Building a satellite data business with Python - some examples from Cibo Labs

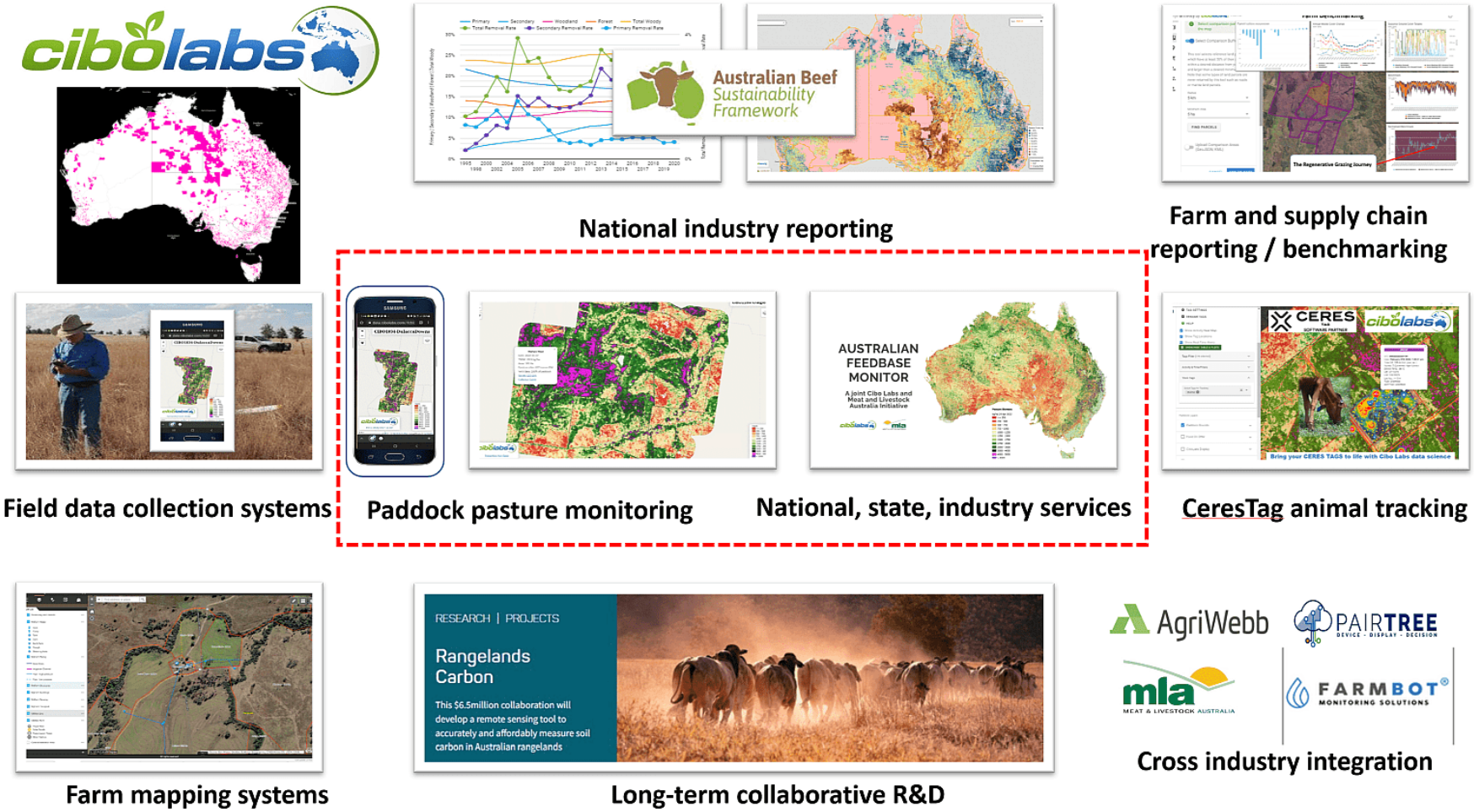
By Peter Scarth

This notebook contains some examples of how we access and use cloud based earth observation data to build an operational data pileline at Cibo Labs. We'll cover:

- What Cibo Labs is, and who our users are
- How we collect field data
- How we access satellite imagery
- How we use Python to process and analyse satellite imagery
- Our cloud architecture

Is my Grass & Ground Going to Last?
New approaches to mapping and monitoring the feed base and natural capital for Livestock producers

cibolabs



Building a cloud based pasture prediction stack

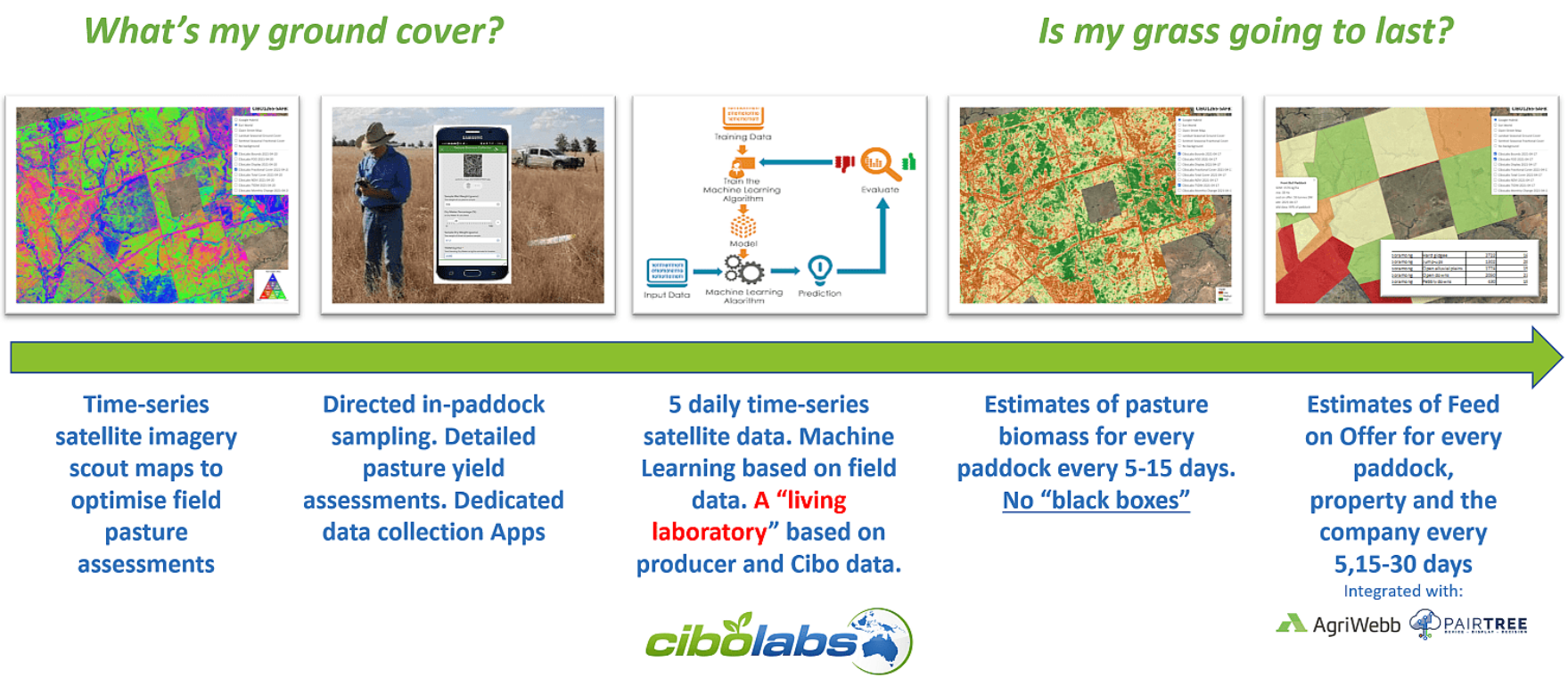
This session outlines a cloud-based workflow for building a total standing dry matter (TSDM) prediction model by calibrating field data to coincident Sentinel 2 imagery and then training a machine learning model to predict across the landscape.

It is a simplified version of the prediction and delivery method used by the startup company Cibo Labs.

Being able to work using data sets on the cloud has a number of benefits.

- Having the compute next to the imagery makes operations fast
- Having access to fast data science platforms like SageMaker or Google Colaboratory makes running experiments fast
- Having access to scalable compute and free data makes the starting cost very low, and allows you to grow as your customer base grows

The Cibo Labs PastureKey process that we'll build a much simplified version of here is summarised in this diagram:



The first step is to import all the python libraries we'll need

If you're new to this, I'd recommend starting with Miniconda then installing the required packages in a new environment. I'd also recommend replacing conda with mamba to speed up the solving of python dependencies. The Python geospatial stack is unfortunately full of

dependency conflicts that can often cause issues so always use environments. In CiboLabs, we use docker and maintain a base container onto which we add components as required.

Something like this should work:

```
conda create --name geom3001 python=3.10
conda activate geom3001
conda install -c conda-forge mamba
mamba install -c conda-forge boto3 cramjam fastapi folium gcc gdal geopandas imageio jupyterlab matplotlib
numexpr numpy pip plotly pystac-client rasterio rust uvicorn tensorflow rio-tiler
pip install titiler.application
```

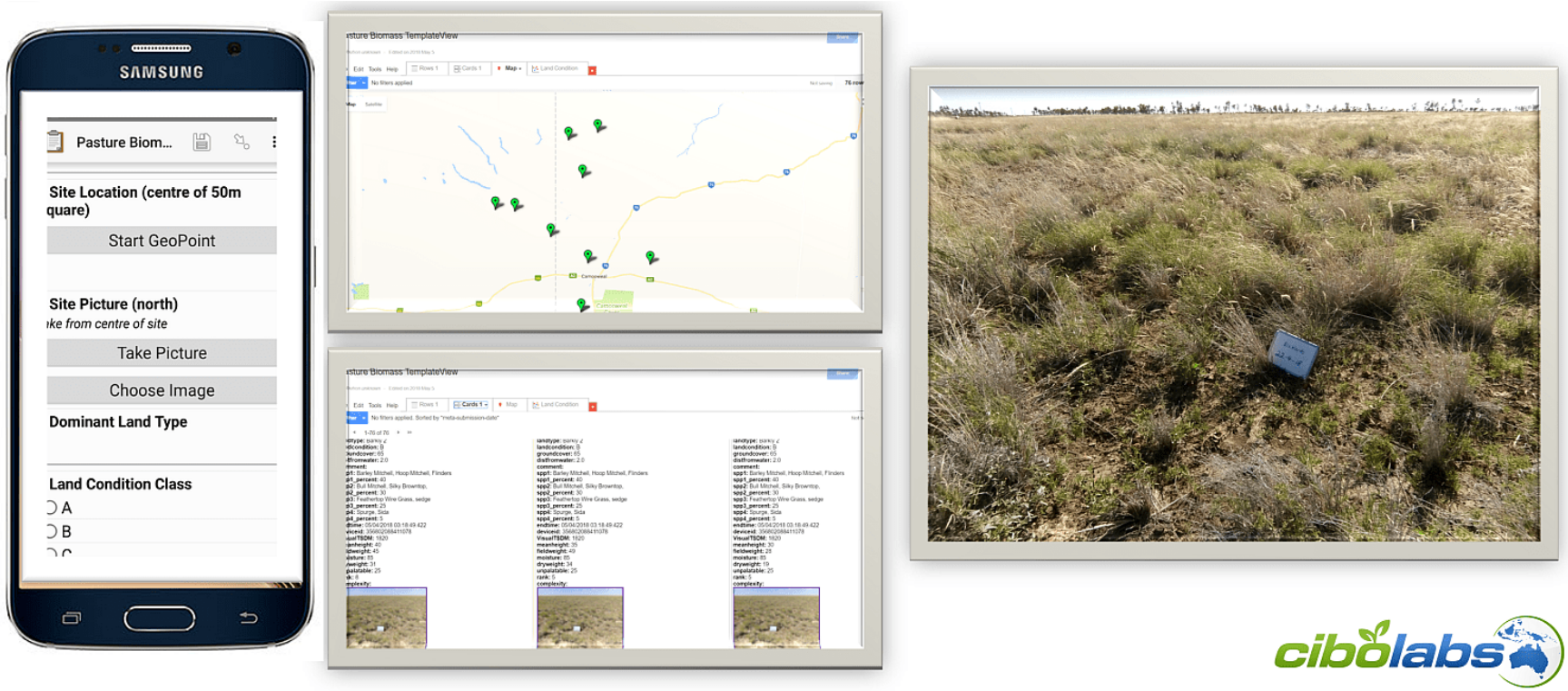
In [22]:
```python
# Importing 'timedelta' from 'datetime' module. This can be used to perform operations with time durations
from datetime import timedelta

# Importing numpy module (popularly aliased as np). Numpy is used for handling arrays, matrices and related mathematical opera
import numpy as np

# Importing matplotlib's pyplot (aliased as plt), a plotting library used for 2D graphics
import matplotlib.pyplot as plt

# Importing imread function from imageio.v2, used to read an image from a file
from imageio.v2 import imread

# Importing GDAL module from osgeo package for handling geospatial data
from osgeo import gdal

# Importing geopandas (aliased as gp) for working with geospatial data in Python
import geopandas as gp

# Importing Folium for visualizing geospatial data on interactive map
import folium

# Importing rasterio, a Python library for reading and writing geospatial raster datasets
import rasterio

# Importing tensorflow (aliased as tf), a machine learning framework
import tensorflow as tf
```

```
# Importing Client from pystac_client. STAC makes finding geospatial data easy by providing common metadata and API.
from pystac_client import Client

# Calling function 'UseExceptions' under 'gdal'.
# This makes GDAL raise Python exceptions for GDAL errors,
# allowing the programmer to handle these exceptions using Python exception handling mechanisms.
gdal.UseExceptions()
```

Organize Field Data

Field data is collected in the field using Open Data Kit (ODK) A form that allows the user to capture location, a photo and the Total Standing Dry Matter (TSDM) of pasture is given to producers.

Grass is cut from a known area using a quadrat, weighed when cut, then dried and weighed again. These measurements are scaled to give TSDM, or the weight of dry grass in the paddock which is the primary input to building farm forage budgets. Cattle will eat 2.5 to 3% of their body weight in dry matter per day, so a 300kg steer or heifer requires a pasture allowance of about 10kg DM/day to achieve potential animal growth.



Load the data

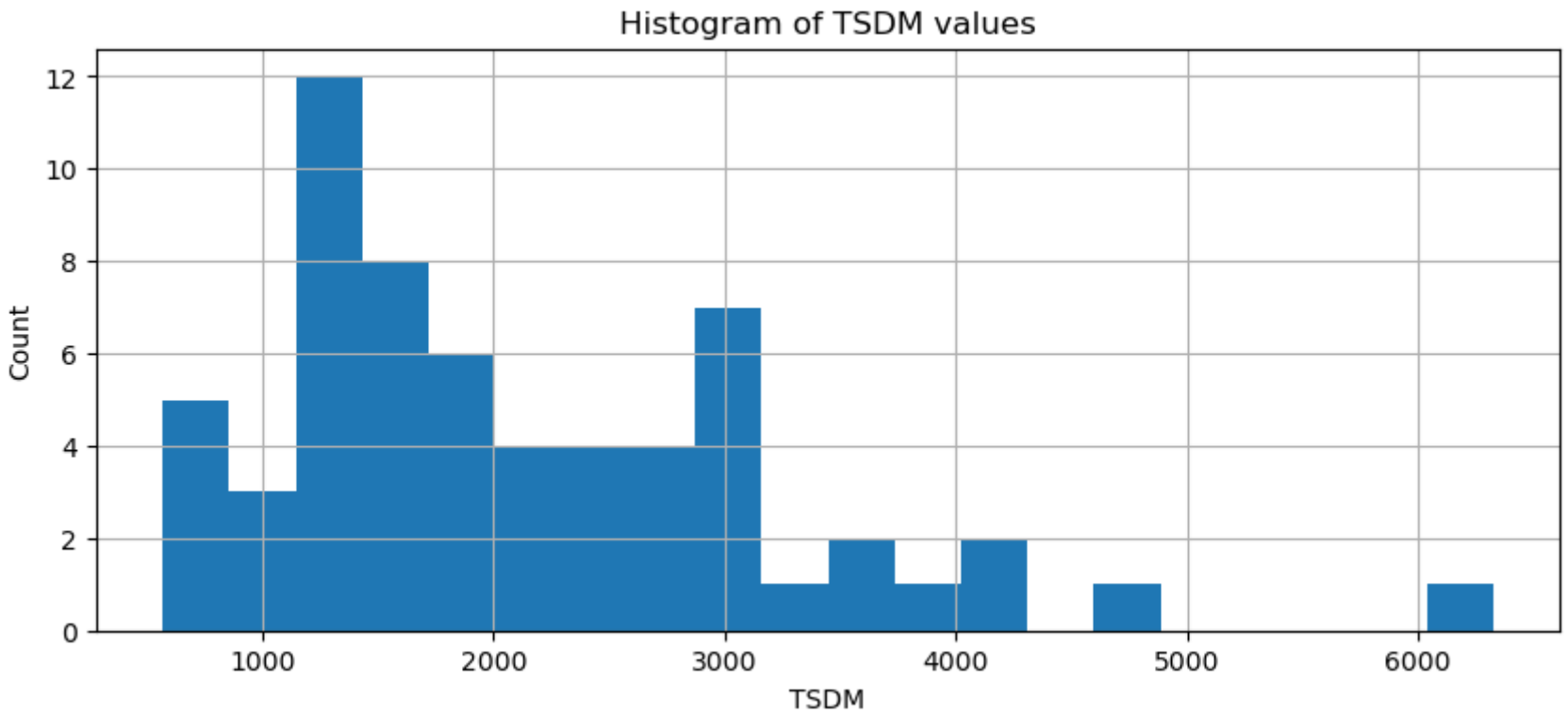
Normally we'd pull the data directly from a PostGIS database, but here we will use a FlatGeoBuf file hosted on github. It's one of the few cloud optimised vector file formats. We prefix the location with /vsicurl/ to indicate to Geopandas that we want to read from a web endpoint. We could also use /vsis3/ to read directly from a AWS S3 bucket if the file was hosted there.

In [23]:
```
# Read the field data directly from the web without downloading the dataset
FIELD_DATA = '/vsicurl/https://github.com/petescarth/GEOM3001-7001/raw/main/data/odkFieldData.fgb'

# Load the field data into a geopandas dataframe
fieldData = gp.read_file(FIELD_DATA)

##########################################################################################
# Plot a histogram of the TSDM values to check the data range
```

```
fig, ax = plt.subplots(1,1,figsize=(10,4))
ax.hist(fieldData.tsdm, bins=20);
ax.set_ylabel('Count')
ax.set_xlabel('TSDM')
ax.set_title('Histogram of TSDM values')
ax.grid()

#############################################################################################
# Print the header to inspect the data
fieldData.head()
```

Out[23]:

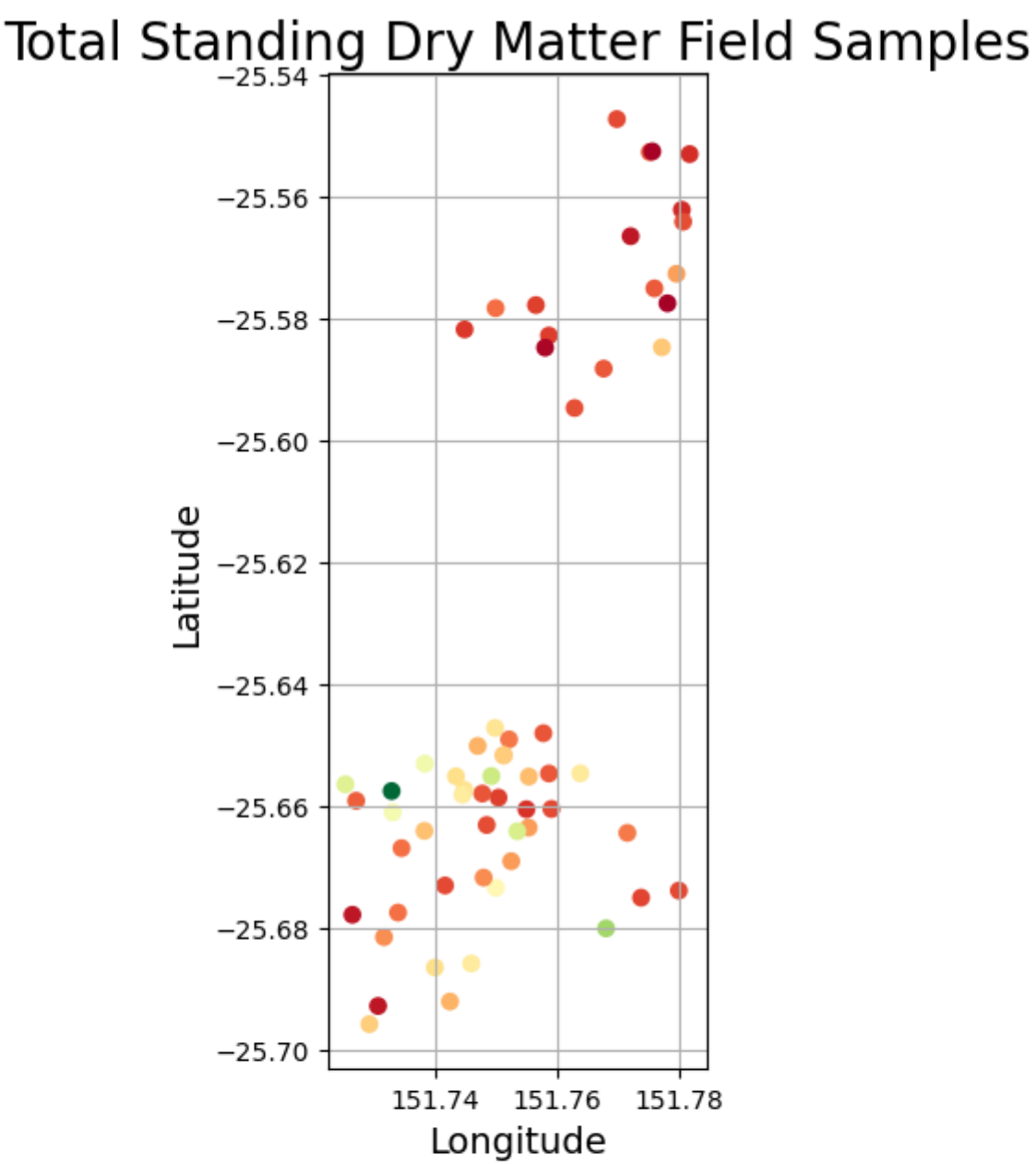
	transect_id	tsdm	points_center_lat	points_center_lng	sample_date	geometry
0	rqgsioklsvaukfyrtprndywruvjfhesx	1400	-25.673810	151.779993	2020-05-21	POINT (151.77999 -25.67381)
1	pmrrncgczplamiqrffexbsevkfjtcxzy	1335	-25.674966	151.773798	2020-05-21	POINT (151.77380 -25.67497)
2	ehtgxxdmqufrmjcpyvylhqmpfjjypaji	1845	-25.664339	151.771542	2020-05-22	POINT (151.77154 -25.66434)
3	kcfisqqjhujgxothionficnmnfaoofkf	4630	-25.680011	151.768055	2020-05-21	POINT (151.76806 -25.68001)
4	criebpvmarctjkawpdcqqidcxfhvmflr	2150	-25.663473	151.755347	2020-05-21	POINT (151.75535 -25.66347)

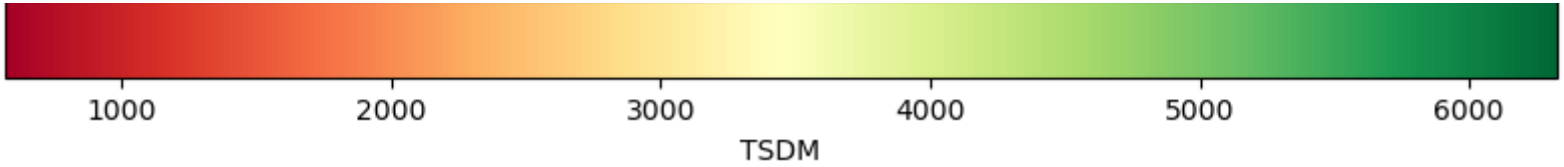


Plot the data using Matplotlib on the Dataframe

Geopandas simplifies working with vector data in Python. Here we simply plot where the data is.

In [24]:
```
# Plot the dataframe using pandas with a grid spacing of 0.02 degrees
ax = fieldData.plot(figsize=(10,10), aspect='equal', column='tsdm',
                    cmap='RdYlGn', legend=True, legend_kwds={'label': "TSDM", 'orientation': "horizontal"})
# Set title and labels
ax.set_title("Total Standing Dry Matter Field Samples", fontsize=20)
ax.set_xlabel("Longitude", fontsize=14)
ax.set_ylabel("Latitude", fontsize=14)
# Create a grid
ax.grid(True,)
# Get rid of the offset on the x-axis
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```


Total Standing Dry Matter Field Samples
−25.54
−25.56
−25.58
−25.60
−25.62
−25.64
−25.66
−25.68
−25.70
Latitude
151.74
151.76
151.78
Longitude



Select the best Sentinel 2 image from AWS

We want to find the most cloud free image close to the field data acquisition date. Pasture mass can change quickly in the growing season, or if grazed so we try to get imagery as close as possible to the date people were in the field. We could also use the cloud masks that come with the imagery if no cloud free scenes were available - it's just a slightly more complicated workflow. To do this we:

- filter from the 1,000,000's of images using the dates and bounds of the field data
- select the tile with the lowest cloud to extract pixel values from to build a prediction model

First create the time range and bounding box for the search:

In [25]:
```
bufferDate = 5 # days buffer around field work dates

# Work out the date range of the field data buffered by the specified number of days
earliestImageDate = (fieldData.sample_date.min() - timedelta(days=5)).strftime("%Y-%m-%d")
latestImageDate = (fieldData.sample_date.max() + timedelta(days=5)).strftime("%Y-%m-%d")

# Use thesxe dates to make a time range filter to use in the STAC query
timeRange = f"{earliestImageDate}/{latestImageDate}"

# Work out the bounding box of the field data to use in the STAC query
bbox = fieldData.total_bounds

# Print some info to the screen
print(f'Searching for imagery between {earliestImageDate} and {latestImageDate} using a bounding box of\n{bbox}')
```

```
Searching for imagery between 2020-05-14 and 2020-06-01 using a bounding box of
[151.7252935 -25.695717  151.7817585 -25.5472235]
```

Now search the STAC catalog

The SpatioTemporal Asset Catalogs (STAC) specification is a common language to describe geospatial information, so it can more easily be worked with, indexed, and discovered.

A spatiotemporal asset is any file that represents information about the earth captured in a certain space and time.

Here we use PySTAC, a library for working with STAC, to find the Sentinel 2 tiles of interest in the Sentinel-2 Cloud-Optimized GeoTIFFs AWS Bucket

In [26]:
```
# Search the Element84 STAC catalog
client = Client.open("https://earth-search.aws.element84.com/v1")
s2Search = client.search(
    collections=['sentinel-2-l2a'],
    datetime = timeRange,
    bbox=bbox
    )

# Show the results of the search
print(f"{s2Search.matched()} items found:\n{list(s2Search.items())}")
```

```
4 items found:
[<Item id=S2A_56JLS_20200529_0_L2A>, <Item id=S2B_56JLS_20200524_0_L2A>, <Item id=S2A_56JLS_20200519_0_L2A>, <Item id=S2B_56JLS_20200514_0_L2A>]
```

Show thumbnails and some information on the found Sentinel 2 Tiles

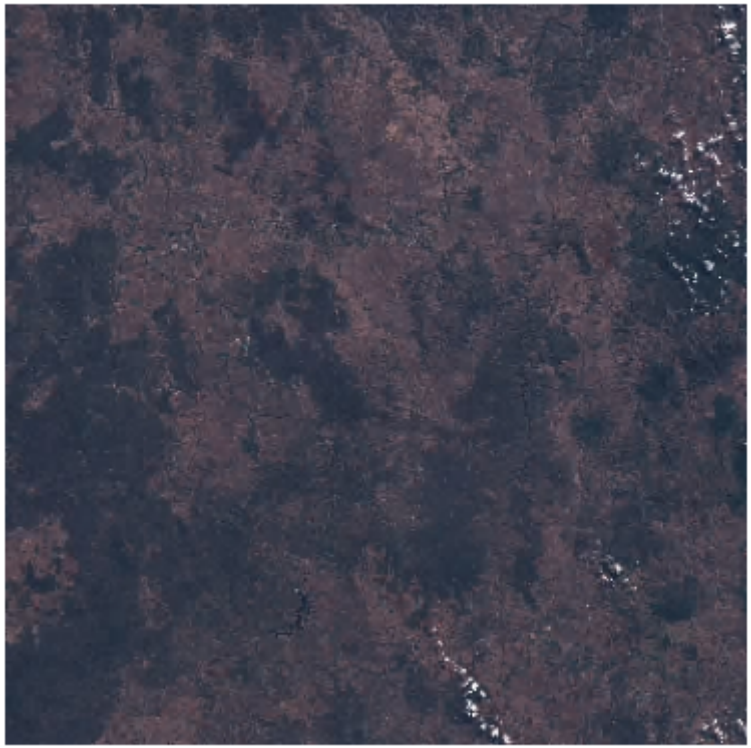
STAC results are returned as "Pages" of results, to avoid overwhelming the calling process with 1000's of results. The logic in the following code is to:

- look through each "page" of results,
- look at the items in each page and download the JPEG thumbnail for each tile as a quick look
- keep track of the item with the lowest cloud for further analysis - the 'bestItem'

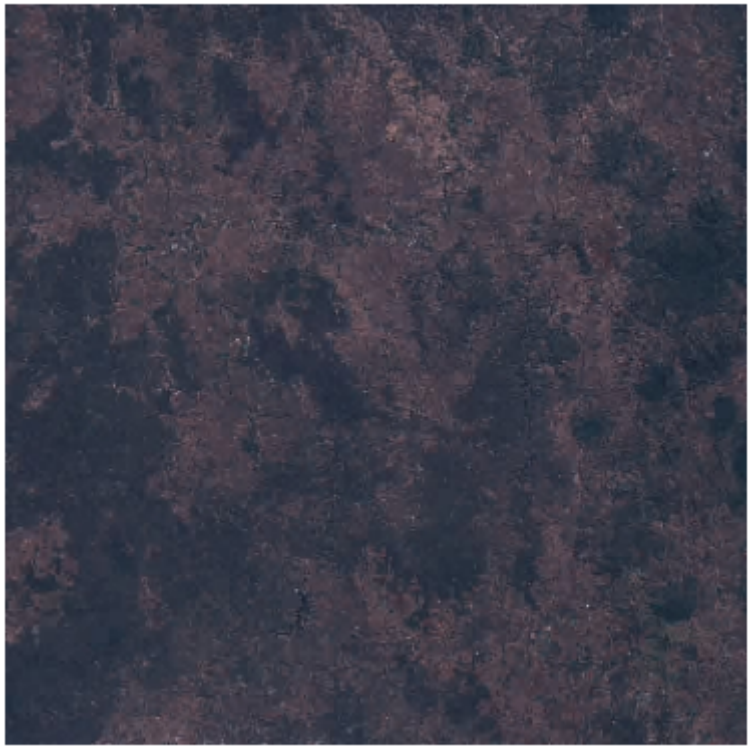
In [27]:
```
# Variables to keep track of the Sentinel 2 asset, or item, with the lowest cloud cover
lowestCloudCover = 101
bestItem = None

# Loop for each page of results
```

```
for page in s2Search.pages():

    # Loop for each item in a page
    for item in page:

        # Get the amount of cloud from the item
        cloudAmount = item.properties['eo:cloud_cover']

        # If the cloud amount is lower than the lowest cloud cover so far, update the lowest cloud cover and the best item
        if cloudAmount < lowestCloudCover:
            lowestCloudCover = cloudAmount
            bestItem = item

        # Read the thumbnail of the Sentinel 2 asset, or item, and display it in a figure.
        # The thumbnail is a small image of the asset, and is used to show the user what the asset looks like
        # The imread libraryreads the image, which is a jpeg, as a numpy array
        thumbnail = imread(item.assets["thumbnail"].href);

        # Show the thumbnail - This sometimes times out on slow connections
        plt.imshow(thumbnail)

        # Add some metadata as the title
        dateString = item.get_datetime().strftime('%Y-%m-%d')
        plt.title(f"{item.id}{chr(10)}Date:{dateString} Cloud:{cloudAmount:.2f}%")
        plt.axis('off')

        # Notebook Trick: pause allows the notebook to render the plot
        plt.pause(1)

# Print some information on the best item
print(f"{bestItem.id} had the lowest cloud cover of {lowestCloudCover}%")
```

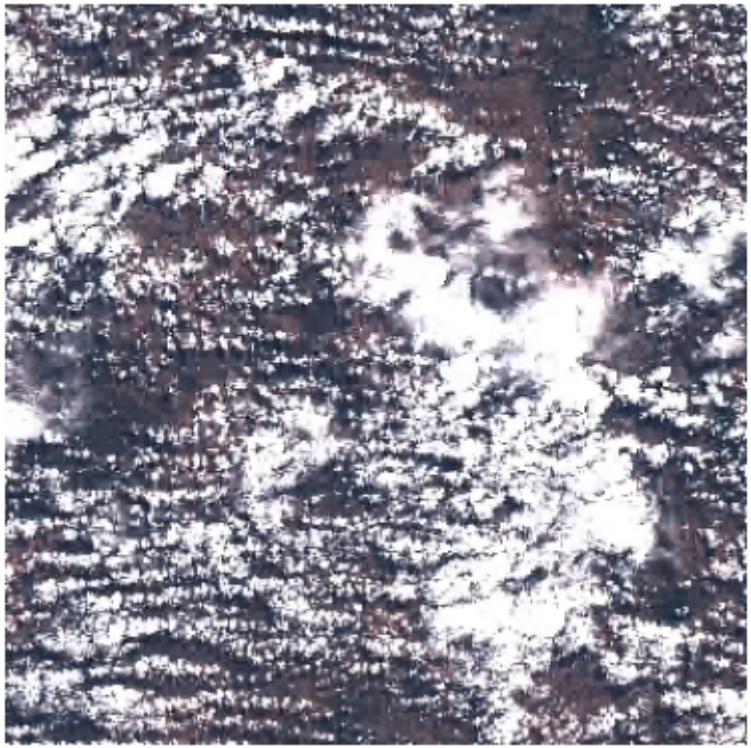
S2A_56JLS_20200529_0_L2A
Date:2020-05-29 Cloud:0.14%

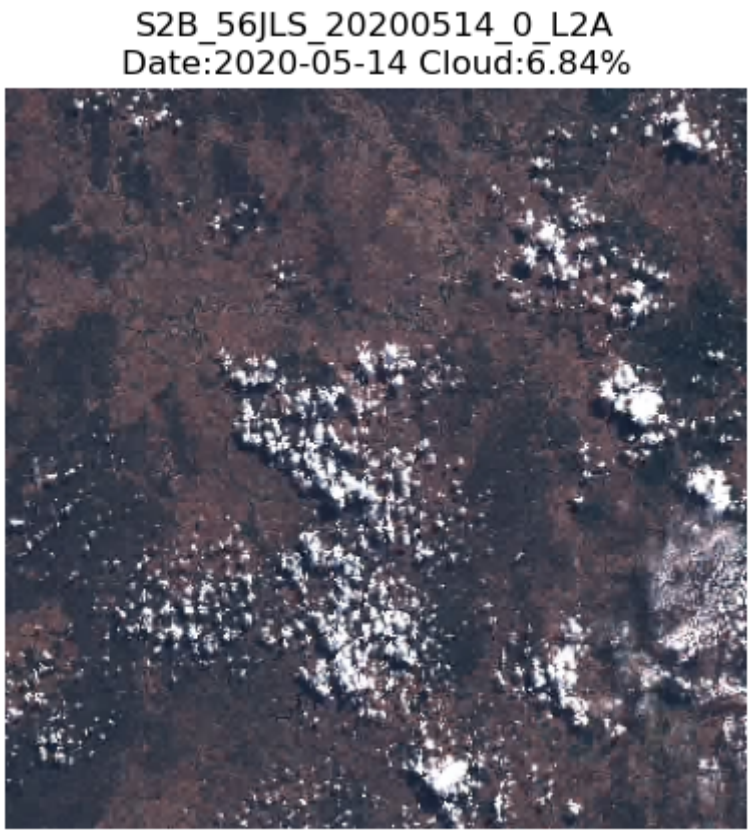


S2B_56JLS_20200524_0_L2A
Date:2020-05-24 Cloud:0.02%



S2A_56JLS_20200519_0_L2A
Date:2020-05-19 Cloud:38.37%






```
S2B_56JLS_20200524_0_L2A had the lowest cloud cover of 0.017366%
```

Have a look at the best item properties

A Spatio-Temporal Catalog item is actually just a python dictionary, and they are quite easy to see directly in a notebook.

In [28]:
```
bestItem
```

Out[28]:

- **type** "Feature"
- **stac_version** "1.0.0"
- **id** "S2B_56JLS_20200524_0_L2A"
- ▶ **properties**
- ▶ **geometry**
- ▶ **links** *[] 8 items*
- ▶ **assets**
- ▶ **bbox** *[] 4 items*
- ▶ **stac_extensions** *[] 7 items*
- **collection** "sentinel-2-l2a"

Show the sites and the bounds of the 'Best Image' on a Slippy map

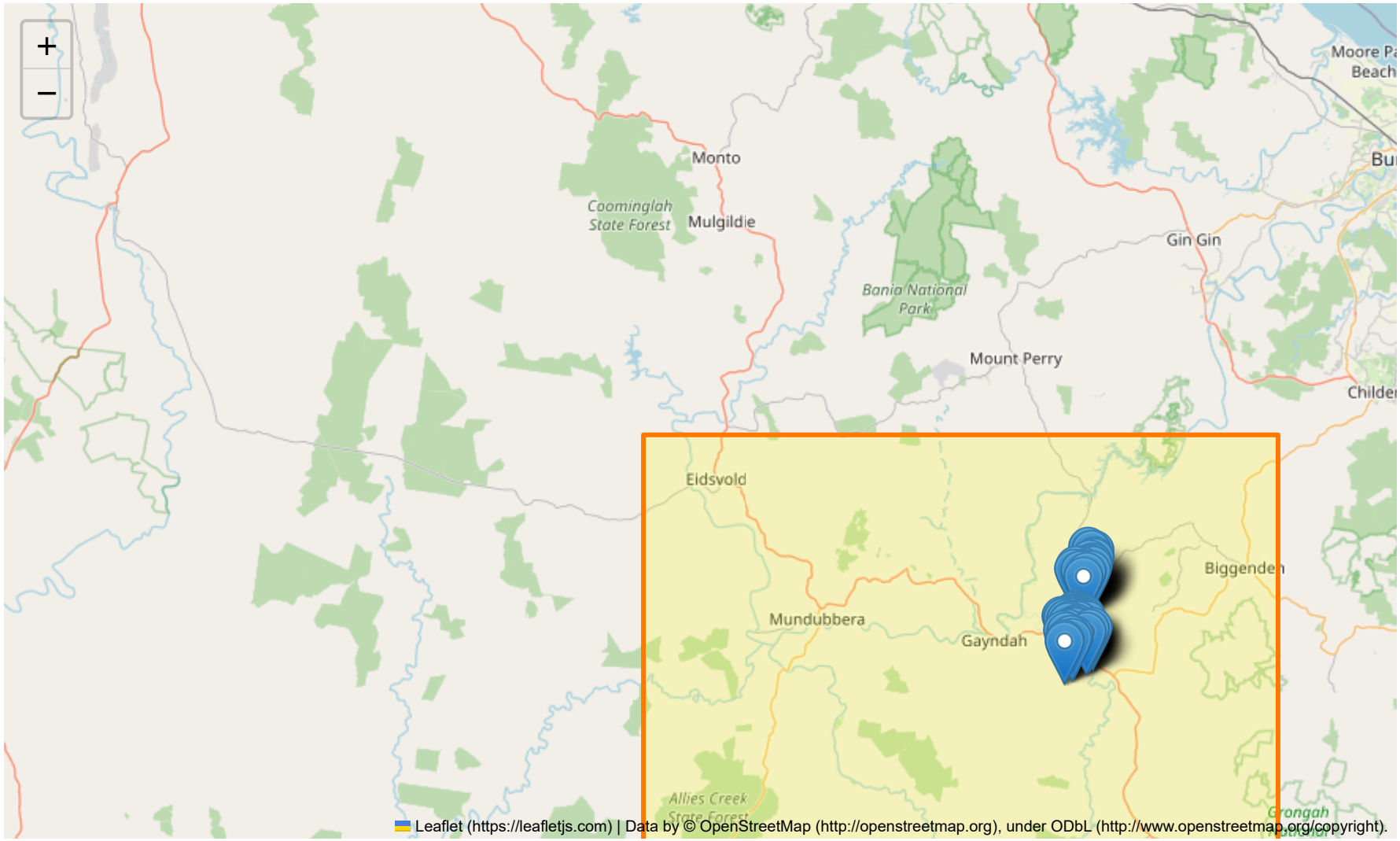
Folium uses Leaflet (he leading open-source JavaScript library for mobile-friendly interactive maps) in the background to embed "Slippy" maps directly into notebooks.

In [29]:

```
# Plot the dataframe using folium - first make the map with some defaults
map = folium.Map(
    location = [fieldData.points_center_lat.mean(),fieldData.points_center_lng.mean()],
     tiles = "OpenStreetMap", zoom_start = 9)

##############################################################################################
# Add the field points to the map
folium.features.GeoJson(
    fieldData[['geometry','tsdm']].to_json(),
     tooltip=folium.GeoJsonTooltip(fields=['tsdm'])
     ).add_to(map)

##############################################################################################
# Get the bounding box from the STAC item
bbox = bestItem.bbox

# Create a rectangle corresponding to the best image
rectangle = folium.vector_layers.Rectangle(
    bounds=[[bbox[1], bbox[0]], [bbox[3], bbox[2]]],
    color="#ff7800",
    fill=True,
    fill_color="#ffff00",
```

```
    fill_opacity=0.2,
).add_to(map)

##########################################################################################################
# Show the map
map
```

Out[29]:



Build a GDAL VRT file to access the Sentinel 2 data directly from AWS

There are a few ways of doing this, but the key is using GDAL's virtual filesystem support

GDAL's Virtual format (VRT) method allows the user to control things like:

- resampling
- mosaicing of tiles
- output spatial resolution
- output radiometric resolution and datatype
- band ordering
- output projection
- output data type
- color tables
- even band math on the fly

On AWS, the Sentinel bands are kept as separate assets called B01.tif, B02.tif etc. Because the Sentinel 2 bands have differing spatial resolutions this makes sense.

I like being able to access one Sentinel Tile as one file so I stack all the individual bands.

Here we use the "best" Sentinel 2 tile identified above and build a virtual representation of it using gdalbuildvrt

In [30]:
```
# This can take a little while to run as GDAL needs to open each band before making the VRT

# Make a name for the VRT file
vrtName = bestItem.id+'.vrt'
print(f'Making: {vrtName}')


# List of the band names we're looking for to stack in the VRT
s2BandNames = ['blue','green','red','rededge1','rededge2','rededge3','nir','swir16','swir22']

# List to hold the individual band URLs
urlList = []
```

```
# Loop for each bandname
for band in s2BandNames:
    # Get the url for the band
    url = bestItem.assets[band].href
    print(url)
    # Add it to the list
    urlList.append('/vsicurl/' + url)


# First set some options for the VRT file
vrtOptions = gdal.BuildVRTOptions(resampleAlg='average',
                                  separate=True,
                                  resolution='lowest',
                                  )
# Then make the VRT file using the list of URLs
vrtData = gdal.BuildVRT(vrtName, urlList, options = vrtOptions)
# Close the VRT file after we're done
vrtData = None
```

```
Making: S2B_56JLS_20200524_0_L2A.vrt
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B02.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B03.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B04.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B05.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B06.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B07.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B08.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B11.tif
https://sentinel-cogs.s3.us-west-2.amazonaws.com/sentinel-s2-l2a-cogs/56/J/LS/2020/5/S2B_56JLS_20200524_0_L2A/B12.tif
```

Test the VRT by calling gdalinfo

gdalinfo simply lists information about a raster data set.

In [ ]:
```
!gdalinfo {vrtName}
```

Access the Sentinel Data through the VRT

One of the cool things about Cloud Optimized GeoTIFFs is that if you request a reduced spatial resolution it will read from the file overviews.

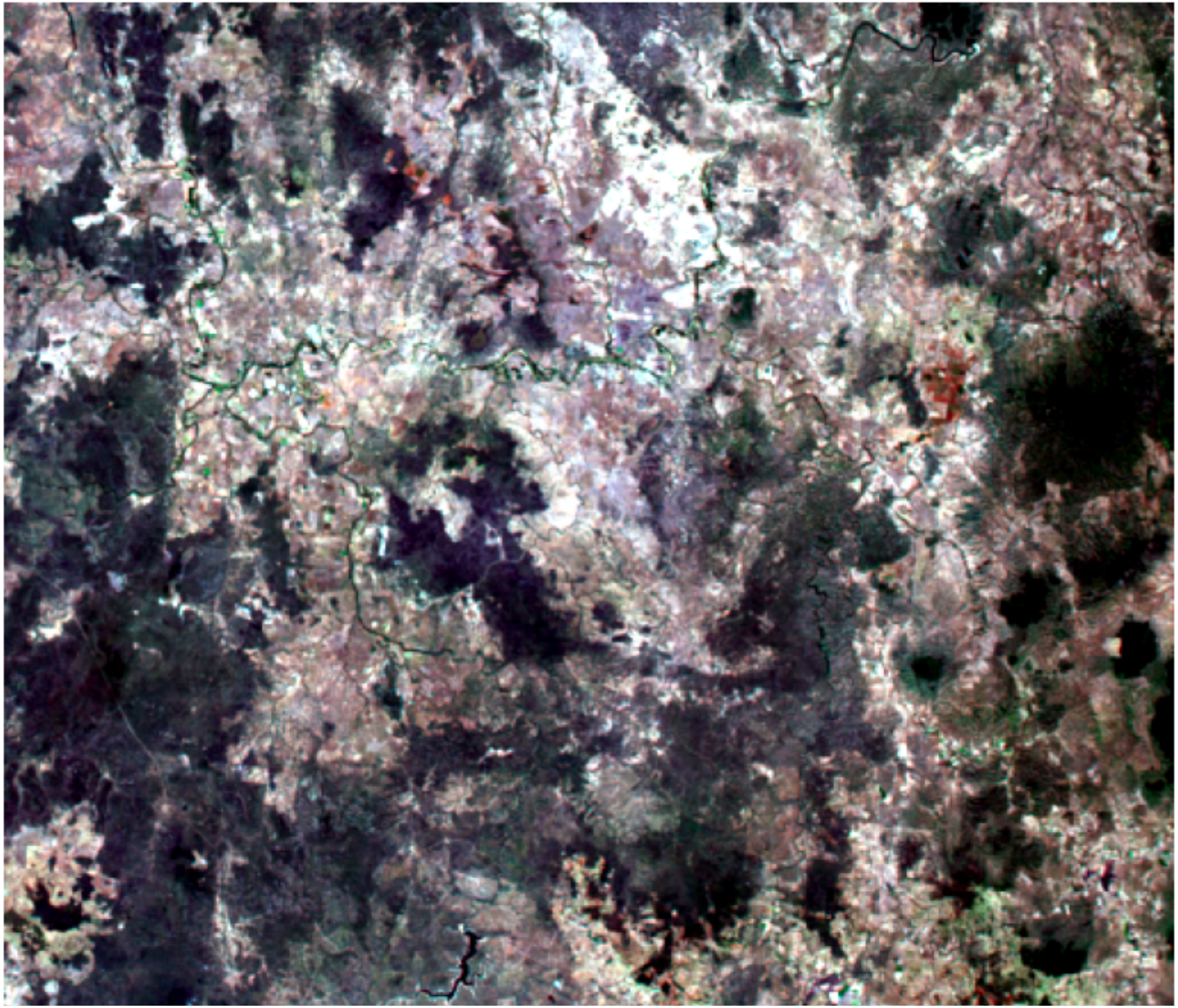
This keeps data transfers low.

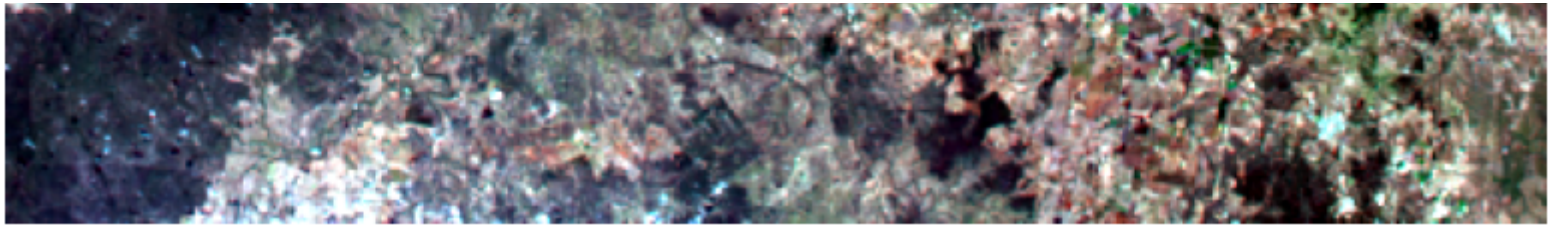
The cell below:

- Reads a three band reduced resolution (512 x 512 pixel) thumbnail of the Sentinel data using rasterio on the VRT
- Stretches the data using a percentile stretch to make a pleasing visual image
- Displays the image in the notebook

In [32]:
```
# List of the bands we want to read from the VRT
bandList = [3,2,1]

# Open the VRT using Rasterio
cloudRaster = rasterio.open(vrtName )

# Get only the data for the bands we want using a decimated read
falseColourThumbnail = cloudRaster.read(bandList,out_shape=(3, 512, 512))

# Change the ordering for matplotlib that wants the bands in the right order
falseColourThumbnail = np.rollaxis(falseColourThumbnail,0,3)

# Scale the data using a 2:98% stretch
scaleStats = np.percentile(falseColourThumbnail,[2,98],axis=(0,1))
falseColourThumbnail = (falseColourThumbnail - scaleStats[0]) / (scaleStats[1] - scaleStats[0])

# Clip the data to between 0 and 1
falseColourThumbnail = np.clip(falseColourThumbnail,0,1)

# Plot the data
fig, ax = plt.subplots(1,1,figsize=(10,10))
ax.imshow(falseColourThumbnail,aspect=1)

# Add the VRT Name to the title
ax.set_title(vrtName, fontsize=18)
ax.axis('off');
```

S2B_56JLS_20200524_0_L2A.vrt





Link the field TSDM data to the image

We want to link the field data to the reflectance of the coincident pixel so that we can build a prediction model for TSDM.

This is pretty easy using Rasterio's sample module once we have a list of coordinates in the same projection as the image data.

In [33]:
```
# Open the VRT using Rasterio
cloudRaster = rasterio.open(vrtName)

# Reproject the geometry to match the raster coordinate system
fieldProjectedGeometry = fieldData.geometry.to_crs(cloudRaster.crs)

# Build a list of coordinate pairs to pass to Rasterio's sample function
coordList = [(x,y) for x,y in zip(fieldProjectedGeometry.x , fieldProjectedGeometry.y)]

# Print the number of coordinates in the list and some information on the raster
print(f'There are {len(coordList)} coordinates to be extracted from the raster')

# Sample the raster (without downloading the entire raster) and add it to the dataframe as a list of values
fieldData['Sentinel2'] = [x for x in cloudRaster.sample(coordList)]

# Show what the fielddata dataframe looks like now
fieldData.head()
```

```
There are 61 coordinates to be extracted from the raster
```

Out[33]:

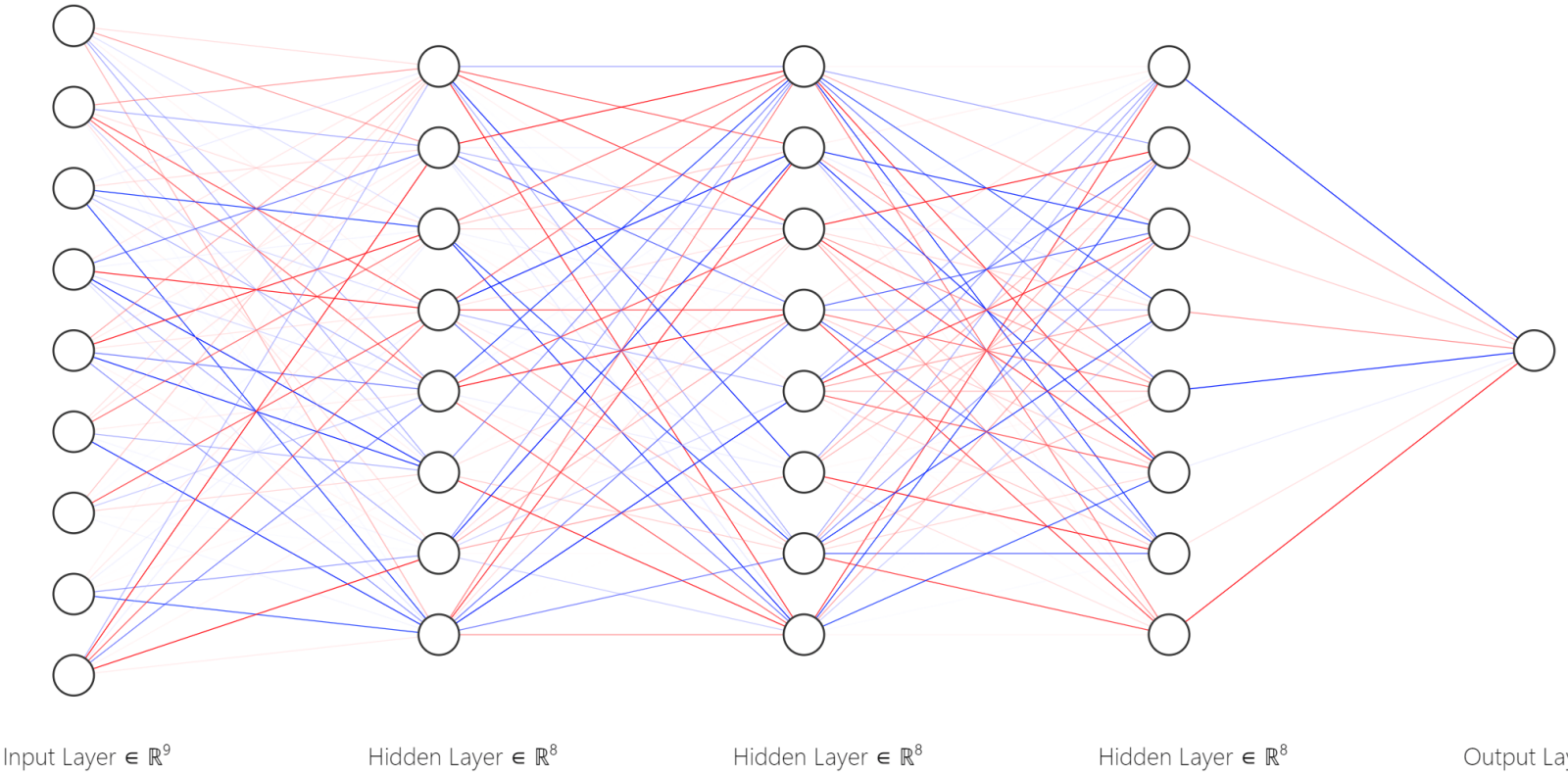
	transect_id	tsdm	points_center_lat	points_center_lng	sample_date	geometry	Sentinel2
0	rqgsioklsvaukfyrtprndywruvjfhesx	1400	-25.673810	151.779993	2020-05-21	POINT (151.77999 -25.67381)	[460, 641, 820, 1223, 1654, 1816, 2035, 2701, ...
1	pmrrncgczplamiqrffexbsevkfjtcxzy	1335	-25.674966	151.773798	2020-05-21	POINT (151.77380 -25.67497)	[473, 636, 955, 1220, 1444, 1625, 1869, 3115, ...
2	ehtgxxdmqufrmjcpyvylhqmpfjjypaji	1845	-25.664339	151.771542	2020-05-22	POINT (151.77154 -25.66434)	[271, 418, 485, 864, 1461, 1672, 1817, 1903, 1...
3	kcfisqqjhujgxothionficnmnfaoofkf	4630	-25.680011	151.768055	2020-05-21	POINT (151.76806 -25.68001)	[462, 648, 1008, 1215, 1508, 1665, 1987, 2563,...
4	criebpvmarctjkawpdcqqidcxfhvmflr	2150	-25.663473	151.755347	2020-05-21	POINT (151.75535 -25.66347)	[403, 580, 751, 1087, 1446, 1599, 1793, 2614, ...

Build a Tensorflow Multi Layer Perceptron (MLP) model to predict Total Standing Dry Matter

We define and train a very simple MLP Model in Tensorflow Keras. This is just one of many machine learning models that could be appropriate for this task, and I introduce it here as an alternative to the very popular (in remote sensing) Random Forest model. We use Tensorflow because it trains fast on a GPU, the Keras API is reasonably accessible, and it has great hardware support running on x86 and ARM CPUs, TPUs, and even in the browser.

- We scale the field and image data into the 0-1 range
- Use the Huber loss function due to it's robustness to outliers from dodgy field data
- Use the Adam optimiser
- Train it for 500 iterations, using 33% of the field data for validation
- Plot the training and validation losses

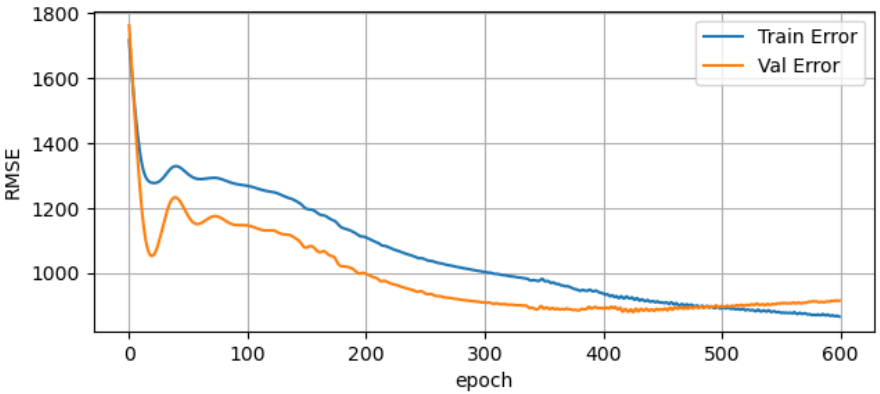
The model architecture we're using here is represented in the diagram below.

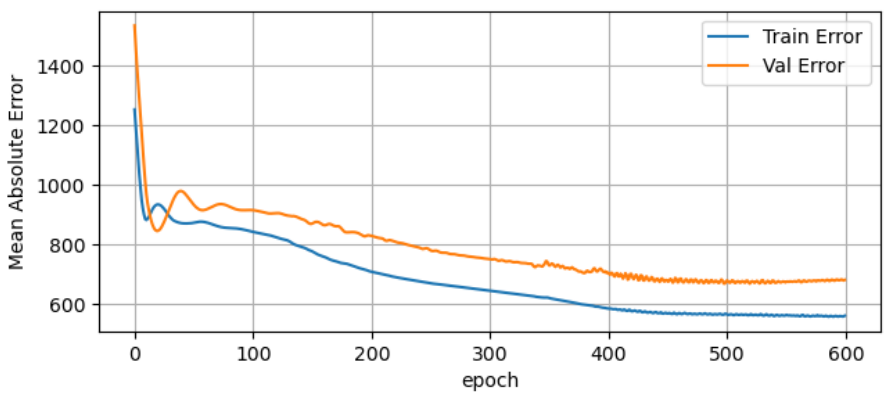


In [34]:

```
# Shuffle the data so remove spatial and temporal correlations
fieldData.sample()

# Extract X and y data from the dataframe into numpy arrays
# X is the Sentinel2 data, y is the field TSDM data
X = np.array(fieldData['Sentinel2'].to_list(),dtype=np.float32)
y = np.array(fieldData['tsdm'],dtype=np.float32)

# Scale the data from 0 to 1
X = X / 10000 # Scale to reflectance
y = y / 10000 # Scale to t/Ha/10
```

```
# Build a simple 3 layer neural network using the KERAS library
tsdmMLP = tf.keras.Sequential([
tf.keras.layers.Dense(8, activation='relu', input_shape=[X.shape[1]]),
tf.keras.layers.Dense(8, activation='relu'),
tf.keras.layers.Dense(8, activation='relu',),
tf.keras.layers.Dense(1)
])

# Compile the model using the huber loss function and the Adam optimizer
tsdmMLP.compile(loss=tf.keras.losses.Huber(delta = 0.05),
            optimizer=tf.keras.optimizers.Adam(),
            metrics=['mean_absolute_error', 'mean_squared_error'])

# Fit the model to the data using 500 training epochs and a validation split of 33%
history = tsdmMLP.fit(X,y, epochs = 600,
                    validation_split = 0.33,
                    shuffle = True,
                    batch_size = 1024,
                    verbose=0)

##########################################################################################################
# Make a plot of the training run
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(16,3))

# Plot the mean squared error loss function (remembering to unscale the data)
ax1.plot(history.epoch,  10000* np.sqrt(history.history['mean_squared_error']),label='Train Error')
ax1.plot(history.epoch,  10000* np.sqrt(history.history['val_mean_squared_error']),label = 'Val Error')
ax1.set_xlabel('epoch')
ax1.set_ylabel('RMSE')
ax1.grid()
ax1.legend();
# Plot the mean absolute error loss function
ax2.plot(history.epoch,  10000* np.array(history.history['mean_absolute_error']),label='Train Error')
ax2.plot(history.epoch,  10000* np.array(history.history['val_mean_absolute_error']),label = 'Val Error')
ax2.set_xlabel('epoch')
ax2.set_ylabel('Mean Absolute Error')
ax2.grid()
ax2.legend();
```





Use the Tensorflow model to predict on the full data set

In this example we're accessing the image from the cloud and writing locally, but this could happen entirely on the cloud.

First we define a function that takes the Sentinel 2 bands and returns a predicted TSDM value.

Then we apply that model block by block to the Sentinel 2 data read from AWS in Oregon.

Because we are processing the full image, and pulling the data across from the US, this takes a while.

In operational use, you would typically process only your region of interest at the desired spatial resolution to save compute.

Rather than reading the entire image, we use blocks of imagery called windows to minimise memory use, and can be used to parallelize the process if more CPU's are available.

We are careful to compress the tiff file, write it in tiles, and generate overviews (or pyramid layers) so that processes using it can efficiently access it on the cloud as well.

This is not a fully Cloud Optimized GeoTiff but in practice it is quite close. The final GeoTiff can be made fully COG complient by using a tool like Airbus' COGGER which reads an internally tiled geotiff and rewrites it as a Cloud Optimized Geotiff (COG) by reshuffling the original geotiff's bytes.

In [ ]:

```
# Make a filename for the prediction using the VRT name
tsdmImageName = vrtName.replace('.vrt','_tsdm.tif')
print(f'Making: {tsdmImageName}')

#########################################################################################################
# The prediction function that takes a numpy array of data and a tensorflow model object
# Returns a numpy array of TSDM predictions
def predictTsdm(nbar, tsdmMLP):

    # Get the shape of the input sentinel 2 array
    inshape = nbar.shape

    # Flatten the array to Bands x Pixels and scale the reflectance like when we trained the model
    nbar = np.reshape(nbar,(inshape[0],-1)) / 10000.0

    # Run the prediction on the tensorflow model. Increase the batch size to 512 x 512 to speed up the prediction
    tsdm = tsdmMLP.predict(nbar.T,batch_size=262144, verbose =0)

    # Round and clip the predictions from 1 to 10000 kg/Ha
    tsdm = np.clip(np.round(tsdm * 10000),1,10000)

    # Handle nodata in the input by making output nodata values 0
    tsdm[nbar[0] ==0 ] = 0
    tsdm[nbar[0] > 1] = 0

    # Reshape the output back to the input shape and return it
    return np.reshape(tsdm,(1,inshape[1],inshape[2])).astype(np.uint16)

#########################################################################################################
# Apply the prediction function to the raster
# First open the raster using Rasterio
with rasterio.open(vrtName) as src:

    # Print the size and number of bands in the raster
    print(f'Raster size: {src.width} x {src.height}')
    print(f'Tile size: {src.block_windows(0).__next__()[1].width} x {src.block_windows(0).__next__()[1].height}')
    print(f'Number of bands: {src.count}')

    # Set the output file metadata from the input raster
    kwargs = src.meta
```

```
    # Set the output driver to GTiff
    kwargs['driver']='GTIFF'
    # Set the number of bands to 1
    kwargs['count']=1
    # Set the output nodata value to 0
    kwargs['nodata']=0
    # Set the output datatype to uint16
    kwargs['dtype']='uint16'

    # Use the rasterio windowed tiling iterator
    tiles = src.block_windows(0)

    # Open the output file for writing with compression and tiling
    with rasterio.open(tsdmImageName, 'w', **kwargs,
                       compress="DEFLATE",
                       tiled=True,
                       blockxsize=128,
                       blockysize=128) as dst:

        # Iterate through the tiles
        for idx, window in tiles:
            # Read the data from the input raster
            srcData = src.read(window=window).astype(np.float32)

            # Run the prediction function defined above on the data
            dstData = predictTsdm(srcData,tsdmMLP)

            # Write the data to the output raster
            dst.write(dstData, window=window)

            # Print the tile progress followed by a carriage return to overwrite the line
            print(f'Completed tile: {idx[0]} x {idx[1]}  \r', end='')


        # Before closing the file, build overviews so that the raster can be viewed quickly in QGIS or on the cloud
        dst.build_overviews([2, 4, 8, 16, 32, 64, 128, 256])
```

Plot the TSDM image

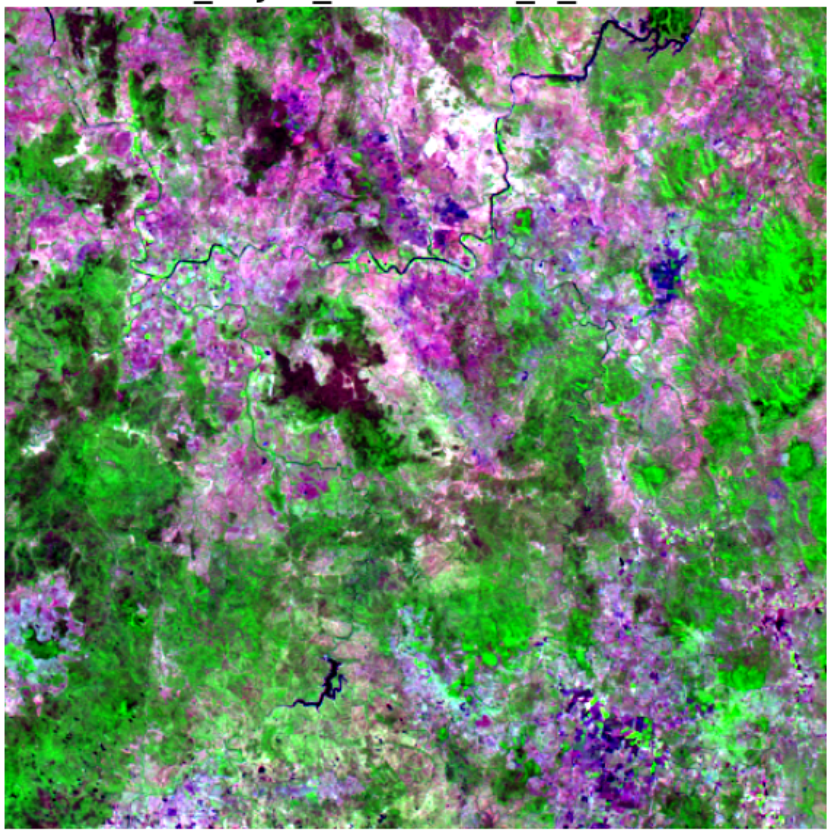
Now we've run the prediction across the entire image, we can view it next to the original data to check the prediction.

We use the same Rasterio based image reading and the stretch introduced previously to view the two images side by side.

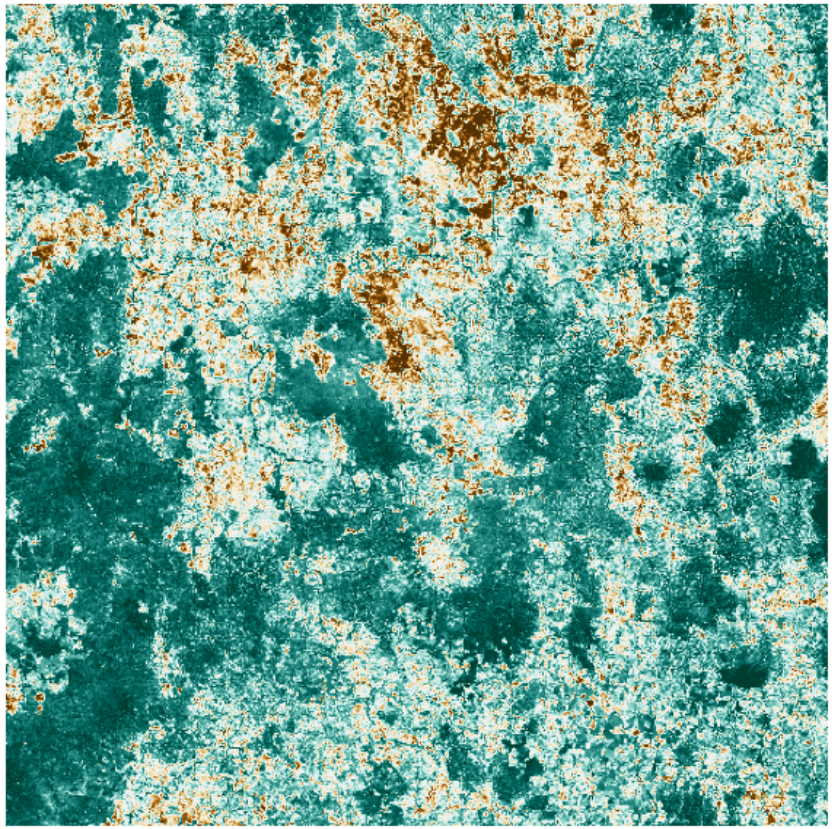
In [35]:
```
# Make a filename for the prediction
tsdmImageName = vrtName.replace('.vrt','_tsdm.tif')

#############################################################################################
# Load the Sentinel 2 data into an array ready to plot
cloudRaster = rasterio.open(vrtName )

# Get the data for the bands we want using a decimated read
falseColourThumbnail = cloudRaster.read([8,7,3],out_shape=(3, 512, 512))

# Change the band ordering for matplotlib
falseColourThumbnail = np.rollaxis(falseColourThumbnail,0,3)

# Scale the data using a 2:98% stretch
scaleStats = np.percentile(falseColourThumbnail,[2,98],axis=(0,1))
falseColourThumbnail = (falseColourThumbnail - scaleStats[0]) / (scaleStats[1] - scaleStats[0])

# Clip the data from 0 to 1
falseColourThumbnail = np.clip(falseColourThumbnail,0,1)

#############################################################################################
# Load the TSDM raster into an array ready to plot
tsdmRaster = rasterio.open(tsdmImageName)

# Get the data for the bands using a decimated read
tsdmThumbnail = tsdmRaster.read(1,out_shape=(1, 512, 512))

# Scale the data using a 2:98% stretch
scaleStats = np.percentile(tsdmThumbnail,[2,98],axis=(0,1))
tsdmThumbnail = (tsdmThumbnail - scaleStats[0]) / (scaleStats[1] - scaleStats[0])

# Clip the data to between 0 and 1 then scale by the maximum TSDM
tsdmThumbnail = np.clip(tsdmThumbnail,0,1) * scaleStats[1]

#############################################################################################
# Plot the image and prediction side by side with a legend
```

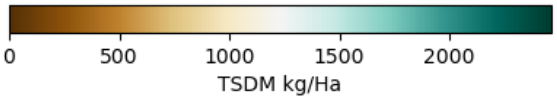
```
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(16,12))

# Sentinel 2 Image
ax1.imshow(falseColourThumbnail,aspect=1)
ax1.set_title(vrtName, fontsize=18)
ax1.axis('off');

# TSDM prediction
im = ax2.imshow(tsdmThumbnail,aspect=1,cmap='BrBG')
ax2.set_title('Predicted Total Standing Dry Matter', fontsize=18)
ax2.axis('off');

# Legend
cbar = fig.colorbar(im, ax=[ax1,ax2],location='bottom', shrink=0.3)
cbar.set_label('TSDM kg/Ha')
```

S2B_56JLS_20200524_0_L2A.vrt



Predicted Total Standing Dry Matter





Make the result available as a web service

This uses the Developmentseed TiTiler library.

"A modern dynamic tile server built on top of FastAPI and Rasterio/GDAL"

It is a cloud-first image tiler that takes an image (or a mosaic of images) and serves them up to users on the web via a Swagger api.

It can be hosted on a regular server, and also as a serverless AWS Lambda function or as a Docker container on an Elastic Container Service.

It is also able to apply stretches, color tables, calculate statistics and compute band math functions on the fly.

For this to work, the image must be hosted on a web server, preferably in cloud optimized format.

For this demonstration I'll run this using a local webserver, but in production the image would be on a cloud web server or in a cloud bucket (like S3 etc).

If you are running this yourself, the server will run at `localhost` on your local machine.

Links - Only active while webserver is running - I will route geom3001.space to this machine.

- http://geom3001.space
- http://geom3001.space/cog/viewer
- https://tinyurl.com/best-image
- https://tinyurl.com/grass-mass

In [ ]:
```
# Run TiTiler to serve a web map
# The API documentation is then available at http://127.0.0.1:8000/docs
# A COG viewer is available at http://127.0.0.1:8000/cog/viewer
!uvicorn titiler.application.main:app
```

Call the XYZ tile server and add them to our webmap

The tiler API exposes lots of endpoints that you can see by visiting the servers /api.html page

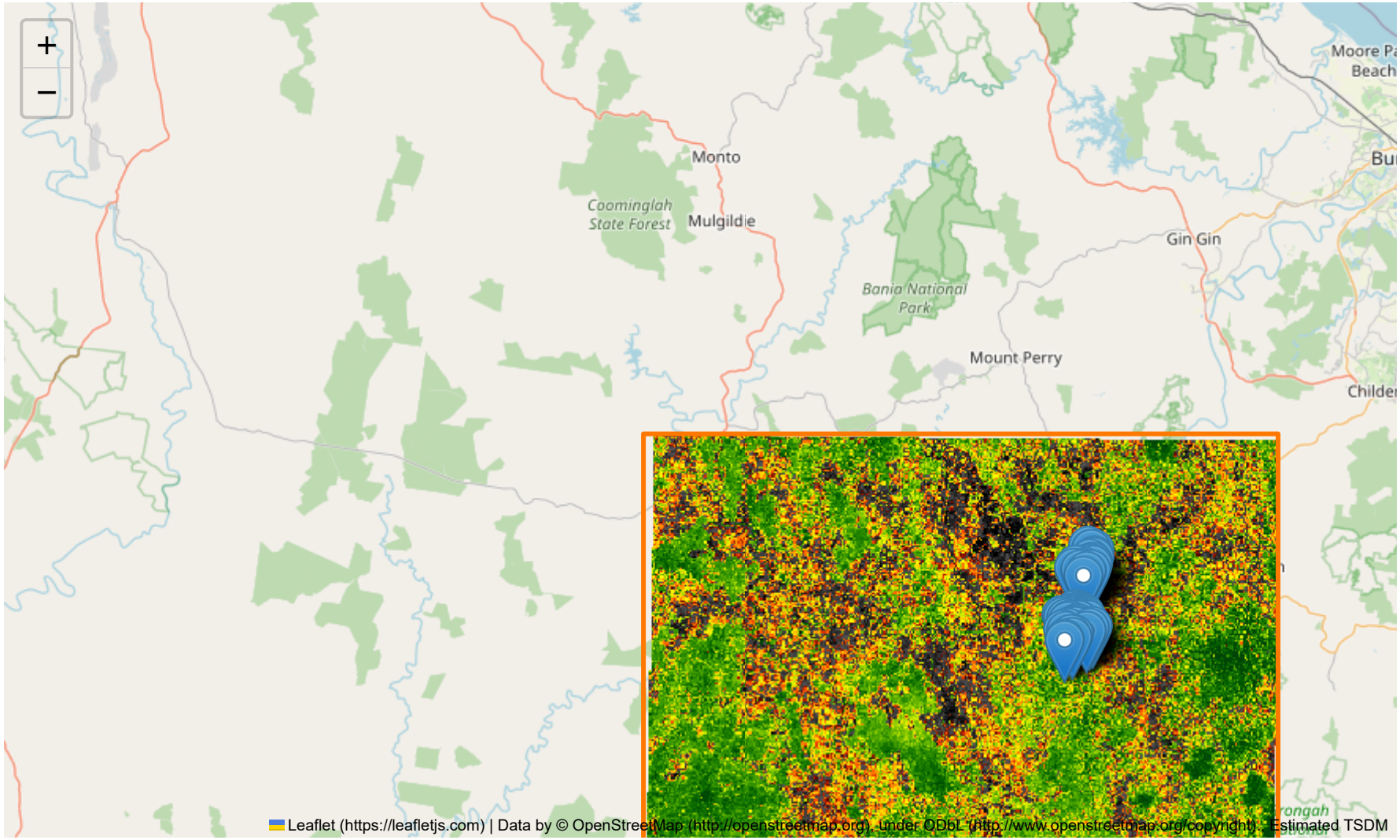
Here we just use the standard XYZ tile server method, as used by Google Maps, Open Streetmap etc and add it as an additional tile layer to our map.

In [37]:

```
# Plot the dataframe using folium - first make the map with some defaults
map = folium.Map(
    location = [fieldData.points_center_lat.mean(),fieldData.points_center_lng.mean()],
     tiles = "OpenStreetMap", zoom_start = 9)

##########################################################################################################
# Add the field points to the map
folium.features.GeoJson(
    fieldData[['geometry','tsdm']].to_json(),
     tooltip=folium.GeoJsonTooltip(fields=['tsdm'])
     ).add_to(map)

##########################################################################################################
# Get the bounding box from the STAC item
bbox = bestItem.bbox

# Create a rectangle corresponding to the best image
rectangle = folium.vector_layers.Rectangle(
    bounds=[[bbox[1], bbox[0]], [bbox[3], bbox[2]]],
    color="#ff7800",
    fill=False
).add_to(map)

##########################################################################################################
# The xyz tile layer from titiler is below.
# The documentation is at http://geom3001.space/api.html
# Note the "Stretch" between 500 and 2500 and the color map "rplumbo"
tileset = 'http://geom3001.space/cog/tiles/WebMercatorQuad/{z}/{x}/{y}?url=S2B_56JLS_20200524_0_L2A_tsdm.tif&rescale=500%2C250

# Add the tile layer to the map
tile_layer = folium.TileLayer(
    tiles= tileset,
    attr="Estimated TSDM"
)

tile_layer.add_to(map)

##########################################################################################################
```

```
# Show the map
map
```

Out[37]:



Summary

This cloud workflow example is only touching the surface of what is possible, but the take home message is that being able to understand the opportunities, and/or be able to work programmatically in cloud based systems, is a very useful (*and employable*) skill to acquire.

Processing massive amounts of satellite data is now available to anyone, through frameworks like the Google Earth Engine, Open Data Cube, Sentinel Hub and Microsoft Planetary Computer. The process outlined in this notebook could be easily undertaken in any of these.

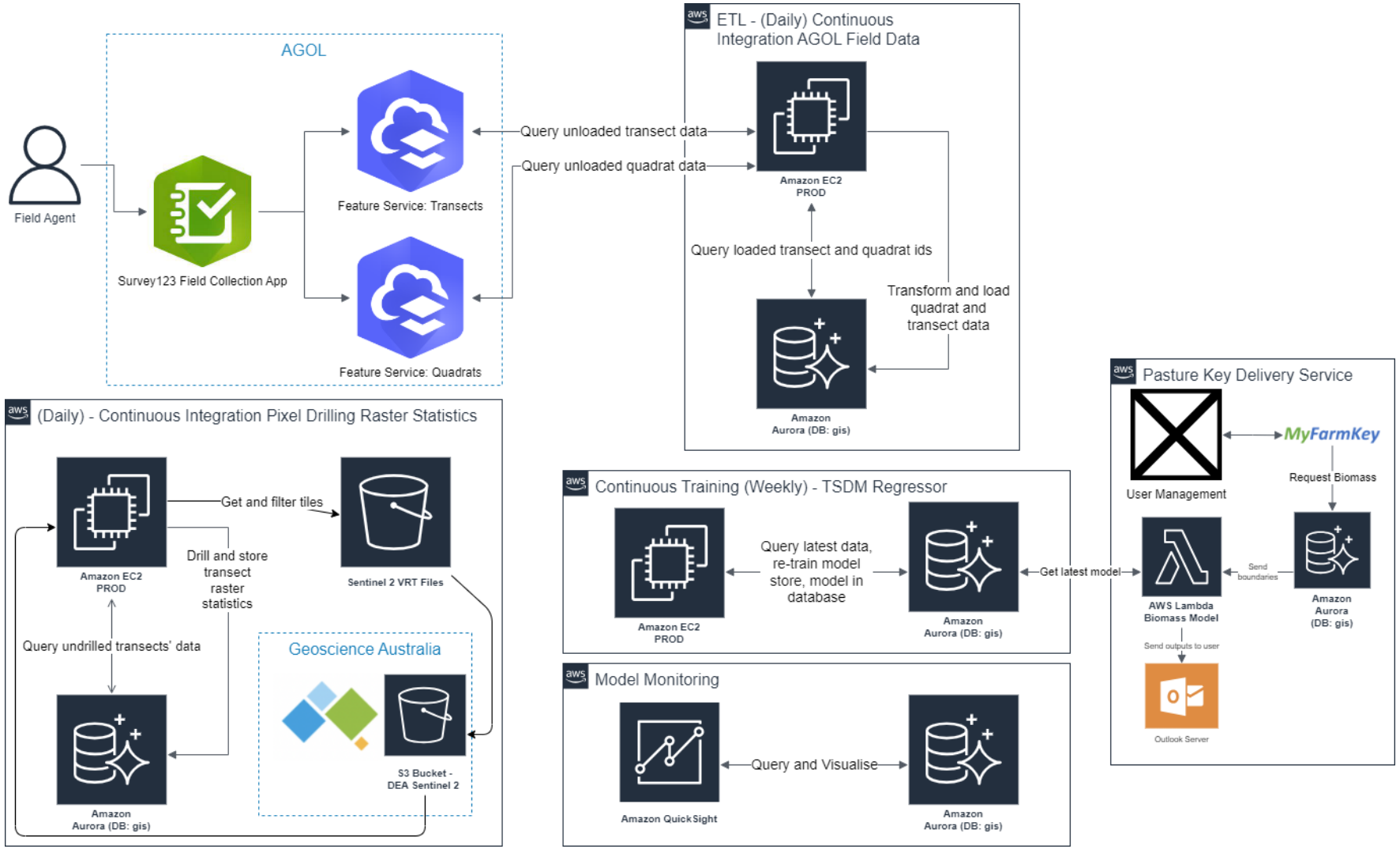
The reasons you might want to build your own cloud processing workflow could be to:

- reduce costs,
- utilize a technology stack not available elsewhere,
- utilize data formats not available elsewhere,
- integrate datasets not available natively on a chosen platform; or
- to just have greater certainty around platform changes or deprecations.

For Cibo Labs, the critical cloud technologies used are based on a more traditional HPC computing stack rather then tools like docker, kubernetes and AWS Lambda (although we also leverage these in production where appropriate). In particular we make heavy use of:

- AWS Batch to run Gravitron (ARM) instances using spot pricing to minimize costs when processing machine learning models on thousands of Sentinel 2 tiles or running zonal statistics across millions of polygons, currently costing around USD4600/month.
- Aurora Serverless PostGIS databases which give us a fully scalable spatial database to store, aggregate, cluster and deliver billions of precomputed statistics against various spatial aggregations (like Cadastre, NRM regions, Postcodes etc) currently costing around USD 1600/month.

A cloud solution will typically take a number of these technologies and connect them to make a production workflow. As a final example, our full pasture prediction process is captured in this diagram:


AGOL
Field Agent
Survey123 Field Collection App
Feature Service: Transects
Feature Service: Quadrats
Query unloaded transect data
Query unloaded quadrat data
ETL - (Daily) Continuous Integration AGOL Field Data
Amazon EC2 PROD
Query loaded transect and quadrat ids
Transform and load quadrat and transect data
Amazon Aurora (DB: gis)
(Daily) - Continuous Integration Pixel Drilling Raster Statistics
Amazon EC2 PROD
Get and filter tiles
Sentinel 2 VRT Files
Drill and store transect raster statistics
Query undrilled transects' data
Geoscience Australia
S3 Bucket - DEA Sentinel 2
Amazon Aurora (DB: gis)
Continuous Training (Weekly) - TSDM Regressor
Amazon EC2 PROD
Query latest data, re-train model store, model in database
Amazon Aurora (DB: gis)
Model Monitoring
Amazon QuickSight
Query and Visualise
Amazon Aurora (DB: gis)
Pasture Key Delivery Service
User Management
MyFarmKey
Request Biomass
Get latest model
AWS Lambda Biomass Model
Send boundaries
Amazon Aurora (DB: gis)
Send outputs to user
Outlook Server