be used for searches (right)

As part of OGC's testbed (Testbed-16), the idea of a models catalogue was proposed based on OGC's standards (OGC API).

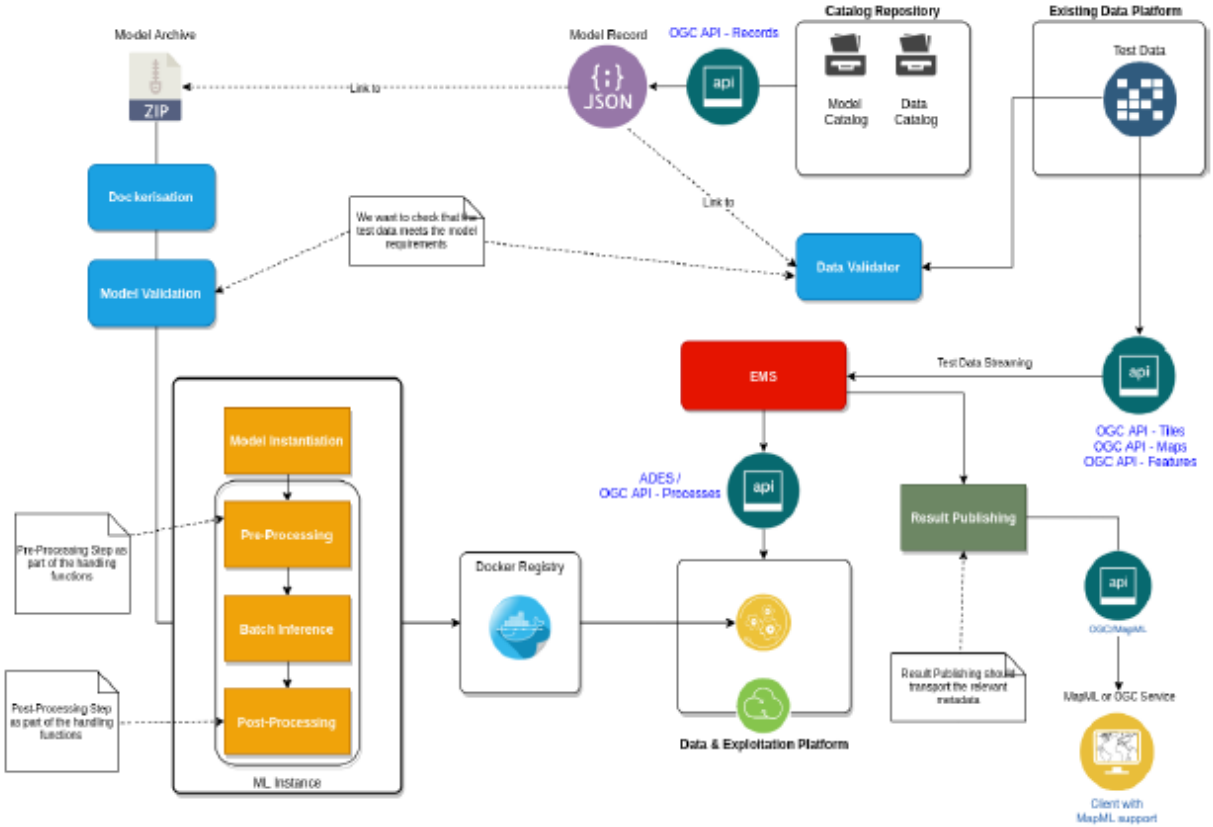


Figure 4: Principle of a models catalogue proposed in OGC's Testbed-16.

3.1.3 Other tools

There are other solutions; we mention only the most popular:

- **Catalogue Web Service (CSW)** is an OGC standard that permits publication and searches of metadata sets (*metadata records)* for geospatial data, services and related information. The catalogue metadata represent the features of the resources that can be searched and presented for evaluation and future processing by other services. The CSW standard is being re-developed as part of the OpenAPI as discussed in section 3.1.2.
- **GeoNetwork:** GeoNetwork is a catalogue application for management of spatial reference resources. It offers metadata editing and search functions as well as an interactive Web map viewer. The latest version (GN 4.0) uses OpenAPI for the REST API and includes the ElasticSearch search engine.
- **GeoNode:** is a geospatial content management system; a platform for the management and publication of geospatial data based on GeoServer.
- **ESRI GeoPortal Server:** based on the Apache Tomcat server and the PostgreSQL database works only with Windows.
- **Data Catalog Vocabulary (DCAT):** is a methodology that was developed by W3C to ensure interoperability between the various data catalogues published on the Web. In many respects, DCAT is a very open method because it does not have any requirements regarding the formats in which the catalogues must be published, i.e. the catalogues may be published in XML, RDF or xlsx. It also permits decentralized publication of catalogues and facilitates the federated search of all data on several sites. The aggregated DCAT metadata can be used as a manifest file to facilitate digital preservation.
- **CKAN:** is a powerful data management system based on DCAT that makes data accessible – by providing tools to rationalize the publication, sharing, search and use of data. CKAN is intended for data editors (national and regional governments, businesses and organizations) who want to make their data open and available. The portal used by CKAN include http://data.gov.uk and http://open-data.europa.eu. In the United States, http://data.gov uses a version of CKAN that is presented as the open government platform.

References:

1. CWS 3.0, https://www.ogc.org/standards/cat
2. GeoNetwork, https://geonetwork-opensource.org/
3. GeoNode, https://docs.geonode.org/en/master/about/index.html
4. ESRI GeoPortal Server, https://geoportal.sourceforge.io/
5. DCAT, https://www.w3.org/TR/vocab-dcat/
6. CKAN, https://ckan.org/

3.2 Machine Learning pipeline management tools

The goal here is to present a few software tools that could facilitate the management and reproducibility of ML pipelines. The first two tools (MXNet Multi Model Server and TorchServe) use an archive of the model to deploy the model in inference mode as a Web service. The MLflow and TensorFlow Extended (TFX) libraries are tools that cover a broader range in the machine learning pipeline development cycle, from training to deployment and going into production.

3.2.1 MxNet Multi Model Server

The Multi Model Server (MMS) allows all model artefacts to be grouped together in a single model archive (.mar). As such, it is easy to share and deploy (serve) the models. To archive a model, the model-archiver function is used:

```
model-archiver --model-name squeezenet_v1.1 --model-path squeezenet
--handler mxnet_vision_service:handle
```

The model's information is stored in a file, specified by model-path, which contains all of the files necessary to run the model's inference code. All of the files and subfiles (except excluded files) will be grouped together in the same .mar. A generic template is available[9]. An input function (*handling function*) is a Python entry point that the MMS can call to run the inference code. The format of that Python function is as follows:

```
python_module_name[:nom_fonction]    (par    exemple    :    lstm-
service:handle).
```

The name of the function is optional if the Python module provided follows one of the predefined conventions:

- A handle() function is available in the module
- The module contains only one class and that class contains a handle() function.

The models can then be "served" in inference mode.

```
multi-model-server --start --models
squeezenet=https://s3.amazonaws.com/model-
server/model_archive_1.0/squeezenet_v1.1.mar
```

[9] https://github.com/awslabs/multi-model-server/tree/master/examples/model_service_template

This is the content of an .mar for a squeezenet model

```
./squeezenet_v1.1
├── MAR-INF
│   └── MANIFEST.json
├── model_handler.py
├── mxnet_model_service.py
├── mxnet_utils
│   ├── image.py
│   ├── __init__.py
│   └── ndarray.py
├── mxnet_vision_service.py
├── signature.json
├── squeezenet_v1.1-0000.params
├── squeezenet_v1.1-symbol.json
└── synset.txt
```

The MANIFEST.json file contains metadata on the implementation of the model, the version, the author and the call function:

```
{
  "specificationVersion": "1.0",
  "implementationVersion": "1.0",
  "description": "squeezenet",
  "modelServerVersion": "1.0",
  "license": "Apache 2.0",
  "runtime": "python",
  "engine": {
    "engineName": "MXNet",
    "engineVersion": "^1.2"
  },
  "model": {
    "modelName": "squeezenet_v1.1",
    "description": "squeezenet",
    "modelVersion": "1.0",
    "handler": "mxnet_vision_service:handle"
  },
  "publisher": {
    "author": "MXNet SDK team",
    "email": "noreply@amazon.com"
  }
}
```

Three types of models can be archived:

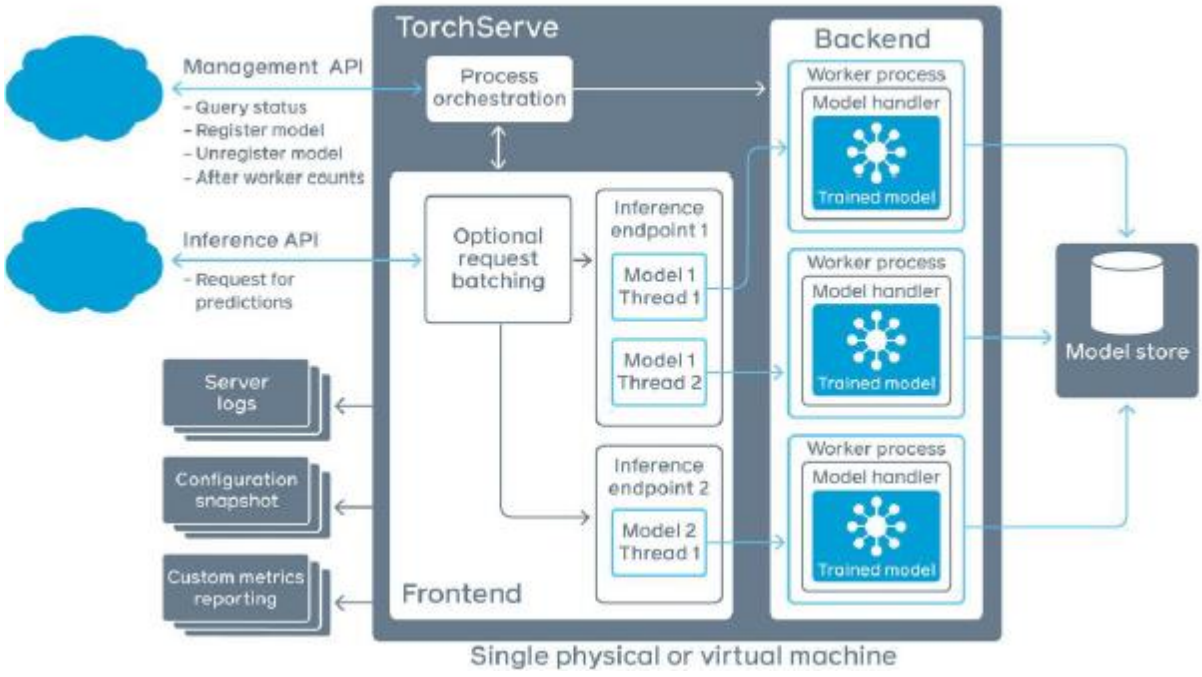
1. Gluon Models
2. Models in symbolic format
3. Models from PyTorch

A network signature file (signature.json) must be specified describing the structure of the input tensor:

```
{
  "inputs": [
    {
      "data_name": "data",
      "data_shape": [
        0,
        3,
        224,
        224
      ]
    }
  ]
}
```

3.2.2 TorchServe

TorchServe follows the same principle as MXNet, the inference service architecture is illustrated in the figure below.



The torch-model-archiver service permits the creation of a model archive (.mar) containing the following information:

- **MAR-INF:** is a reserved file name that will be used in the .mar file. This file contains archive metadata model files.
- **Model file:** a valid model name, the class is derived from torch.nn

- **Serialized file:** A serialized file (.pt or .pth) should be a control point in the case of torchscript and state_dict() in the case of eager mode.
- **Handler:** can be the name of the handler incorporated into TorchServe or the access path to a .py file to manage TorchServe's customized inference logic. TorchServe supports the following handlers outside or inside the box:
  - image_classifier
  - object_detector
  - text_classifier
  - image_segmenter[10]
  - customized handler in the form of Python file:
    --handler /home/serve/examples/custom_image_classifier:my_entry_point_func

  - Extra file: an index_to_name.json file can be added for the mapping between the model's output indices and class names

```
torch-model-archiver --model-name <model-name> --version
<model_version_number> --handler
model_handler[:<entry_point_function_name>] [--model-file
<path_to_model_architecture_file>] --serialized-file
<path_to_state_dict_file> [--extra-files
<comma_seperarted_additional_files>] [--export-path <output-dir> -
-model-path <model_dir>] [--runtime python3]
```

https://github.com/pytorch/serve/tree/master/examples

Model Zoo: https://github.com/pytorch/serve/blob/master/docs/model_zoo.md

Custom model: https://gitub.com/pytorch/serve/blob/master/docs/custom_service.md

Notes:

1. The various handlers transform the *ByteStream* into a 24-bit image, which meets the computer vision needs, but not necessarily in satellite imaging.
2. The main use of TorchServe is to serve models in inference mode via a Web service (REST), which is not necessarily the best mode of interaction.

[10] https://github.com/pytorch/serve/blob/master/ts/torch_handler/image_segmenter.py

3.2.3 Tensorflow Extended

Source code: https://github.com/tensorflow/tfx

Documentation: https://www.tensorflow.org/tfx

TensorFlow Extended (TFX)[11],[12] is a library developed by Google, which permits the definition of a complete pipeline going from training of a model, its statistical analysis as well as the data used for its training, and deployment into an infrastructure for inference in the form of a service. It uses a set of subprojects, each with specific tasks that interact based on a common metadata format.

Components:

- **ML Metadata:** this library defines the relevant information for the monitoring of references in relation to a training dataset, the hyperparameters used, the model creation pipeline, the TensorFlow version and other details specific to the training and experiment process or other interaction events with TFX. This can be saved in a *Metadata Store* in the form of a standard schema with the traditional saving and search operations. The schemas use Google's *proto* format. https://github.com/google/ml-metadata
- **TensorFlow Data Validation:** this tool permits the analysis of data from the dataset that will be used for model training or inference of new predictions with that model, when incorporated into the TFX pipeline. It mainly permits one to ensure that the data follows the expected statistical pace or to detect suspicious situations (e.g.: *data shift, unbalanced distribution, etc.),* using multiple visual interfaces for their observation along with several typical data analysis metrics.
  https://github.com/tensorflow/data-validation
- **TensorFlow Transform:** this library provides the transformation definitions useful in doing the pre-processing of data from the dataset, which will be transferred to input to the trained model or the model used for inference. The difference compared to the multiple other libraries that offer the same type of operators to transform the data into Tensors useable by the model is the definition of the operation schema that is incorporated into the metadata model. https://github.com/tensforflow/transform

[11] https://arxiv.org/abs/2010.02013
[12] https://research.google/pubs/pub46484/

- **TensorFlow (Training):** this library is the very heart of the definition of dataloaders, models and their training. It is practically always included and used in tandem with the other models given that it incorporates the implementation of low-level classes, such as Tensors and Layers defining the models. This library may be combined with the TFX component called Trainer to conserve the training and experiment metadata completed for future use with the other modules. https://github.com/tensorflow/tensorflow
- **TensorFlow Model Analysis:** this analysis tool is relatively similar to the other one, but highlights its use around generated models, rather than the data. It allows you to monitor the quality of the models obtained based on the metrics applicable over time and between different experiments. The tool is also used to validate when making decisions regarding the update of models deployed in the form of a service to ensure that the minimal performance criteria are met. https://github.com/tensorflow/model-analysis
- **TensorFlow Serving:** set of modules responsible for the deployment of new models or their update to offer their inference in the form of service API. It uses model analysis for their validation prior to ingestion into the target infrastructure, which can be based on Airflow, Kubeflow or a minimal JavaScript Web interface. The integration with the TFX pipeline allows it to determine how to load the model and the pre-processing of the data specified to produce the inference result. It also offers plug-ins for the validation of a given infrastructure to ensure the completeness of the solution as well as its complete support of the model to be deployed to ensure that it can be run. https://github.com/tensorflow/serving

As a complete solution in and of itself, coupling between the components is very highly dependent on the TensorFlow framework. The use of a different framework to define an external model (e.g.: PyTorch) very likely requires a prior conversion to that framework. Nonetheless, TensorFlow is considered the current leader in terms of industrial deployment of models. Supported by a vast community, it quickly addresses the problems specific to ML with great robustness, but a few models resulting from very recent research may take some time before they are integrated in reliable manner.

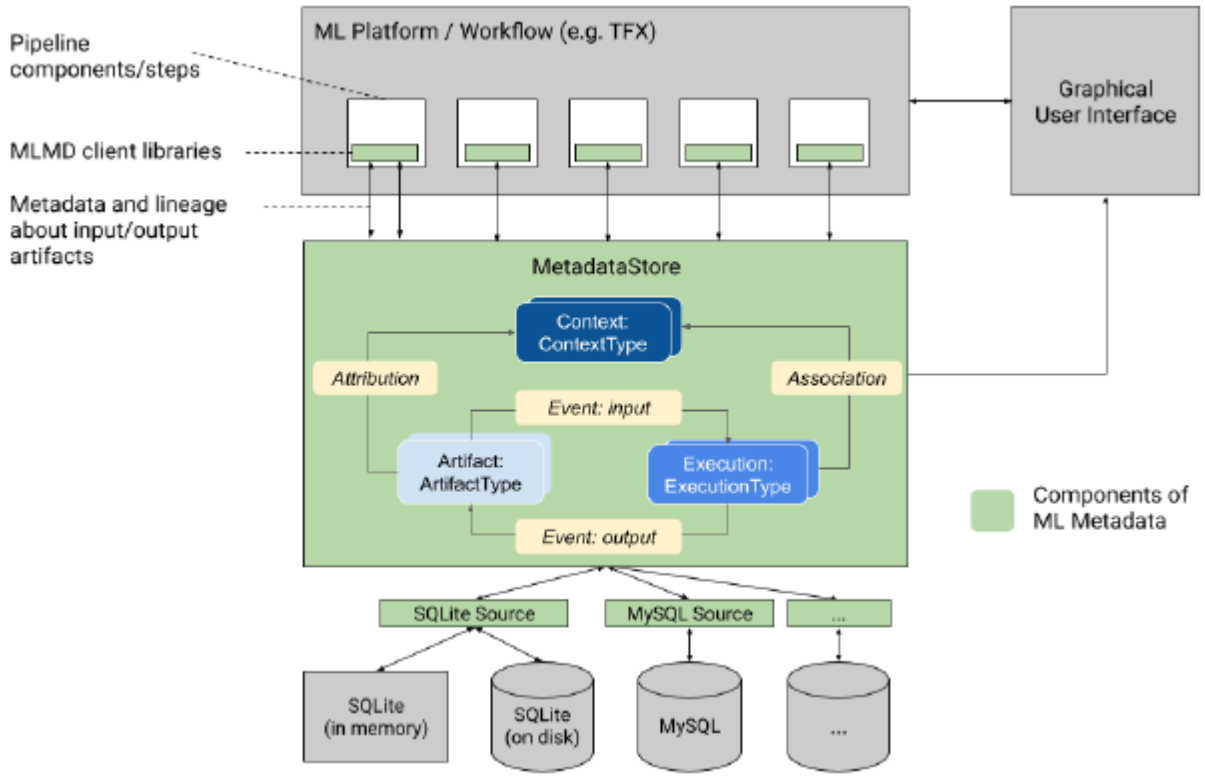


Figure 5: Overview of MLMD components

3.2.4 MLflow

Source code: https://github.com/mlflow/mlflow

Documentation: https://mlflow.org

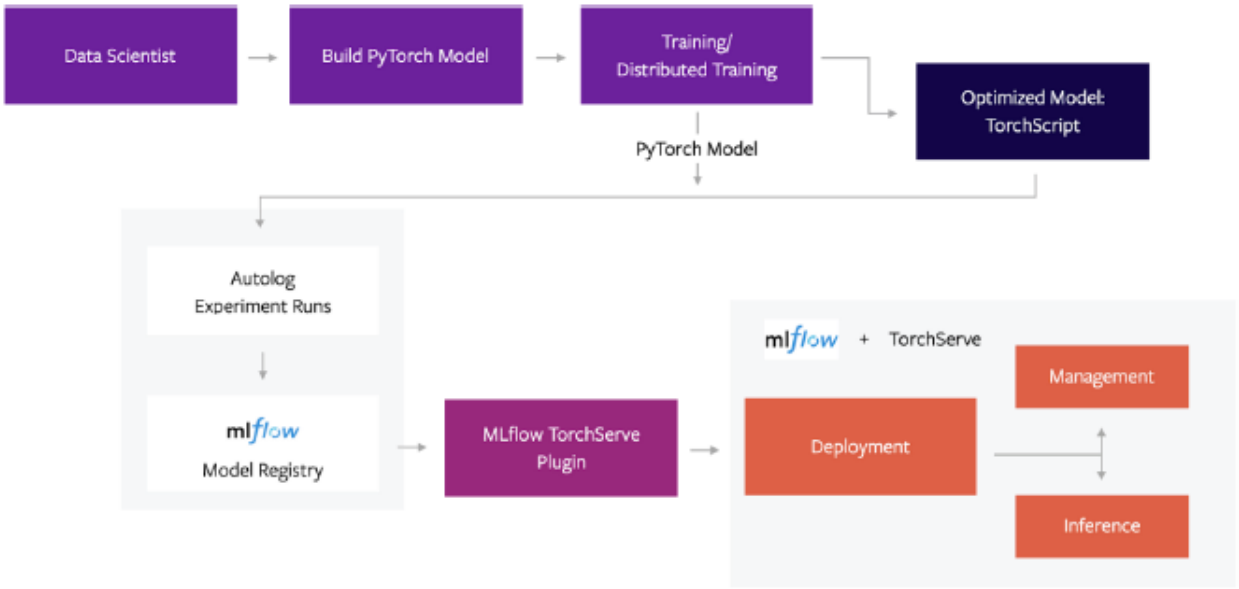
MLFlow is agnostic to the ML library and works with most libraries typically found to define a ML Pipeline (PyTorch, Tensorflow, Keras, etc.). A comprehensive list of the libraries supported is illustrated in the Tutorials and Examples as well as the model flavours (Built-In Model Flavors).

Components:

- **MLflow tracking:** essentially permits following and logging of experiences (*runs*) and the change in metrics during model training. This can be practical in maintaining monitoring of the version changes of a given code and its impact on the results produced. The output files (*artefacts*) as well as the parameters used for the run are recorded so as to allow for comparison and future viewing. The run metaparameters (*start/end time, etc.*) permit further research of the models thanks to several APIs (REST, Python, etc.). Moreover, most traditional ML libraries automatically have experimental log monitoring support.

- **MLflow projects:** permits a form of *packaging* code dependencies required to run a certain model in a reproducible manner. An *MLproject* file not only indicates the required dependencies, but also the commands to install them, create the environment and run the code. The *MLproject* created may then be used to run experiences (*runs*).
- **MLflow models:** allows you to define the flavours that a model supports. The flavours indicate the MLflow commands and functionalities with which the model can interface, whether it be a CLI call, an integration with the specific library implementing it (e.g.: *PyTorch, TensorFlow,* etc.) or more generic operations (*log, run, save,* etc.). The *MLmodel* file also allows you to define the model signature, input and output, which facilitates interfacing with the other components and tools to integrate it with the multiple operations offered (e.g.: *batch inference, serve model,* etc.).
- **MLflow registry:** allows you to keep the saved models for reuse through future experiences or inferences. As such, the saved models hold information such as the version (iterative), the status of the associated step (*Production, Archived,* etc.) as well as a flexible annotation in Markdown format. Several APIs and UIs can then interface with the registry to search and change existing models.
- **MLflow plugins:** the above components are implemented so as to permit the addition of extensions using the interfaces (Python classes), which define the components interacting in MLflow. As such, they permit extending the particular functionalities specific to a field that it not natively supported by MLflow's basic classes.

A recent announcement by PyTorch stipulates MLflow support in PyTorch pipelines:



3.2.5 thelper

Source code: https://github.com/pstcharles/thelper

Documentation: https://thelper.readthedocs.io

Description:

The *thelper* library implemented in Python is an open source code developed jointly by the CRIM and the MILA and aims to facilitate the configuration of deep learning pipelines through configuration files in JSON or YAML format. This allows for saving of "checkpoints" *enriched* not only by the model with its weights, but also by the DataLoader parameters, applicable transformations between the data and the model input, and the evaluation metrics, among others. As such, training and test environments can be saved and restarted from the command line with great certainty regarding result replicability.

An example of use of *thelper* is available in the context of application of the GeoImageNet platform (https://github.com/crim-ca/geoimagenet). The application of *thelper* in this case allows one to define a complete model training pipeline, the fine tuning and their use for inference with specific monitoring of the data source (with transformations applied), the source of the code and the complete definition of the model generated. The detailed *packaging* format provided by *thelper* permits the use of a background to define the detailed models, as is the case in the GeoImageNet's ML API (https://geoimagenet.ca/ml/api), which uses a nomenclature aligned with the OGC API – Processes. An example with thelper is given in section 4.1.

3.3 Deployment tools

These tools facilitate the creation of containers with a stable and reproducible environment.

3.3.1 repo2docker

repo2docker[13] is a simple tool that allows one to containerize the environments to promote scientific reproducibility. It uses a simple online command interface to test software reproducibility of a repository and permits reproducibility of the code in an environment agnostic in terms of language and platform. repo2docker takes the entry of a path or URL toward a repository with standard configuration files and produces a Docker image containing the depository's files with an environment built using the dependencies indicated in the….

[13] https://repo2docker.readthedocs.io/en/latest/

…configuration files. repo2docker detects various configuration files[14]: Dockerfile, environment.yml, requirements.txt, apt.txt, postBuild, runtime.txt, setup.py, etc. These various configuration files may be combined to form a rich runtime environment. When a Dockerfile is present, then all of the other files are ignored.

Notes:

1. The repository may contain documentation, notebooks, the model's code, and test data, however the weights file could not be included immediately due to its size, which is often large.
2. The targeted interaction through Python notebooks, therefore repo2docker uses a basic image based on JupyterHub to start a Jupyter server. It seems possible to change the basic image to support a GPU, for example.
3. The configuration through configuration files (apt.txt, etc.) is likely sufficient for a simple environment and does not permit all of the richness of Docker commands.

3.3.2 nvidia-docker

The nividia-docker[15] command tool is an extension to Docker that adds specific arguments to the GPU Nvidia hardware (number, indices, capacities, etc.) as well as automatic placement (mount, config) of the system's libraries and dependencies where Docker runs to provide access to the resources available within a container. Without that type of plugin (or Singularity alternative presented later), the GPUs particularly necessary for ML are not visible from inside a container, thereby preventing them from being used by underlying algorithms and greatly increasing the model training and inference times.

The Docker images built so as to permit operation of GPU Nvidia resources also need to be based on the nvidia/cuda[16] images to provide access to the CUDA and CuDNN libraries usually used by the ML *frameworks,* otherwise the dependencies must install themselves. The nvidia-docker plug-in adds only the minimal requirements to make the resources accessible (e.g.: see the *devices* with a call of the nvidia-smi command), but does not add the CUDA/CuDNN calculation libraries that the GPUs operate during operations.

[14] https://repo2docker.readthedocs.io/en/latest/config_files.html
[15] https://github.com/NVIDIA/nvidia-docker
[16] https://hub.docker.com/r/nvidia/cuda

3.3.3 Singularity

Singularity[17],[18] is a tool that permits running of containers in a manner similar to Docker, but is focused more on optimizing calculation resources, thereby facilitating access to the GPUs or requiring operations based on HPC. An argument specific to Nvidia (--nv) is available to perform the operations equivalent to nvidia-docker, which make the GPUs visible from inside the container. Again, the CUDA/CuDNN resources must be available in the target image to permit use of Nvidia GPUs in a ML context.

This plug-in permits a direct conversion between the Docker images and the Singularity images (often annotated .sif for Singularity Image Format), from a local tag or remote Docker directory, thereby making the transition transparent. The manner in which the "layers" that define the resulting image are defined nonetheless permit a more controlled and safe architecture than Docker, which requires *root* access by default. Singularity uses the context of the source system's user (and not the *scope* of permissions), that runs the container, removing the dangers associated with Docker. This particular control also permits running of untrusted containers in terms of security, in a regulated context such that it does not give the user high permissions. This is clearly more appropriate in a server deployment context where the source code of the Docker image may contain any operation (potentially malicious). Docker permissions and access privileges can be limited using various options (see the many additional Docker Security chapters), but Singularity does this more naturally. The SIF containers are unchanged and their useful load is managed directly, without extraction to disk. This means that the container can always be verified, even at the time of running, and that the encrypted content is not exposed on the disk.

References:

- Kurtzer, Gregory M. et al. Singularity – Linux application and environment containers for science. 10.5281/zenodo.1310023, htpps://doi.or/10.5281/zenodo.1310023
- Sochat VV, Prybol CJ, Kurtzer GM (2017) Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers. PLOS ONE 12911): e0188511. https://doi.org/10.1371/journal.pone.0188511
- Un example complet de construction et de deployment d'un service complet dans une instance: https://sylabs.io/guides/3.7/user-guide/running_services.html#putting-all-together

[17] https://github.com/hpcng/singularity
[18] https://sylabs.io/guides/latest/

3.4 Summary

The following table summarizes the tools described in the sections above.

TOOL	PROS	CONS	REMARKS
STAC	• Expandable • Covers data and artefacts • Bridges to OGC API Features	• Not standardized • Partial catalogues • Geospatial data only?	• Rapidly evolving
OGC API Record & Feature	• Future standard • Based on OpenAPI	• Limited implementations for now	• Specifications being finalized
Metadata serve / search solutions			
sat-api / search	• Complies with STAC API specifications • Deployable on AWS		• Used by Earth Search: https://aws.amazon.com/earth/
pygeoapi	• Extension in the form of plug-in • Based on OpenAPI • Deployable on AWS	• Implementation ongoing	
ML pipelines/deep learning management tools			
MLflow	• Complete management suite from training to deployment	• Relatively complex • Use case primarily in computer vision	• Integration into PyTorch underway
TensorFlow Extended	• Touches on production and continuous deployment	• Complex • Unclear if everything is open source	• Targets production (continuous deployment)
TorchServe	• Permits deployment of a Web service (REST) from an archive (.mar)	• Entry points are RGB oriented (24-bit) • The tool requires a specific version of PyTorch	• Interesting for a Web demo
MXNet-MMS	• Permits deployment of a Web service (REST) from an archive (.mar)	• Entry points are RGB oriented (24-bit) • The tool requires a specific version of PyTorch	• Interesting for a Web demo • The entry points are RGB oriented (24-bit)
thelper	• Incorporates functions compatible with eo data (gdal) • Permits definition of a complete pipeline in the form of config files	• The library is training configuration oriented	• Research oriented • Used by the **GeoImageNet** platform

Environment reproducibility tools			
repo2docker	• Standardization of the Git directory for definition of the executable Docker image • Facilitates reproducibility of experiments using source history	• Adds a level of abstraction during debugging • Requires mapping of the data to be processed • No access to GPUs (Docker native)	• Basic image based on Jupyter server • Container configuration should be simple
nvidia-docker	• Provides access to Nvidia GPUs from a Docker container	• Adds a level of abstraction during debugging • Requires mapping of the data to be processed	• Deployment oriented
Singularity	• Optimizations and accessibility to the HPC and GPU resources • Adds aspects of security control for container execution	• Adds a level of abstraction during debugging • Requires mapping of the data to be processed	• Deployment oriented • Configuration file similar to a Dockerfile

4 INSTANTIATION EXAMPLES

Goals: provide a few concrete cases of pipeline instantiation

4.1 Example in the GeoImageNet platform

The model is a fine-tuned SegNet on Pleiades data (4 bands). Below we are simulating the creation of a catalogue item:

- The training notebook allows a checkpoint to be generated:
  - https://github.com/crim-ca/geoimagenet/blob/master/fine_tuning_segnet.ipynb
- Possible content for a model archive (.mar):
  - Mapping between the index and class names: `index_to_name.json`
  - Checkpoint resulting from the training: `ckpt.best.pth`
  - Additional requirement: `requirements.txt`
  - Configuration file for inference in thelper: config.yml
  - The .mar is created using the `torch-model-archiver` but any archiving application can be used here (tar, zip, etc.):
    - ```
      torch-model-archiver --model-name resnet18-unet-scse --version 1.0 --serialized-file model_store/ckpt.best.pth --extra-files model_store/config.yml,model_store/requirements.
      ```
      ```

```
txt,model_store/index_to_name.json,model_store/t
est_pleiade/257/annotation_61_fixed_crop.tif --
handler image_segmenter --force
```

- The .mar is stored on google drive
- The .mar content is as follows:

```
├── annotation_61_fixed_crop.tif
├── ckpt.best.pth
├── config.yml
├── index_to_name.json
├── MAR-INF
│   └── MANIFEST.json
└── requirements.txt
```

- Basic image docker with thelper and the gdal library installed:
  - The image is based on an NVIDIA image (nvidia/cuda: 10.0-cudnn7-devel-ubuntu16.04):
  - https://github.com/crim-ca/CCCOT03/blob/main/docker/thelper-geo.dockerfile .
- Run with the thelper inference mode:
  - The .mar content should be decompressed in the working directory: `/workspace`
  - The inference test is launched directly with the instance:
  -

```
docker run -it --rm thelper-geo:base thelper
infer --config /workspace/config.yml --save-dir
/workspace --ckpt-path /workspace/ckpt.best.pth
```

An example of a catalogue item in yaml format is available here:

https://github.com/crim-ca/CCCOT03/blob/main/extension/dl-model/examples/example-thelper-item.yml

5 DISCUSSION AND RECOMMENDATIONS

Recommendations on metadata organization:

1. A model catalogue item should show mainly the metadata useful to the search:
   a. The characteristics of EO input data with a dlm:data section
   b. The references and characteristics of the model with the dlm:inputs and dlm:architecture sections
   c. The output classes (semantic)
   d. The rest of the information can be accessed through links contained in the item.
2. It is important to accurately specify the data compatible with the model considered. However, the risk here is to try to replicate a data schema with all of the relevant observation parameters while it is not possible to satisfy all of the scenarios. As such, the data schema proposed here in `dlm:data` would be inadequate to inform models resulting from SAR data, for example. It…

...would, therefore, be preferable for the model's metadata to point to a catalogue item of EO data compatible with the model. Generic metadata (instrument, platform) should suffice to filter models at an initial level.

3. If a STAC input is based on GeoJSON[19], the geographic region specified could be `nil` or describe the geographic region on which the model was trained.
4. Models should also be organized in sets either by type of instrument (optical, SAR, etc.) or by type of machine learning task.
5. A performance section could be considered to compare the performance of various models however, the lack of standard testbeds is still an issue in the field.
6. A section on the runtime environment (`dlm:runtime`) gathers the information necessary for reproduction of the model in inference mode. However, this section does not really improve quality of searches in a model catalogue. Therefore this information could be available in a register to be determined and referenced in the item.
7. The weights of the model are logically part of the assets and could be in the form of a pth file or in the form of an .mar file (a zip file) given that it is done using the torchserve and mxnet tools (sections 3.3.1 and 3.3.2). In the end, a link (with a tag or hash value) to an MLflow registry repository could also be an interesting solution, particularly if MLflow becomes a component of PyTorch in the near future.

Recommendations on the implementation of a model catalogue:

Catalogue implementation:

1. We recommend a STAC extension even if a model is not really a geospatial entity at first glance. However, one could argue that a model trained on data acquired over a determined geographic area, for example a model trained on a training set like EuroSAT, could then justify specifying a geographic area in the metadata that thereby defines its area of validity.
2. A STAC model catalogue should be compatible with the OGC's standards currently being re-developed (OGC API Record and Feature) thereby allowing for entry points through the OGC's API. All of the properties of the STAC item then become automatically queryable.
3. In terms of solutions compatible with the STAC API, the choice will depend on the size of the catalogues being considered:

[19] https://github.com/radiantearth/stac-spec/blob/master/best-practices.md#unlocated-items

a. For local catalogues, the SAT-SEARCH library combined with INTAKE-STAC permits searches and access to assets through a local path[20].
b. For a larger catalogue with search services, SAT-API permits the use of endpoints (OpenAPI) on AWS, see, for example, the earth-search service. STAC suggests also using content management tools (Drupal) or open data catalogue, such as CKAN[21].

Use of code repositories (GitHub):

1. Increasingly, most search results are in code repositories, such as GitHub. Therefore, it is preferable to treat those repositories like assets that can be viewed. The documentation, model implementation, etc., are then organized and versioned to a single address. Nonetheless, the files containing the weights should be managed separately. Therefore there is added value in defining a catalogue item that brings together that dispersed information. Figure 5 below gives an overview of the possible interactions with a GitHub repository as a starting point.
2. An archiver, similar to torch-model-archiver could be proposed, which could generate an archive (.mar) as well as a catalogue item from a GitHub repository.
3. It would be interesting to implement a tool that could automatically build a Docker or Singularity container using a catalogue item along the lines of the repo2docker tool. Moreover, that tool could help validate that the information is complete in the item.
4. It would also be interesting to propose a sort of "wrapper" or pattern to standardize the model's entry (and exit) points in inference mode. Another possibility is to use libraries like thelper or geo-deep-learning that help package inference pipelines and specify the parameters through configuration files.
5. We can pre-resolve the locally compatible environments with conda-lock[22] to convert the `environment.yml` file into a `conda-linux-64.lock`, which is an explicit list of compatible packages resolved by Conda. The key benefit of this method is that if you reconstruct in the future, the resulting Conda environment is identical, which improves reproducibility.
6. For the specification of a run container, we recommend deriving the environment of a basic image containing at least the basic libraries (cuda, etc.) to permit deep learning. To maintain good reproducibility, avoid referring to the tag latest, but rather a specific SHA key.

[20] https://github.come/intake/intake-stac/blob/master/examples/aws-earth-search.ipynb
[21] https://github.com/radiantearth/stac-spec/blob/master/best-practices.md#dynamic-catalogs
[22] https://pypi.org/project/conda-lock/

Other considerations:

1. The following tools, currently emerging, could impact the organization of model catalogues:
   a. Cloud Optimized Geotiff (COG)[23]: offers the possibility of "streaming" the geospatial data in tiles from a URL. The key use scenario is viewing large datasets in python notebooks or light Web clients. Initially, nothing prevents using those flows to perform the analysis and processing. However, those flows are generally optimized on the server side to obtain compressed data flows, which could impact the inference result.
   b. Along the same lines, PyTorch just launched WebDataset[24] for distribution of training data *"batches"* in flux. This tool could permit dissemination of test data, however the available decoders are mainly oriented toward computer vision.
   c. A STAC extension for data annotation[25] was just proposed. It is used by MLHub, which is a repository of training sets[26] based on STAC technology. In the end, one could imagine adding links to training sets on MLHub.

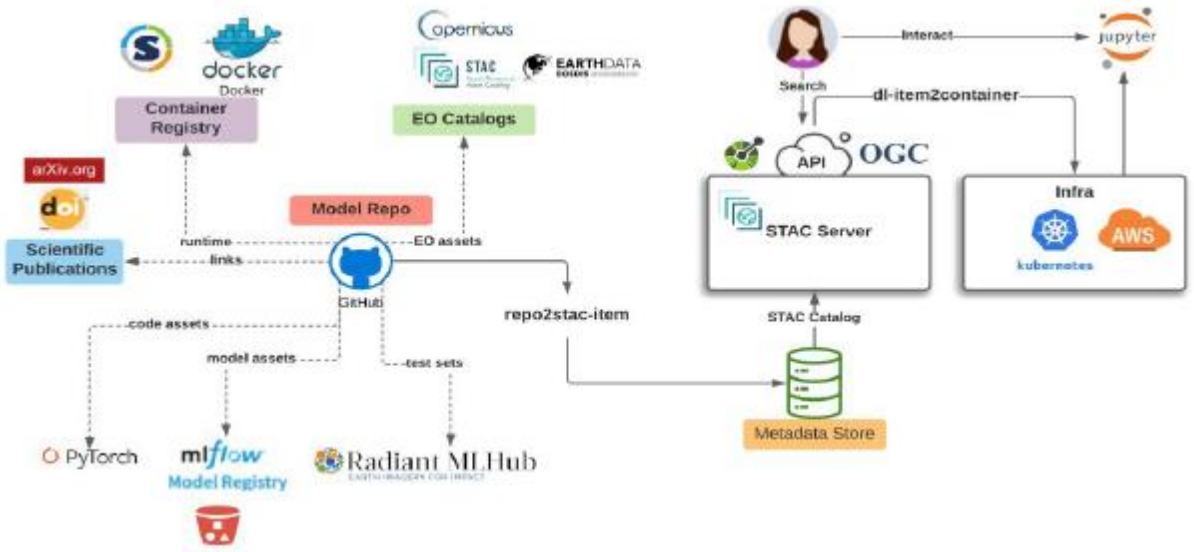


Figure 6: Top-level diagram illustrating what could be the ecosystem surrounding the deep-learning-model catalogue.

[23] https://www.cogeo.org/
[24] https://webdataset.readthedocs.io/en/latest/
[25] https://github.com/radiantearth/stac-spec/tree/master/extensions/label
[26] http://registry.mlhub.earth

APPENDIX 1 – REVIEW OF THE LITERATURE ON AUTOMATIC GRADING