MSIN0166 Coursework Group 3

Data Engineering for Environmental Insight: Deforestation in Indonesia

Word Count: 3803

Team Members:

- Alessandra Cerutti, Student ID : 23228473
- Amita Sujith, Student ID : 23153341
- Nefeli Zampeta Marketou, Student ID : 23165493
- Kevin Hayes, Student ID : 23146448
- Rowan Alexander, Student ID : 23226266

Table of Contents

1. Introduction

1.1 Business Problem

Over the last few decades, Indonesia's forest cover has experienced significantly high deforestation rates, with only 64% of its original forest remaining (DGB Group, 2024). Deforestation in Indonesia has led to greenhouse emissions that account for 6-8% of global emissions (RAN, 2024). As carbon emissions increase global temperatures and lead to adverse environmental conditions (Risser, 2024), there is a critical and timely need to monitor and understand the dynamics of deforestation and its effects on climate.

1.2 Objective

This project seeks to develop a scalable data pipeline for extracting and transforming deforestation and weather data, offering insights into their correlation through visualization. We utilise a versatile storage approach for flexibility, revealing patterns between Indonesia's deforestation and climate change. This infrastructure supports the use of advanced analysis, machine learning, and data monitoring for environmental studies.

In [ ]:
```
### ATTENTION: DO NOT TO ATTEMPT TO RUN THIS NOTEBOOK IF YOU DO NOT HAVE ALL DATA CONNECTED TO I
# Skip this block if you are running on local computer
# connection with google drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

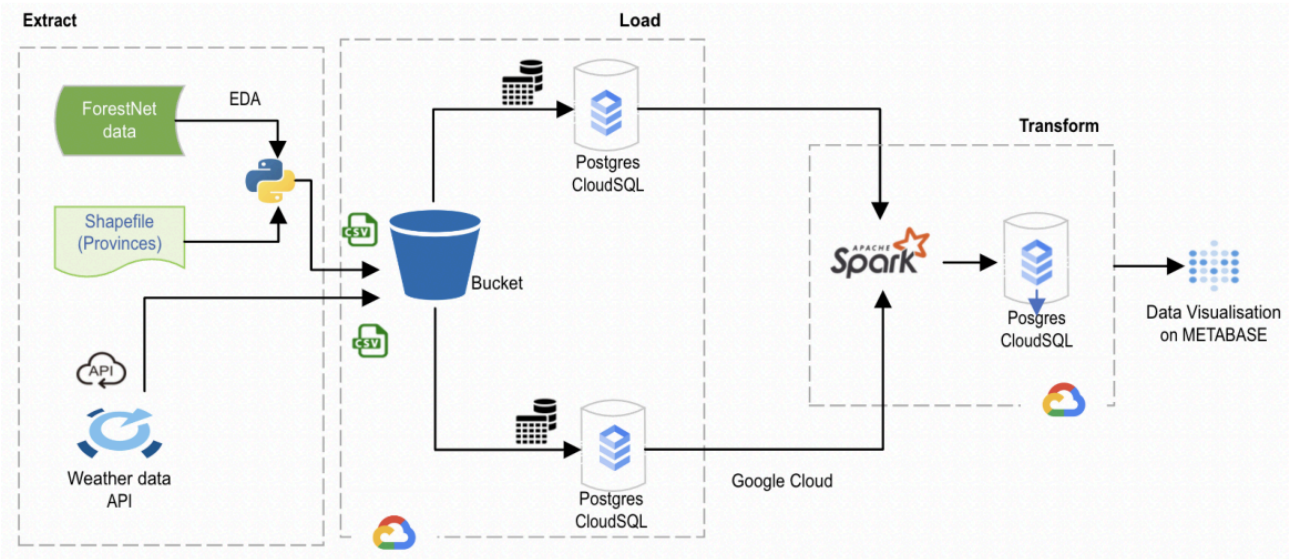
In [ ]:
```
# import libraries and modules needed for the project
from PIL import Image
import pandas as pd   # !pip install pandas pyarrow
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import folium
import geopandas as gpd  # !pip install geopandas
import requests   # !pip install requests
import calendar
import os
from PIL import Image
import matplotlib.pyplot as plt
from google.colab import files
from folium.plugins import MarkerCluster
!pip install adjustText
from adjustText import adjust_text
from shapely.geometry import Point  # merge geodata
```

```
Requirement already satisfied: adjustText in /usr/local/lib/python3.10/dist-packages (1.0.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from adjustText)
(1.25.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from adjust
Text) (3.7.1)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from adjustText)
(1.11.4)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from
matplotlib->adjustText) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matp
lotlib->adjustText) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from
matplotlib->adjustText) (4.49.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from
matplotlib->adjustText) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from m
atplotlib->adjustText) (23.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from mat
plotlib->adjustText) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from
matplotlib->adjustText) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (f
rom matplotlib->adjustText) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-d
ateutil>=2.7->matplotlib->adjustText) (1.16.0)
```

2. Architecture

The diagram illustrates the data pipeline, which streamlines the workflow from data collection to insight generation. The pipeline is divided into four stages: Extract, Load, Transform (ETL Process), and Visualize.

In [ ]:
```
# Open the image file
image = Image.open('/content/drive/MyDrive/ForestNetDataset/Architecture diagram.png')

# Set the figure size to a larger value to make the displayed image bigger
plt.figure(figsize=(15, 10))  # Adjust the width and height as needed

# Display the image
plt.imshow(image)

# Remove the axis for a cleaner look
plt.axis('off')

# Show the plot
plt.show()
```



3. Extract Data

In this section, we extract the required data, including satellite images, the causes of deforestation, and weather data.

3.1 Forest Net Data

The dataset features 2,756 satellite images of deforestation events, each annotated with the cause, such as Plantation or Smallholder Agriculture, based on expert analysis of Global Forest Change maps and high-resolution Google Earth imagery. Landsat 8 satellite images of these events were captured using a cloud-minimizing technique, processed via Descartes Labs. (Irvin et al., 2020)

To facilitate merging weather and deforestation data, the next steps highlight how we create a shared 'province' column in both datasets. The upcoming steps will demonstrate how we utilize province polygons stored in a shapefile to assign a province to each deforestation entry based on their longitude and latitude coordinates.

In [ ]:
```
# Initially, the individual data files comprising the ForestNet dataset are imported.
# import dataframes /content/drive/MyDrive/ForestNetDataset
path_train = '/content/drive/MyDrive/ForestNetDataset/data_forest_tabular/train.csv'
path_test = '/content/drive/MyDrive/ForestNetDataset/data_forest_tabular/test.csv'
path_val = '/content/drive/MyDrive/ForestNetDataset/data_forest_tabular/val.csv'

# Read the files
df_train = pd.read_csv(path_train)
df_test = pd.read_csv(path_test)
df_val = pd.read_csv(path_val)

# add a column to each dataframe so that we know their origin
df_train['origin'] = 'train'
df_test['origin'] = 'test'
df_val['origin'] = 'val'
```

In [ ]:
```
# for better usability merge all dataframes together
df = pd.concat([df_train, df_test, df_val], ignore_index=True)
```

In [ ]:
```
# Displaying an example of how the data looks
df.head()
```

Out[ ]:

	label	merged_label	latitude	longitude	year	example_path
0	Timber plantation	Plantation	4.430849	96.101634	2014	examples/4.430849118860583_96.1016343478138
1	Other	Other	1.332341	109.374229	2012	examples/1.3323406178609702_109.37422873130464
2	Grassland shrubland	Grassland shrubland	-1.720266	115.006996	2016	examples/-1.720266384577504_115.00699582064485
3	Small-scale agriculture	Smallholder agriculture	-2.248346	104.135786	2011	examples/-2.248346072674411_104.1357857482906
4	Other large-scale plantations	Plantation	-2.100800	113.022376	2008	examples/-2.100800102991412_113.0223763234016

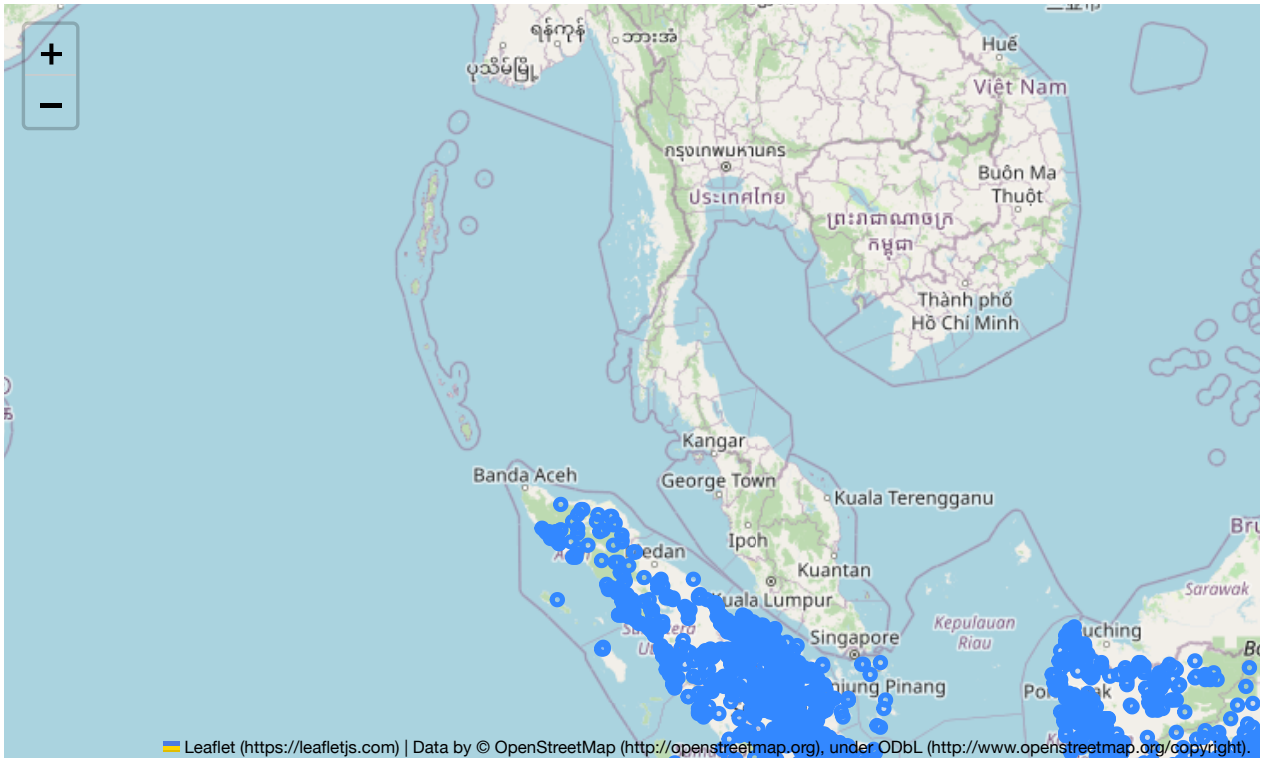
The dataset provides a framework for analysing deforestation in Indonesia, with each entry pinpointing a location via latitude and longitude, the year of deforestation, and the specific cause of deforestation (`label`). The `merged_label` simplifies these into broader categories, while 'example_path' connects each entry to its satellite image file, organised by coordinates and time, requiring extra steps to access due to its non-relational file structure.

The map below depicts the geo-coordinates for which we have the relevant deforestation data for :

In [ ]:
```
# map centered around the approximate center of Indonesia
map = folium.Map(location=[-2.2151, 115.6628], zoom_start=5)

# Add points to the map
```

```
for idx, row in df.iterrows():
    folium.CircleMarker(location=[row['latitude'], row['longitude']],
                        radius=3,
                        fill=True).add_to(map)

# Display the map
map
```

Out[ ]:



This map displays numerous blue dots spread across the Indonesian archipelago, each representing a specific location of forest loss contained in the dataset. The data is concentrated on the islands of Sumatra, Borneo, and Papua, presumably indicating higher deforestation activities in these regions.

3.2 Regions Data

The regions dataset, sourced from MIT GeoData (2015), details geographical areas with attributes like coordinates and provincial boundaries to enable the analysis of deforestation by matching locations with provinces.

In [ ]:
```
# Load the regions dataset as a GeoDataFrame
regions = gpd.read_file('/content/drive/MyDrive/ForestNetDataset/data_provinces/BATAS_PROVINSI_D
```

In [ ]:
```
# Convert the DataFrame to a GeoDataFrame
# to be able to then match the coordinates with the regions
gdf = gpd.GeoDataFrame( # converting the df to geodf
    df, geometry = [Point(xy) for xy in zip(df.longitude, df.latitude)])
```

In [ ]:
```
gdf.head()
```

Out[ ]:

	label	merged_label	latitude	longitude	year	example_path
0	Timber plantation	Plantation	4.430849	96.101634	2014	examples/4.430849118860583_96.1016343478138
1	Other	Other	1.332341	109.374229	2012	examples/1.3323406178609702_109.37422873130464
2	Grassland shrubland	Grassland shrubland	-1.720266	115.006996	2016	examples/-1.720266384577504_115.00699582064485
3	Small-scale agriculture	Smallholder agriculture	-2.248346	104.135786	2011	examples/-2.248346072674411_104.1357857482906
4	Other large-scale plantations	Plantation	-2.100800	113.022376	2008	examples/-2.100800102991412_113.0223763234016

In [ ]:
```
# Display the initial number of rows in the regions dataset
num_rows = regions.shape[0]
print(f"The number of Provinces is: {num_rows}")
```

```
The number of Provinces is: 34
```

In [ ]:
```
# Perform the spatial join to match deforestation points with their corresponding provinces
gdf = gpd.sjoin(gdf, regions, how="left", op='within')
```

```
/usr/local/lib/python3.10/dist-packages/IPython/core/interactiveshell.py:3473: FutureWarning: The
`op` parameter is deprecated and will be removed in a future release. Please use the `predicate`
parameter instead.
  if (await self.run_code(code, result,  async_=asy)):
<ipython-input-16-316c481b0557>:2: UserWarning: CRS mismatch between the CRS of left geometries a
nd the CRS of right geometries.
Use `to_crs()` to reproject one of the input geometries to match the CRS of the other.

Left CRS: None
Right CRS: EPSG:4326

  gdf = gpd.sjoin(gdf, regions, how="left", op='within')
```

In [ ]:
```
# Rename the 'PROVINSI' column to 'province' for clarity
gdf = gdf.rename(columns={'PROVINSI': 'province'})
gdf.head()
```

Out[ ]:

	label	merged_label	latitude	longitude	year	example_path
0	Timber plantation	Plantation	4.430849	96.101634	2014	examples/4.430849118860583_96.1016343478138
1	Other	Other	1.332341	109.374229	2012	examples/1.3323406178609702_109.37422873130464
2	Grassland shrubland	Grassland shrubland	-1.720266	115.006996	2016	examples/-1.720266384577504_115.00699582064485
3	Small-scale agriculture	Smallholder agriculture	-2.248346	104.135786	2011	examples/-2.248346072674411_104.1357857482906
4	Other large-scale plantations	Plantation	-2.100800	113.022376	2008	examples/-2.100800102991412_113.0223763234016

In [ ]:
```
# Display the initial number of rows after the join
num_rows = gdf.shape[0]
print(f"After the join, the number of rows is: {num_rows}")
```

```
After the join, the number of rows is: 2757
```

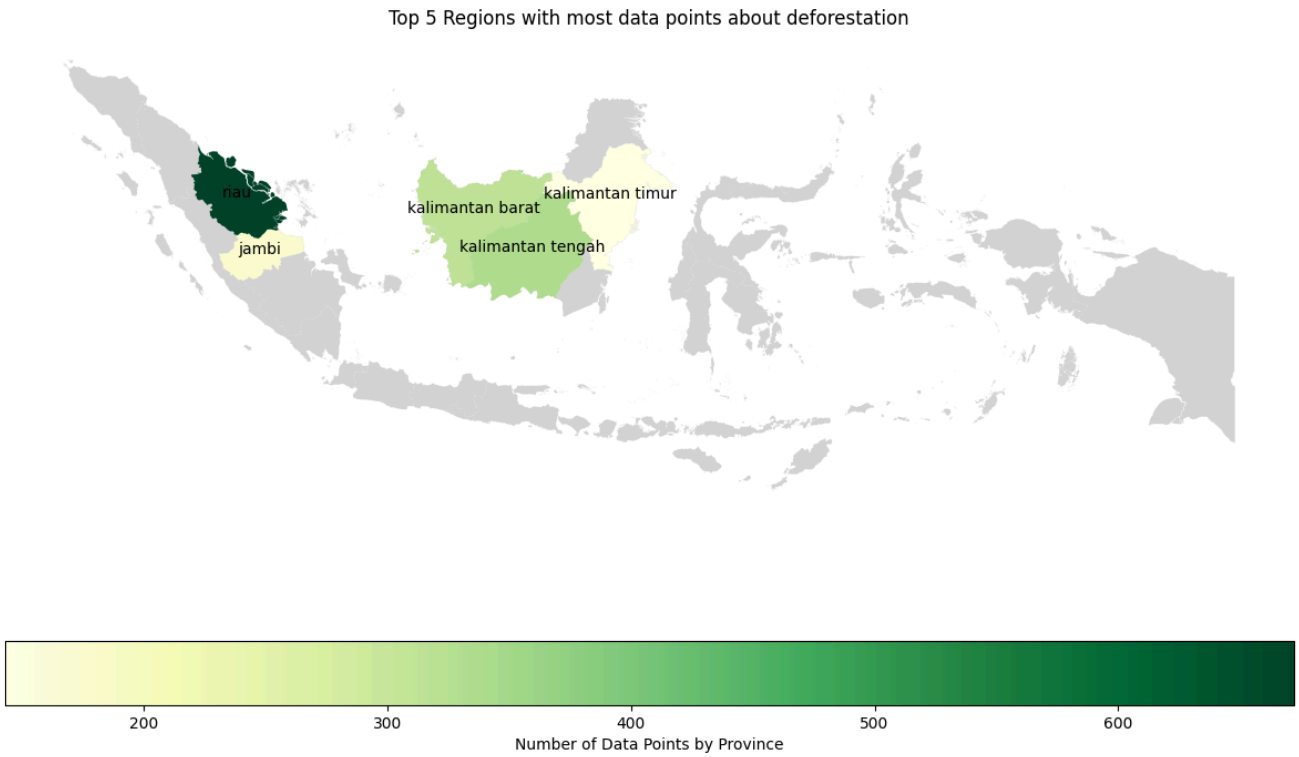
In [ ]:
```
# Drop columns not needed
gdf.drop(['index_right', 'OBJECTID', 'Shape_Leng', 'Shape_Area', 'geometry'], axis=1, inplace=Tr
```

In [ ]:
```
# Make all the regions name lower case for clarirty
gdf['province'] = gdf['province'].str.lower()
gdf.head()
```

Out[ ]:

	label	merged_label	latitude	longitude	year	example_path
0	Timber plantation	Plantation	4.430849	96.101634	2014	examples/4.430849118860583_96.1016343478138
1	Other	Other	1.332341	109.374229	2012	examples/1.3323406178609702_109.37422873130464
2	Grassland shrubland	Grassland shrubland	-1.720266	115.006996	2016	examples/-1.720266384577504_115.00699582064485
3	Small-scale agriculture	Smallholder agriculture	-2.248346	104.135786	2011	examples/-2.248346072674411_104.1357857482906
4	Other large-scale plantations	Plantation	-2.100800	113.022376	2008	examples/-2.100800102991412_113.0223763234016

In [ ]:
```
# Regions Map
indonesia_map = regions
indonesia_map['province'] = indonesia_map['PROVINSI'].str.lower()

# Aggregate data points by province
province_counts = gdf['province'].value_counts().reset_index()
province_counts.columns = ['province', 'data_point_count']

# Sort and get the top five provinces
top_provinces = province_counts.nlargest(5, 'data_point_count')

# Merge the shapefile with your data points count
map_with_data = indonesia_map.merge(top_provinces, on='province', how='left')

# Plot the map with all provinces
fig, ax = plt.subplots(1, 1, figsize=(15, 10))
indonesia_map.plot(ax=ax, color='lightgrey')  # Plot all provinces in a light grey color

# Now plot the provinces with data on top
map_with_data.plot(column='data_point_count', ax=ax, legend=True,
                   legend_kwds={'label': "Number of Data Points by Province",
                                'orientation': "horizontal"},
                   cmap='YlGn')

# Annotate only the top five provinces
for idx, row in map_with_data.iterrows():
    if pd.notnull(row['data_point_count']) and row['province'] in top_provinces['province'].valu
        plt.annotate(text=row['province'], xy=(row['geometry'].centroid.x, row['geometry'].centr
                     horizontalalignment='center', fontsize=10, color='black')

plt.title("Top 5 Regions with most data points about deforestation", fontdict=None, loc='center'

# Remove the axis for a cleaner look and display the map
plt.axis('off')
plt.show()
```



3.3 Weather Data

For our weather data, we decided to utilise Visual Crossing's API (a low-cost, easy-to-use weather API that allows users to obtain weather data from any GPS coordinate). An API is preferable to an online CSV for our deforestation dashboard because it offers real-time access to updated weather information, which is necessary for having up-to-date data. Given the budget constraints and the project scope, we included only historical weather data.

The variables of interest are average rainfall and temperature, which are necessary for understanding deforestation impacts such as forest fires and droughts. We wanted a broad timeframe to offer insights into long-term climate trends, which is essential for assessing the impact of climate change.

Initially, we registered for Visual Crossing, obtaining an API key for data requests. When crafting the query, we had two options for specifying location: GPS coordinates or address. We opted to provide a GPS coordinate of a central point in each province provided by Geokeo (2024). We recognised that weather there might not fully represent the entire province or deforestation areas. Nevertheless, with 38 provinces in Indonesia, we deemed province-level granularity sufficient to mitigate errors. We iterated through the provinces and years to retrieve data and then saved it as a CSV.

3.3.1 Integrating The API into the pipeline

For seamless integration with Docker and the weather API, we create a standalone Python script to retrieve and save the weather data. Furthermore, the script has been adapted to receive the API key through the terminal, bolstering security while providing greater adaptability. The code below is an example of the script, which is saved as weather_data_processor.py on our Github repository. Please note that the system was originally designed to save the output as .csv and parquet files directly to Google Drive.

In [ ]:
```
# Retrieve API key from environment variables
API_KEY = os.getenv('VISUAL_CROSSING_API_KEY')
# Check if API key is available, if not raise an error
if not API_KEY:
    raise ValueError("No API Key found. Set the VISUAL_CROSSING_API_KEY environment variable.")

# Base URL for Visual Crossing Weather API
BASE_URL = 'https://weather.visualcrossing.com/VisualCrossingWebServices/rest/services/timeline/

# Define the range of years for which to fetch weather data
start_year = 2002
end_year = 2016
```

```python
# Initialize an empty list to store the data
data = []

# Define the list of provinces with their names and geographic coordinates
provinces = [
    {"name": "Riau", "latitude": 0.5004112, "longitude": 101.5475811},
    {"name": "Central Kalimantan", "latitude": -1.499583, "longitude": 113.2903307},
    {"name": "West Kalimantan", "latitude": -0.1322387, "longitude": 111.0968901},
    {"name": "Jambi", "latitude": -1.611572, "longitude": 102.779699},
    {"name": "East Kalimantan", "latitude": 0.7884397, "longitude": 116.2419977},
    {"name": "North Sumatra", "latitude": 2.1923519, "longitude": 99.3812201},
    {"name": "North Kalimantan", "latitude": 3.0235817, "longitude": 116.2049306},
    {"name": "Papua", "latitude": -3.9885728, "longitude": 138.3485219},
    {"name": "Central Sulawesi", "latitude": -1.6937786, "longitude": 120.8088555},
    {"name": "West Sumatra", "latitude": -0.5827529, "longitude": 100.6133379},
    {"name": "Aceh", "latitude": 4.3685491, "longitude": 97.0253024},
    {"name": "Southeast Sulawesi", "latitude": -3.5491199, "longitude": 121.7279646},
    {"name": "South Sumatra", "latitude": -3.1266842, "longitude": 104.0930554},
    {"name": "West Papua", "latitude": -1.3842356, "longitude": 132.902528},
    {"name": "North Maluku", "latitude": 0.6301215, "longitude": 127.9720219},
    {"name": "Bangka Belitung Islands", "latitude": -2.7410513, "longitude": 106.4405872},
    {"name": "South Kalimantan", "latitude": -2.9285686, "longitude": 115.3700718},
    {"name": "West Sulawesi", "latitude": -2.4974546, "longitude": 119.3918955},
    {"name": "South Sulawesi", "latitude": -3.6446718, "longitude": 119.9471906},
    {"name": "Bengkulu", "latitude": -3.5186763, "longitude": 102.5359834},
    {"name": "Maluku", "latitude": -3.118837, "longitude": 129.4207759},
    {"name": "Gorontalo", "latitude": 0.7186174, "longitude": 122.4555927},
    {"name": "North Sulawesi", "latitude": 0.6555692, "longitude": 124.090151},
    {"name": "Lampung", "latitude": -4.8555039, "longitude": 105.0272986},
    {"name": "East Java", "latitude": -7.6977397, "longitude": 112.4914199},
    {"name": "Riau Islands", "latitude": -0.1547846, "longitude": 104.5803745},
    {"name": "West Java", "latitude": -6.8891904, "longitude": 107.6404716},
    {"name": "Banten", "latitude": -6.4453801, "longitude": 106.1375586},
    {"name": "West Nusa Tenggara", "latitude": -8.6529334, "longitude": 117.3616476},
]

# Loop over each province and year to fetch weather data
for province in provinces:
    for year in range(start_year, end_year + 1):
        start_date = f"{year}-01-01"
        end_date = f"{year}-12-31"

        # Create the location string from the latitude and longitude
        location = f"{province['latitude']},{province['longitude']}"
        # Construct the URL for the API request
        url = f"{BASE_URL}{location}/{start_date}/{end_date}?unitGroup=metric&key={API_KEY}&incl

        # Make the API request
        response = requests.get(url)
        # Check if the request was successful
        if response.status_code == 200:
            yearly_data = response.json()
            # Calculate the total rainfall for the year
            total_rainfall = sum(day.get('precip', 0) for day in yearly_data.get('days', []) if

            # Calculate the average temperature for the year
            valid_temps = [day.get('temp') for day in yearly_data.get('days', []) if day.get('te
            if valid_temps:  # Check if the list is not empty
                avg_temp = sum(valid_temps) / len(valid_temps)
            else:
                avg_temp = None  # Placeholder value indicating no data

            # Append the results to the data list
            data.append({
                "Province": province['name'],
                "Year": year,
                "Average Temperature (°C)": avg_temp,
                "Total Rainfall (mm)": total_rainfall,
            })
            print(f"Processed {province['name']} for {year}")
        else:
            print(f"Failed to retrieve data for {province['name']} for {year}. Status code: {res
```

```
# Convert the data list into a pandas DataFrame
df = pd.DataFrame(data)

# Group by province and year, then calculate mean temperature and rainfall
grouped = df.groupby(['Province', 'Year']).agg({
    'Average Temperature (°C)': 'mean',
    'Total Rainfall (mm)': 'mean'
}).reset_index()

# Pivot the table for average temperature
avg_temp_pivot = grouped.pivot(index='Province', columns='Year', values='Average Temperature (°C
avg_temp_pivot.columns = [f'{col} Average Temp' for col in avg_temp_pivot.columns]

# Pivot the table for total rainfall
avg_rain_pivot = grouped.pivot(index='Province', columns='Year', values='Total Rainfall (mm)')
avg_rain_pivot.columns = [f'{col} Average Rain' for col in avg_rain_pivot.columns]

# Join the pivoted tables
result = avg_temp_pivot.join(avg_rain_pivot).reset_index()

# Define the CSV file path where the result will be saved
csv_file_path = "data_weather/historical_weather_data_annual.csv"
# Save the result to a CSV file
result.to_csv(csv_file_path, index=False)

# Print completion message with the CSV file path
print(f"Data saved to {csv_file_path}")
```

In [ ]:
```
#Change to parquet file
uploaded = files.upload()

csv_file = 'historical_weather_data_annual.csv'
df = pd.read_csv(csv_file)

parquet_file = 'historical_weather_data_annual.parquet'
df.to_parquet(parquet_file, index=False)

files.download(parquet_file)
```

3.3.2 Docker integration

Docker streamlines the creation, deployment, and running of applications by enclosing them in containers with their dependencies, guaranteeing consistent functionality on diverse Linux systems and facilitating easier deployments. To manage project dependencies, a requirements.txt file enumerates all necessary Python packages, simplifying installation during environment setup. Additionally, a Dockerfile has been meticulously prepared, providing step-by-step instructions to construct the Docker image, which includes selecting a base image, designating the working directory, transferring essential files, installing dependencies specified in requirements.txt, and defining the execution command for the script.

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Run weather_data_processor.py when the container launches
CMD ["python", "./weather_data_processor.py"]
```

Commands were executed in the Terminal to build the Docker image and run the container. The process of building the image utilised a Docker command referencing the Dockerfile, while running the container involved setting the API key through an environment variable and mounting a volume for the CSV output.

```
(base) k@KMB ~ % cd /Users/k/Documents/de/DockerApiProj

(base) k@KMB DockerApiProj % docker build -t weather-data-processor .

[+] Building 20.7s (10/10) FINISHED                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                          0.0s
 => => transferring dockerfile: 1.03kB                                        0.0s
 => [internal] load metadata for docker.io/library/python:3.8-slim            4.2s
 => [auth] library/python:pull token for registry-1.docker.io                 0.0s
 => [internal] load .dockerignore                                             0.0s
 => => transferring context: 2B                                               0.0s
 => [1/4] FROM docker.io/library/python:3.8-slim@sha256:23252009f10b4af8a     4.7s
 => => resolve docker.io/library/python:3.8-slim@sha256:23252009f10b4af8a     0.0s
 => => sha256:b7885ba641278df6e419c1c888a8981d86aeb940c 13.74MB / 13.74MB     3.3s
 => => sha256:23252009f10b4af8a8c90409c54a866473a251b001b 1.86kB / 1.86kB     0.0s
 => => sha256:515463c9b88285509388c9d127a8a56cf9602cd56ce 1.37kB / 1.37kB     0.0s
 => => sha256:054df0836f5230590646d8b7c62c54009a80c5c95c0 6.98kB / 6.98kB     0.0s
 => => sha256:f546e941f15b76df3d982d56985432b05bc065e39 29.16MB / 29.16MB     3.3s
 => => sha256:24935aba99a712c5a6efc86118762b0f914e1577a2c 3.32MB / 3.32MB     2.4s
 => => sha256:ab172a9f49e99303f4e3f13978812bad85a6ccb74545e95 244B / 244B     2.7s
 => => sha256:f5995b23a281d55dc1aada4458690373f1fe4cf0ac1 3.13MB / 3.13MB     3.6s
 => => extracting sha256:f546e941f15b76df3d982d56985432b05bc065e3923fb35b     0.8s
 => => extracting sha256:24935aba99a712c5a6efc86118762b0f914e1577a2c072db     0.1s
 => => extracting sha256:b7885ba641278df6e419c1c888a8981d86aeb940cc1e17c9     0.3s
 => => extracting sha256:ab172a9f49e99303f4e3f13978812bad85a6ccb74545e95f     0.0s
 => => extracting sha256:f5995b23a281d55dc1aada4458690373f1fe4cf0ac1d711a     0.1s
 => [internal] load build context                                             0.0s
 => => transferring context: 7.22kB                                           0.0s
 => [2/4] WORKDIR /app                                                        0.1s
 => [3/4] COPY . /app                                                         0.0s
 => [4/4] RUN pip install --no-cache-dir -r requirements.txt                 11.2s
 => exporting to image                                                        0.3s
 => => exporting layers                                                       0.3s
 => => writing image sha256:38ad81e5610ff1562de737319e62278b3414100d0704d     0.0s
 => => naming to docker.io/library/weather-data-processor                     0.0s
```

Here we set the working directory and build a docker image according to the instructions from the Dockerfile created.

```
(base) k@KMB DockerApiProj % docker run -e VISUAL_CROSSING_API_KEY=[redacted] -v /Users/k/Documents/de/DockerApiProj/data_weather:/app/data_weat
her weather-data-processor

Processed Riau for 2010
Processed Riau for 2011
Processed Central Kalimantan for 2010
Processed Central Kalimantan for 2011
Processed West Kalimantan for 2010
Processed West Kalimantan for 2011
Processed Jambi for 2010
Processed Jambi for 2011
Processed East Kalimantan for 2010
Processed East Kalimantan for 2011
Processed Banten for 2011
Processed West Nusa Tenggara for 2010
Processed West Nusa Tenggara for 2011
Data saved to data_weather/historical_weather_data_annual.csv
(base) k@KMB DockerApiProj %
```

The next step involved running the python script we saved, by inputting the API key into the terminal. Here we begin to receive data. Finally, after we finish receiving the data,we save the dockerised csv file locally.

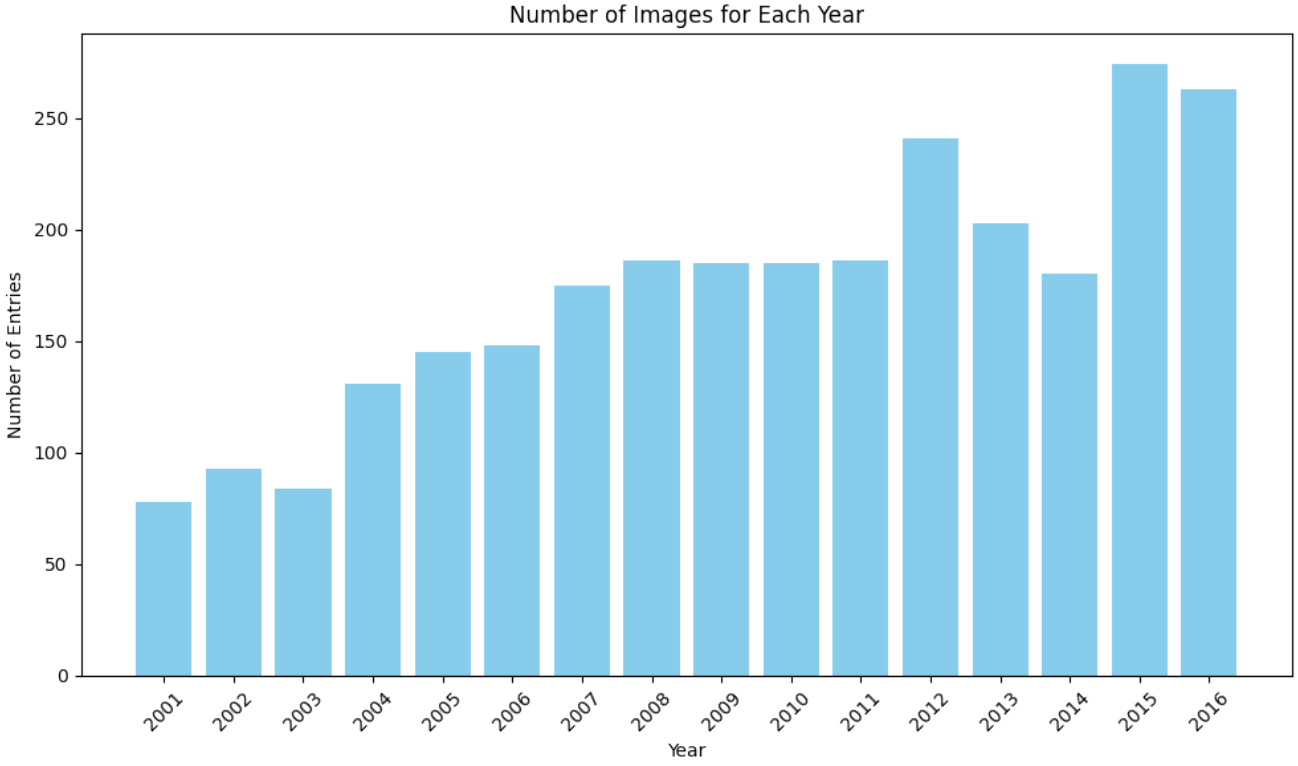
4. Exploratory Data Analysis

4.1 EDA on ForestNet

Exploratory Data Analysis (EDA) of the forest dataset can reveal key spatial and temporal trends, aiding in the assessment of the deforestation causes.

The early EDA focuses on revealing the connections between provinces, time, and deforestation causes

In [ ]:
```
# Extract the different years from the dataset
unique_years = df['year'].unique()

# Put the years in ascending order
unique_years.sort()

# Count the number of entries for each year
entry_counts = [len(df[df['year'] == year]) for year in unique_years]

# Plotting
plt.figure(figsize=(10, 6), facecolor = 'none', edgecolor='none')
plt.bar(unique_years, entry_counts, color='skyblue')
plt.xlabel('Year')
plt.ylabel('Number of Entries')
plt.title('Number of Images for Each Year')
plt.grid(False)
plt.xticks(unique_years, rotation=45)
plt.tight_layout()
plt.show()
```

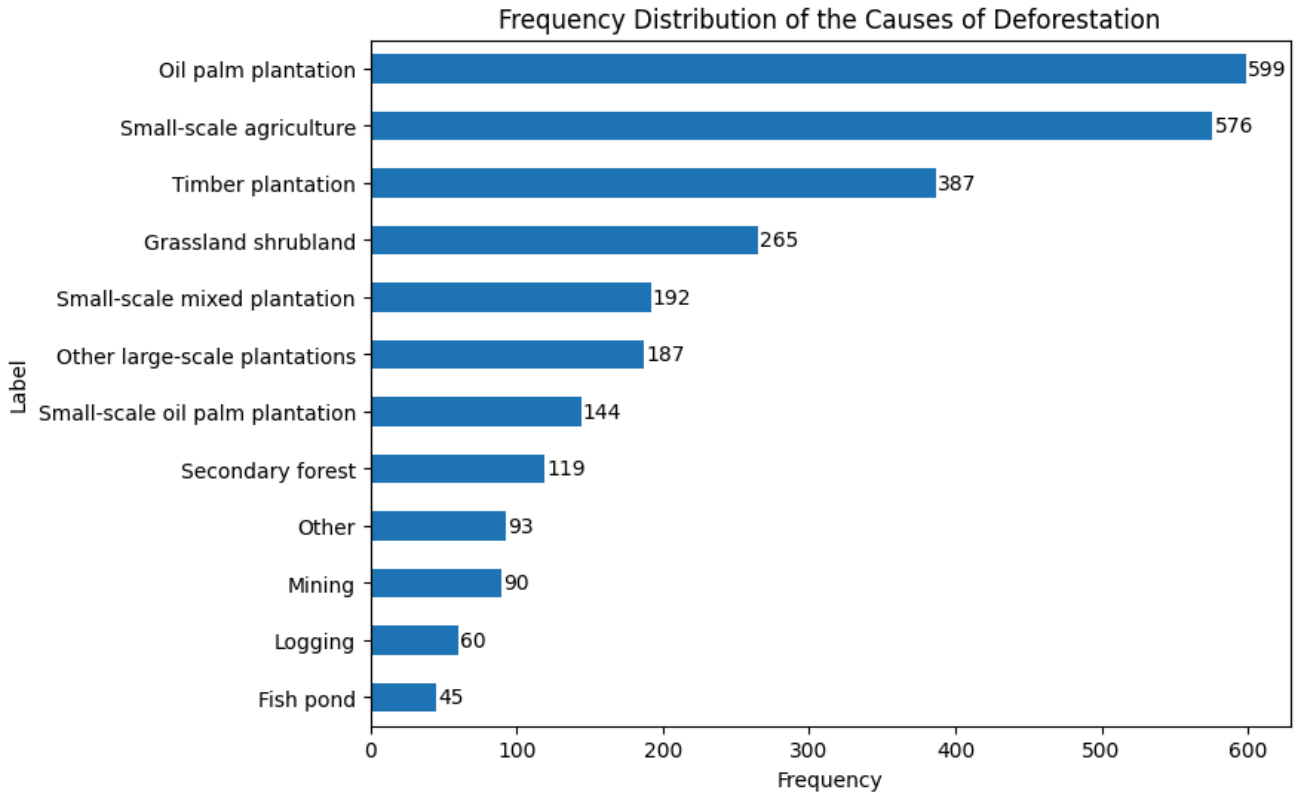


In [ ]:
```
#Calculating the total number of unique data points

# Create a set of tuples where each tuple is a unique location (latitude, longitude)
unique_locations = set(zip(df['latitude'], df['longitude']))

# Convert the set back to a list
unique_locations_list = list(unique_locations)

# Print the first few locations to verify
#print(unique_locations_list[:5])

# Count the number of unique locations on the dataframe
number_of_unique_locations = len(unique_locations)

# Display the number of unique locations
print("Total unique locations:", number_of_unique_locations)
```

```
Total unique locations: 2757
```

The bar chart below shows the distribution of deforestation causes. Oil palm plantations are the leading cause with 599 entries, followed closely by small-scale agriculture at 576, with other factors like timber plantations, grassland shrubland, and mining contributing to a lesser extent. This suggests that agricultural expansion, particularly oil palm cultivation, is a major driver of deforestation.

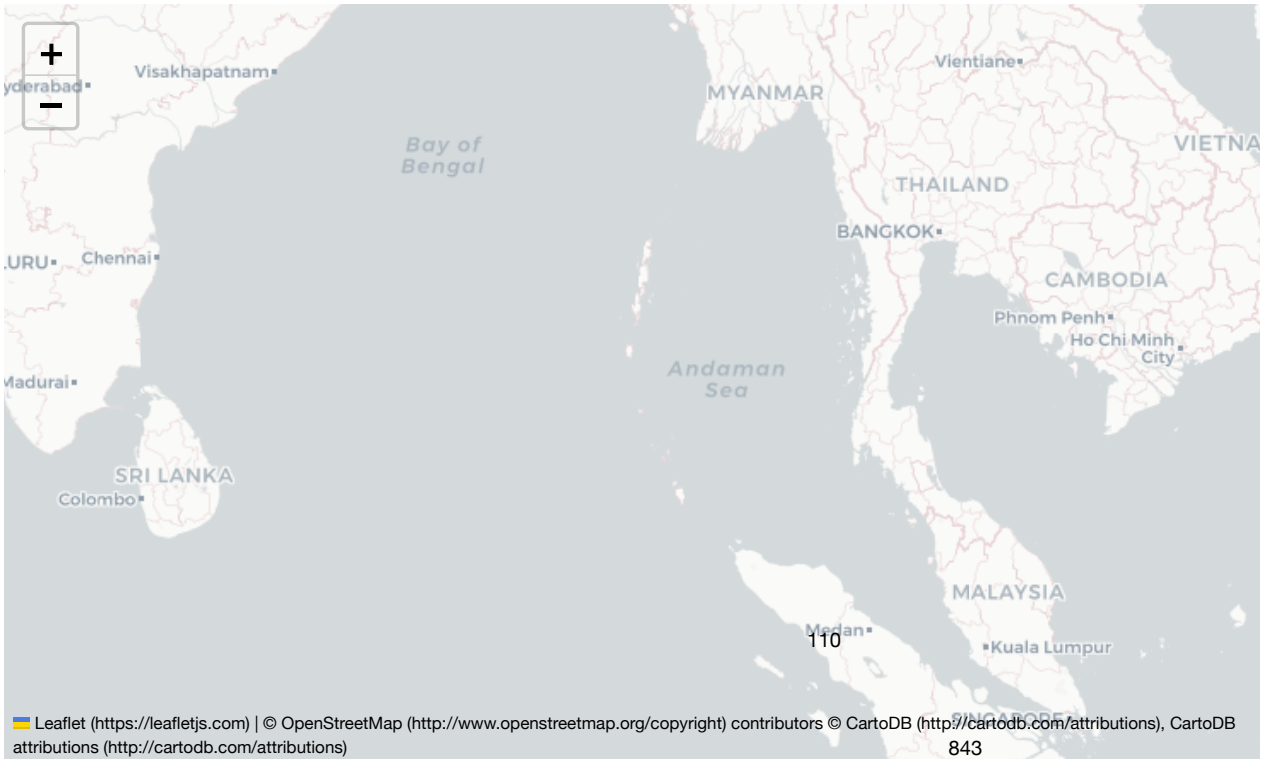
In [ ]:
```
# Calculate the frequency distribution of the 'label' column
label_counts = df['label'].value_counts().sort_values(ascending=True)  # Sort from smallest to l

# Plotting the frequency distribution as a horizontal bar chart
plt.figure(figsize=(8, 6))
ax = label_counts.plot(kind='barh')  # Use 'barh' for horizontal bar chart
plt.title('Frequency Distribution of the Causes of Deforestation')
plt.xlabel('Frequency')
plt.ylabel('Label')
ax.set_facecolor('none')  # Set the axes background to transparent

# Iterate over the bars and add text labels
for i, v in enumerate(label_counts):
    ax.text(v + 1, i, str(v), color='black', va='center')

plt.show()
```



In [ ]:
```
# Causes and locations

# Create a base map
map = folium.Map(location=[df['latitude'].mean(), df['longitude'].mean()], zoom_start=5, tiles='

# MarkerCluster to cluster close markers together
marker_cluster = MarkerCluster().add_to(map)

# Define a fixed list of colors
colors = ['red', 'blue', 'green', 'purple', 'orange', 'darkred', 'lightred', 'beige', 'darkblue'

# Generate a color for each unique 'merged_label'
unique_labels = df['merged_label'].unique()
color_dict = {label: colors[i % len(colors)] for i, label in enumerate(unique_labels)}

# Add markers to the map, ensuring different colors for each 'merged_label'
for _, row in df.iterrows():
    icon_color = color_dict.get(row['merged_label'], 'lightgray')  # Default to 'lightgray' if n
    folium.Marker(
        location=[row['latitude'], row['longitude']],
        popup=f"Cause: {row['merged_label']}",
```

```
        icon=folium.Icon(color=icon_color)
    ).add_to(marker_cluster)

# Display the map
map
```

Out[ ]:



4.2 EDA on Weather Data

EDA on weather data can reveal insights into how average temperature and rainfall vary over time and provinces.

In [ ]:
```
# Read the CSV file into a DataFrame
weather_data = pd.read_csv("/content/drive/My Drive/ForestNetDataset/data_weather/combined_weath

weather_data.head()
```

Out[ ]:

	province	year	average_temp	average_rain
0	Aceh	2001	NaN	0.000
1	Bangka Belitung Islands	2001	26.933688	2757.200
2	Banten	2001	27.213151	2459.296
3	Bengkulu	2001	26.818207	2255.800
4	Central Kalimantan	2001	NaN	0.000

In [ ]:
```
weather_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 464 entries, 0 to 463
Data columns (total 4 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   province      464 non-null    object
 1   year          464 non-null    int64
 2   average_temp  367 non-null    float64
 3   average_rain  464 non-null    float64
dtypes: float64(2), int64(1), object(1)
memory usage: 14.6+ KB
```

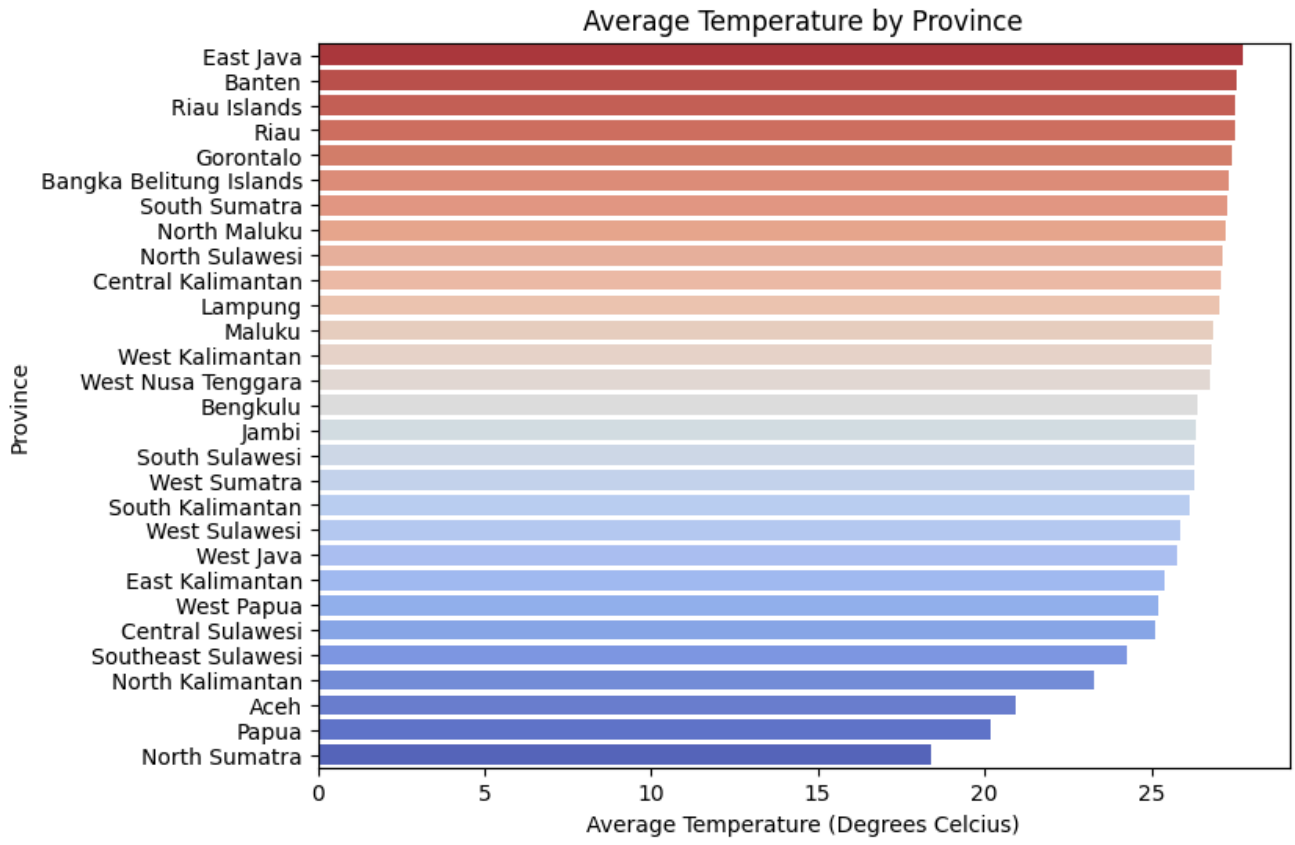
In [ ]:
```
weather_data.nunique()
```

Out[ ]:
```
province         29
year             16
average_temp    367
average_rain    355
dtype: int64
```

There are 29 of Indonesia's provinces in the dataset. Before 2010, there was incomplete average temperature data (as suggested by the lower non-null count).

In [ ]:
```
# Drop rows with missing values for now
weather_data_cleaned = weather_data.dropna()

# Taking the average temperature of each province and sorting
province_grouped_temp = weather_data_cleaned.groupby('province').mean().reset_index().sort_value

plt.figure(figsize=(8, 6))
palette_reversed = sns.color_palette("coolwarm", len(province_grouped_temp))[::-1]
sns.barplot(x='average_temp', y='province', data=province_grouped_temp, palette=palette_reversed
plt.title('Average Temperature by Province')
plt.xlabel('Average Temperature (Degrees Celcius)')
plt.ylabel('Province')
plt.show()
```

```
<ipython-input-33-dcb35ffb2a3e>:9: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign th
e `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x='average_temp', y='province', data=province_grouped_temp, palette=palette_reverse
d)
```

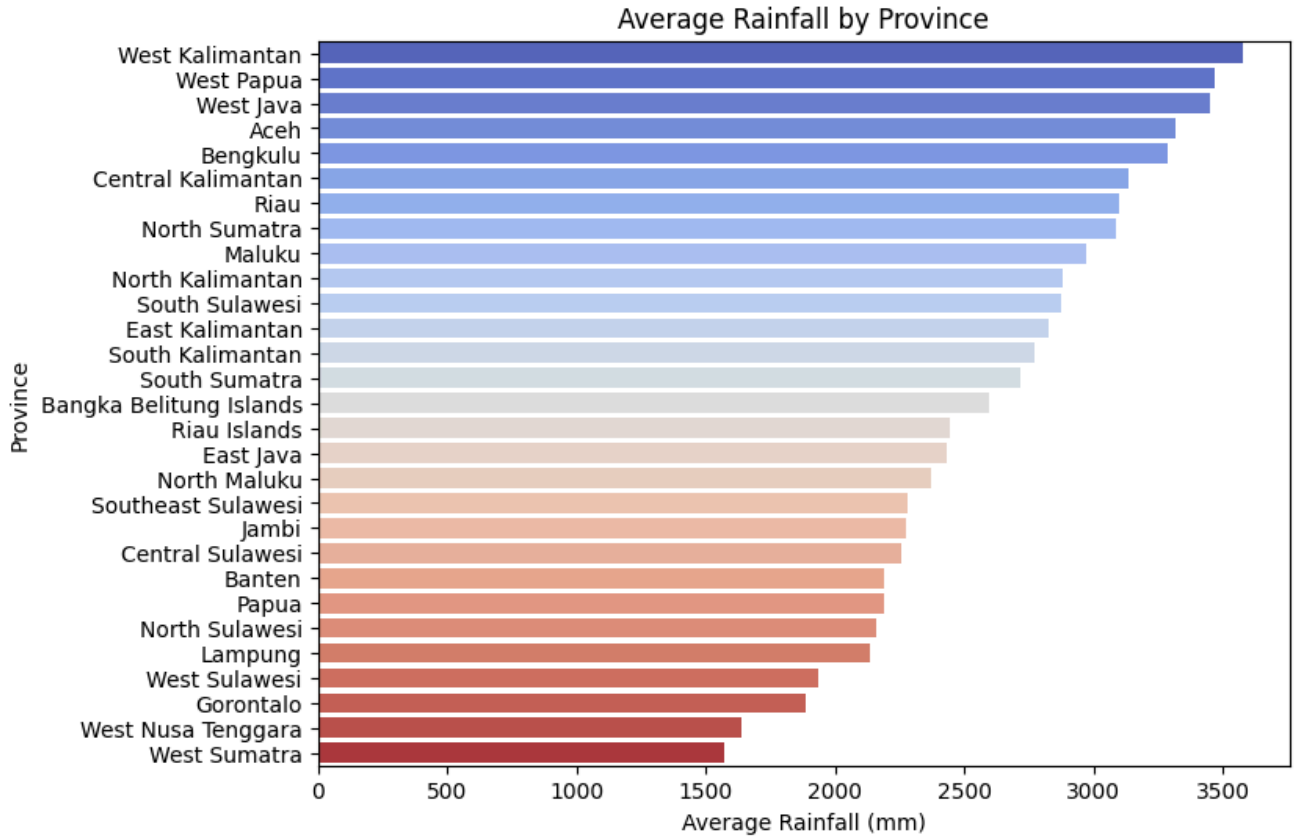


The average temperatures across provinces range from 18.4 degrees in North Sumatra to 27.7 degrees in East Java.

In [ ]:
```
# Taking the average rainfall of each province and sorting
province_grouped_rain = weather_data_cleaned.groupby('province').mean().reset_index().sort_value

plt.figure(figsize=(8, 6))
sns.barplot(x='average_rain', y='province', data=province_grouped_rain, palette='coolwarm')
plt.title('Average Rainfall by Province')
plt.xlabel('Average Rainfall (mm)')
```

```
plt.ylabel('Province')
plt.show()
```

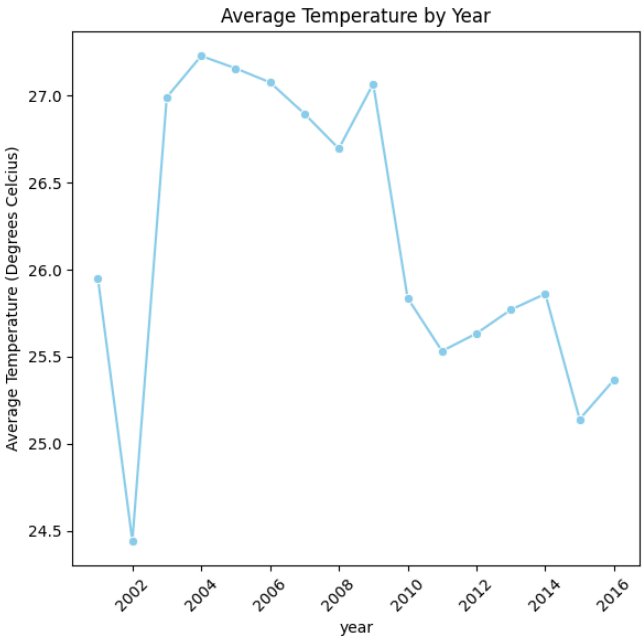
```
<ipython-input-19-da9c73509f8a>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign th
e `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x='average_rain', y='province', data=province_grouped_rain, palette='coolwarm')
```

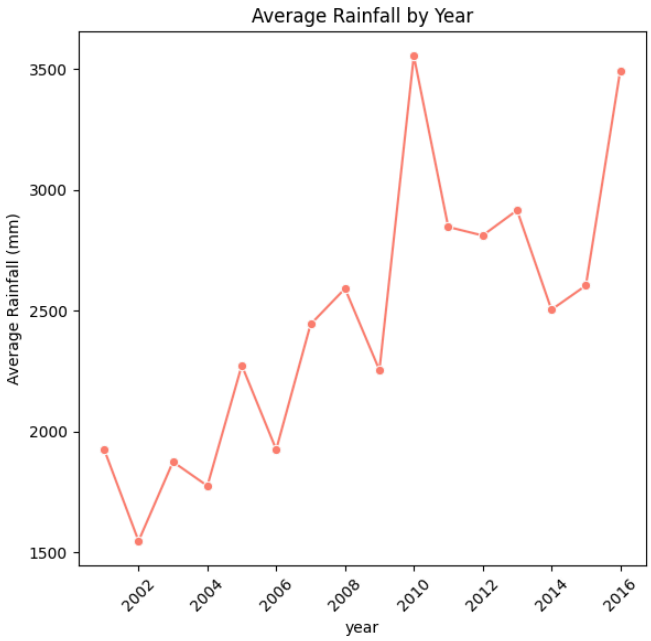


The average rainfall spans from 1574mm in West Sumatra to 3576mm in West Kalimantan.

In [ ]:
```
# Grouping by Year
year_grouped = weather_data_cleaned.groupby('year').mean().reset_index()

# Visualising trends over the years
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.lineplot(x='year', y='average_temp', data=year_grouped, marker='o', color='skyblue')
plt.title('Average Temperature by Year')
plt.xticks(rotation=45)
plt.ylabel('Average Temperature (Degrees Celcius)')

plt.subplot(1, 2, 2)
sns.lineplot(x='year', y='average_rain', data=year_grouped, marker='o', color='salmon')
plt.title('Average Rainfall by Year')
plt.xticks(rotation=45)
plt.yticks(np.arange(1500, max(year_grouped['average_rain'])+100, 500))
plt.ylabel('Average Rainfall (mm)')

plt.tight_layout()
plt.show()
```

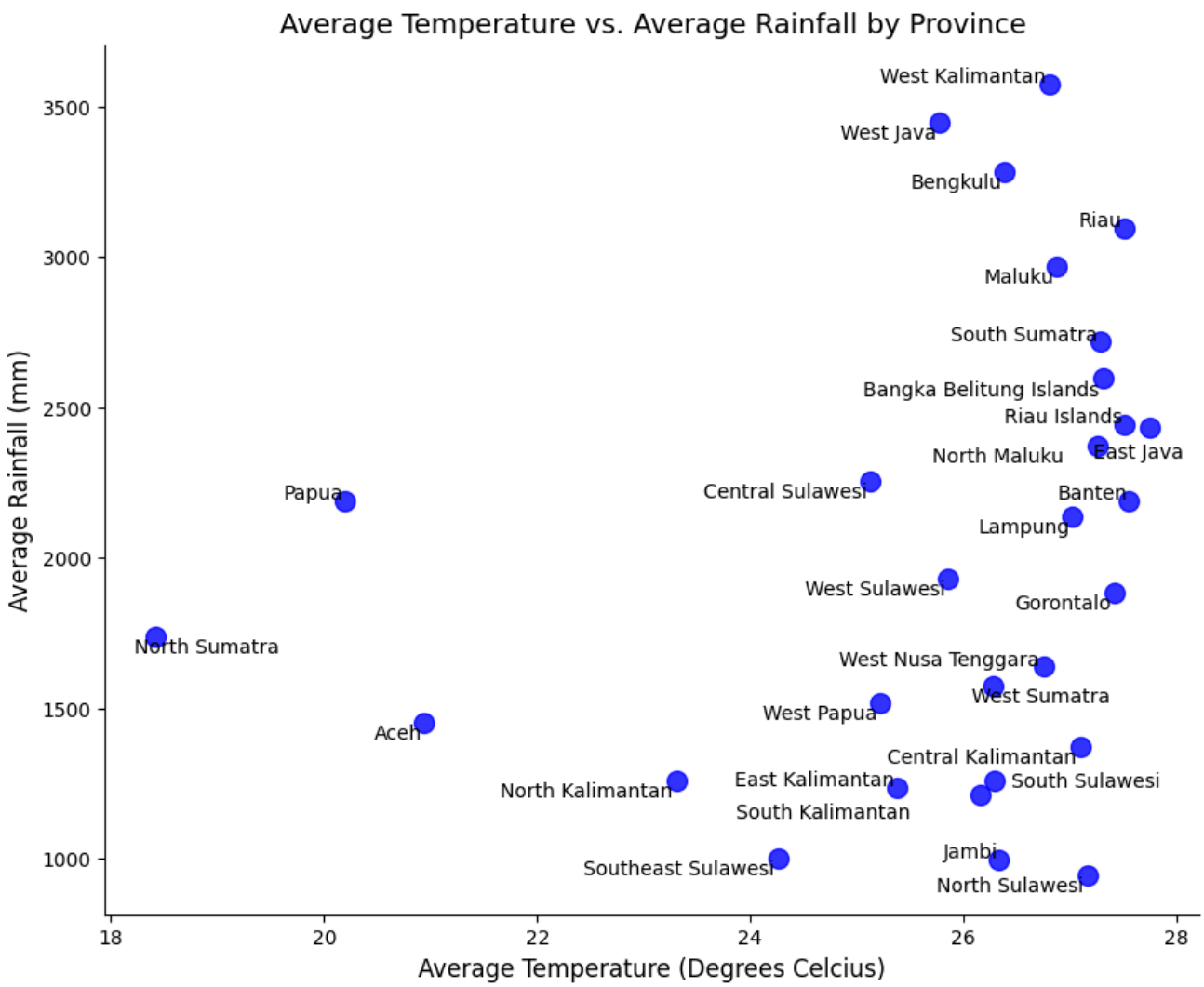
```
<ipython-input-21-cafddd92c241>:2: FutureWarning: The default value of numeric_only in DataFrameG
roupBy.mean is deprecated. In a future version, numeric_only will default to False. Either specif
y numeric_only or select only columns which should be valid for the function.
  year_grouped = weather_data_cleaned.groupby('year').mean().reset_index()
```





Plotting average temperature amd rainfall for all of Indonesia provides interesting results. From 2000 to 2016, average rainfall increased from a low of 1546.78mm in 2002 to 3493.68mm in 2016. In contrast, average temperature peaks around the 27 degrees mark between 2003 and 2009 and then plummets to around 25 degrees from 2010 to 2016.

In [ ]:
```
# Calculate the mean values for each province
province_means = weather_data.groupby('province').mean()

# Create the scatter plot with averaged values
plt.figure(figsize=(10, 8))
scatter = plt.scatter(x=province_means['average_temp'], y=province_means['average_rain'], s=100,

# Add labels and title
plt.xlabel('Average Temperature (Degrees Celcius)', fontsize=12)
plt.ylabel('Average Rainfall (mm)', fontsize=12)
plt.title('Average Temperature vs. Average Rainfall by Province', fontsize=14)

# Add province names as annotations
texts = [plt.text(province_means['average_temp'][i], province_means['average_rain'][i], province

# Adjust label positions to minimize overlap and set distance between label and point
adjust_text(texts, force_text=(0.5, 0.5), force_static=(0.5, 0.5), force_pull=(0.1, 0.1), force_

# Remove top and right spines
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

plt.show()
```



This is a scatter plot of average rainfall and temperature for all provinces. West Kalimantan stands out as a province that experiences both high temperatures and rainfall. In contrast, North Sumatra stands out as a province that experiences low rainfall and temperatures.

5. Load

This project utilises the Google Cloud Platfrom (GCP) as it provides a wide range of effective, easy-to-implement cloud solutions with modern infrastructures (Google Cloud, 2024). Within the GCP ecosystem, components used for the below pipeline are: Google Cloud Storage buckets and Cloud SQL for PostgreSQL. The choice of Cloud-Based Storage provides inherent scaability as it enables an easy expansion of IT resources to meet fluctuating demands (VMWARE, 2024).

5.1 Load Forest Net Image Data and Weather Data Into Bucket

Google Cloud Storage buckets have been selected for storing data in the cloud. Buckets enable a centralisation of the data storage which increases the efficiency of the processes, they are a scalable solution for data storage and they ensure flexible and secure accesibility to data (LogicMonitor, 2022).

The process involves creating and uploading data into cloud storage buckets in GCP. Initially, access had to be secured to the GCP services and storage capabilties by authenticating user's credentials.

Two buckets are created:

- **ucl-forest** - for storing the deforestation data
- **ucl-weather** - for storing the weather data.

The corresponding csv files and image data are uploaded to the buckets via Python script by defining a function connecting to the bucket. Hence, after the completion of these steps, the buckets serve the purpose of being repositories for storing the datasets. The above process can be seen on the file bucket.py in our GitHub repo.

Example of Function for the Bucket Creation

```
# Instantiates a client
storage_client = storage.Client()

## WEATHER BUCKET ##
# The name for the new bucket for the weather

bucket_name = "ucl-weather"
bucket = storage_client.create_bucket(bucket_name)
print(f"Bucket {bucket.name} created.")
```

To upload files on the bucket, we created two different functions: one for uploading individual files and one for uploading pictures folders.

Individual Files

```
def upload_blob(bucket_name, source_file_name, destination_blob_name):
  """Uploads a file to the bucket."""
  storage_client = storage.Client()
  bucket = storage_client.get_bucket(bucket_name)
  blob = bucket.blob(destination_blob_name)

  blob.upload_from_filename(source_file_name)

  print('File {} uploaded to {}.'.format(
      source_file_name,
      destination_blob_name))
```

Folder with Picture

This function recursively uploads all files from a specified directory on the local filesystem to a Google Cloud Storage bucket, maintaining the directory structure.

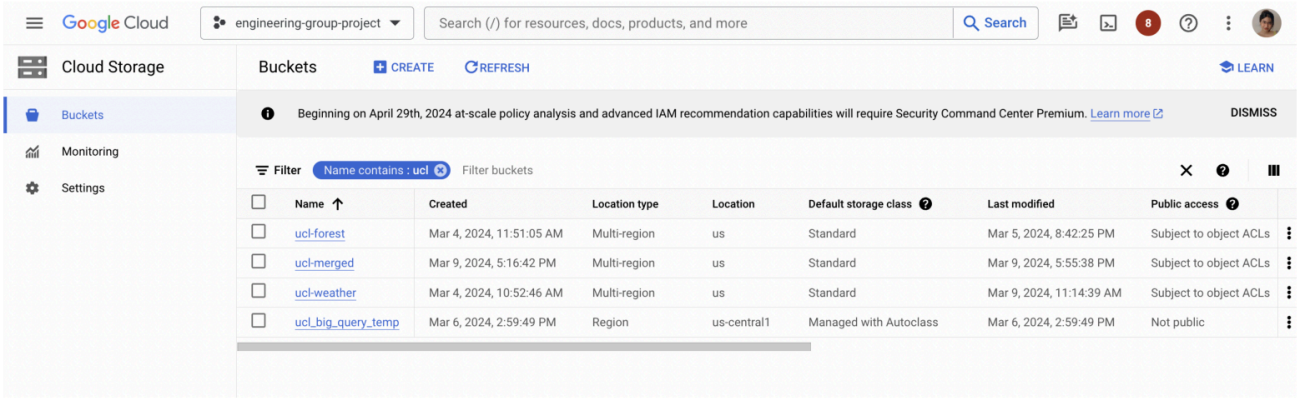
```
def upload_directory_to_bucket(bucket_name, source_directory):
    """Uploads a directory to the bucket."""
    storage_client = storage.Client()
    bucket = storage_client.get_bucket(bucket_name)

    for local_dir, _, files in os.walk(source_directory):
        for file in files:
            local_file_path = os.path.join(local_dir, file)

            # Construct the full path for the destination blob
            # Using the folder names as the blob prefix
            relative_path = os.path.relpath(local_file_path, source_directory)
            destination_blob_name = relative_path.replace("\\", "/")  # Ensure
proper path format for GCS

            blob = bucket.blob(destination_blob_name)
            blob.upload_from_filename(local_file_path)

            print(f'File {local_file_path} uploaded to
{destination_blob_name}.')
```

For storage costs reasons, we only uploaded five folders to the bucket (see Appendix)

In [ ]:
```
# The buckets were created
image_1 = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/buckets1.png')
# Set the figure size to a larger value to make the displayed image bigger
plt.figure(figsize=(20, 10))  # Adjust the width and height as needed
# Remove the axis for a cleaner look
plt.axis('off')
```

```
# Display the image
plt.imshow(image_1)
```

Out[ ]: <matplotlib.image.AxesImage at 0x7cf47bba90f0>



5.2 Connecting Buckets to Postgres in GCP

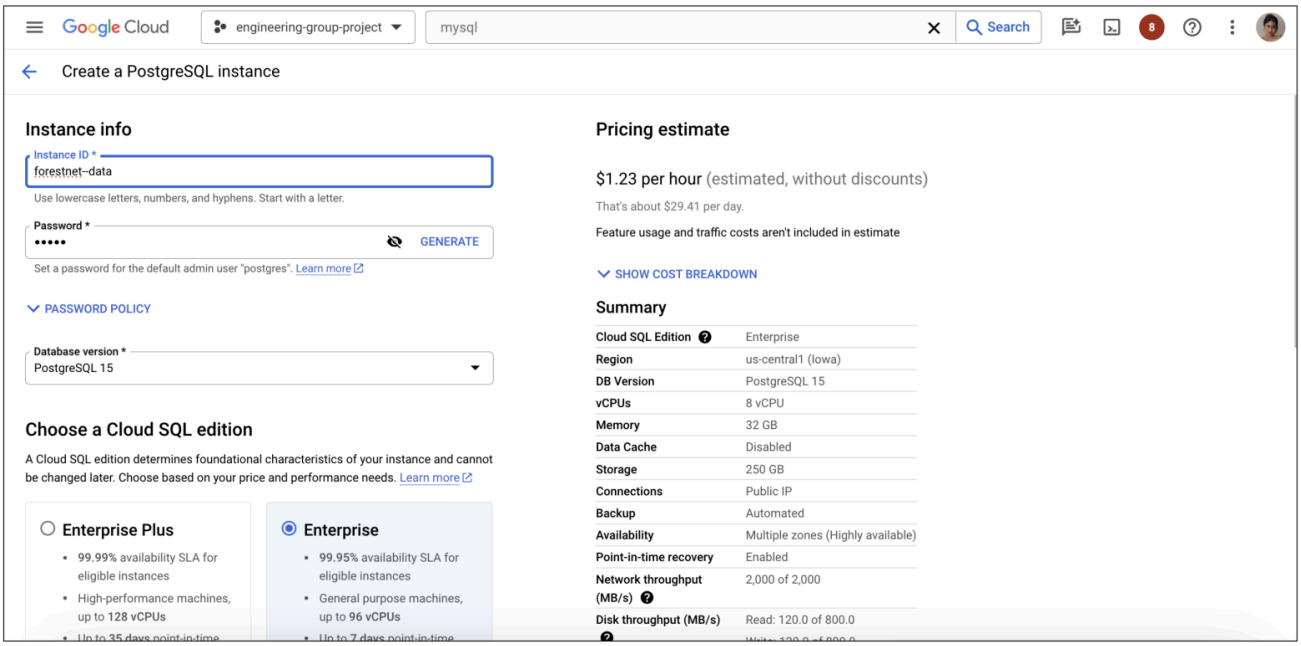
Once the data has been loaded into the buckets, we can connect them to CloudSQL - Postgres to then populate the postgres with the data present in the buckets.

PostgreSQL database within GCP is a highly reliable and scalable database system. The initial step requires the creation of an instance in PostgreSQL, where a **forestnet-data instance** is created.

The image below shows the set up of the postgreSQL15 database, with instance ID forestnet-data. In order to minimise expenses, the us-central1(Iowa) was selected as it offers a single-zone availability.

In [ ]:
```
# Open the image file
forestnet = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/forestnet.png')
# Set the figure size to a larger value to make the displayed image bigger
plt.figure(figsize=(20, 10))  # Adjust the width and height as needed

plt.xticks([])
plt.yticks([])

# Display the image
plt.imshow(forestnet)
```

Out[ ]: <matplotlib.image.AxesImage at 0x7cf47bc83a60>



Storing the data in a relational database like PostgreSQL involves creating structured tables that support querying operations. The next step includes creating tables in the database to achieve organisation, populating them with data, and facilitating efficient querying operations. Direct Interaction with the PostgreSQL database has been

achieved through the connection with the Command Line Interface(CLI), which enables more rapid, automated, and direct command execution (Abdishakur, 2023) and via the script `postgres_sql_queries.py`.

Executing the following command in CLI **`gcloud sql connect forestnet-data --user=postgres`**, secures access to the PostgreSQL environment, indicated by the terminal prompt changing to postgres=>, which welcomes SQL statements for direct database management. Kindly note that the database is named Postgres, the default database, and is authorised by the user named Postgres.

In [ ]:
```
image_3 = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/Picture3.1.png')

plt.figure(figsize=(20, 10))

plt.xticks([])
plt.yticks([])

plt.imshow(image_3)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7cf47bd19ab0>
```

Once access to psql client is established, the two tables for deforestaion and weather data can be created. The schemas are designed in a manner that fits the data, facilitating the loading and querying of data in the environment.

- Table: **deforestation_data:**

```
CREATE TABLE deforestation_data (
label VARCHAR(255),
merged_label VARCHAR(255),
latitude FLOAT,
longitude FLOAT,
year INT,
example_path TEXT,
province TEXT
);
```

- Table: **weather_data:**

```
CREATE TABLE weather_data (
    province TEXT,
    year INT,
    month INT,
    average_temp FLOAT,
    average_rain FLOAT
);
```

The gsutil tool is used to upload the data from the Buckets to the schemas via CLI, as it is considered an efficient practice for automating tasks (Almeida, 2024).

We execute the three commands below to populate a table with data in Postgres. The example is a replica of our approach, where we add to the table `deforestation_data` the data from the file **deforestaion_causes_regions.csv**, which is stored in the ucl-forest bucket.

1. Extract the relevant csv file from the bucket and remove the head row for compatatbility with the schemas.

> `gsutil cat gs://ucl-forest/deforestation_causes_regions.csv | tail -n +2 | gsutil cp - gs://ucl-forest/deforestation_causes_regions_no_header.csv`

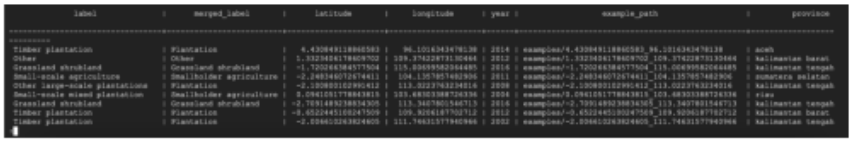
2. Import the clean data into the PostgreSQL database.

> `gcloud sql import csv forestnet-data gs://ucl-forest/deforestation_causes_regions_no_header.csv --database=postgres --table=deforestation_data --project=engineering-group-project`

3. The `SELECT * FROM deforestation_data;` output verifies the succesfull import. The image below shows that the table has been populated with data.

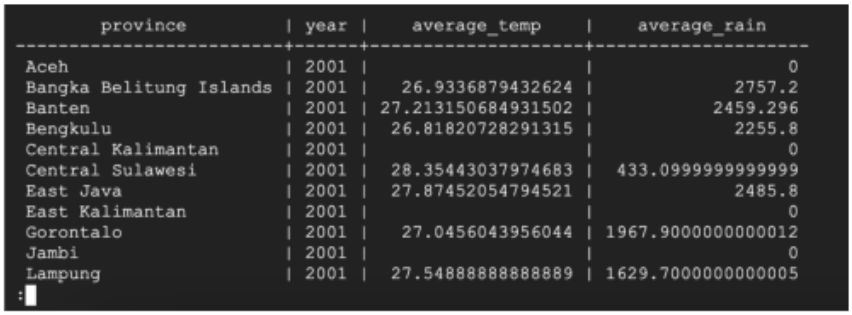
In [ ]:
```
#Deforestion table has been populated succesfully

deforestation_data = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/deforestati
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(deforestation_data)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7d0ece6628c0>
```



In [ ]:
```
#Weather table has been populated succesfully

weather_data = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/weather_data.png'
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(weather_data)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7d0ece663ca0>
```



province	year	average_temp	average_rain
Aceh	2001		0
Bangka Belitung Islands	2001	26.9336879432624	2757.2
Banten	2001	27.213150684931502	2459.296
Bengkulu	2001	26.81820728291315	2255.8
Central Kalimantan	2001		0
Central Sulawesi	2001	28.35443037974683	433.0999999999999
East Java	2001	27.87452054794521	2485.8
East Kalimantan	2001		0
Gorontalo	2001	27.0456043956044	1967.9000000000012
Jambi	2001		0
Lampung	2001	27.54888888888889	1629.7000000000005

6. Transform

6.1 Connect Postgres to Spark Pipeline for merging data

Connecting PostgreSQL to Spark is beneficial as it combines Spark's ability to handle large-scale data processing with PostgreSQL's robust data storage and retrieval capabilities, enabling efficient analysis of large datasets.

To facilitate effective data analysis, the goal is to integrate the deforestation and weather tables. Merging these two tables is crucial as it helps us build compelling visualisations and the base for future data tasks, such as machine learning. Merging Postgres tables can be done via Spark or directly in Postgres. This section covers the steps to effectively create a spark pipeline to create a new merged table.

6.1.1 Connect to Postgres via Python Script

The details steps for the connection can be found on on the postgres_Spark.py. The code is designed to interface with PostgreSQL database using the cloud-sql-python connector and the sqlalchemy libraries.

First, we set up the necessary variables for connecting to a GCP Cloud SQL instance, including the project ID, region, instance name, and the credentials of the database user.

```
project_id = "engineering-group-project"
region = "us-central1"
instance_name = "forestnet-data"

# grant Cloud SQL Client role to authenticated user
current_user = ['user@gmail.com']

DB_USER = "db_user_name"
DB_PASS = "db_pw"
DB_NAME = "db_name"
```

Second, we define the instance connection name, which is a unique identifier for the Cloud SQL instance, constructed from the project ID, region, and instance name.

```
INSTANCE_CONNECTION_NAME = f"{project_id}:{region}:{instance_name}"
print(f"Your instance connection name is: {INSTANCE_CONNECTION_NAME}")
```

Third, a connector object from the cloud-sql-python-connector library is initialised to facilitate connections to the Cloud SQL instance.

```
connector = Connector()
```

Fourth, we define the getconn function to create and return a new database connection object using the connector.connect method with the Cloud SQL instance connection name, database type (pg8000 is a PostgreSQL adapter), and authentication details.

```
def getconn():
    conn = connector.connect(
        INSTANCE_CONNECTION_NAME,
        "pg8000",
        user=DB_USER,
        password=DB_PASS,
        db=DB_NAME
    )
    return conn
```

Fifth, we create an SQLAlchemy create_engine object with a connection pool, which uses the getconn function to establish connections to the database.

```
pool = sqlalchemy.create_engine(
    "postgresql+pg8000://",
    creator=getconn,
    )
```

Last, to check the connection we print out a list of all tables in the database. This is achieved through the list_tables function, which connects to the database using the getconn function, creates a cursor object, and executes a SQL query to retrieve all table names from the public schema. It then prints the names of these tables.

```
def list_tables():
    # Use the getconn function to connect to the database
    conn = getconn()

    # Create a cursor from the connection
    cursor = conn.cursor()

    # Query to select all table names from the 'public' schema
    list_tables_query = """
        SELECT table_name
        FROM information_schema.tables
        WHERE table_schema='public';
    """
```

```
        try:
            # Execute the query
            cursor.execute(list_tables_query)

            # Fetch all the results
            tables = cursor.fetchall()

            # Print the names of the tables
            for table in tables:
                print(table[0])

        finally:
            # Close the cursor and connection
            cursor.close()
            conn.close()

    list_tables()
```

6.1.2 Import SparkSession and install JDBC Driver

In this script, we establish a connection between Spark and a PostgreSQL, allowing us to read data directly from PostgreSQL tables into Spark DataFrames.The code below initialises a SparkSession, which is the entry point for programming Spark with the Dataset and DataFrame API. It sets up the application with the name "YourAppName," specifies the location of the PostgreSQL JDBC connector jar for database connectivity, and creates a new SparkSession.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import lower

spark = SparkSession.builder \
    .appName("YourAppName") \
    .config("spark.jars", "/Users/amita/Desktop/postgresql-42.7.2.jar") \
    .getOrCreate()
```

Next, we establish a JDBC connection string that tells Spark how to connect to the PostgreSQL instance. It includes the protocol (jdbc:postgresql:), the server's IP address (34.136.83.153), the port number (5432), and the specific database to connect to (postgres).

```
database_url = "jdbc:postgresql://34.136.83.153:5432/postgres"
properties = {
    "user": "db_user_name",
    "password": "db_pw",
    "driver": "org.postgresql.Driver"
}
```

We are retrieving data from two tables, 'deforestation_data' and 'weather_data', and loading them into Spark as spark dataframe for further processing and analysis. Here we are creating 'deforestation_df' and 'weather_df'.

```
deforestation_df = spark.read \
    .jdbc(url=database_url, table="deforestation_data", properties=properties)

weather_df = spark.read \
    .jdbc(url=database_url, table="weather_data", properties=properties)
```

Before merging the two dataframes, we need to make sure the data within the spark dataframes are in the same format, to ensure this, we convert the data in the 'province' column to lowercase.

```
deforestation_df = deforestation_df.withColumn('province',
lower(deforestation_df['province']))
deforestation_df.show()

weather_df = weather_df.withColumn('province', lower(weather_df['province']))
```

```
weather_df.show()
```

Finally, we merge the two tables and create a 'merged_df' which can now be converted into a csv file and uploaded into postgres. The steps can be observed at `postgres_Spark.py`

```
merged_df = deforestation_df.join(weather_df, ['year', 'province'], how='left')
```

For further analysis, the spark dataframe can be converted into a csv file named merged_deforestation_weather.csv.

```
output_path = "/Users/amita/ucl-forest-weather-1/data/merged"

merged_df.write.csv(output_path, header=True, mode="overwrite")
```

6.1.3 Upload the merged CSV file to PostgreSQL

The spark pipeline creates a new CSV file named **`merged_deforestation_weather.csv`**, temporarily saved in the GitHub repository before being uploaded to a new bucket called `merged`. To achieve this, we re-used the functions created in the loading phase as follows:

First, creating the bucket `merged`.

```
storage_client = storage.Client()

bucket_name = "ucl-merged"
bucket = storage_client.create_bucket(bucket_name)
print(f"Bucket {bucket.name} created.")
```

Second, we uploaded the file by importing the function from the main.py to the current script.

```
import upload_blob from main.py
```

Third, we upload the data by calling the function `upload_blob` and passing the destination bucket, the data to upload, and the names of the files in the bucket.

```
upload_blob("ucl-merged",
  "data/merged/merged_deforestation_weather.csv",
  "merged_deforestation_weather.csv")
```

Fourth, in the CLI of our Postgres, we created a table for the merged data:

```
CREATE TABLE merged_deforestation_weather (
  year INT,
  province TEXT,
  label VARCHAR(255),
  merged_label VARCHAR(255),
  latitude FLOAT,
  longitude FLOAT,
  example_path TEXT,
  average_temp FLOAT,
  average_rain FLOAT
  );
```

Fifth, we populate the table with the merged data from the bucket.

```
gcloud sql import csv forestnet-data gs://ucl-
merged/merged_deforestation_weather_no_header.csv --database=postgres --
table=merged_deforestation_weather --project=engineering-group-project
```

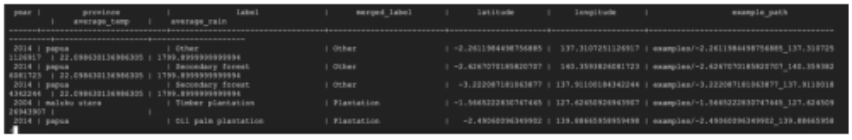
The image below shows that by running the query,

```
SELECT * FROM merged_deforestation_weather;
```

we verify that the data has been successfully uploaded.

In [ ]:
```
merged_deforestation_weather= Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/me
plt.axis('off')
plt.figure(figsize=(15, 10))
plt.imshow(merged_deforestation_weather)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7d0ece3f01c0>
```



The table has been succesfully populated!

6.2 Alternate Method: Merge data in Postgres

We observed above the method of using spark data frames to merge deforestation_data and weather_data tables. Alternatively, SQL Queries could be run on PostgreSQL in Google Cloud to facilitate data merging. The script to achieve table merging, table creation and inserting data into a new table after the merging can be seen at postgres_SQL_queries.py

7. Data Visualizations

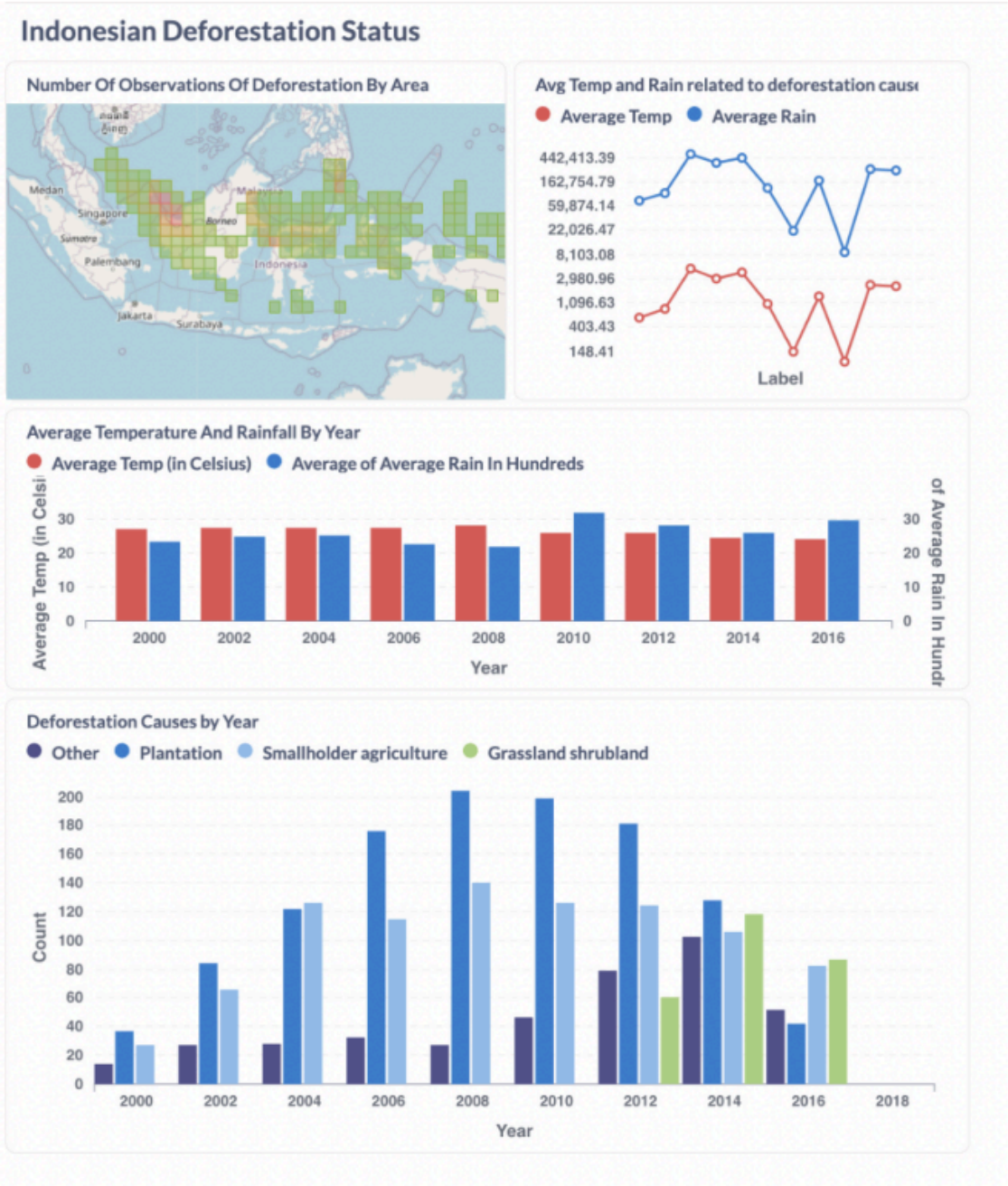
For the purpose of creating the visualisations, the PostgreSQL database hosted on Google Cloud was connected to Metabase using the provided public IP address as the hostname, along with the necessary database name, username, password, and port for access.

Metabase was selected for its ease of use and approachable interface, allowing for the rapid creation of visualisations and dashboards. The key advantages of using Metabase include its simplicity in transforming data into insights, the ability to generate and share reports efficiently, and its supportive features for collaborative data exploration.

The link to the dashboard can be found here

In [ ]:
```
dashboard1= Image.open('/content/drive/MyDrive/ForestNetDataset/dashboard1.png')
plt.figure(figsize=(15, 10))
plt.axis('off')
plt.imshow(dashboard1)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7d0ecbf4b190>
```



Observation:

We have a heatmap that identifies which regions are most affected by deforestation by analysing the colour density on the map. Darker shades typically represent higher counts of deforestation instances.

The bar chart *'Average Temperature and Rainfall by Year'* displays the patterns of rainfall and temperature variations from the years 2000 to 2016.

Deforestation can be caused due to multiple reasons such as excess planation, smallholder agriculture , grassland shrubland etc. The bar chart *'Deforestation causes by Year'* indicates that one of the biggest contributing factors of deforestation is plantation and the effects of these were most observed from 2008 to 2012.

It is interesting to observe that in the years that sees less rainfall and higher temperature- specifically the years 2006 and 2008, deforestation is seemingly high.

8. Limitations

8.1 Exploratory Attempt: Integrating PostgreSQL with BigQuery

As part of the data transformation step, we explored the idea of using BigQuery. Google BigQuery is a serverless, fully managed enterprise data warehouse. It utilises BigTable and the Google Cloud Platform and is able to join, query, and analyze data of any size, from a few rows to petabytes, in just a few seconds.

Why use Bigquery?

BigQuery provides a highly scalable and serverless data warehouse solution. By connecting BigQuery to PostgreSQL, we can leverage BigQuery's powerful analytics and ML capabilities on the PostgreSQL data, making it ideal for running complex queries and extracting insights from large datasets in near real-time.

To upload data, we need to build a CDC pipeline using datastream. Building a CDC pipeline is a straightforward way to continuously pipeline our data from Postgres to Google BigQuery in near real-time making it easily available for analytics, reporting, and machine learning.

The region we selected for building our datastream was US-Central 1 Region Iowa.

Choosing a single region for all services in a project helps us reduce latency, as data doesn't have to travel as far, leading to faster response times and a better user experience. This also reduces data transfer costs, which are often higher for inter-regional data movement and simplifies operational logistics.

8.1.1 Create a dataset and table in Bigquery

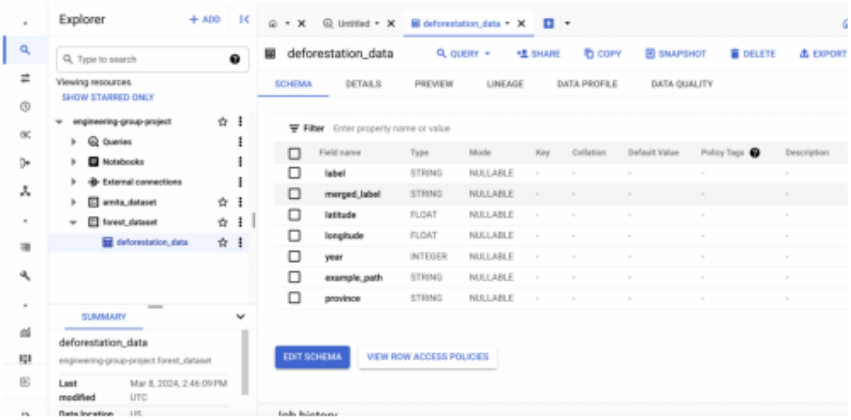
Before building the datastream to connect Postgres to BigQuery, we need to build a dataset and a table schema. A well-defined dataset and schema ensure that the data transferred from PostgreSQL is organized and stored correctly in BigQuery and helps us effectively manage relationships.

A dataset named 'forestnet_dataset' and a schema for table named 'deforestation_data' was created.

Note: The table can also be created via VS Code, the steps we followed can be observed here.

In [ ]:
```
from PIL import Image
bq1 = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/bigq1.png')
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(bq1)
```

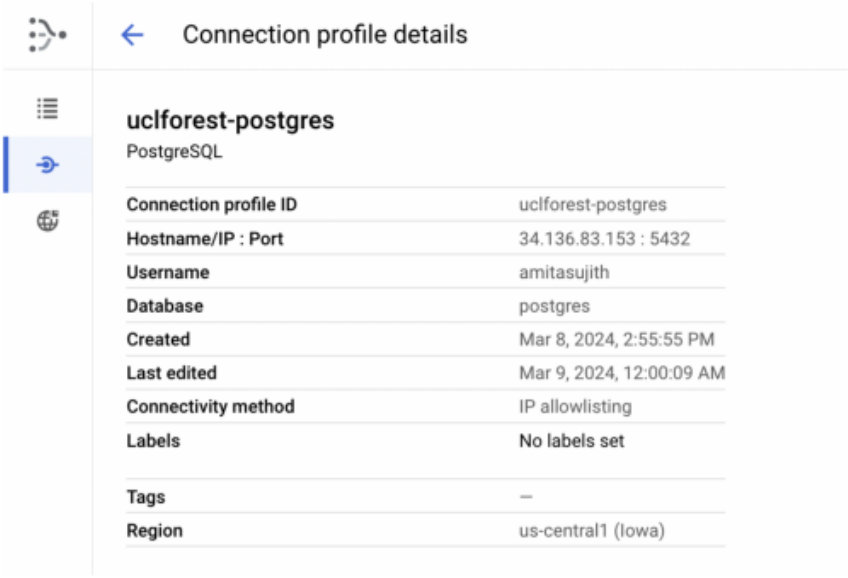
Out[ ]:
```
<matplotlib.image.AxesImage at 0x7c0ebd45b760>
```



8.1.2 Create connection profiles

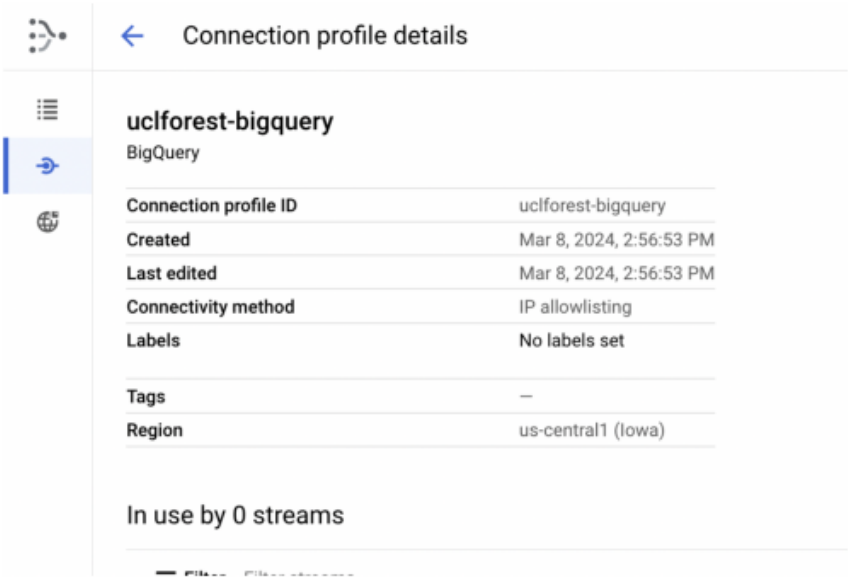
In [ ]:
```
# Source connection profile
bq2 = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/bigq2.png')
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(bq2)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x78ad5f190b80>
```



In [ ]:
```
# Destination connection profile
bq3 = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/bigq3.png')
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(bq3)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x78ad5cffc430>
```

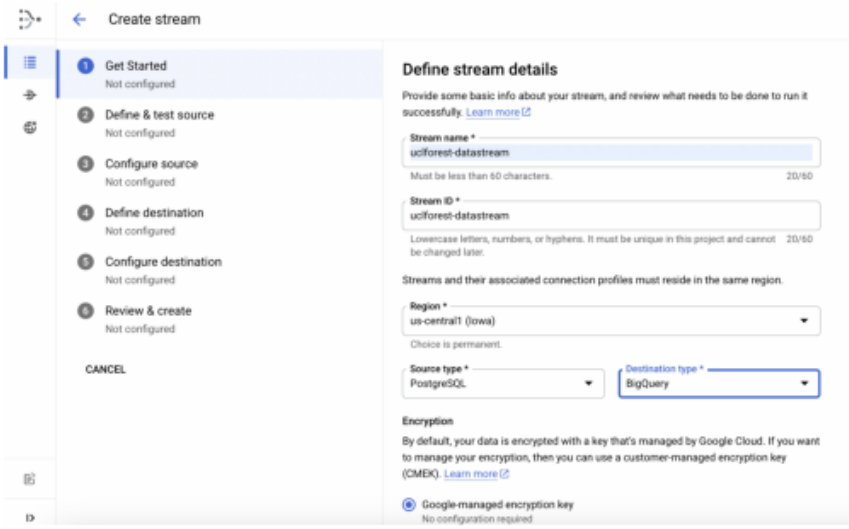


8.1.3 Create the Datastream

Datastreams facilitate real-time or near-real-time data ingestion into BigQuery, ensuring that the data warehouse is continuously updated with the latest information.

In [ ]:
```
# Defining stream details
bq4 = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/bigq4.png')
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(bq4)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7c0eee01d960>
```



We created a datastream named 'uclforest-datastream'. As part of this step we also created a publication and replication slot in the CLI.

Publication: In PostgreSQL, a publication is a set of database changes (inserts, updates, deletes) that can be replicated. We created it by running the following command:

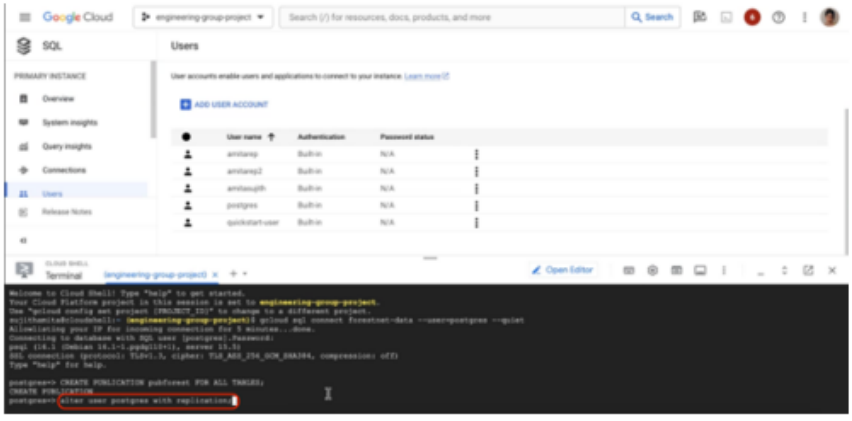
CREATE PUBLICATION pubforest FOR ALL TABLES;

Replication slot: It is a feature that ensures changes to the database are not removed before they have been replicated to all subscribers. After connecting with a user with sufficient privileges, we can create a replication slot by running the following command:

SELECT PG_CREATE_LOGICAL_REPLICATION_SLOT('repforest', 'pgoutput');

In [ ]:
```
# Replication slot
replication = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/replicationslot.pn
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(publication)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7c0ebd525e40>
```



Followingly, a unique a user called "ucidatastream" was generated and it was granted permission to read and access objects within this schema.

> CREATE USER ucidatastream WITH REPLICATION IN ROLE cloudsqlsuperuser LOGIN PASSWORD 'bauci';

Once the role was created, we granted the user all permissions by running the command:

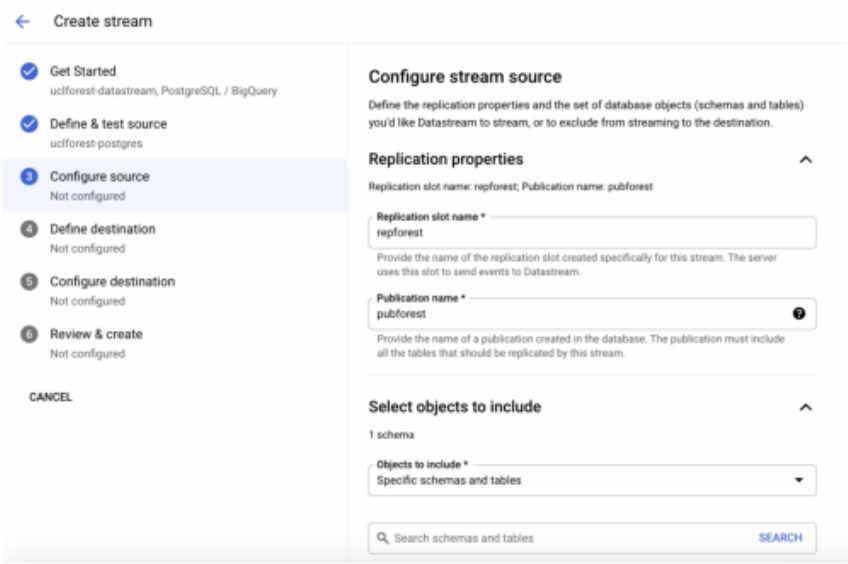
> GRANT SELECT ON ALL TABLES IN SCHEMA public TO ueldatastream;

GRANT USAGE ON SCHEMA public TO ucidatastream; ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO ucidatastream;

Subsequently, within the web interface, configurations are being made for both the data source and the destination. This involves specifying the necessary tables to be extracted from the database, as well as setting up the destination.

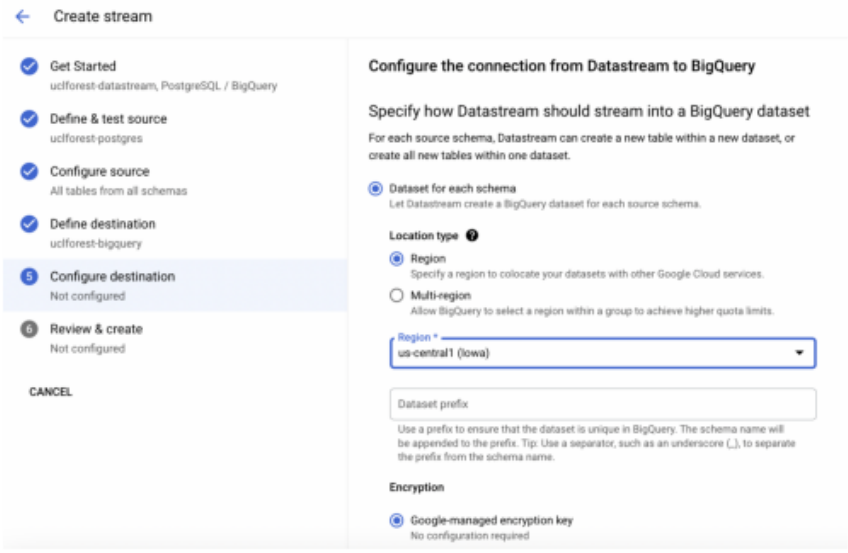
In [ ]:
```
# Source Configuration
source= Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/source-bq.png')
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(source)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7d0ec9b9b4c0>
```



In [ ]:
```
# Destination Configuration
destination= Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/destination-bq.png'
# Set the figure size to a larger value to make the displayed image bigger

plt.axis('off')
# Display the image
plt.imshow(destination)
```

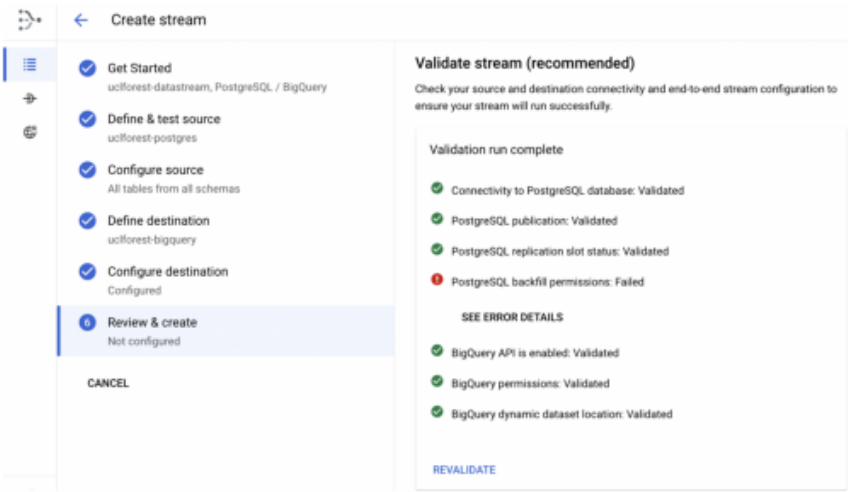
Out[ ]:
```
<matplotlib.image.AxesImage at 0x7d0ecbf4a380>
```



In [ ]:
```
# Running validation to create data stream
validation_bq = Image.open('/content/drive/MyDrive/ForestNetDataset/Screenshots/validation_bq.pn
# Set the figure size to a larger value to make the displayed image bigger
```

```
plt.axis('off')
# Display the image
plt.imshow(validation_bq)
```

Out[ ]: <matplotlib.image.AxesImage at 0x7c0eb897d870>



Following this setup, the connectivity is then tested throgh the following commands :

```
gsutil cat gs://ucl-weather/historical_weather_data_top3.csv | tail -n +2 | gsutil cp
- gs://ucl-weather/historical_weather_data_top3_no_headers.csv
gcloud sql import csv forestnet-data gs://ucl-
weather/historical_weather_data_top3_no_headers.csv --database=postgres --
table=weather_data --project=engineering-group-project```
```

Why we couldn't proceed with BigQuery

As observed above, the validation did not pass and we were unable to create the datastream due to the error: *'PostgreSQL backfill permissions: Failed'*

The error statement suggested that "One or more tables in the stream's include list weren't found or Datastream couldn't read from them. Make sure that the tables exist and that Datastream has the necessary permissions."

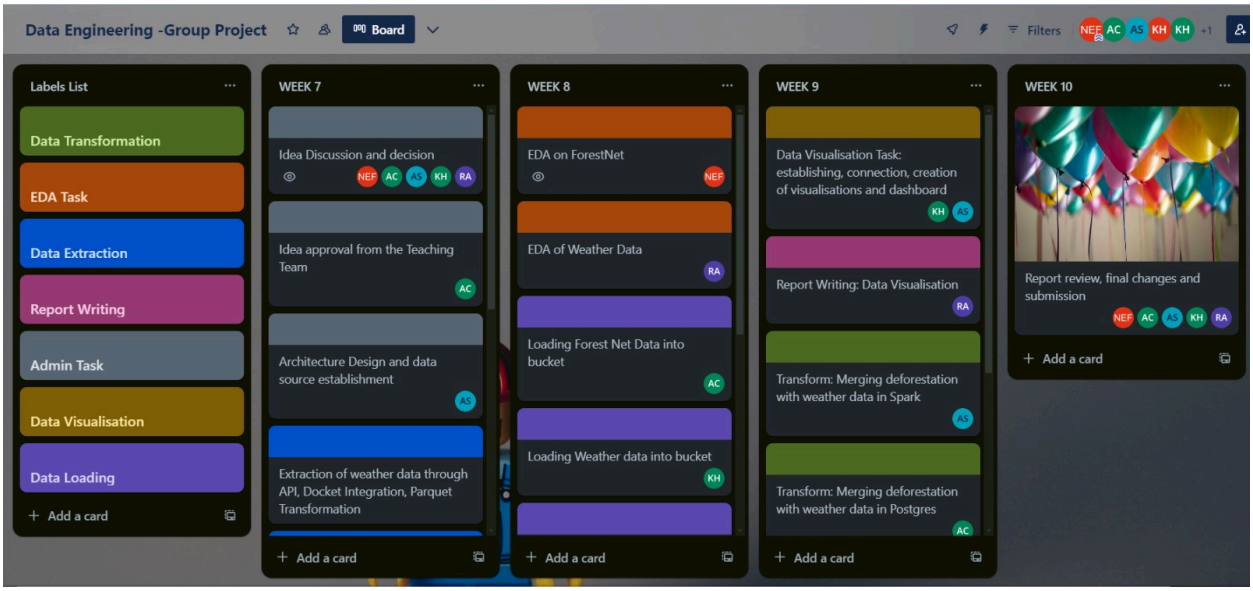
8.2 Limitations for scalability

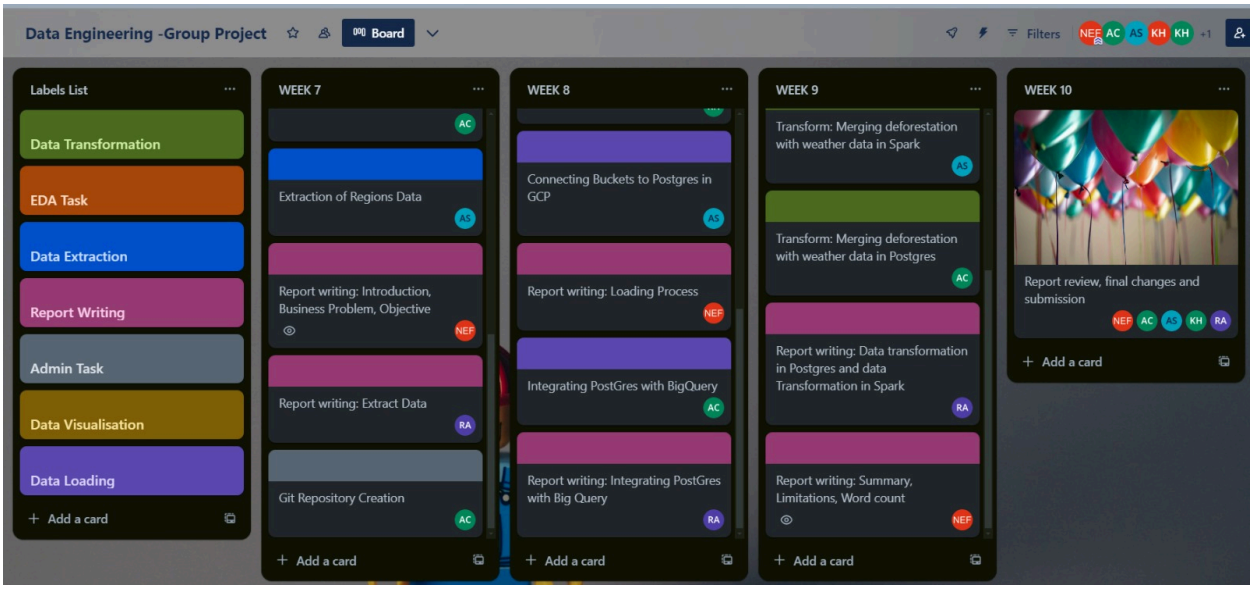
After performing transformations using Spark, data must be routed through a storage bucket before being loaded into the PostgreSQL database. This intermediate step may increase latency and act as a bottleneck in the pipeline, especially as the volume of data scales up.

Source Version Control(SCM)

For the operations of our project, Git has been used for version control system purposes. It was chosen based on its efficiency for collaborating, reviewing code and tracking changes (Wayne 2023). Hence a detailed view of the project's codebase can be found through the project's GitHub repository: https://github.com/acerutti/ucl-forest-weather

Project Management





Summary

The project successfully implemented a fully cloud-based, scalable architecture that leveraged diverse data extracting, loading and transformative techniques. The subsequent phase could involve creating a dynamic deforestation dashboard featuring live weather data and risk alerts, coupled with advanced analysis and machine learning for better predictions and insights to further combat deforestation.

References

Abdishakur H.(2023)What Is a Command-Line Interface? Built In. Available at: https://builtin.com/software-engineering-perspectives/command-line-interface (Accessed: 9 March 2024).

Almeida, T. A. (2024) gsutil: Command-Line Control of Google Cloud Storage, NetApp BlueXP. Available at: https://bluexp.netapp.com/blog/gsutil-command-line-control-of-google-cloud-storage (Accessed: 9 March 2024).

DGB Group(2024) Countries with the highest deforestation rates in the world, DGB Group. Dutch Green Business Group. Available at: https://www.green.earth/blog/countries-highest-deforestation-rates (Accessed: 10 March 2024).

Geodata.mit.edu. (2015). First-level Administrative Divisions, Indonesia, 2015 - GeoWeb. Available at: https://geodata.mit.edu/catalog/stanford-ky614gm0524.

Geokeo. (2024). List of all states in the world with Latitude and Longitude. Available at: https://geokeo.com/database/state/id/ (Accessed: 11 March 2024).

Google (2024) Google Cloud. Why Google Cloud?. Available at: https://cloud.google.com/why-google-cloud?hl=en (Accessed: 9 March 2024).

Hijmans and California, U. of (2015) First-level Administrative Divisions, Indonesia, 2015, GeoWeb. University of California, Berkeley. Museum of Vertebrate Zoology. Available at: https://geodata.mit.edu/catalog/stanford-ky614gm0524 (Accessed: 10 March 2024).

Irvin, J., Sheng, H., Ramachandran, N., Johnson-Yu, S., Zhou, S., Story, K.T., Rustowicz, R., Elsworth, C.W., Austin, K. and Ng, A.Y. (2020). ForestNet: Classifying Drivers of Deforestation in Indonesia using Deep Learning on Satellite Imagery. arXiv (Cornell University). doi:https://doi.org/10.48550/arxiv.2011.05479.

LogicMonitor (2023) What is a bucket in GCP? GCP buckets explained, LogicMonitor. Available at: https://www.logicmonitor.com/blog/what-is-a-bucket-in-gcp-gcp-buckets-explained (Accessed: 10 March 2024).

RAN(2024)Indonesia Climate Change and Rainforests Available at: https://www.ran.org/wp-content/uploads/2018/06/indonesia_climatechange_rainforests.pdf (Accessed: 11 March 2024).

Risser MD(2024), Collins WD, Wehner MF, O'Brien TA, Huang H, Ullrich PA. Anthropogenic aerosols mask increases in US rainfall by greenhouse gases. Nature Communications.;15(1):1318. doi: 10.1038/s41467-024-45504-8

Ritchie (2023) Deforestation and Forest Loss, Our World in Data. Available at: https://ourworldindata.org/deforestation#:~:text=Globally%20we%20deforest%20around%20ten%20million%20hect (Accessed: 10 March 2024).

Stanford ML Group(2020)ForestNet: Classifying Drivers of Deforestation in Indonesia using Deep Learning on Satellite Imagery. Available at: https://stanfordmlgroup.github.io/projects/forestnet/ (Accessed: 10 March 2024).

VMWARE(2024). Available at: https://www.vmware.com/topics/glossary/ content/cloud-scalability.html#:~:text=Cloud%20scalability%20in%20cloud%20computing,its%20exploding%20popularity%20with (Accessed: 10 March 2024).

Wayne, R. (2023) Unlocking the Power of Git: The Benefits of Using Git for Source Code Management, Medium. Medium. Available at: https://medium.com/@rick.wayne.2022/unlocking-the-power-of-git-the-benefits-of-using-git-for-source-code-management-8e4eae6782c7#:~:text=Git%20provides%20a%20complete%20version,to%20identify%20and%20resolve%20issues (Accessed: 10 March 2024).

In [ ]:
```
### WORD COUNT ###
import nbformat

# Load the current notebook
notebook_path = '/content/drive/MyDrive/ForestNetDataset/DE_Forest_Net_Draft.ipynb'
with open(notebook_path, 'r', encoding='utf-8') as nb_file:
    nb_contents = nbformat.read(nb_file, as_version=4)

# Initialize word count
total_word_count = 0

# Iterate through each cell in the notebook
for cell in nb_contents['cells']:
    if cell['cell_type'] == 'markdown':
        total_word_count += len(cell['source'].split())
    elif cell['cell_type'] == 'code':
        # Count words in code comments
        comments = [line for line in cell['source'].split('\n') if line.strip().startswith('#')]
        comment_text = ' '.join(comments)
        total_word_count += len(comment_text.split())

# Print the total word count
print(f"Total Word Count: {total_word_count}")
```

```
Total Word Count: 5080
```

Appendix

Appendix I: Selection of Uploaded Images

In [ ]:
```
image_names = [
    "-0.002226324002905_109.97159881327198",
    "-0.025186616919319_113.71975529073988",
    "-0.03027428009349_101.59766149324824",
    "-0.22946911191326_101.62682788208392",
    "-0.23142888714857_99.87426629402542"
]

# Function to find matching row in DataFrame for given image name
def find_matching_row(gdf, image_name):
    # Split image name into latitude and longitude
    lat_str, long_str = image_name.split('_')
    latitude = float(lat_str)
    longitude = float(long_str)

    # Find row in DataFrame where latitude and longitude match
    match = df[(df['latitude'].astype(str).str.contains(lat_str)) & (df['longitude'].astype(str)
    return match

# Iterate through image names and find matching rows
matches = []
for image_name in image_names:
    match = find_matching_row(df, image_name)
    if not match.empty:
        matches.append((image_name, match.index.tolist()))

matches
```

Out[ ]:
```
[('-0.002226324002905_109.97159881327198', [392]),
 ('-0.025186616919319_113.71975529073988', [2502]),
 ('-0.03027428009349_101.59766149324824', [1513]),
 ('-0.22946911191326_101.62682788208392', [162]),
 ('-0.23142888714857_99.87426629402542', [2584])]
```