

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
22 ~ ALTER TABLE weather
23   ADD weather_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;
24   # Alter table's datatypes of the columns and set primary key:
25 • ALTER TABLE `gans`.`weather`
26   CHANGE COLUMN `precip_prob` `precip_prob` FLOAT NULL DEFAULT NULL,
27   CHANGE COLUMN `forecast_time` `forecast_time` DATETIME NULL DEFAULT NULL,
28   CHANGE COLUMN `temperature` `temperature` FLOAT NULL DEFAULT NULL,
29   CHANGE COLUMN `feels_like` `feels_like` FLOAT NULL DEFAULT NULL,
30   CHANGE COLUMN `humidity` `humidity` INT NULL DEFAULT NULL,
31   CHANGE COLUMN `cloudiness` `cloudiness` INT NULL DEFAULT NULL,
32   CHANGE COLUMN `wind_speed` `wind_speed` FLOAT NULL DEFAULT NULL,
33   CHANGE COLUMN `wind_gust` `wind_gust` FLOAT NULL DEFAULT NULL,
34   CHANGE COLUMN `city` `city` VARCHAR(100) NULL DEFAULT NULL,
35   CHANGE COLUMN `city_id` `city_id` VARCHAR(10) NULL DEFAULT NULL;
36   # Setting foreign key:
37 • ALTER TABLE `gans`.`weather`
38   ADD INDEX `weather_city_id_idx` (`city_id` ASC) VISIBLE;
39 • ALTER TABLE `gans`.`weather`
40   ADD CONSTRAINT `weather_city_id`
41     FOREIGN KEY (`city_id`)
42     REFERENCES `gans`.`cities` (`city_id`)
43     ON DELETE NO ACTION
        ON UPDATE NO ACTION;

```

Name: **airports** Schema: gans

Column	Datatype	PK	NN	UQ	B...	UN	ZF	AI	G	Default / Expression
icao	CHAR(4)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>						NULL
iata	CHAR(3)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
airport_name	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
municipality...	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
country_code	VARCHAR(5)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
airport_latitude	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
airport_longitude	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
city_id	VARCHAR(10)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

<click to edit>

Column details 'icao'

Column Name: **icao**

Charset/Collation: Default Charset Default Collation

Comments:

Datatype: **CHAR(4)**

Default:

Storage: VIRTUAL STORED

Primary Key Not NULL Unique
 Binary Unsigned ZeroFill
 Auto Increment Generated

Apply **Revert**

Columns Indexes Foreign Keys Triggers Partitioning Options

Name: **airports** Schema: gans

Foreign Key	Referenced Table
airports_city_id	'gans'`.`cities`

<click to edit>

Foreign key details 'airports_city_id'

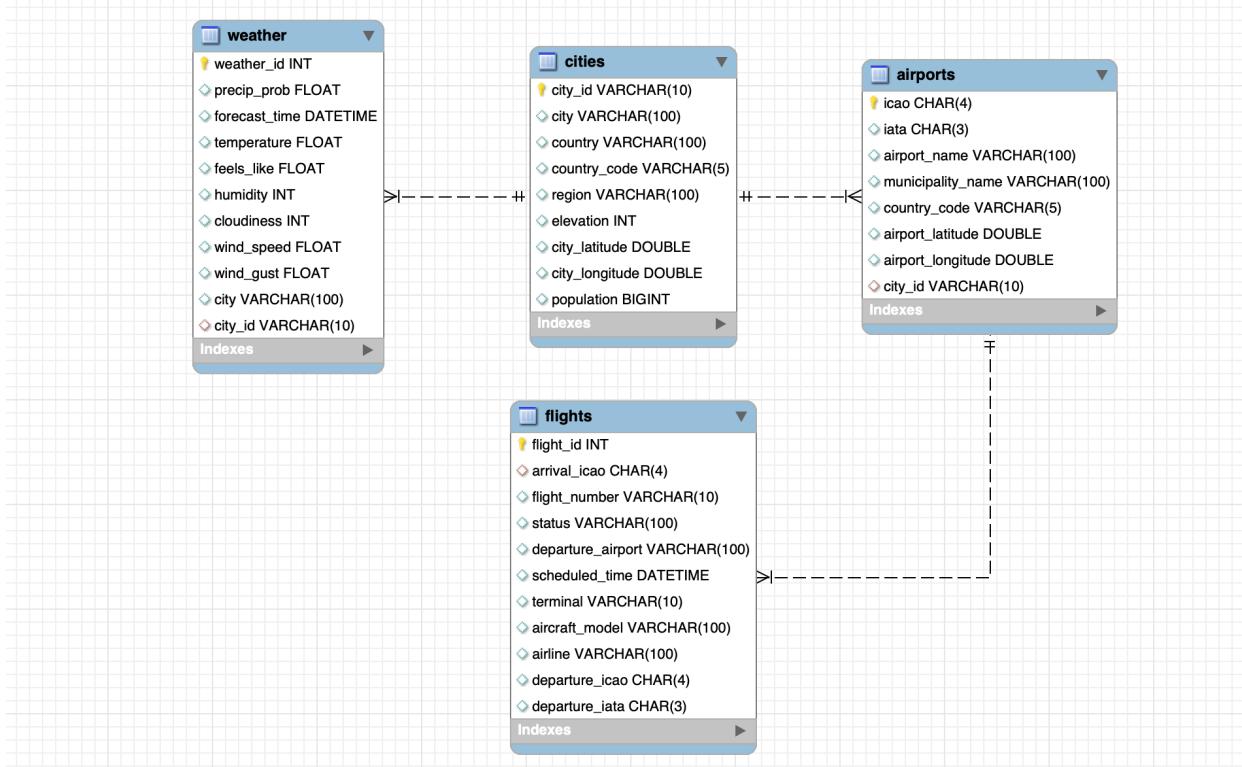
Column	Referenced Column
<input type="checkbox"/> icao	
<input type="checkbox"/> iata	
<input type="checkbox"/> airport_name	
<input type="checkbox"/> municipality_name	
<input type="checkbox"/> country_code	
<input type="checkbox"/> airport_latitude	
<input type="checkbox"/> airport_longitude	
<input checked="" type="checkbox"/> city_id	city_id

On Update: RESTRICT
On Delete: RESTRICT
Comment:

Skip on SQL generation

Columns | Indexes | Foreign Keys | Triggers | Partitioning | Options | Apply | Revert

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

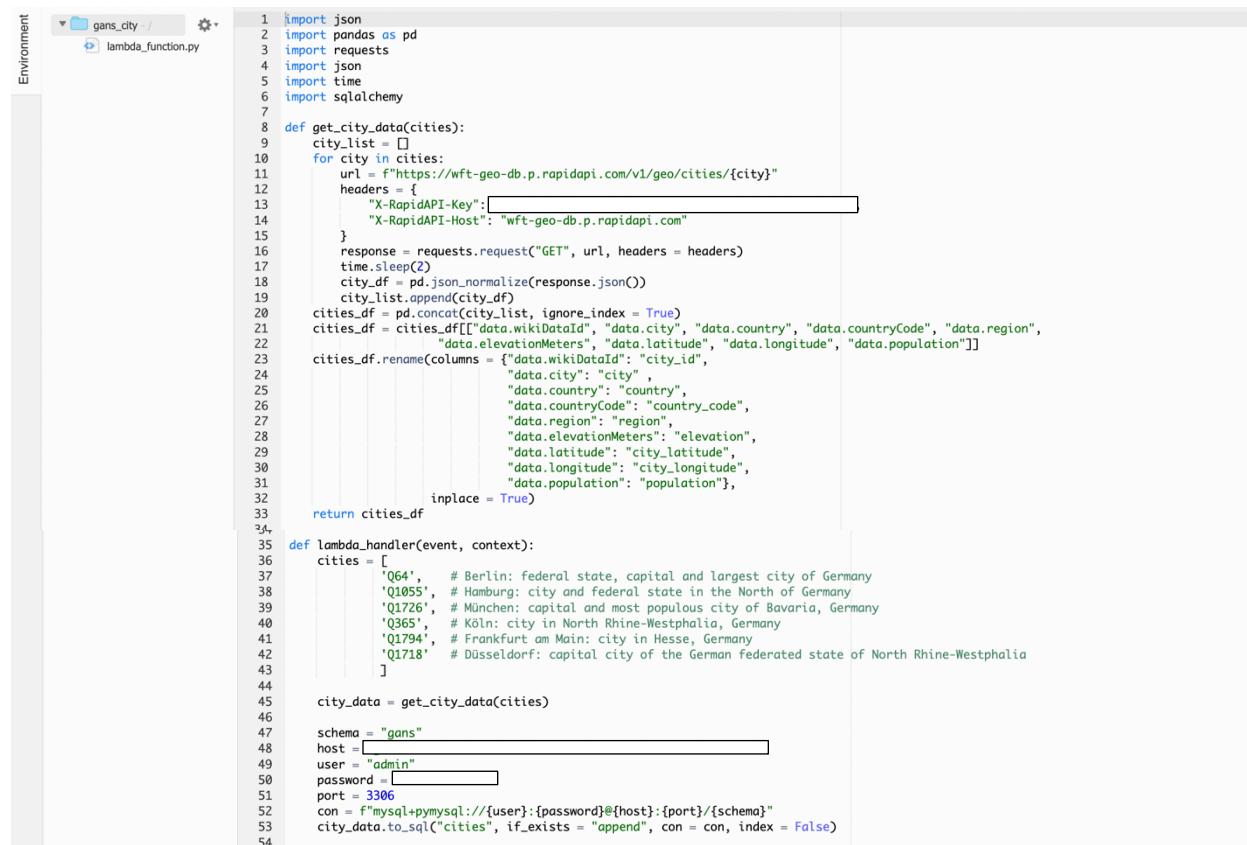


Cloud computing with AWS

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.



The screenshot shows the AWS Lambda function editor interface. On the left, there's a sidebar labeled "Environment" with a "gans_city" folder containing a file named "lambda_function.py". The main area displays the Python code for the Lambda function:

```
1 import json
2 import pandas as pd
3 import requests
4 import json
5 import time
6 import sqlalchemy
7
8 def get_city_data(cities):
9     city_list = []
10    for city in cities:
11        url = f"https://wft-geo-db.p.rapidapi.com/v1/geo/cities/{city}"
12        headers = {
13            "X-RapidAPI-Key": "REDACTED",
14            "X-RapidAPI-Host": "wft-geo-db.p.rapidapi.com"
15        }
16        response = requests.request("GET", url, headers = headers)
17        time.sleep(2)
18        city_df = pd.json_normalize(response.json())
19        city_list.append(city_df)
20    cities_df = pd.concat(city_list, ignore_index = True)
21    cities_df = cities_df[["data.wikiDataId", "data.city", "data.country", "data.countryCode", "data.region",
22                           "data.elevationMeters", "data.latitude", "data.longitude", "data.population"]]
23    cities_df.rename(columns = {"data.wikiDataId": "city_id",
24                           "data.city": "city",
25                           "data.country": "country",
26                           "data.countryCode": "country_code",
27                           "data.region": "region",
28                           "data.elevationMeters": "elevation",
29                           "data.latitude": "city_latitude",
30                           "data.longitude": "city_longitude",
31                           "data.population": "population"}, inplace = True)
32
33    return cities_df
34
35 def lambda_handler(event, context):
36    cities = [
37        'Q64', # Berlin: federal state, capital and largest city of Germany
38        'Q1055', # Hamburg: city and federal state in the North of Germany
39        'Q1726', # München: capital and most populous city of Bavaria, Germany
40        'Q365', # Köln: city in North Rhine-Westphalia, Germany
41        'Q1794', # Frankfurt am Main: city in Hesse, Germany
42        'Q1718' # Düsseldorf: capital city of the German federated state of North Rhine-Westphalia
43    ]
44
45    city_data = get_city_data(cities)
46
47    schema = "gans"
48    host = "REDACTED"
49    user = "admin"
50    password = "REDACTED"
51    port = 3306
52    con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"
53    city_data.to_sql("cities", if_exists = "append", con = con, index = False)
```

The code uses the AWS Data Wrangler library to interact with a MySQL database. It defines a function `get_city_data` that sends multiple GET requests to a RapidAPI endpoint for city data and concatenates the results into a single DataFrame. It then renames columns and returns the DataFrame. The `lambda_handler` function calls `get_city_data` with a list of city IDs and connects to a MySQL database using the `pymysql` library. The connection string is defined in the code, with host, user, and password fields redacted.

```

8 def get_weather_data(cities):
9     weather_list = []
10    for city in cities:
11        url = f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid=[REDACTED]&units=metric"
12        weather = requests.get(url)
13        weather_df = pd.json_normalize(weather.json()["list"])
14        weather_df["city"] = city
15        weather_list.append(weather_df)
16    weather_combined = pd.concat(weather_list, ignore_index = True)
17    weather_combined = weather_combined[["pop", "dt_txt", "main.temp", "main.feels_like", "main.humidity", "clouds.all",
18                                         "wind.speed", "wind.gust", "city"]]
19    weather_combined.rename(columns = {"pop": "precip_prob",
20                                "dt_txt": "forecast_time",
21                                "main.temp": "temperature",
22                                "main.feels_like": "feels_like",
23                                "main.humidity": "humidity",
24                                "clouds.all": "cloudiness",
25                                "wind.speed": "wind_speed",
26                                "wind.gust": "wind_gust"}, inplace = True)
27
28    return weather_combined
29
30 def lambda_handler(event, context):
31    schema = "gans"
32    host = [REDACTED]
33    user = "admin"
34    password = [REDACTED]
35    port = 3306
36    con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"
37
38    city_df = pd.read_sql("select city_id, city from cities", con = con)
39    city_list = city_df["city"].tolist()
40
41    weather_data = get_weather_data(city_list)
42    weather_data = weather_data.merge(city_df, how = "left", on = "city")
43
44    weather_data.to_sql("weather", if_exists = "append", con = con, index = False)
45

```

Environment

gans_airport - /

lambda_function.py

```

1 import json
2 import pandas as pd
3 import requests
4
5 import sqlalchemy
6
7 def icao_airport_code(latitude, longitude):
8     airport_list = []
9     assert len(latitude) == len(longitude)
10    for i in range(len(latitude)):
11        url = f"https://aerodatabox.p.rapidapi.com/airports/search/location/{latitude[i]}/{longitude[i]}/km/50/10"
12        querystring = {"withFlightInfoOnly": "true"}
13        headers = {"X-RapidAPI-Key": [REDACTED],
14                   "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"}
15
16        response = requests.request("GET", url, headers = headers, params = querystring)
17        airport_df = pd.json_normalize(response.json()["items"])
18        airport_list.append(airport_df)
19
20    airports_df = pd.concat(airport_list, ignore_index = True)
21    airports_df = airports_df[~airports_df.name.str.contains("Air Base", case = False)]
22    airports_df.drop_duplicates(subset = "icao", inplace = True)
23    airports_df.drop(columns = ["shortName"], inplace = True)
24    airports_df.rename(columns = {"name": "airport_name",
25                           "municipalityName": "municipality_name",
26                           "countryCode": "country_code",
27                           "location.lat": "airport_latitude",
28                           "location.lon": "airport_longitude"}, inplace = True)
29
30    airports_df["city_id"] = [
31        'Q64',
32        'Q64',
33        'Q1055',
34        'Q1726',
35        'Q365',
36        'Q1718',
37        'Q1794'
38    ]
39
40    return airports_df
41
42 def lambda_handler(event, context):
43    schema = "gans"
44    host = [REDACTED]
45    user = "admin"
46    password = [REDACTED]
47    port = 3306
48    con = f"mysql+pymysql://{user}:{password}@{host}:{port}/{schema}"
49
50    city_geo = pd.read_sql("select city_latitude, city_longitude from cities", con = con)
51    lat = city_geo["city_latitude"].tolist()
52    lon = city_geo["city_longitude"].tolist()
53
54    airport_data = icao_airport_code(lat, lon)
55
56    airport_data.to_sql("airports", if_exists = "append", con = con, index = False)

```

```

Environment gans_flight / lambda_function.py

1 import json
2 import pandas as pd
3 import requests
4 from datetime import datetime, date, timedelta
5 from pytz import timezone
6 import sqlalchemy
7
8 def get_flight_data(icao):
9     today = datetime.now().astimezone(timezone("Europe/Berlin")).date()
10    tomorrow = (today + timedelta(days = 1))
11    arrival_list = []
12    for code in icao:
13        url = f"https://aerodata.p.rapidapi.com/flights/airports/icao/{code}/{tomorrow}T10:00/{tomorrow}T22:00"
14        querystring = {"withLeg": "false", "direction": "Arrival", "withCancelled": "false", "withCodeshare": "true",
15                      "withCargo": "false", "withPrivate": "false", "withLocation": "false"}
16        headers = {"X-RapidAPI-Key": "REDACTED", "X-RapidAPI-Host": "aerodata.p.rapidapi.com"}
17
18        response = requests.request("GET", url, headers = headers, params = querystring)
19        arrival_df = pd.json_normalize(response.json()["arrivals"])
20        arrival_df["arrival_icao"] = code
21        arrival_list.append(arrival_df)
22    arrival_df = pd.concat(arrival_list, ignore_index = True)
23    arrivals_df = pd.concat([arrival_df, ignore_index = True])
24    arrivals_df.drop(columns = ["codeshareStatus", "isCargo", "movement.scheduledTimeUtc", "movement.quality",
25                           "aircraft.reg", "aircraft.mode5", "callSign", "movement.actualTimeLocal",
26                           "movement.actualTimeUtc"], inplace = True)
27    arrivals_df.rename(columns = {"number": "flight_number",
28                           "movement.airport.icao": "departure_icao",
29                           "movement.airport.iata": "departure_iata",
30                           "movement.airport.name": "departure_airport",
31                           "movement.scheduledTimeLocal": "scheduled_time",
32                           "movement.terminal": "terminal",
33                           "aircraft.model": "aircraft_model",
34                           "airline.name": "airline"}, inplace = True)
35
36    return arrivals_df
37
38 def lambda_handler(event, context):
39     schema = "gans"
40     host = "REDACTED"
41     user = "admin"
42     password = "REDACTED"
43     port = 3306
44     con = f"mysql+pymysql://user:{password}@{host}:{port}/{schema}"
45
46     icao_df = pd.read_sql("select icao from airports", con = con)
47     icao_list = icao_df["icao"].to_list()
48
49     flight_data = get_flight_data(icao_list)
50     flight_data.to_sql("flights", if_exists = "append", con = con, index = False)
51

```

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.

The screenshot shows the AWS Lambda function configuration for the 'weather_update' function. The code is identical to the one shown in the previous code block. The environment variables section contains:

- X-RapidAPI-Key: REDACTED
- X-RapidAPI-Host: aerodata.p.rapidapi.com
- host: REDACTED
- user: admin
- password: REDACTED
- port: 3306

The function is set to run on a schedule. The 'Event schedule' tab shows a cron expression of '59 3 ? * * *'. The next 10 trigger dates are listed as:

- Sat, 18 Jun 2022 03:59:00 UTC
- Sun, 19 Jun 2022 03:59:00 UTC
- Mon, 20 Jun 2022 03:59:00 UTC

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.

