Cloud ETL Report

For the Cloud ETL requirement of our project, we leveraged our knowledge of Kafka and tools provide by Microsoft Azure to build an automated data pipeline on CO2 data from https://global-warming.org. With the creation of this pipeline, we can continually update the data gathered via the API and load it into SQL for use in visuals or statistical models.

The first step of this process was finding and gathering data from an API. We decided to use the API found here: https://global-warming.org/api/co2-api. This is a simple API, that contains one dataset that receives one new record entry every day. Additionally, previous entries are occasionally updated, but these changes are only made to roughly the 50 most recent entries. Each of these entries contain five fields: year, month, day, cycle, and trend. Cycle and trend are atmospheric CO2 measurements in parts per million (ppm). Cycle is the true atmospheric reading, while trend represents the reading adjusted to ignore the Earth CO2 cycle.

Now that we had built an understanding of the data, we needed to extract it. To extract the data from the API, we used Azure data bricks and the requests, json, and pandas libraries. Using the link to the API above, we perform a get request. Then, we turned the request into json format with the json library and loaded the json into a dataframe. If you click the link to the API above, you will see that the data is in a dictionary with a single key of 'co2'. Using the json.normalize() method on the created dataframe, using the 'co2' key, we get a dataframe with five columns representing the five fields of the data. Finally, we then transfer the dataframe to spark, so that we can use Kafka to send the data as messages to a consumer. The reproducible code for this step can be seen below, with added error handling features.

```
url = 'https://global-warming.org/api/co2-api'

loop = True
while loop:
    try:
        r = requests.get(url)
        if r.status_code == 200:
            loop = False
        else:
            print("Request failed. Trying again...")
            continue
    except Exception as e:
        print(f"Error: {e}")
        print("Trying again...")
        continue

r = r.json()
r = json.dumps(r)

data_pandas = pd.read_json(r)
data_pandas = pd.json_normalize(data_pandas["co2"])

data = spark.createDataFrame(data_pandas)
```

The next step was to build a Kafka producer that would be able to send rows of the dataframe as messages to a Kafka consumer. In order to perform this step, you will need access to Kafka via a broker where you will be able to set up a topic and send messages. These steps are dependent on which broker you choose, but the basic setup of a Kafka producer should be easily repeatable from an example online using whichever broker you prefer.

Once we have setup a producer, we can then write the code to produce the messages. Since up to the 50 most recent entries can get updated, we want to send the last 100 entries every time we run our pipeline to ensure we don't miss an update. This means we must loop through the final 100 rows of the

dataframe, compile each row into a json (python dictionary) format, and then produce that message to the broker. The following code shows how our group accomplished this, with some extra error handling.

```
condition = True
success, failure = 0, 0

while condition:
    for i in range(1,101):
        try:
            aDict = {}
            aDict = data.take(data.count())[-i].asDict()
            p.produce('group-6-th',json.dumps(aDict))
            p.flush()
            success += 1
            time.sleep(0.1)
        except Exception as e:
            print(f"Error: {e}")
            failure += 1
    condition = False
```

Once we had completed this step, we began to build a Kafka consumer. This consumer would be used to gather the messages sent by the producer, transform the data, and then load the data into SQL. Similarly, to building a Kafka producer, the consumer will depend on which broker you are using, but should be easily repeatable from an online example. Once, the consumer is setup, we can receive the data and perform the needed cleaning and transformations upon it.

To receive the messages from our producer, we set up a loop that would collect the 105 most recent messages and would stop once a message was not returned for the 6th time. This allows for us to ensure that it collects each message that is sent by the producer, while also ensuring it doesn't run forever if no messages are found. Once we receive the message, we want to append them into a dataframe to hold them all. The code for this portion of the consumer can be seen below.

```
condition = True
breakpoint, success = 0, 0

while condition:
    df = pd.DataFrame()
    for i in range(106):
        try:
            msg = consumer.poll(1.0)
            output = display_msg(msg)
            if output != 'err':
                success += 1
                temp = pd.DataFrame([output])
                df = pd.concat([df, temp])
            else:
                breakpoint+=1
                if breakpoint > 5:
                    condition = False
                    break
        except Exception as e:
            print(f"Error: {e}")
    condition = False
consumer.close()
```

Now that the consumer has received the messages and they are in a dataframe, we can transform the rows. To begin, we make sure the columns have the correct types. Year, month, and day are integers while cycle and trend are floats. Then, we load in the data that is currently being stored in our SQL server into another dataframe. We append the new messages to the previous data, then drop the duplicates within the data on the subset columns of year, month, and day. Additionally, we make sure that we drop the

oldest entry of these columns and keep the most recent. This allows for the data to update to the most recent values for the given date. Once this is done, we then add a column to track the number of million tons of CO2 in the atmosphere. This number is found by multiplying the cycle and trend columns by 7,800 to convert from ppm to million tons. Our final step is to order the final dataframe by year, month, and day. Then we load the final data to a SQL table. These steps can be seen in the code below. (in the concat statement, df represents the new messages and data_pandas represents the old SQL data)

```
if df.shape != (0,0):
    df[["year", "month", "day"]] = df[["year", "month", "day"]].astype("int64")
    df[["cycle", "trend"]] = df[["cycle", "trend"]].astype("float64")

data_pandas = pd.concat([data_pandas, df])

data_pandas.drop_duplicates(subset=["year", "month", "day"], inplace=True, keep= "last")
data_pandas["atmospheric_cycle"] = data_pandas['cycle'] * 7800
data_pandas["atmospheric_trend"] = data_pandas['trend'] * 7800

df_daily_emissions = spark.createDataFrame(data_pandas)
df_daily_emissions = df_daily_emissions.orderBy("year","month", "day")
```

Now the data is stored within a SQL server and can be pulled into a different python notebook and used to make visuals, create a machine learning algorithm, or make a statistical model.