Data Engineering Pipeline with Python, MySQL and AWS

Introduction

The e-scooter company Gans is looking to expanding into the European market. They are most interested in the German speaking markets where there are already some well established companies such as Tier and Bird in the scene. Gans' operational success depends on whether they can optimize the supply and demand of their scooters, which means they have to place their scooters at locations where the users need. They have discovered some user patterns such as people tend to ride uphill with a scooter and walk downhill, scooter usage decreases when there is rain, and young tourists like to ride scooters from the airport to the city. With that, Gans need to collect data such as geographical information and population, weather, airports of the city and flight arrivals to those airports.

Data collection

City data

Since Gans sought to expand into the German speaking markets. We have decided to focus on the top 10 German cities and top 5 Austrian cities by population. Initially, we performed web scraping on the Wikipedia pages of the respective cities in order to collect geo and population data. However, there are discrepancies in the infobox of the Wikipedia entries and we were not able to obtain uniform data across all cities. Therefore, we have decided to use an API instead. We have found the GeoDB Cities API from RapidAPI to be a good source of city data. This API takes the WikiData ID of the cities as input and return data such as latitude, longitude, elevation and population of the cities. We have programmed our requests such that all city data are returned as one concatenated dataframe. We have dropped the redundant columns and renamed the rest with more intuitive names. This is necessary since some column names contain dots which will pose problems when we try to query them in MySQL later.

```
# loop through the list of cities, makes API calls and concatenate all city data into one dataframe
city_list = []
for city in cities:
    url = f"https://wft-geo-db.p.rapidapi.com/v1/geo/cities/{city}"
    headers = {
        "X-RapidAPI-Key": API_key,
        "X-RapidAPI-Host": "wft-geo-db.p.rapidapi.com"
    }
    response = requests.request("GET", url, headers = headers)
    time.sleep(2)
    city_df = pd.json_normalize(response.json())
    city_list.append(city_df)
cities_df = pd.concat(city_list, ignore_index = True)
cities_df = cities_df[["data.id", "data.wikiDataId", "data.city", "data.country", "data.elevationMeters",
                       "data.latitude", "data.longitude", "data.population"]]
cities_df
```

Weather data

We used the OpenWeatherMap API to collect data and the procedure of doing so is similar to the steps above for city. We have also dropped the unnecessary columns and renamed some of them.

Airport data

AeroDataBox from Rapid API is a good resource for both airport and flight arrival information. It is capable of searching airports when given a set of latitude and longitude as input. We have defined the parameters to show only 10 results within 50km of the input location. We believe it is sufficient to capture all major airports within a target city.

Flight arrival data

There are additional parameters that need to be set in order to get flights information. First is the date and we have used the datetime function in Python to determine today's date, so that we can infer tomorrow's date from there. We are only interested in arrival information and have excluded return, cancelled, cargo and private flights since our target demographic is young tourists arriving into the city from commercial flights.

Setup MySQL Database

Once we have all the data as dataframes, we created an empty Gans database in MySQL. We have opted to first push all the tables to the database before defining a relationship between them, instead of creating all the empty tables in MySQL first. By importing the SQLAlchemy package in Python, we are able to push the dataframes to MySQL in a simple manner with just a few lines of code

```
schema = "gans"
host = "127.0.0.1"
user = "root"
password = os.getenv("MYSQL_PASSWORD")
port = 3306
con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"
```

```
city_df.to_sql("cities", if_exists = "append", con = con, index = False)
```

```
15
```

```
weather_df.to_sql("weather", if_exists = "append", con = con, index = False)
```

```
600
```

```
airport_df.to_sql("airports", if_exists = "append", con = con, index = False)
```

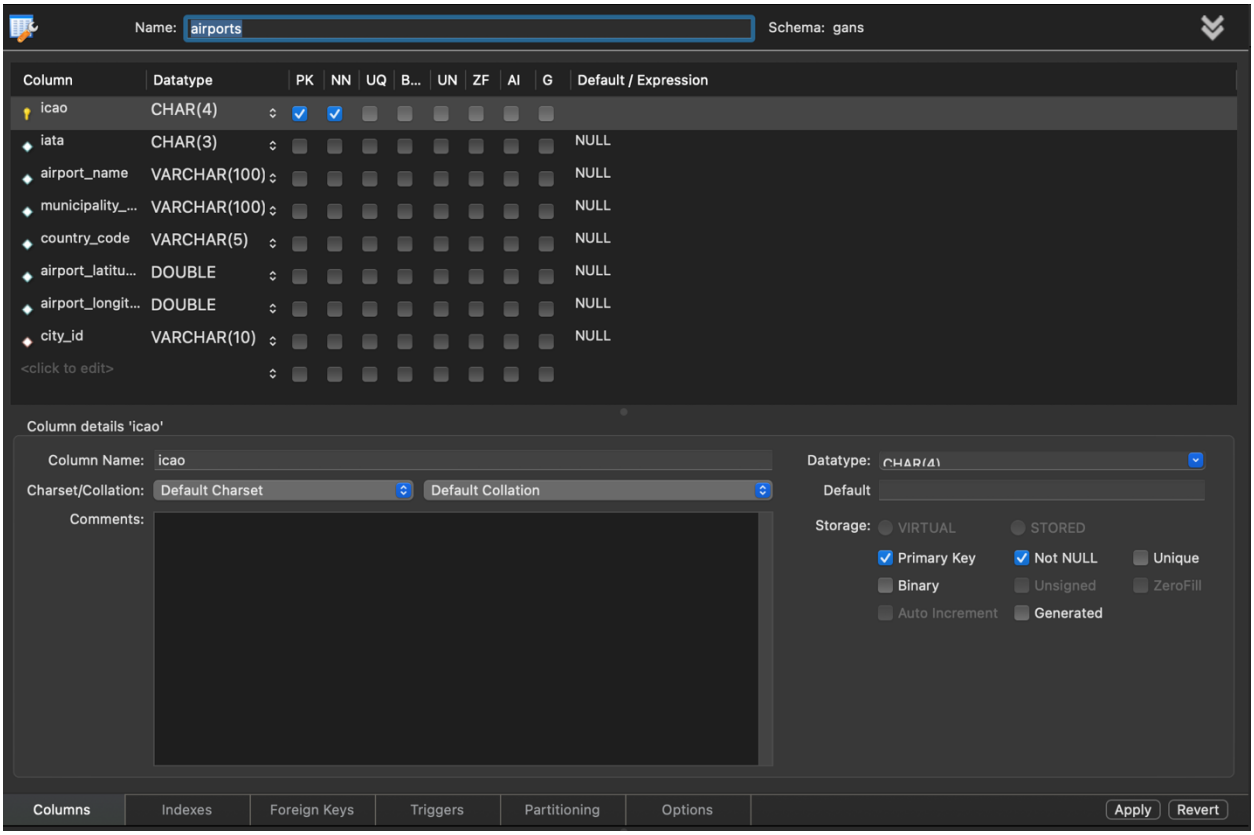
```
15
```

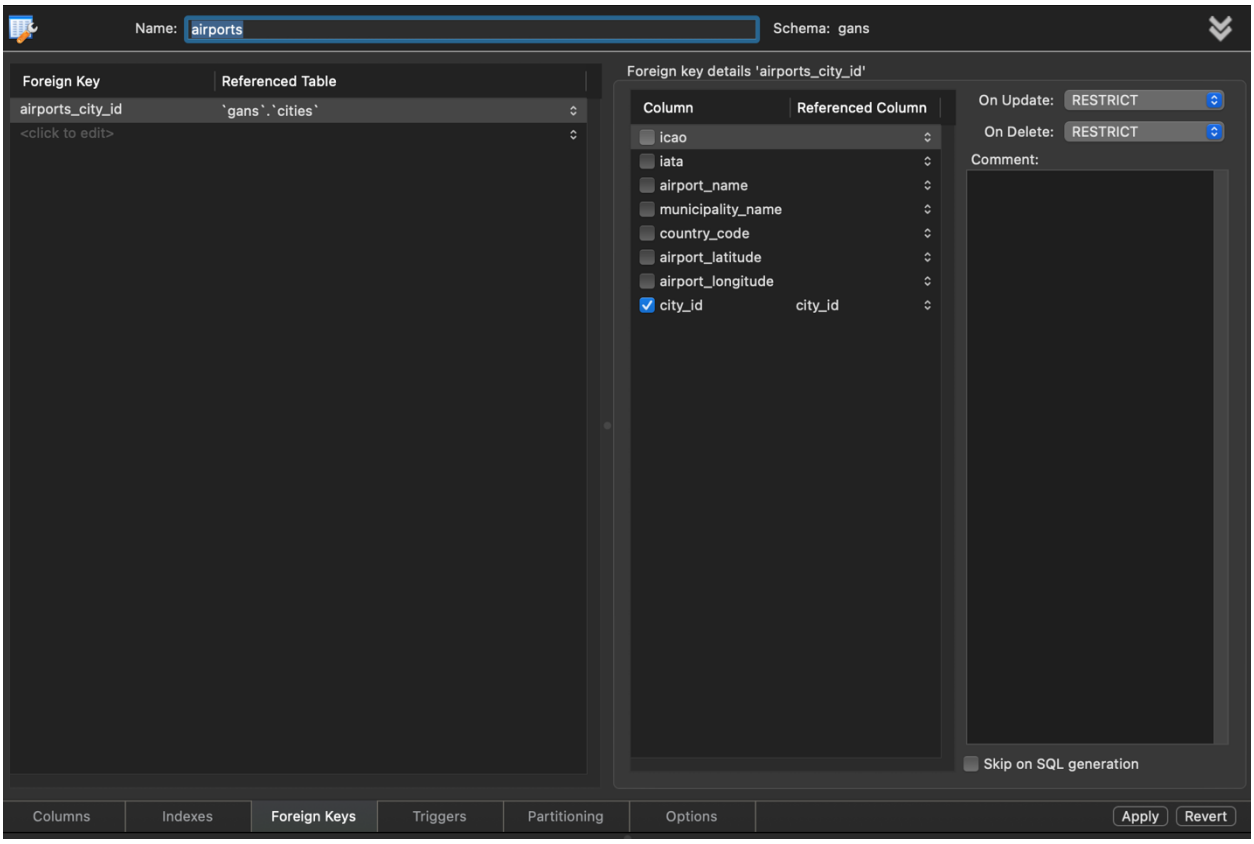
```
flight_df.to_sql("flights", if_exists = "append", con = con, index = False)
```

```
2451
```

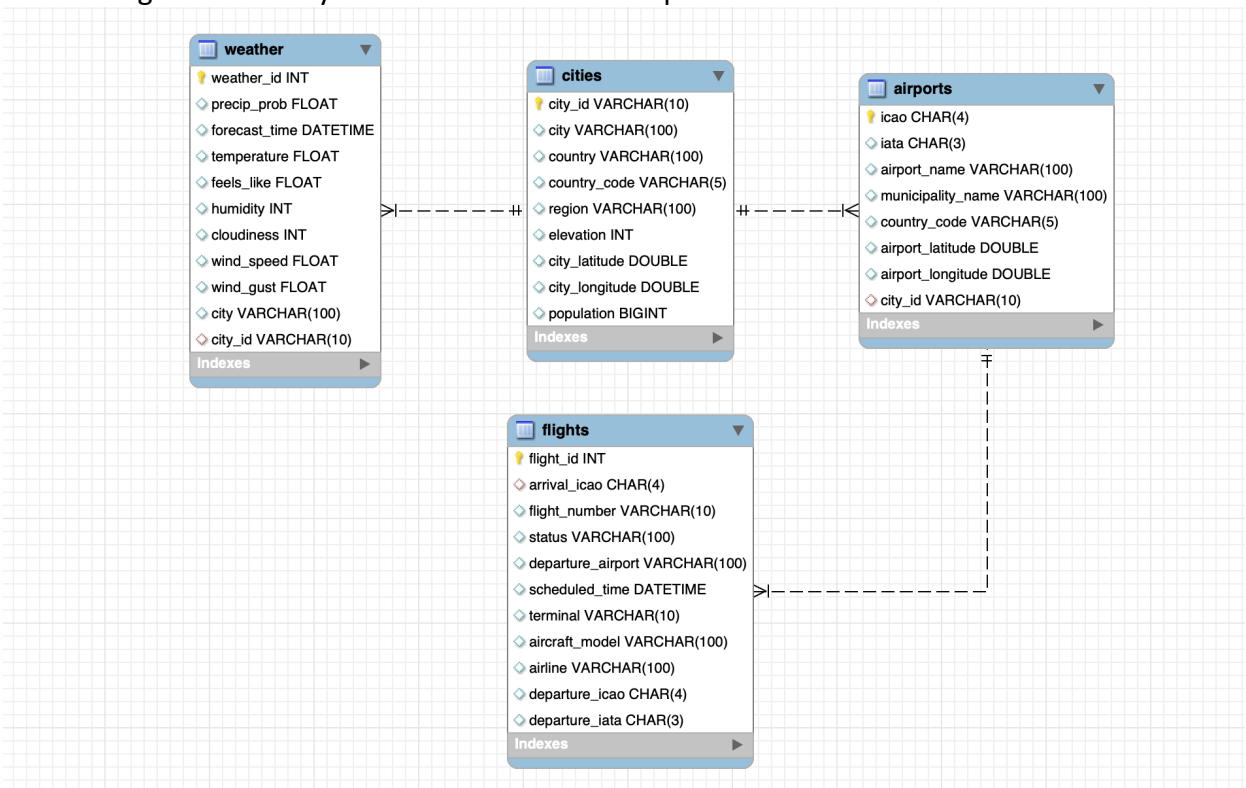
After all the dataframes are pushed to MySQL, they became four standalone tables with no connections between them. We have to define a relationship between them by first setting a primary key for each table, then set up a foreign key to connect the tables. In order to set up foreign key relationships between two columns, we need to first set those columns to have the same datatype and constraint. There are two ways of achieving that, we can write queries to alter the tables, or we can use the GUI in MySQL Workbench.

```
# a new ID column with unique values is necessary for the primary key
ALTER TABLE weather
ADD weather_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;
# Alter table's datatypes of the columns and set primary key:
ALTER TABLE `gans`.`weather`
CHANGE COLUMN `precip_prob` `precip_prob` FLOAT NULL DEFAULT NULL,
CHANGE COLUMN `forecast_time` `forecast_time` DATETIME NULL DEFAULT NULL,
CHANGE COLUMN `temperature` `temperature` FLOAT NULL DEFAULT NULL,
CHANGE COLUMN `feels_like` `feels_like` FLOAT NULL DEFAULT NULL,
CHANGE COLUMN `humidity` `humidity` INT NULL DEFAULT NULL,
CHANGE COLUMN `cloudiness` `cloudiness` INT NULL DEFAULT NULL,
CHANGE COLUMN `wind_speed` `wind_speed` FLOAT NULL DEFAULT NULL,
CHANGE COLUMN `wind_gust` `wind_gust` FLOAT NULL DEFAULT NULL,
CHANGE COLUMN `city` `city` VARCHAR(100) NULL DEFAULT NULL,
CHANGE COLUMN `city_id` `city_id` VARCHAR(10) NULL DEFAULT NULL;
# Setting foreign key:
ALTER TABLE `gans`.`weather`
ADD INDEX `weather_city_id_idx` (`city_id` ASC) VISIBLE;
ALTER TABLE `gans`.`weather`
ADD CONSTRAINT `weather_city_id`
FOREIGN KEY (`city_id`)
REFERENCES `gans`.`cities` (`city_id`)
ON DELETE NO ACTION
ON UPDATE NO ACTION;
```





After all primary keys and foreign keys between the tables have been defined. We can generate an EER diagram to clearly visualize their relationship

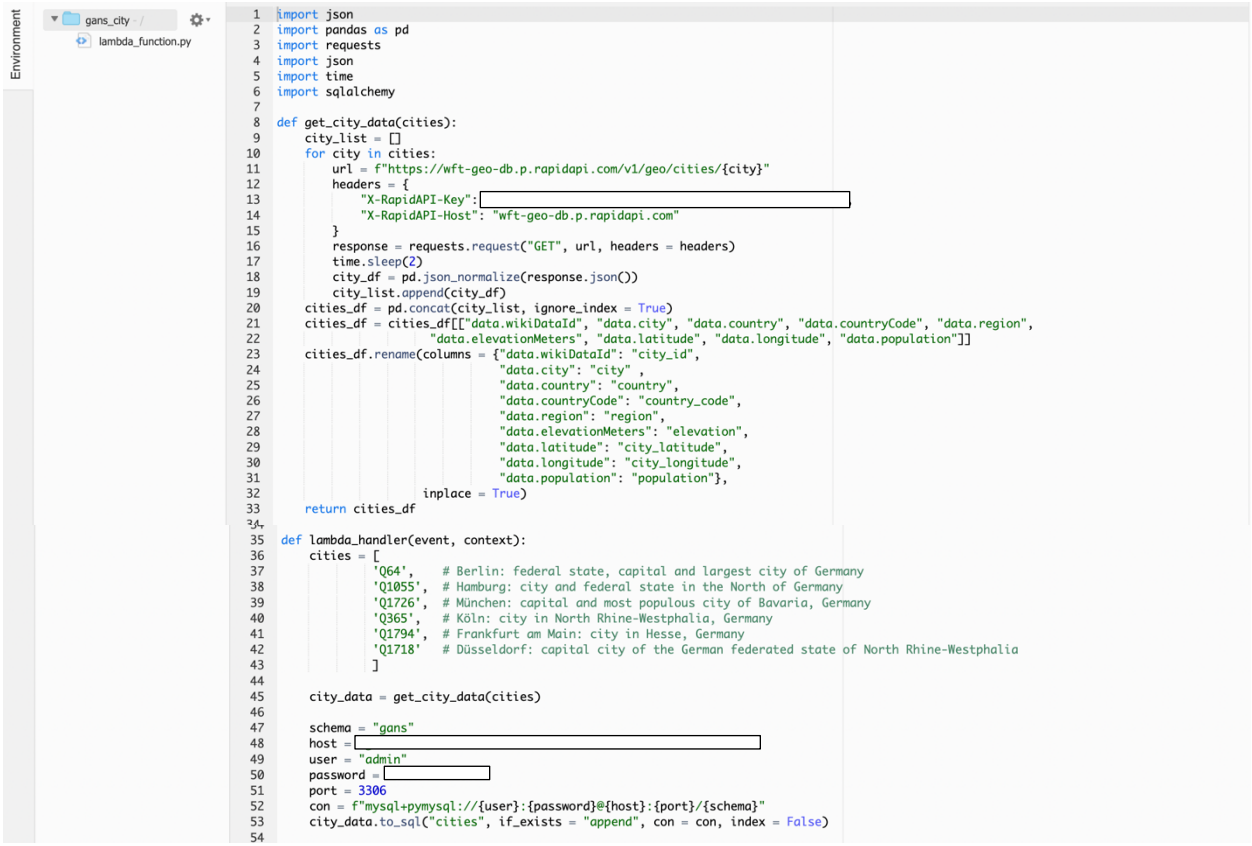


Cloud computing with AWS

In order to automate the process of data collection, we need to move the scripts that call the API to the cloud. AWS is a great service for doing so. We first set up a cloud instance of RDS on AWS, then we tested the connection by setting up a test table and pushed several rows of data to the RDS. Once we know the correct connection has been established. We are ready to move the scripts to the cloud by setting up Lambda functions on AWS.

Since every piece of code that runs on AWS has to be a function, we have rewritten our scripts such that the calls to the APIs are in the form of a function and the lambda handler will make the function call. The SQLAlchemy script also needs to be modified such that the host is the cloud database instance, not the local MySQL database instance. We did that by changing the host as the endpoint of the AWS RDS.

Once all the scripts have been changed, they can simply be copied from the local jupyter notebook and pasted to Amazon lambda function. Since these functions are run on the cloud, the packages that we imported to run these functions on our local machine has to be set up as layers. The AWS Data Wrangler has packages such as pandas and numpy. We set up SQLAlchemy as a K Layer by specifying an ARN found on developer Keith Rozario's github repo. We then configure the run time to be 1 minute or longer such that the function does not time out after 3 seconds.



```python
def get_weather_data(cities):
    weather_list = []
    for city in cities:
        url = f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid=[redacted]&units=metric"
        weather = requests.get(url)
        weather_df = pd.json_normalize(weather.json()["list"])
        weather_df["city"] = city
        weather_list.append(weather_df)
    weather_combined = pd.concat(weather_list, ignore_index = True)
    weather_combined = weather_combined[["pop", "dt_txt", "main.temp", "main.feels_like", "main.humidity", "clouds.all",
                                         "wind.speed", "wind.gust", "city"]]
    weather_combined.rename(columns = {"pop": "precip_prob",
                                       "dt_txt": "forecast_time",
                                       "main.temp": "temperature",
                                       "main.feels_like": "feels_like",
                                       "main.humidity": "humidity",
                                       "clouds.all": "cloudiness",
                                       "wind.speed": "wind_speed",
                                       "wind.gust": "wind_gust",},
                            inplace = True)
    return weather_combined

def lambda_handler(event, context):
    schema = "gans"
    host = [redacted]
    user = "admin"
    password = [redacted]
    port = 3306
    con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"

    city_df = pd.read_sql("select city_id, city from cities", con = con)
    city_list = city_df["city"].to_list()

    weather_data = get_weather_data(city_list)
    weather_data = weather_data.merge(city_df, how = "left", on = "city")

    weather_data.to_sql("weather", if_exists = "append", con = con, index = False)

```

Environment

gans_airport

lambda_function.py

```python
import json
import pandas as pd
import requests

import sqlalchemy

def icao_airport_code(latitude, longitude):
    airport_list = []
    assert len(latitude) == len(longitude)
    for i in range(len(latitude)):
        url = f"https://aerodatabox.p.rapidapi.com/airports/search/location/{latitude[i]}/{longitude[i]}/km/50/10"
        querystring = {"withFlightInfoOnly":"true"}
        headers = {"X-RapidAPI-Key": [redacted],
                   "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"
        }
        response = requests.request("GET", url, headers = headers, params = querystring)
        airport_df = pd.json_normalize(response.json()["items"])
        airport_list.append(airport_df)
    airports_df = pd.concat(airport_list, ignore_index = True)
    airports_df = airports_df[~airports_df.name.str.contains("Air Base", case = False)]
    airports_df.drop_duplicates(subset = "icao", inplace = True)
    airports_df.drop(columns = ["shortName"], inplace = True)
    airports_df.rename(columns = {"name": "airport_name",
                                  "municipalityName": "municipality_name",
                                  "countryCode": "country_code",
                                  "location.lat": "airport_latitude",
                                  "location.lon": "airport_longitude"},
                       inplace = True)
    airports_df["city_id"] = [
                              'Q64',
                              'Q64',
                              'Q1055',
                              'Q1726',
                              'Q365',
                              'Q1718',
                              'Q1794'
                              ]
    return airports_df

def lambda_handler(event, context):
    schema = "gans"
    host = [redacted]
    user = "admin"
    password = [redacted]
    port = 3306
    con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"

    city_geo = pd.read_sql("select city_latitude, city_longitude from cities", con = con)
    lat = city_geo["city_latitude"].to_list()
    lon = city_geo["city_longitude"].to_list()

    airport_data = icao_airport_code(lat, lon)

    airport_data.to_sql("airports", if_exists = "append", con = con, index = False)

```

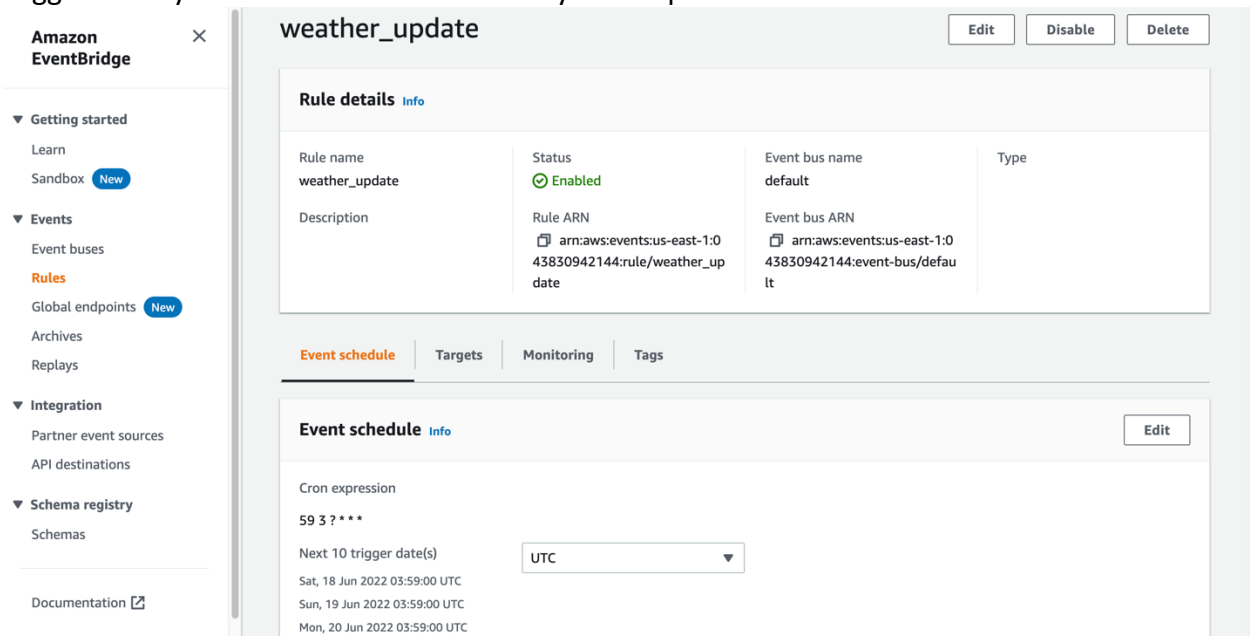
Environment

gans_flight /

lambda_function.py

```python
import json
import pandas as pd
import requests
from datetime import datetime, date, timedelta
from pytz import timezone
import sqlalchemy

def get_flight_data(icao):
    today = datetime.now().astimezone(timezone("Europe/Berlin")).date()
    tomorrow = (today + timedelta(days = 1))
    arrival_list = []
    for code in icao:
        url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{code}/{tomorrow}T10:00/{tomorrow}T22:00"
        querystring = {"withLeg":"false","direction":"Arrival","withCancelled":"false","withCodeshared":"true",
                       "withCargo":"false","withPrivate":"false","withLocation":"false"}
        headers = {"X-RapidAPI-Key": [redacted],
                   "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"
        }
        response = requests.request("GET", url, headers = headers, params = querystring)
        arrival_df = pd.json_normalize(response.json()["arrivals"])
        arrival_df["arrival_icao"] = code
        arrival_list.append(arrival_df)
    arrivals_df = pd.concat(arrival_list, ignore_index = True)
    arrivals_df.drop(columns = ["codeshareStatus", "isCargo", "movement.scheduledTimeUtc", "movement.quality",
                                "aircraft.reg", "aircraft.modeS", "callSign", "movement.actualTimeLocal",
                                "movement.actualTimeUtc"], inplace = True)
    arrivals_df.rename(columns = {"number": "flight_number",
                                  "movement.airport.icao": "departure_icao",
                                  "movement.airport.iata": "departure_iata",
                                  "movement.airport.name": "departure_airport",
                                  "movement.scheduledTimeLocal": "scheduled_time",
                                  "movement.terminal": "terminal",
                                  "aircraft.model": "aircraft_model",
                                  "airline.name": "airline"},
                       inplace = True)
    return arrivals_df

def lambda_handler(event, context):
    schema = "gans"
    host = [redacted]
    user = "admin"
    password = [redacted]
    port = 3306
    con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"

    icao_df = pd.read_sql("select icao from airports", con = con)
    icao_list = icao_df["icao"].to_list()

    flight_data = get_flight_data(icao_list)
    flight_data.to_sql("flights", if_exists = "append", con = con, index = False)
```

After setting up all the Lambda functions, we set up triggers via EventBridge. Once the conditions of the triggers are met, the event will happen. We set up the event using cron expression and specify the pattern to be everyday at 11:59pm local time. However, the scheduler is in UTC so we need to convert our desired local time to UTC before setting up the cron expression. Since only weather and flight data has to be updated everyday, we applied the trigger to only those two functions. The city and airport functions will remain static.



After this step is completed, our entire data pipeline is on the cloud and is automated. New data will be collected at our previously defined time and they will be pushed to our cloud instance RDS for further analysis. The entire data pipeline can be visualized as follows.