Combining and Enriching Data with Amazon Managed Workflows for Apache Airflow

Description

Amazon Managed Workflows for Apache Airflow (MWAA) is a secure and highly available workflow orchestration tool. Using Amazon MWAA saves you from the technical complexity of creating, managing, and configuring the servers and other resources required by an Apache Airflow environment.

Learning how to create and use Amazon MWAA will make you more effective at creating and working with data processing systems in the AWS public cloud.

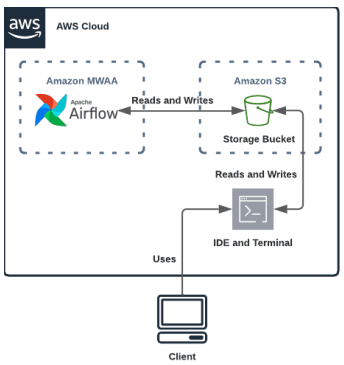
In this hands-on lab, you will tour the Amazon MWAA environment creation options, create a data source for use with Amazon MWAA, and you will create a workflow and run it in Amazon MWAA.

Learning Objectives

Upon completion of this intermediate level lab, you will be able to:

- Understand the requirements and options for a new Amazon MWAA environment
- Update and configure an AWS Lambda function
- Use an IDE to create a directed acyclic graph (DAG) in Python
- Use Apache Airflow to run your DAG

Environment after



Lab steps

- Logging In to the Amazon Web Services Console
- Touring the Amazon Managed Workflows for Apache Airflow Console
- Creating a Data Source
- Creating a Directed Acyclic Graph

- Running your Directed Acyclic Graph

Touring the Amazon Managed Workflows for Apache Airflow Console

Introduction

Apache Airflow is a popular platform for building, running, scheduling, and monitoring complex workflows. It's aimed at people working with big datasets that have a need to apply multiple stages of processing that comprise a pipeline. Whilst it can be used in an ad-hoc manner for once-off tasks, it's most suitable for data processing pipelines that:

- Have many discrete elements
- Are expected to be re-run often as new data becomes available
- Should be easy to modify or update
- May need to scale significantly

Amazon Managed Workflows for Apache Airflow (often shorted to MWAA) is a fully managed service that saves you from having to deal with the technical complexity of:

- Provisioning servers
- Configuring Apache Airflow
- Scaling the components of Apache Airflow
- Ensuring high availability

Uses-cases for Apache Airflow include:

- Extract, transform, and load (ETL) tasks
- Training machine learning models
- Generating insights from datasets
- Any kind of batch processing task

Although primarily aimed at machine learning and ETL, Apache Airflow can potentially be used in many other situations where automation and scheduling are key requirements, including typical DevOps activities such as making and storing backups of data.

Amazon MWAA is similar in purpose to AWS Step Functions. Both allow you to create complex pipelines that can automate significant and regular data processing tasks. In comparison, Amazon MWAA is:

- More code focused
- More configurable
- Fully-managed but not serverless like AWS Step Functions
- Less easy to integrate with other AWS services

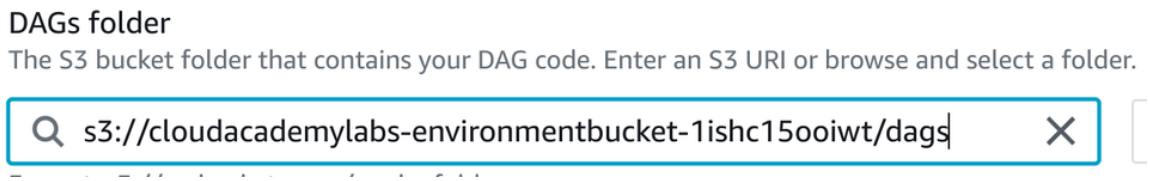
Which service you should choose will be dependent upon the data pipeline requirements of you and your organization.

In this lab step, you will tour the Amazon MWAA console and see how to use it to configure an environment.

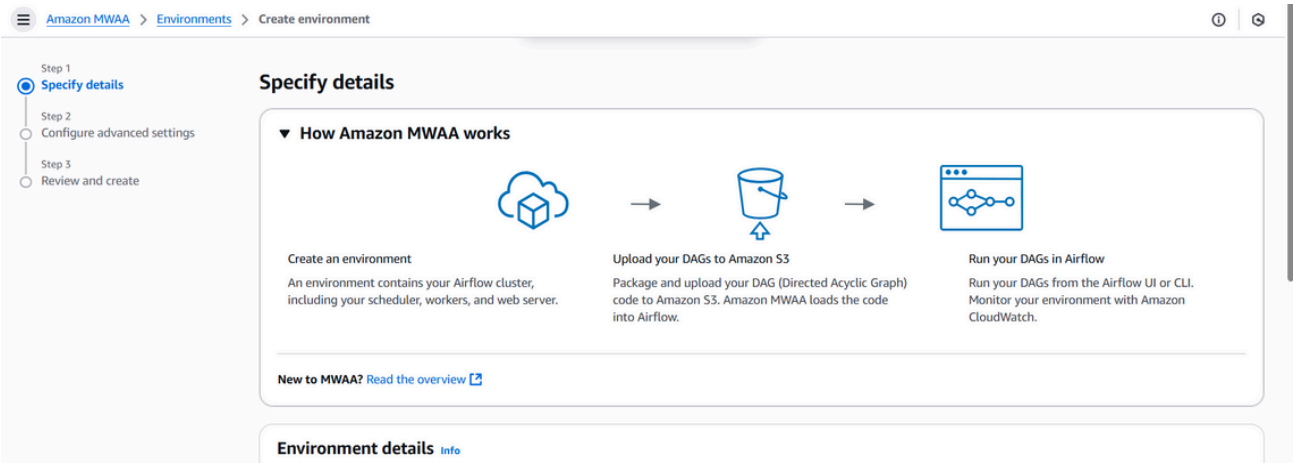
Note: This lab can take up 25 minutes for lab setup to complete. You can proceed and complete this and the next lab step whilst the environment setup is still in progress.

Instructions

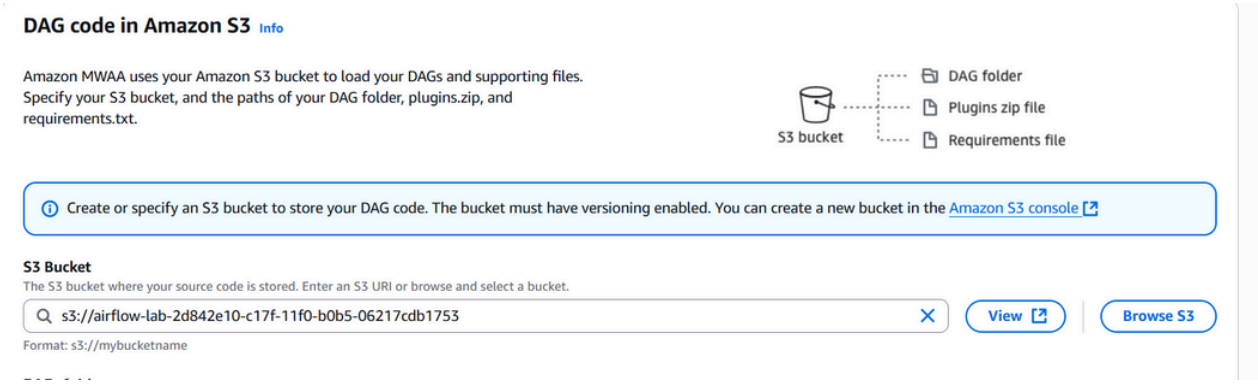
1. In the Amazon Management Console, in the search bar at the top, enter *MWAA*, and under **Services**, click the **Managed Apache Airflow** result
2. To navigate to the environments page, in the left-hand menu, click **Environments**
3. *Note*: If the **Airflow envrionments** page loads by default, proceed to the next instruction.
4. Amazon MWAA uses the term Environment to denote an instance of the Apache Airflow platform. An environment includes the following Apache Airflow components:
   - A web server that provides a user interface for creating workflows
   - Workers that perform tasks
   - Schedulers that assign tasks to workers
   - A database to store metadata
5. In addition, an Amazon MWAA environment is backed by the following AWS services:
   - Amazon CloudWatch for storing logs
   - Amazon Simple Storage Service (S3) for storing code, data, and other needed storage
   - Amazon Simple Queue Service (SQS) for queuing tasks
   - Optionally Amazon MWAA can use Amazon Elastic Container Registry (ECR) for docker image storage
   - AWS Key Management Service (KMS) for encrypting data
6. This lab is creating a Managed Apache Airflow environment for you. Environment creation can take up to 25 minutes to complete.
7. In this lab step, you will walk through the process of creating a new environment to see the available configuration options before cancelling the creation of a second new environment.



1. To load the environment creation wizard, in the top-right corner, click **Create environment**
2. A multi-step wizard will load.
3. The shows a high-level diagram about how Amazon MWAA works.
4. Observe the **Environment details** section. As well as naming your environment, you can specify the version of Apache Airflow Amazon MWAA should use. By default, the version is the latest available supported by Amazon MWAA.
5. To learn more about how the service supports Apache Airflow versions, including the full list of supported versions, see the Apache Airflow versions on Amazon Managed Workflows for Apache Airflow (MWAA), opens in a new tab page of the Amazon MWAA User Guide.



1. Scroll to the bottom of the page and observe the **DAG code in Amazon S3** section
2. DAG is an acronym, short for Directed Acyclic Graph. A DAG is a graph that cannot contain cyclical closed loops. This property makes them popular for defining workflows as there is no scope for looping indefinitely.
3. DAGs are a core part of the data structures used by the well-known source control system Git.
4. This section of the form allows you to specify an Amazon S3 bucket to store code that defines a DAG.
5. To open a bucket selection dialog, click **Browse S3**
6. Select the bucket whose name begins with **airflow-lab-** and click **Choose**:



1. To specify a folder for DAG code, copy the value of the **S3 Bucket** field, paste it into the **DAGs folder** textbox, and append */dags* to it:

1. *Note*: Your bucket name will be different due to Amazon S3 bucket names being required to be globally unique.
2. You have filled in this field because it's required to proceed to the next step.
3. At the bottom of the page, click **Next**, to proceed
4. The **Configure advanced settings** page will load.
5. The following describes the options available on this page

6. **Networking**

- [ ] Amazon MWAA must be deployed into a Virtual Private Cloud (VPC) and supports two kinds of deployments, private or public. You must create and configure an appropriate VPC before creating an Amazon MWAA environment.
- [ ] Once an environment is deployed, it can be accessed in two ways, through the Apache Airflow web user interface, or through a web API. Both require you to authenticate using AWS IAM.
- [ ] A private network deployment should be selected when you want maximum control over access to the Amazon MWAA environment. This may be due to organization based or compliance based security requirements. A private network deployment will require the host VPC to have VPC endpoint resources created for accessing services such as Amazon S3 and Amazon SQS.
- [ ] There are three main ways that you can use to access a private network environment:
  - [ ] AWS Client VPN
  - [ ] Bastion server
  - [ ] Load balancer
- [ ] Details of these methods are out of scope for this lab. But be aware that because Amazon MWAA uses AWS IAM to authenticate access, configuration to integrate this is required for access to a private network environment.
- [ ] To learn more about accessing a private MWAA environment, see the Accessing the VPC endpoint for your Apache Airflow Web server (private network access) section of the Amazon MWAA User Guide.
- [ ] A public network environment requires that the host VPC has private subnets with NAT Gateways or other NAT devices to provide access to Amazon services through network address translation. VPC endpoints aren't necessary for public Amazon MWAA environments.
- [ ] The full details of the host VPC's requirements can be found in the Amazon MWAA User Guide on the About networking on Amazon MWAA page.

7. **Environment class**

- [ ] The class of the environment dictates how much compute resource the Apache Airflow components have. You can also set the maximum number of workers that Amazon MWAA can create.
- [ ] This section determines how much your Amazon MWAA environment will cost you to run. For full pricing and examples, see the Amazon Managed Workflows for Apache Airflow Pricing page.
- [ ] Be aware that the cost of the host VPC, including bandwidth charges and NAT Gateways or instances (required for a public environment), are not included and need to be accounted for separately.

8. **Encryption**

- [ ] If you want to use your own KMS key to encrypt Amazon MWAA data, you can specify that in this section, or you can use the default key that Amazon manages for you.
- [ ] Using a custom key may be required to satisfy compliance and regulatory requirements.

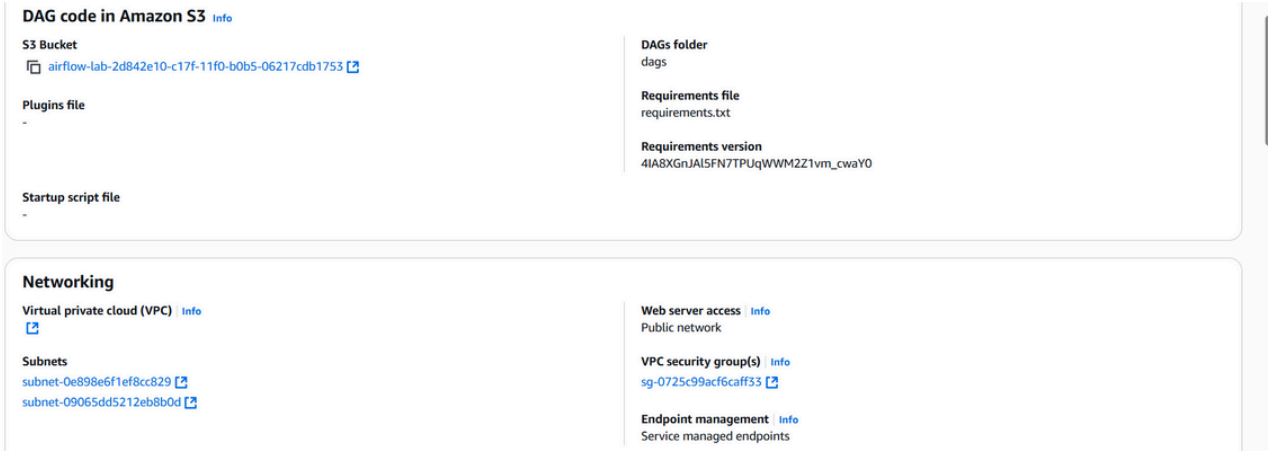
9. **Monitoring**

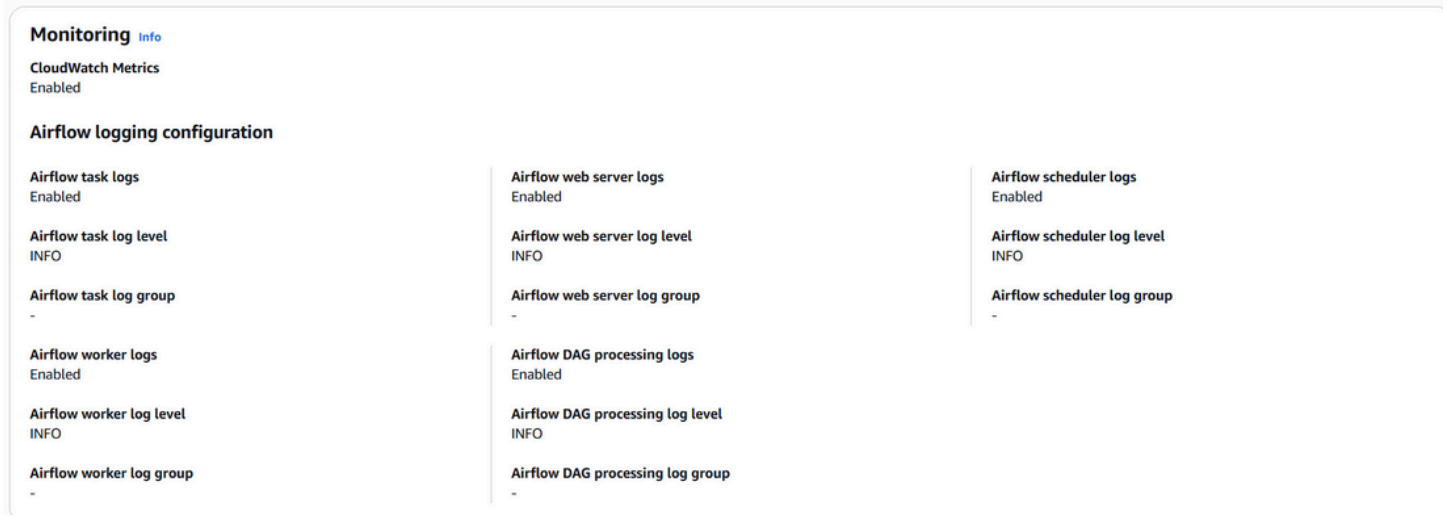
- [ ] Amazon MWAA uses Amazon CloudWatch to store logs. Here you can configure the amount of the logging and enable or disable logging for specific Apache Airflow components.

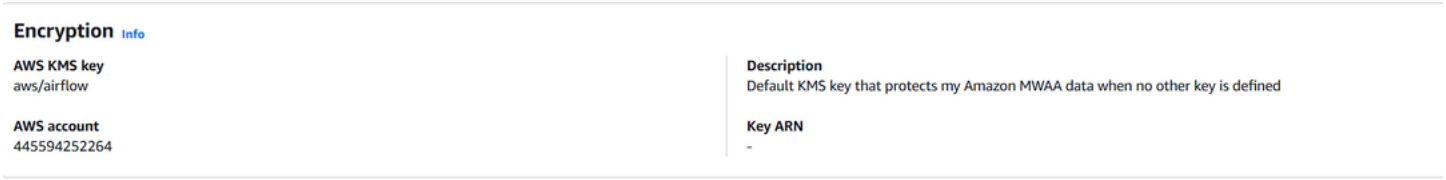
10. **Airflow configuration options**
- [ ] Using this section you can define Apache Airflow specific configuration options
- [ ] Feel free to click **Add custom configuration value** and then click in the **Configuration option** field to see the available options.
- [ ] **Tags**
- [ ] Tagging your AWS resources is a best practice that helps to make managing resources, finding and filtering them, and analysing resource usage easier.

11. **Permissions**
- [ ] An Amazon MWAA environment requires access to other AWS resources. You can restrict the access it has by specifying an AWS IAM role here.
- [ ] The role's requirements are documented on the Amazon MWAA execution role page of the Amazon MWAA User Guide.

12. To cancel the creation of a new environment, scroll to the bottom and click **Cancel**
13. The final step of the wizard is the **Review** step. It would have shown options that have been selected prior to creating a new environment. The review page doesn't show any new options.
14. You will be returned to the **Airflow environments** page and you will see an environment listed named **airflow-lab-env**. Since an environment can take over 20 minutes to create, you may still see the environment's status as **Creating**.
15. To see the details of this lab's environment, in the **Environments** table, click **airflow-lab-env**
16. This page displays the configuration of the Amazon MWAA env that has been created for you in this lab. Feel free to scroll down and observe the configuration options.







1. You can proceed to the next lab step even if the **Status** field is not **Available** yet.
2. Leave this browser tab open, you will return to it later in the lab to access the Apache Airflow user interface.

Summary

In this lab step, you toured the Amazon MWAA creation form wizard and learned about the requirements and configuration options of a new Amazon MWAA environment.

Creating a Data Source

Introduction

Amazon Managed Workflows for Apache Airflow is designed for pipelines that comprise of many different tasks. Examples of typical tasks include:

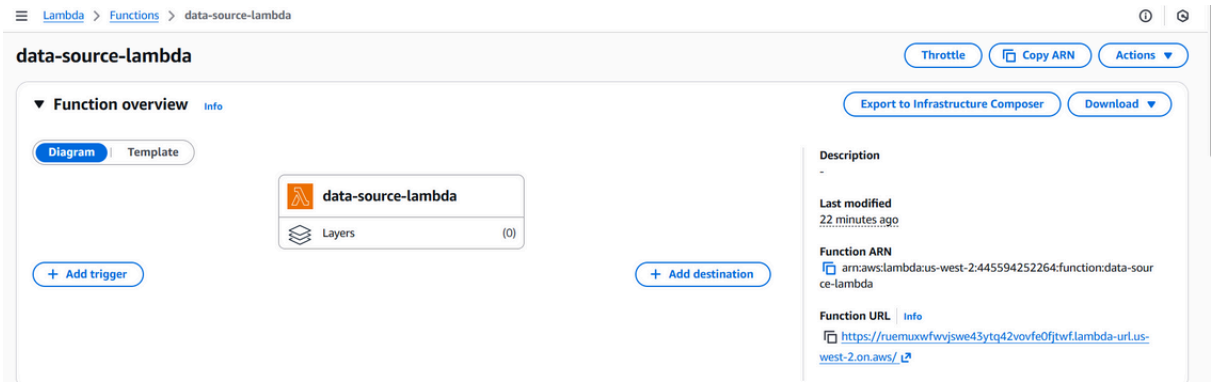
- Fetching data
- Transforming data
- Cleansing data
- Creating new assets from data such as training machine learning models

To use Amazon MWAA, you need a source of data to work with.

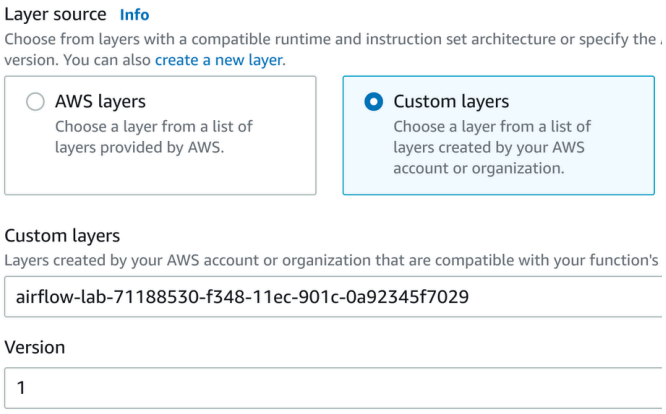
In this lab step, you will configure an AWS Lambda function to provide randomly generated test data.

Instructions

1. In the AWS Management Console, in the search bar at the top, enter *Lambda*, and under **Services**, click the **Lambda** result
2. The **Functions** page will load and you will see a function listed with a named **data-source-lambda**. This is a Python function that has been pre-created for you in this lab.
3. To navigate to the function's details page, click the function named **data-source-lambda**:



1. A page will load with **Function overview** at the top, and a row of tabs below.
2. Next, you will add a layer to the AWS Lambda function that provides dependencies.
3. To add a layer to this function, scroll down to the **Layers** section at the bottom of the page, and on the right-hand side, click **Add a layer**:
4. The **Add layer** form page will load.
5. A Lambda layer is a mechanism provided by AWS that enables you to add code to an AWS Lambda function. In this case you are using it to add a Python library named faker that is useful for generating sample test data.
6. Select the following:
   - **Layer source**: Select **Custom layers**
     - *Note*: The other options won't be visible until you select this
   - **Custom layers**: Select the only available layer whose name begins with **airflow-lab-**
   - **Version**: Select the only available version



1. To finish adding a layer, at the bottom of the page, click **Add**
2. You will be returned to the AWS Lambda function's details page.
3. Scroll to the **Code source** section of the page and observe the contents of the text editor:
4. This is the default code that exists when a new Python AWS Lambda function is created from scratch
5. Replace the contents of the **index.py** code editor with the following:

```
import json, calendar
```

```
from datetime import datetime, date
from faker import Faker
from random import randrange, choice


CUSTOMERS = 20
ORDERS = 50
REVIEWS = 30


REVIEW_LINES = [
    [
        "This is an excellent product, I would happily recommend it.",
        "The product works as expected, no major issues.",
        "I'm unsatisfied, this product broke after only using it for two days.",
    ],
    [
        "The company's sales and support was very good.",
        "I didn't need to contact the company after placing my order.",
        "I sent the company a message and it took a long time to get a response.",
    ],
    [
        "I'm satisfied and would order again.",
        "The cost was reasonable, I may order again.",
        "I would not order this product again.",
    ],
]
```

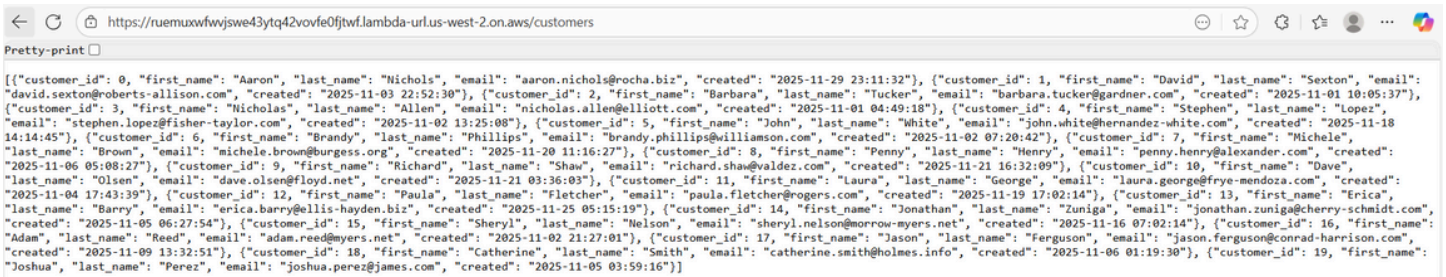
```
YEAR = date.today().year

MONTH = date.today().month

LAST_DAY = calendar.monthrange(YEAR, MONTH)[1]


fake = Faker()



def lambda_handler(event, context):


    path = event.get("rawPath", None)


    if path.endswith("/customers"):

        return json.dumps(create_customers(CUSTOMERS, YEAR, MONTH, LAST_DAY))


    if path.endswith("/orders"):

        return json.dumps(create_orders(ORDERS, CUSTOMERS, YEAR, MONTH, LAST_DAY))


    if path.endswith("/reviews"):

        return json.dumps(create_reviews(REVIEWS, ORDERS, YEAR, MONTH, LAST_DAY))


    return json.dumps({})



def create_customers(number, year, month, last_day):

    customers = []

    for i in range(number):
```

```
        create_date = fake.date_time_between_dates(
            datetime_start=datetime(year, month, 1),
            datetime_end=datetime(year, month, last_day),
        )
        first_name = fake.first_name()
        last_name = fake.last_name()
        email = f"{first_name}.{last_name}@{fake.domain_name()}".lower()
        customers.append(
            {
                "customer_id": i,
                "first_name": first_name,
                "last_name": last_name,
                "email": email,
                "created": str(create_date),
            }
        )

    return customers


def create_orders(number, customer_range, year, month, last_day):
    orders = []

    for i in range(number):
        amount = fake.pydecimal(left_digits=3, right_digits=2, positive=True)
        order_datetime = fake.date_time_between_dates(
```

```
            datetime_start=datetime(year, month, 1),
            datetime_end=datetime(year, month, last_day),
        )
        orders.append(
            {
                "order_id": i,
                "customer_id": randrange(1, customer_range),
                "amount": str(amount),
                "datetime": str(order_datetime),
            }
        )

    return orders


def create_reviews(number, order_range, year, month, last_day):
    reviews = []
    for i in range(number):
        review = "/n/n".join(
            [
                choice(REVIEW_LINES[0]),
                choice(REVIEW_LINES[1]),
                choice(REVIEW_LINES[2]),
            ]
        )
        reviews.append(
```

```
        {
            "review_id": i,
            "order_id": randrange(1, order_range),
            "review": review,
        }
    )

  return reviews
```

1. This code does the following:
    - The handler method checks the event payload to see if the request has a URL path
    - If the path matches one of customers, orders, or reviews, it calls a method to create sample data
    - The create_customers and create_orders methods generate random dates from the current month and details for orders and customers such as the customers first and last name
    - The create_reviews method generates a review using a one line each from three sets of sample review sentences
    - The data is returned using json.dumps which converts the data to the JavaScript Object Notation (JSON) format
2. To save your changes to the function's code, at the top of the **Code source** section, click the **Deploy** button
3. You will see a notification at the top that the function has been updated:
4. To test your AWS Lambda function, in the **Function overview** section, on the right under **Function URL**, click the copy icon
5. In a new browser tab, paste the URL from your clipboard into your browser's address bar, append customers to the end of the URL, and press enter:



In response, you will see randomly generated customer data that looks similar to

The AWS Lambda function returns the data in the JSON data format.

Optional: Feel free to try the orders and reviews URL paths to see the data and verify that they are working.

Summary

In this lab step, you implemented and tested an AWS Lambda function generate data to use with Amazon Managed Workflows for Apache Airflow.

Creating a Directed Acyclic Graph

Introduction

A workflow in Apache Airflow is defined as a Directed Ayclic Graph, commonly referred to as a DAG. A DAG is Python code.

In this lab step, you will define a new DAG and upload it to Amazon S3.

Instructions

1. Navigate to the lab's development environment:
2. This is a web-based IDE and is similar in style and functionality to Visual Studio Code.
3. You will use this IDE to create a DAG using Python code. Once complete, you will upload the DAG to an Amazon S3 bucket.
4. If you wish to use your own IDE such as VS Code, feel free to do so. All that is required for this lab step is to produce a Python code file.
5. In the IDE, click **File** and **New File**:
6. You will see a tab appear in the code editor pane:
7. To save and name your file, click **File** and click **Save As.**
8. You will see a dialog box appear.
9. Click the first dropdown and select the **/home/project** option to set the file location:
   In the **Name** textbox at the bottom of the **Save As..** dialog box, replace the contents with **dag.py**, and click **Save**:
10. At the top of the file, add the following import and setup statements:

Copy code

```
from datetime import datetime, timedelta

import os

os.environ["PATH"] = f"/usr/local/airflow/.local/bin:{os.environ['PATH']}"

import tempfile

import boto3

import requests
```

```
import json

from airflow.decorators import dag, task

from airflow.utils.dates import days_ago

from airflow.operators.bash import BashOperator

from airflow.operators.python import PythonOperator

from airflow.models import Variable

s3_bucket = Variable.get("S3_BUCKET")

data_source_url = Variable.get("DATA_SOURCE_URL")
```

1. This code is doing the following:
   - Importing modules for working with temporary files, accessing AWS, making web requests and parsing JSON
   - Importing modules required for defining a DAG and its components
   - Creating two Apache Airflow variables for the S3 bucket name and the data source URL
2. You will create variables for the S3 bucket and data source URL using the Apache Airflow web UI later.
3. To define two helper functions and a dictionary for configuring a DAG, add the following to the end of the **dag.py** file:

Copy code

```
def fetch_url_data(url, bucket, key):

    s3_client = boto3.client("s3")

    temp_file = tempfile.mkstemp()[1]

    resp = requests.get(url)

    with open(temp_file, "w") as fd:

        json.dump(resp.json(), fd)

    s3_client.upload_file(temp_file, bucket, key)

    os.remove(temp_file)

def fetch_s3_data(client, key, bucket):

    temp_file = tempfile.mkstemp()[1]

    client.download_file(bucket, key, temp_file)
```

```
    data = None
    with open(temp_file) as fd:
        data = json.load(fd)
    os.remove(temp_file)
    return data
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2015, 6, 1),
    "email": ["airflow@airflow.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}
```

1. The fetch_url_data and fetch_s3_data functions will be used to fetch customer, order, and review data from the AWS Lambda data source you configured previously, and from the S3 bucket. Both functions use a temporary file to store data, and then upload or download to and from S3 using the boto3 Python library.
2. One important expectation about tasks in an Apache Airflow DAG is that they should be self-contained, with no unexpected side effects, and ideally idempotent. This means if you run the task with the same inputs twice, you should get the same result.
3. The **default_args** dictionary contains arguments that are required when creating a new DAG. One notable thing to mention is that the start_date argument refers to the DAG's data interval, and not when the DAG is first scheduled to run. The start_date must always be a date in the past. To learn more, see the What's the deal with start_date? section of the Apache Airflow FAQ.
4. All possible arguments for a DAG are documentated on the airflow.models.baseoperator, page of the Apache Airflow documentation.
5. To define the DAG, add the following code to the **dag.py** file:

Copy code

```
@dag(
```

```
    default_args=default_args,
    schedule_interval="@monthly",
    start_date=days_ago(1),
)
def combine_and_enrich_dag():
    @task
    def get_customer_data():
        url = f"{data_source_url}/customers"
        fetch_url_data(url, s3_bucket, "data/customers.json")
    @task
    def get_order_data():
        url = f"{data_source_url}/orders"
        fetch_url_data(url, s3_bucket, "data/orders.json")
    @task
    def get_review_data():
        url = f"{data_source_url}/reviews"
        fetch_url_data(url, s3_bucket, "data/reviews.json")
    @task.virtualenv(
        requirements=["boto3==1.24.16", "textblob==0.17.1"],
        system_site_packages=False,
    )
    def analyse_reviews(s3_bucket):
        from textblob import TextBlob
        import tempfile, boto3, os, json
        s3_client = boto3.client("s3")
        temp_file = tempfile.mkstemp()[1]
```

```
    s3_client.download_file(s3_bucket, "data/reviews.json", temp_file)
    data = None
    with open(temp_file) as fd:
        data = json.load(fd)
    os.remove(temp_file)
    analysed_data = []
    for review in data:
        analysis = TextBlob(review["review"]).sentiment
        review["sentiment"] = analysis
        analysed_data.append(review)
    temp_file = tempfile.mkstemp()[1]
    with open(temp_file, "w") as fd:
        json.dump(analysed_data, fd)
    s3_client.upload_file(temp_file, s3_bucket, "data/reviews.json")
    os.remove(temp_file)
@task
def combine_data():
    s3_client = boto3.client("s3")
    customers = fetch_s3_data(s3_client, "data/customers.json", s3_bucket)
    orders = fetch_s3_data(s3_client, "data/orders.json", s3_bucket)
    reviews = fetch_s3_data(s3_client, "data/reviews.json", s3_bucket)
    combined_customer_data = []
    for customer in customers:
        combined = dict(customer)
        combined["orders"] = list([
            o for o in orders if customer["customer_id"] == o["customer_id"]
```

```
        ])

        order_ids = [o["order_id"] for o in combined["orders"]]

        combined["reviews"] = list([r for r in reviews if r["order_id"] in order_ids])

        combined_customer_data.append(dict(combined))

    temp_file = tempfile.mkstemp()[1]

    with open(temp_file, "w") as fd:

        json.dump(combined_customer_data, fd)

    s3_client.upload_file(temp_file, s3_bucket, "data/combined.json")

    os.remove(temp_file)

  (

    [get_customer_data(), get_review_data(), get_order_data()]

    >> analyse_reviews(s3_bucket)

    >> combine_data()

  )

dag = combine_and_enrich_dag()
```

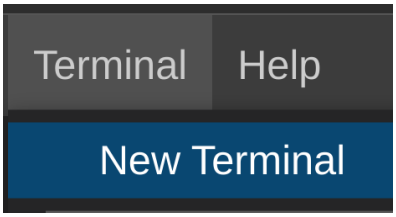
1. This code defines the DAG and five tasks. Here are some things to note about this code:
2. Python decorators are used to define the DAG and the task methods. The alternative is to create a DAG object and separate objects for each task without decorators. To see an example of this, vist the Source code for airflow.example_dags.tutorial page of the Apache Airflow documentation.
3. There's no significant difference between the two styles of defining a DAG. Decorators are a popular feature of the Python scripting language and the decision to use them or not comes down to developer preference or organization coding guidelines.
4. The first three tasks of the DAG use one of the functions mentioned earlier to separately download data from the AWS Lambda data source and place the data in Amazon S3. These tasks are using the PythonOperator to run python code that executes the logic required in a task.
5. The fourth task uses the PythonVirtualenvOperator operator included with Apache Airflow. This type of task creates a new virtual environment where packages can be installed in isolation. Packages to be installed are specified in the requirements argument to the task decorator.
6. This task uses the textblob module to perform sentiment analysis on the review data.
7. Lastly, the **combine_data** task is another Python Operator does the following:
    - Fetches all the data from S3

- Combines the orders and customers using the customer_id
- Combines the reviews and customers using the order_id
- Uploads the combined data to the S3 bucket

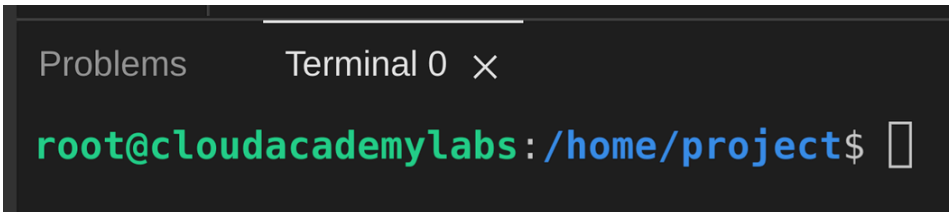
8. In database terms, you can think of the last task as taking denormalized data and normalizing it into a single view.
9. Observe this section of the code at the end of the combine_and_enrich_dag function after the tasks:

```
(
    [get_customer_data(), get_review_data(), get_order_data()]
    >> analyse_reviews(s3_bucket)
    >> combine_data()
)
```

1. This specifies the order the tasks should be run in and their dependencies. In this case it says that the get data functions can run in parallel and that the fourth and fifth tasks are dependent on the previous tasks having completed.
2. To open a terminal, at the top of the IDE, click **Terminal**, and click **New Terminal**:



You will see a terminal pane appear at the bottom of the IDE:



In the terminal, paste the following, replacing <REPLACE_ME> with the values of the **AccessKeyID** and **SecretAccessKeyId** found in the crendentials section of the lab:

Copy code

```
mkdir -p ~/.aws

cat << EOF > ~/.aws/credentials

[default]

region=us-west-2
```

aws_access_key_id=AKIA4AXIFE773NDUJDDS

aws_secret_access_key=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

EOF

1. *Note*: If the lab has not finished setting up the environment, you will have to wait for it to finish before the Access Key and Secret become available. Once lab setup is complete, you will see the values in the top-left corner of the lab.
2. *Note*: You may find it easier to copy this to a text editor or a draft email temporarily whilst replacing the values.
3. *Note*: When pasting commands, you may need to press *Shift* along with *Cmd* (Mac) / *Ctrl* (Windows) and *V*.
4. This command uses two features of the Bash shell:
   - Heredoc syntax to define a multi-line string
   - Redirection of the multi-line string into a file
5. To copy your DAG to the Amazon S3 bucket, enter the following command:

Copy code

aws s3 cp dag.py s3://

airflow-lab-2d842e10-c17f-11f0-b0b5-06217cdb1753

/dags/dag.py

In response, you will see output similar to:

```
root@cloudacademylabs:/home/project$ aws s3 cp dag.py s3://airflow-lab-fe6b67a0-c1f7-11f0-8c30-066a98c3bc05/dags/dag.py
upload: ./dag.py to s3://airflow-lab-fe6b67a0-c1f7-11f0-8c30-066a98c3bc05/dags/dag.py
root@cloudacademylabs:/home/project$
```

This command uses the AWS CLI to copy your dag.py file to the Amazon S3 bucket that was created as part of this lab for use with Amazon MWAA. The path that Amazon MWAA should use to look for DAGs is configurable.

Keep your IDE browser tab open. You will return to it later after running your DAG.

Summary

In this lab step, you use an IDE to create and define a new Python DAG file. You implemented five tasks that fetch data from a data source, analyse the data, and combine the data.

Running your Directed Acyclic Graph

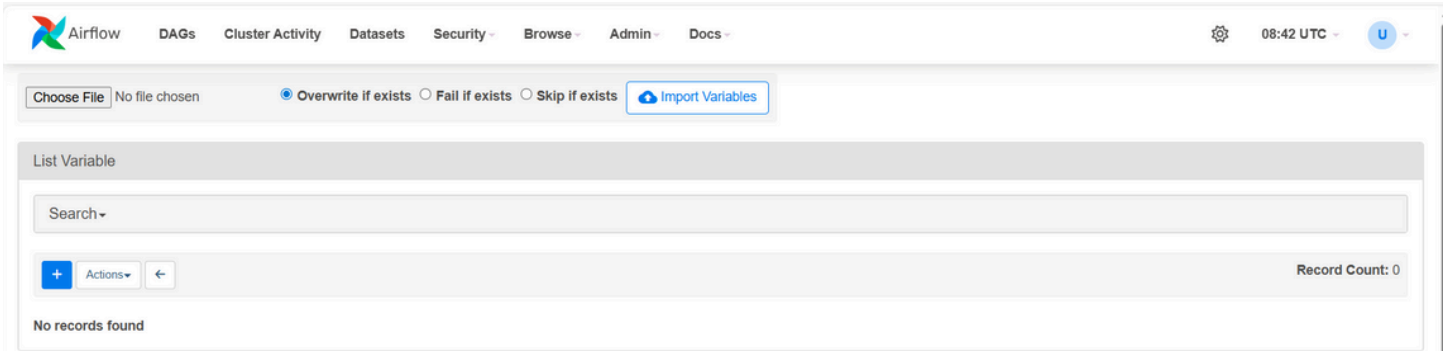
Introduction

In this lab step, you will access the Apache Airflow web user interface and run your directed acyclic graph.

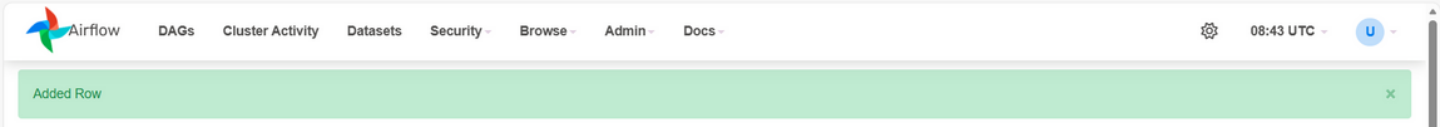
Instructions

1. Navigate to the Environments page of Amazon MWAA console, opens in a new tab.
2. To open the Apache Airflow web user interface in a new browser tab, on the right-hand side of the **Environments** table, click **Open Airflow UI**
3. The Apache Airflow web UI will load in a new browser tab:

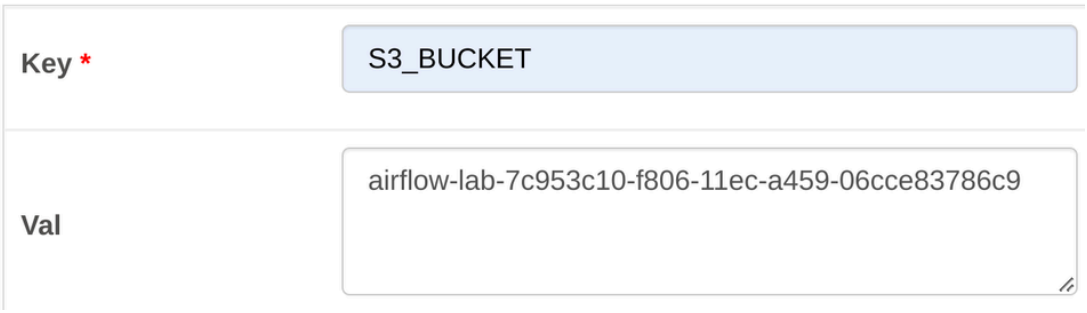
1. Amazon MWAA integrates authentication with Apache Airflow and AWS Identity and Access Management (IAM). You can use IAM roles and policies to control access to an Amazon MWAA environment.
2. *Note*: If you are asked to confirm that you want to log on with SSO say yes.
3. You may see a **DAG Import Errors (1)** notification at the top of the page. This is expected and is caused by missing variables which you will configure next.
4. To navigate to the variables page, at the top, click **Admin** and then **Variables**:



1. A page will load with a section titled **List Variable**.
2. To manually add a new variable, click the blue plus icon
3. A form named **Add Variable** will load.
4. Enter the following in the form fields:
   - **Key**: *DATA_SOURCE_URL*
   - **Val**:
   - *https://h4mb625hphhkg2xqx7gvifiryy0jlhyp.lambda-url.us-west-2.on.aws/*
5. You are specifying a variable to store the URL of the data source AWS Lambda function you configured earlier.
6. *Note*: Ensure that the **Key** is in upper-case and that the URL in **Val** has a trailing slash (**/**). Also, ensure that there are no newlines at the beginning or end of the value.
7. To save the variable, click **Save**
8. You will be returned to the **List Variable** page, and you'll see a notification that the variable has been added.

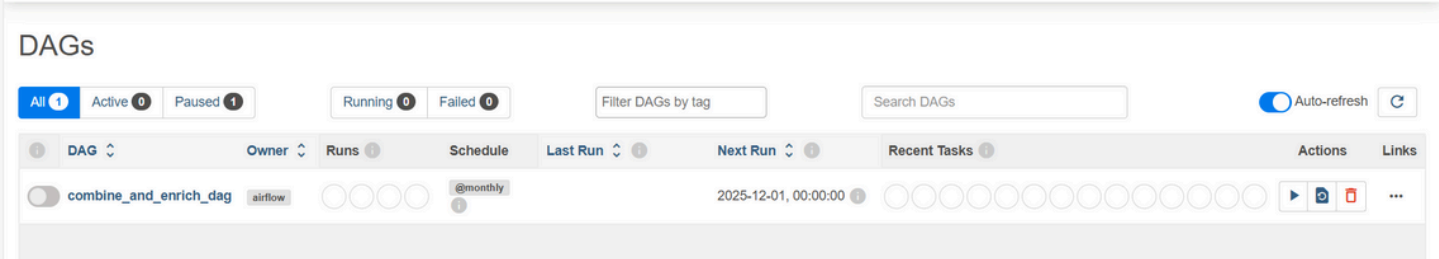


1. Repeat the process for adding a variable using the following key and value:
   - **Key**: *S3_BUCKET*
   - **Val**:
   - *airflow-lab-fe6b67a0-c1f7-11f0-8c30-066a98c3bc05*



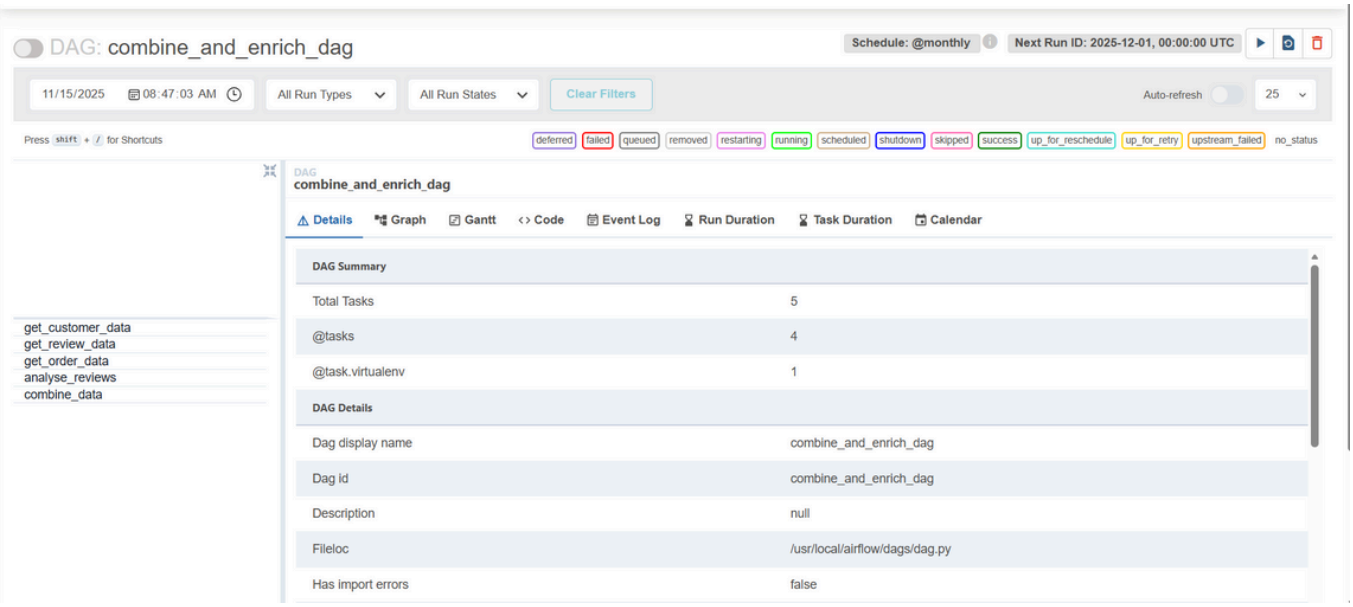
1. *Note*: Your bucket name will be different.
2. You will see two variables lists on the **List Variable** page:

1. Variables in Apache Airflow are used to store configuration data in a generic way. They are accessible within a DAG whilst it is running.
2. Variables are encrypted by default. Whether it is appropriate to use them for sensitive data such as credentials will depend upon your use-case and your organization's policies. Apache Airflow also has support for a Secrets Backend, opens in a new tab which allows you to a custom secret manager service such as AWS Secrets Manager.
3. To return to the DAGs page, at the top, click **DAGs**
4. The error notification will disappear, and you will see your DAG listed:



1. Apache Airflow is constantly monitoring the Amazon S3 bucket for new and update DAGs.
2. *Note*: Apache Airflow can take a few moments to update after uploading files to the S3 bucket or specifying variables. If you do not see your DAG listed, wait a minute and refresh until it appears.
3. DAGs are usually run on a schedule. In this lab, you will trigger your DAG manually.

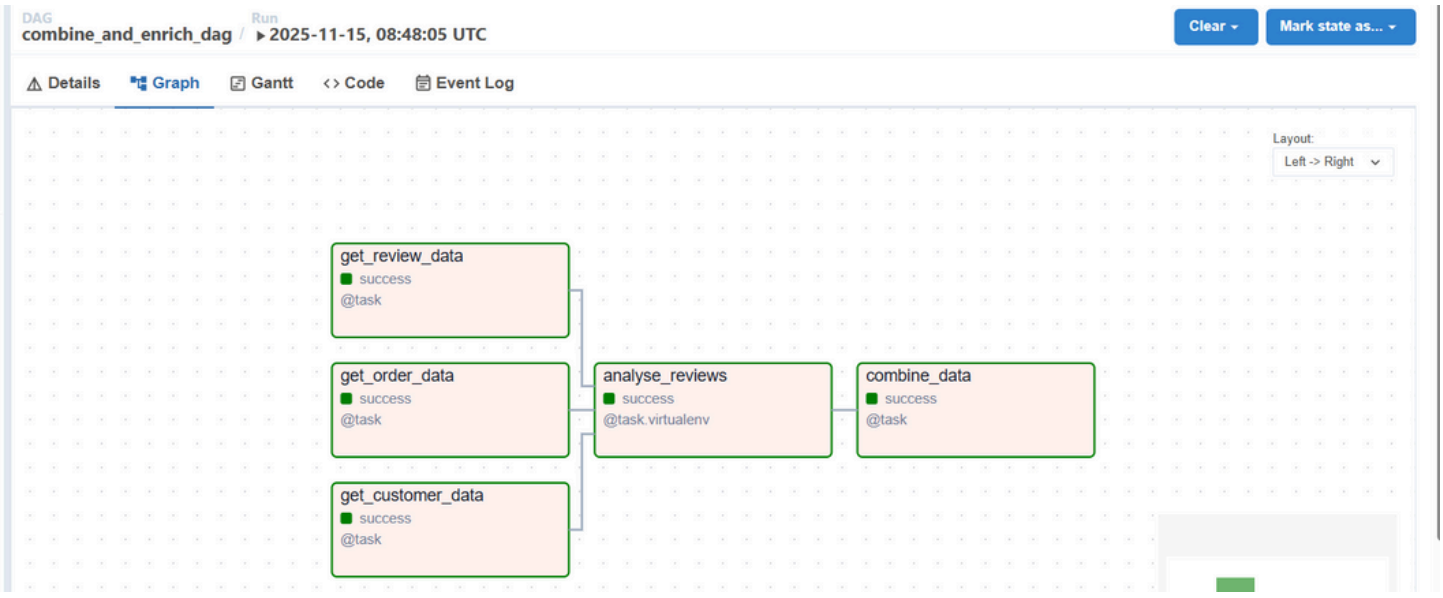
4. To navigate to the details page of the DAG, in the **DAGs** table, click **combine_and_enrich_dag**



1. To start your DAG, on the right-hand side, click the play button and click **Trigger DAG**
2. You will see a notification that your DAG will start shortly:
3. To navigate to a graph view of your DAG, hover your mouse over the three dots on the right-hand side under **Links** and click **Graph**:

Triggered combine_and_enrich_dag with new Run ID manual__2025-11-15T08:48:05.357379+00:00, it should start any moment now.

You will see a graphical representation of the DAG. Here is what it looks like once the DAG run has finished successfully:

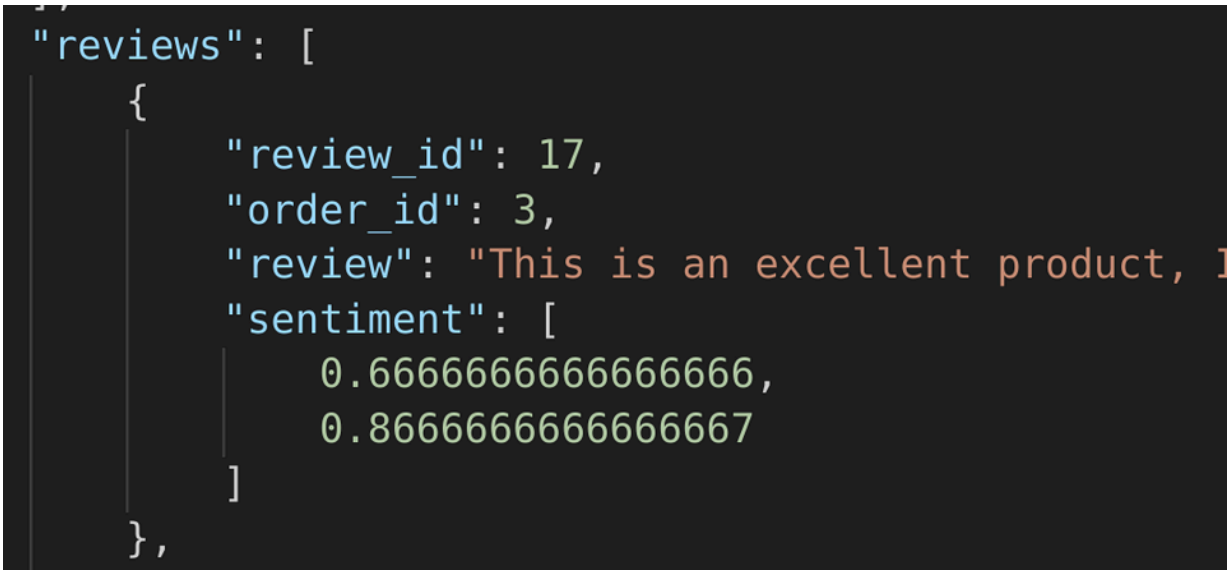


1. Whilst it is running, you will see the tasks highlighted with different colors. Bright green means the task is currently running.
2. The DAG may take a few minutes to complete.
3. *Optional*: You can see the log of a task by click the task in the graph view and clicking **Log** in the dialog box that opens. Feel free to examine the logs of a task.
4. To verify that the DAG has combined and enriched the data, return to the IDE, and enter the following in the terminal:

Copy code

```
aws s3 cp s3://airflow-lab-fe6b67a0-c1f7-11f0-8c30-066a98c3bc05/data/combined.json combined.json
```

1. This command fetches the combined.json file that the DAG produces from the S3 bucket using the AWS CLI.
2. To view the data, on the left-hand side, in the **EXPLORER** pane, double click **combined.json**
3. The file will open in the main editor pane.
4. To format the document and make it easier to inspect, right-click anywhere in the editor pane and click **Format Document**
5. You will see an array of customers. Due to the random generation of the data by the data source, what you see will vary each time you run this DAG.
6. Each customer has reviews and orders attributes (some may be empty).
7. The orders have been associated with the customer using the customer_id and the reviews have been associated with the customer through the order using the order_id.
8. Scroll through the file until you see a customer with at least one review



The sentiment field was populated by the analyse_reviews task in the DAG. The first number is a polarity score between -1.0 and 1.0, and the second is a subjectivity score between 0.0 and 1.0. A lower polarity score indicates that the review has a negative sentiment.

This lab's usage of Apache Airflow is simplified for the sake of time and convenience. A DAG can contain many tasks performing functions such as:

- Cleansing and filtering data
- Checkmarking and batching data operations
- Submitting data for analysis by an external service and receiving the results
- Any kind of the data analysis that you can perform with Python

Summary

In this lab, you saw to how configure a new Amazon MWAA environment. You created a data source that generated random data for use with Amazon MWAA. You created a directed acyclic graph that fetched data from the data source and processed it. Finally, you ran your directed acyclic graph and observed the results.