# **ZILLOW REAL ESTATE DATA PIPELINE WITH AIRFLOW, AWS LAMBDA, REDSHIFT, AND QUICKSIGHT**

In this data engineering project, we demonstrate how to build and automate a Python-based ETL pipeline that extracts real estate property data from the **Zillow Rapid API**, processes it through AWS services, and visualizes it with **Amazon QuickSight**.

|  |  |
| --- | --- |
| Tool/Service | Purpose |
| Python | API call & scripting |
| Apache Airflow (on EC2) | DAG orchestration |
| BashOperator | Move files from EC2 to S3 |
| AWS Lambda | File copy & transformation automation |
| Amazon S3 | Staging, transforming, storing files |
| S3KeySensorOperator | Detect presence of file before Redshift load |
| Amazon Redshift | Data warehouse |
| Amazon QuickSight | BI tool for visualization |

1. **Python ETL Pipeline Design**

### ****Data Extraction & Staging:****

* **Zillow Rapid API** serves as the primary data source.
* Using **Python**, the API is queried to extract structured real estate property information such as number of bedrooms, bathrooms, price, rent estimates, property type, and location.
* The raw data is then loaded into an **Amazon S3 bucket** referred to as the **Landing Zone**.
  1. **S3 Zones & AWS Lambda Triggers:**
* Once data lands in the **Landing Zone**, it triggers a **Lambda function** that copies the data into an **Intermediate Zone bucket**.
* The purpose of this separation is to preserve the integrity of the raw data in the Landing Zone and prevent unintentional modifications. All further processing is done in the Intermediate Zone.
* A second **Lambda function** is triggered from the Intermediate Zone, which performs data **transformation** and **cleansing**, and writes the **transformed CSV** output to another S3 bucket – the **Transformed Data Zone**.

### ****Data Warehouse Integration:****

* The transformed CSV file is loaded into a provisioned **Amazon Redshift cluster**. Redshift serves as the cloud-based data warehouse for persistent storage and analytical querying of the real estate dataset.

### ****Data Visualization with QuickSight:****

* Once data resides in Redshift, **Amazon QuickSight** is connected to the Redshift cluster.
* This enables creation of dynamic visual dashboards to explore real estate market trends such as average price per ZIP code, distribution of home types, and bedroom-price relationships.

## **Apache Airflow Orchestration**

**Apache Airflow**, running on an **Amazon EC2 instance**, orchestrates the entire ETL process through a well-defined **Directed Acyclic Graph (DAG)**.

### ****DAG Components:****

* **PythonOperator**:  
  This task connects to the Zillow Rapid API and extracts the real estate data using Python. The data is first temporarily stored on the EC2 instance.
* **BashOperator**:  
  This task moves the extracted data from the EC2 local environment to the **S3 Landing Zone** bucket using shell commands.
* **AWS Lambda Triggers** (outside Airflow):  
  Once data is placed in the Landing Zone, a Lambda function is triggered to:
  1. Copy data to the Intermediate Zone.
  2. Trigger another Lambda function to transform the data and save it to the Transformed Zone.
* **S3KeySensor**:  
  Before loading data into Redshift, this sensor continuously monitors the **Transformed Data Zone S3 bucket** for the presence of the transformed CSV file. Only once the file is detected does the pipeline proceed.
* **S3ToRedshiftOperator**:  
  This task loads the transformed CSV data into a specified Redshift table using Redshift’s native COPY command, making the data ready for querying and visualization.

## **AWS Setup**

This section outlines the foundational setup of AWS services to support the Zillow Real Estate Data Pipeline project. All cloud infrastructure used in this project is provisioned within the **AWS ecosystem**, ensuring scalability, security, and seamless integration between services.

### ****IAM User Groups and Permissions****

In a production environment, it is a best practice **not to use the root user** for deploying and managing AWS resources. Instead, we:

1. **Create IAM User Groups** to manage permissions more effectively.
   * For instance, if certain engineers only need read access to an S3 bucket while others require full access, we can:
     + Create a **Read-Only Group**.
     + Create a **Read-Write Group**.
   * Permissions are assigned to the group, and users are added to the appropriate group rather than managing individual permissions manually.
2. **Create a Project-Specific IAM User**:
   * Assign this user to the **Administrative Group** created above.
   * Enable **AWS Management Console access** for the user.
   * Generate and download the **Access Key ID** and **Secret Access Key**. These credentials are required for programmatic access via AWS SDKs or the AWS CLI.
   * Log in using this newly created IAM user for all subsequent operations in the project.

### ****Launch Amazon EC2 Instance for Airflow****

To orchestrate the ETL pipeline using Apache Airflow, we set up an **EC2 instance**.

1. **Choose Amazon Machine Image (AMI)**:
   * Use **Ubuntu Server** for compatibility with Airflow and common Python tools.
2. **Select Instance Type**:
   * Avoid using t2.micro (Free Tier) as it lacks the necessary memory for Airflow.
   * Recommended:
     + t2.small (2 GiB) – may work with minor lag.
     + t2.medium (4 GiB) – ideal for smooth operation.
3. **Create or Select Key Pair**:
   * Key pairs are used for **SSH access** to the instance.
   * Note: These keys are not used for AWS API access (IAM credentials handle that).
4. **Configure Security Group**:
   * Create or modify a **security group** to allow essential traffic:
     + HTTP (port 80) – allow from anywhere (0.0.0.0/0)
     + HTTPS (port 443) – allow from anywhere
     + SSH (port 22) – allow from your IP or anywhere (only for testing/development)

### ****Connect to EC2 and Install Apache Airflow****

Once the EC2 instance is launched:

1. Go to the **AWS EC2 Console** > Select your instance > Click **Connect**.
2. Choose the **EC2 Instance Connect** tab to open a browser-based terminal.
3. From the terminal:
   * Start installing required **system dependencies** and **Python packages**.
   * Proceed with installing **Apache Airflow**, setting up a virtual environment, and preparing your project directory.

Bash commands:

sudo apt update

sudo apt install python3-pip

sudo apt update

sudo apt install software-properties-common

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt update

sudo apt install python3.12 python3.12-venv -y

sudo apt install python3.12-venv

python3 -m venv airflow\_venv

source airflow\_venv/bin/activate

pip install --upgrade awscli 3

pip install apache-airflow

airflow standalone

### ****Initialize Apache Airflow and Access the UI****

After installing Airflow on the EC2 instance:

###  **Access Airflow via Browser**

### **Visit: http://<your-ec2-public-ip>:8080**

### **Enter the username and password you created.**

### **Open Port 8080 on EC2 Security Group**

### **Go to EC2 Dashboard > Instances > Security Groups.**

### **Edit the inbound rules:**

### **Click Add Rule**

### **Type: Custom TCP**

### **Port Range: 8080**

### **Source: Anywhere (IPv4) or your specific IP for security**

### ****Connect EC2 to VS Code Using Remote SSH and Setup Airflow****

For better development experience, connect to your EC2 instance from **VS Code** using the **Remote - SSH extension**.

1. **Obtain SSH Command from EC2 Console**:
   * Go to **EC2 Dashboard > Connect > SSH Client**.
   * Copy the SSH command provided.
   * Ensure your .pem key file has the correct permission:
     + Bash: chmod 400 airflow-key.pem
2. **Configure VS Code for Remote Access:**

* Install the **Remote - SSH** extension in VS Code.
* Open a remote window (bottom-left green icon).
* Select: **Connect to Host > Configure SSH Hosts**.
* Add the following entry in your SSH config file:

Host airflow-ec2

HostName <Your EC2 Public IP>

User ubuntu

IdentityFile /path/to/your/airflow-key.pem

1. **Connect to Your Instance:**

* In the **Remote Explorer**, click **Connect to Host**.
* Choose your configured host (e.g., airflow-ec2).
* VS Code will now open your EC2 file system (/home/ubuntu/).

1. Configure Airflow to Use the DAG Folder

* Once connected to your EC2 environment via VS Code, navigate to the Airflow Directory and create *dags* folder:

cd /home/ubuntu/airflow

* Create Your First DAG File inside dags folder: zillow\_analytics.py
* In the airflow folder, locate and open airflow.cfg, **update DAGs Folder Path:**

dags\_folder = /home/ubuntu/airflow/dags

load\_examples = False

1. Restart Airflow to Apply Changes

### ****Create a RapidAPI Account****

* The project requires connecting to a data API.
* Instead of hunting for zillow data APIs, use **RapidAPI** — a popular platform that provides access to many APIs from one place.
* Create a free account on [RapidAPI.com](https://rapidapi.com).
* RapidAPI provides a **mock zillow data API** that mimics the real zillow data structure — perfect for your development and testing.
* This way, you get consistent, API-driven zillow data without worrying about sourcing or hosting it yourself.

### Using RapidAPI with the Zillow API:

1. **Subscribe to the API**
   * After logging into RapidAPI, you need to **subscribe** to the specific API you want to use before you can call it.
   * Go to the API’s page (e.g., the Zillow API), and click **Subscribe to Test** or a similar button.
   * You must subscribe even if the API is free to enable test requests.
2. **Finding the Right Endpoint**
   * Navigate to the **Endpoints** tab on the API page.
   * For this project, use the "search for properties" endpoint or whichever endpoint is relevant to your data needs.
3. **Testing the Endpoint**
   * RapidAPI provides **code snippets** in multiple languages.
   * Select **Python** and then choose the **requests** code snippet.
   * You can click **Test Endpoint** on the RapidAPI UI to see a live response from the API.
4. **Understanding Response**
   * You can view **example responses** to understand the structure of the JSON data you’ll get.
   * Typical property data will include fields like bedrooms, bathrooms, city, country, etc.

**Key Components from the Code Snippet**

**URL**: The API endpoint URL (important for your request).

**Query Parameters**: Contains the location (e.g., "location": "Houston, TX"), which you can change to any city/state of interest.

**Headers**: Includes X-RapidAPI-Key and X-RapidAPI-Host which authenticate your request on RapidAPI.

* + **Making the Request in Code**  
    You’ll use something like:

response = requests.get(URL, headers=headers, params=querystring)

data = response.json()

1. **Create a JSON File for API Keys**

* Inside your Airflow directory on the EC2 instance, create a new file named config\_api.json to store your API credentials:

Path:  
/home/ubuntu/airflow/config\_api.json

{

"x-rapidapi-key": "52c02bf380msh3e00a1cabc82761p197028jsnfcef6f371c79",

"x-rapidapi-host": "zillow56.p.rapidapi.com"

}

* The Python json.load() reads your API keys into the config dictionary. Your Python callable (extract\_data\_from\_rapidapi()) uses those keys from config\_api.json

**Important:**

* Add this **config\_api.json** file to your .gitignore if you are using git, so it won’t be pushed to any repo.
* If you are using an EC2 instance or server, make sure this file is securely stored and not exposed to others.

Use Airflow Variables or Secrets Manager (AWS Secrets Manager or HashiCorp Vault) for production-grade secrets management

## **Creating the First Airflow Task – API Extraction with PythonOperator**

In this step, we implement the **first task in our Airflow DAG**, which is responsible for extracting Zillow data using the RapidAPI.

*Refer to scripts/airflow/dags/zillow\_analytics.py*

### ****6.1. Define DAG Defaults and Instantiate DAG****

Before creating tasks, you must define a set of default arguments (default\_args) and instantiate the DAG:

* **default\_args**:
  + Specify metadata such as task owner, retry policy, and retry delays.
  + These are applied to all tasks unless overridden.
* **DAG Object**:

Define the DAG’s ID, schedule interval, start date, description, and other DAG-level parameters

### ****6.2. Create a Python Callable for API Extraction****

Define a Python function that connects to the **Zillow RapidAPI** endpoint using the requests library:

* This function will be triggered by a PythonOperator.
* It reads API credentials from config\_api.json.
* It writes the API response to a JSON file in your EC2 instance.

The Python callable is the **core logic** of the task. Airflow invokes this function when the task runs.

### ****6.3. Define the PythonOperator in DAG****

Create a PythonOperator that references your callable function:

* The operator is responsible for executing the API call and saving the result to /home/ubuntu/airflow/.

### ****6.4. Trigger and Monitor the DAG in Airflow UI****

* In the Airflow UI, locate your DAG (after it appears).
* **Trigger DAG** button. This will run the task immediately, regardless of the DAG's schedule.
* Use the graph view or tree view in the Airflow UI. Task status colors:
  + **Green** = Success
  + **Yellow** = Running
  + **Red** = Failed
* If the task fails, click on the task box in the graph view.
* Go to the **Logs** tab to view detailed error messages.
* SSH into your EC2 instance.
* Navigate to the output directory:

cd /home/ubuntu/airflow/

ls

* Confirm the presence of the output JSON file containing the extracted data.

1. **Create a Landing S3 bucket in AWS Console** 
   * + Log into your AWS Management Console.
     + Go to the S3 service.
     + Click Create bucket.
     + Give your bucket a globally unique name.
     + Choose a region (preferably the same as your EC2 instance for cost and speed efficiency).
     + Leave other options default or customize based on your needs.
     + Attach an IAM Role to your Airflow EC2 instance
       - Create an IAM Role with AmazonS3FullAccess (or least privilege for your bucket).
       - Attach the role to your EC2 instance.

### Add the Second Task in Your DAG to Upload the File to S3

After extracting data from the Zillow API using a PythonOperator, the next step is to upload the resulting file to Amazon S3 using a BashOperator. The file path produced by the first task is passed dynamically to the second task using Airflow's XCom system.

#### Task Breakdown

* **PythonOperator** returns a list of output file path and file name (e.g., ["/home/ubuntu/airflow/zillow\_data.json"]).
* **XCom** (Cross-communication) allows tasks to exchange data.
* **BashOperator** uses **returned list** values to access the path and file name returned by the first task and construct a dynamic aws s3 mv command.
* **That copies the file from Airflow EC2 environment to landing S3 bucket**

1. **AWS Lambda Functions**

In the previous parts, real estate data was extracted from the **Zillow Rapid API** using a Python-based ETL script. The raw data was initially saved to an **Amazon EC2 instance** and then moved into an **S3 bucket (Landing Zone)** using a **BashOperator** defined in **Apache Airflow**.

In this part, the focus shifts to automating the next stage of the pipeline using **AWS Lambda**:

* A **Lambda function** is triggered automatically whenever a new file land in the **Landing Zone** S3 bucket.
* This function **copies the raw JSON file** into a **second S3 bucket**, referred to as the **Intermediate Zone**.
* The purpose of this step is to **preserve a clean, untouched version of the raw data** before any transformations occur.
* This separation ensures **data integrity** and provides a backup of the original API response.

This design promotes modularity, traceability, and auditability in the pipeline while following data engineering best practices.

**9.1 Function 1: copy-json-to-intermediate-bucket**

1. **Create a New Lambda Function:**
   * Function name: copy-json-to-intermediate-bucket.
   * Runtime: Select Python 3.12
   * Create IAM Role to attach to Lambda Function with permissions:
     + - * **AmazonS3FullAccess**
         * **AWSLambdaBasicExecutionRole** (grants access to write logs to CloudWatch)
2. **Set up S3 Trigger for Lambda:**
   * In the newly created Lambda function, go to **Add trigger**.
   * Choose **S3** as the source.
   * Select the **source S3 bucket** (where the Airflow task places the raw JSON file)
   * Event type: **PUT** (object created).
   * This means: When a new object is uploaded, trigger this Lambda function.
3. **Write Lambda Code** 
   * Use Python’s boto3 and json libraries (pre-installed in Lambda).
   * The function receives two arguments: event and context. Only event is used to extract the source bucket and object key. You can see event structure by choosing event type (s3 put in this case) under Test.

* These are accessed from the deeply nested JSON object passed to the function by the S3 trigger.

1. **Set Lambda Timeout and Memory (Important):**

* In your Lambda function configuration, click **Edit** next to Basic settings.
* Increase the **Timeout** to at least **1 minute (60 seconds)** to allow enough time for large file processing.
* Optionally increase **Memory** to 256 MB or 512 MB depending on data volume.

1. **Deploy and Test Lambda Trigger:**

* Deploy the function using the **Deploy** button.
* Upload a CSV or JSON file into the source S3 bucket. Lambda function is automatically triggered. The function extracts the source bucket and object key from the event JSON.
* These values are printed and used to construct a copy operation.
* The function waits until the object fully exists in the bucket (to avoid copying partial files). Once confirmed, the file is copied to the target S3 bucket.
* Logs can be reviewed in CloudWatch > Logs because the Lambda function is using an IAM role with the AWSLambdaBasicExecutionRole policy, which grants permission to write logs to CloudWatch
  + The Lambda function should:
    - Read the uploaded JSON file from the source bucket.
    - Copy it to the destination (Intermediate Zone) S3 bucket.
    - Preserve the file name and structure.

**9.2 Function 2: transform-json-to-csv-lambda**

#### ****1. Create Second Lambda Function — Transformation Lambda****

To transform the raw JSON into a structured CSV file, follow these steps:

* + Name: transform-json-to-csv-lambda
  + Runtime: Python 3.12
  + IAM Role: Use the same role created earlier with necessary S3 permissions
  + **Timeout**: Set to at least **60 seconds**
  + **Memory**: Allocate **256 MB to 512 MB**, depending on file size
  + **Add Trigger for Lambda Function**
    - **Trigger Source**: S3
    - **Bucket**: copy-of-raw-json-intermediate-bucket (Intermediate Zone)
    - **Event Type**: All object create events

This ensures that whenever a new JSON file is copied into the Intermediate Zone, this Lambda function is automatically triggered to perform the transformation.

#### ****Create S3 Bucket for Clean Data****

This is where the output CSV files will be stored after transformation.

#### Name: transformed-json-data-to-csv-bucket

#### ****Lambda Transformation Logic****

* **Read** the .json file from the intermediate bucket
* **Parse** the contents to extract records from the results key
* **Transform** into a pandas DataFrame
  + Filter relevant fields like bedrooms, bathrooms, homeType, price, etc.
* **Convert** to .csv using to\_csv(index=False)
* **Upload** the transformed CSV to the transformed-json-data-to-csv-bucket

#### ****Install Required Libraries using Lambda Layers****

Since pandas is not available in Lambda by default:

* Go to **Lambda > Layers > Add a Layer**
  + Choose from AWS layers
  + Select **AWSDataWrangler-Python3.12** or a compatible layer that includes pandas

Lambda Layers modularize dependencies and improve performance while reducing function package size.