Updated for Airflow 3

Airflow

Stats

✕ 0 Failed Dags ›   1 Running Dags ›   4 Active Dags ›

First 10 favorite Dags

ai_support_ticket_system   dag_versioning_example   s3_to_snowflake_pipeline

Health

✓ MetaDatabase ✓ Scheduler ✓ Triggerer ✓ Dag Processor

Pool Slots   Manage Pools

✓ 128

History

Last 24 Hours   2025-09-29 11:24:45 - 2025-09-30 11:24:45

35	Dag Runs	
0	Queued	0.00%
1	Running	2.86%
✓ 34	Success	97.14%
✕ 0	Failed	0.00%

29 Asset Events   Newest First

2025-09-30 11:24:02
model_trained
Source: dag_versioning_example

2025-09-30 11:24:02
model_trained
Source: dag_versioning_example

2025-09-30 11:24:01
model_trained
Source: dag_versioning_example

2025-09-30 11:24:01

ETL/ELT patterns with Apache Airflow® 3

9 Practical DAG Code Examples

ASTRONOMER

In this new version of this eBook you'll find ETL and ELT pattern DAGs updated for Airflow 3, as well as an ETL and an ETL sequence using the new @asset decorator!

Introduction

When writing data pipelines, regardless of the end use case, whether you provide data for analytics, complex dashboards, or are fine-tuning large language models, your data must first be **extracted (E)** from its source, **transformed (T)** into a target schema, and **loaded (L)** into your central data storage, whether that's a data warehouse or a data lake.

ETL and ELT pipelines form the core foundation of any data architecture, making the ability to design, build, and understand these pipelines essential for every data engineer.

Originally developed in 2014 specifically for ETL and ELT pipelines, **Apache Airflow®** has since evolved into the open-source standard for full-stack orchestration of any workflow—without losing sight of its roots in moving and transforming data.

According to the **2025 State of Airflow Report**, over 85% of current Airflow users rely on it to run ETL and ELT pipelines, powering everything from some of the world's largest organizations to fast-growing startups.

This eBook contains full example DAG patterns for different ETL and ELT scenarios. The code to all these DAGs can also be found in **this GitHub repository**, which is set up for you to be able to run them locally without needing to configure any connections or additional tools.

While the example DAGs use Postgres as their target system, and MinIO as external storage, they can be adapted to interact with any tool that is accessible through an API. For many tools, pre-built modules exist, see the **Astronomer Registry** for a comprehensive list.

This eBook is aimed at intermediate Airflow users. If you are new to Airflow we recommend first completing the **Apache Airflow® Crash Course: From 0 to running your pipeline in the Cloud** to get familiar with core Airflow concepts before using the code in this eBook.

If you are looking for more guidance on how to use Airflow for ETL and ELT, check out the **Best practices for ELT and ETL pipelines with Apache Airflow® eBook** for all you need to know to write ETL and ELT pipelines.

PART ONE

DAG Code Examples

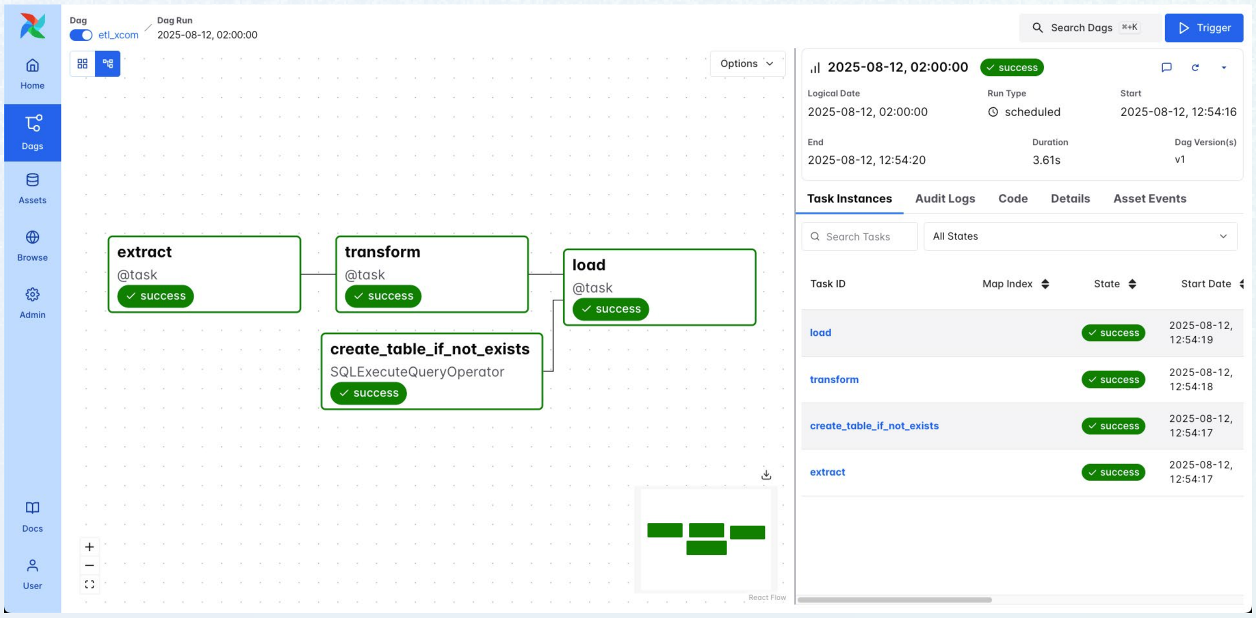
1

ETL pattern DAG using XCom

This DAG shows an ETL pattern to extract data from the Open-Meteo API, transform it and then load it into a Postgres database. The data is passed through XCom between the extraction and transformation task, as well as between the transformation and loading task, which means this setup would need a custom XCom backend in production.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

This is the graph of the DAG that will be created:



Dag: etl_xcom / Dag Run: 2025-08-12, 02:00:00

2025-08-12, 02:00:00 success

Logical Date: 2025-08-12, 02:00:00
Run Type: scheduled
Start: 2025-08-12, 12:54:16
End: 2025-08-12, 12:54:20
Duration: 3.61s
Dag Version(s): v1

Task Instances | Audit Logs | Code | Details | Asset Events

Search Tasks | All States

Task ID	Map Index	State	Start Date
load		success	2025-08-12, 12:54:19
transform		success	2025-08-12, 12:54:18
create_table_if_not_exists		success	2025-08-12, 12:54:17
extract		success	2025-08-12, 12:54:17

```
"""
## Simple ETL DAG loading data from the Open-Meteo API to a Postgres database

This DAG extracts weather data from the Open-Meteo API, transforms it, and
loads it into a Postgres database in an ETL pattern.
It passes the data through XComs.
"""

import os
from datetime import datetime, timedelta

from airflow.sdk import dag, task, chain, Param
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from include.col_orders import WEATHER_COL_ORDER

# ------------------- #
# DAG-level variables #
# ------------------- #

DAG_ID = os.path.basename(__file__).replace(".py", "")

_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"
)

_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"



# -------------- #
# DAG definition #
# -------------- #


@dag(
    dag_id=DAG_ID,
    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled
    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options
    max_active_runs=1,  # maximum number of active DAG runs
    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental
    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial
    default_args={
        "owner": "Astro",  # owner of this DAG in the Airflow UI
        "retries": 3,  # tasks retry 3 times before they fail
        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries
    },
    tags=["Patterns", "ETL", "Postgres", "XCom"],  # add tags in the UI
    params={
        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")
    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params
    template_searchpath=[_SQL_DIR],  # path to the SQL templates
)
def etl_xcom():

    # ---------------- #
    # Task Definitions #
    # ---------------- #
    # the @task decorator turns any Python function into an Airflow task
    # any @task decorated function that is called inside the @dag decorated
    # function is automatically added to the DAG.
    # if one exists for your use case you can use traditional Airflow operators
    # and mix them with @task decorators. Checkout registry.astronomer.io for available
operators
    # see: https://www.astronomer.io/docs/learn/airflow-decorators for information about
@task
    # see: https://www.astronomer.io/docs/learn/what-is-an-operator for information
about traditional operators
```

```
_create_table_if_not_exists = SQLExecuteQueryOperator(
    task_id="create_table_if_not_exists",
    conn_id=_POSTGRES_CONN_ID,
    database=_POSTGRES_DATABASE,
    sql="create_table_if_not_exists.sql",
    params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE},
)

@task
def extract(**context):
    """
    Extract data from the Open-Meteo API
    Returns:
        dict: The full API response
    """
    import requests

    url = os.getenv("WEATHER_API_URL")

    coordinates = context["params"]["coordinates"]
    latitude = coordinates["latitude"]
    longitude = coordinates["longitude"]

    url = url.format(latitude=latitude, longitude=longitude)

    response = requests.get(url)

    return response.json()

@task
def transform(api_response: dict, **context) -> dict:
    """
    Transform the data
    Args:
        api_response (dict): The full API response
    Returns:
        dict: The transformed data
    """

    time = api_response["hourly"]["time"]
    dag_run_timestamp = context["ts"]

    transformed_data = {
        "temperature_2m": api_response["hourly"]["temperature_2m"],
        "relative_humidity_2m": api_response["hourly"]["relative_humidity_2m"],
        "precipitation_probability": api_response["hourly"][
            "precipitation_probability"
        ],
        "timestamp": time,
        "date": [
            datetime.strptime(x, "%Y-%m-%dT%H:%M").date().strftime("%Y-%m-%d")
            for x in time
        ],
        "day": [datetime.strptime(x, "%Y-%m-%dT%H:%M").day for x in time],
        "month": [datetime.strptime(x, "%Y-%m-%dT%H:%M").month for x in time],
        "year": [datetime.strptime(x, "%Y-%m-%dT%H:%M").year for x in time],
        "last_updated": [dag_run_timestamp for i in range(len(time))],
        "latitude": [api_response["latitude"] for i in range(len(time))],
        "longitude": [api_response["longitude"] for i in range(len(time))],
    }

    return transformed_data

@task
def load(transformed_data: dict):
    """
    Load the data to the destination without using a temporary CSV file.
    Args:
        transformed_data (dict): The transformed data
    """
    import csv
    import io
    from airflow.providers.postgres.hooks.postgres import PostgresHook
```

```
        hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

        csv_buffer = io.StringIO()
        writer = csv.writer(csv_buffer)
        writer.writerow(WEATHER_COL_ORDER)
        rows = zip(*[transformed_data[col] for col in WEATHER_COL_ORDER])
        writer.writerows(rows)

        csv_buffer.seek(0)

        with open(f"{_SQL_DIR}/copy_insert.sql") as f:
            sql = f.read()
        sql = sql.replace("{schema}", _POSTGRES_SCHEMA)
        sql = sql.replace("{table}", _POSTGRES_TRANSFORMED_TABLE)

        conn = hook.get_conn()
        cursor = conn.cursor()
        cursor.copy_expert(sql=sql, file=csv_buffer)
        conn.commit()
        cursor.close()
        conn.close()

    _extract = extract()
    _transform = transform(api_response=_extract)
    _load = load(transformed_data=_transform)
    chain(_transform, _load)
    chain(_create_table_if_not_exists, _load)


etl_xcom()
```

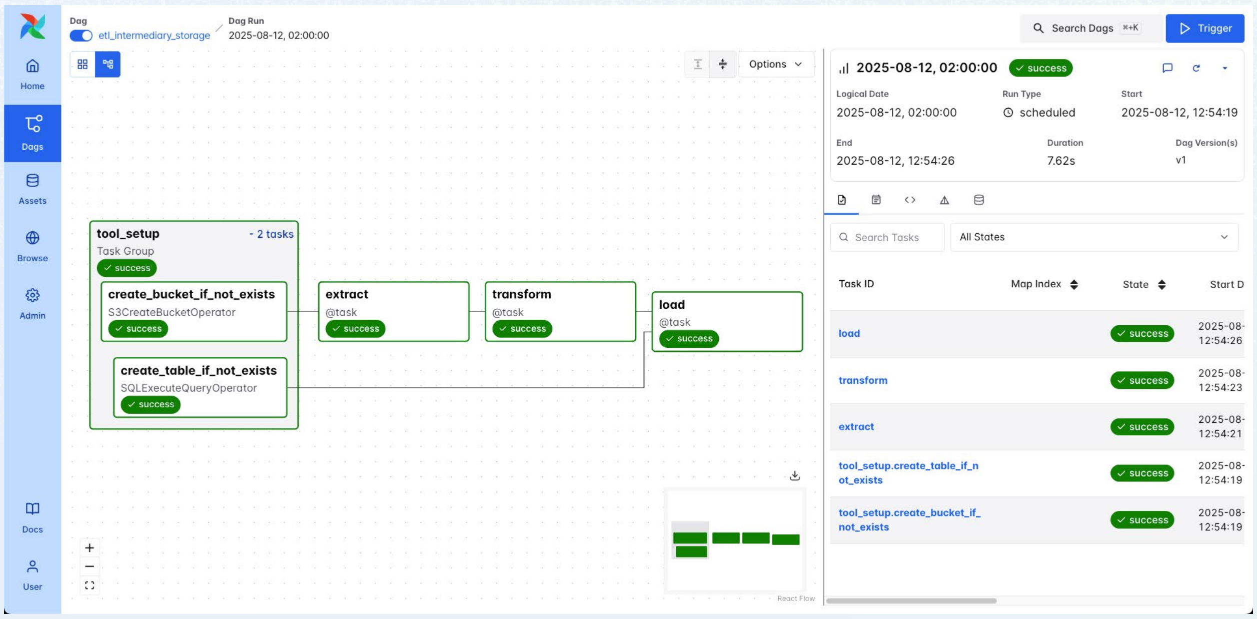
2

ETL pattern DAG using explicit external storage

This DAG accomplishes the same task, extracting data from the Open-Meteo API, transforming it and loading it to a Postgres database, but it does not pass any data through XCom, instead, MinIO is used as explicit external storage.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

This is the graph of the DAG that is created:



2025-08-12, 02:00:00 ✓ success

Logical Date: 2025-08-12, 02:00:00
Run Type: scheduled
Start: 2025-08-12, 12:54:19
End: 2025-08-12, 12:54:26
Duration: 7.62s
Dag Version(s): v1

Task ID	Map Index	State	Start D
load		✓ success	2025-08-12:54:26
transform		✓ success	2025-08-12:54:23
extract		✓ success	2025-08-12:54:21
tool_setup.create_table_if_not_exists		✓ success	2025-08-12:54:19
tool_setup.create_bucket_if_not_exists		✓ success	2025-08-12:54:19

```
"""
## Simple ETL DAG loading data from the Open-Meteo API to a Postgres database

This DAG extracts weather data from the Open-Meteo API, transforms it, and
loads it into a Postgres database in an ETL pattern.
It uses S3 as an intermediary storage.
"""

import json
import os
from datetime import datetime, timedelta

import requests
from airflow.sdk import dag, task, task_group, chain, Param
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.providers.amazon.aws.operators.s3 import S3CreateBucketOperator
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator

# ------------------- #
# DAG-level variables #
# ------------------- #

DAG_ID = os.path.basename(__file__).replace(".py", "")

_AWS_CONN_ID = os.getenv("MINIO_CONN_ID", "minio_local")
_S3_BUCKET = os.getenv("S3_BUCKET", "open-meteo-etl")

_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"
)
```

```
_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"


_EXTRACT_TASK_ID = "extract"

_TRANSFORM_TASK_ID = "transform"


# -------------- #

# DAG definition #

# -------------- #



@dag(

    dag_id=DAG_ID,

    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled

    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options

    max_active_runs=1,  # maximum number of active DAG runs

    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental

    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial

    default_args={

        "owner": "Astro",  # owner of this DAG in the Airflow UI

        "retries": 3,  # tasks retry 3 times before they fail

        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries

    },

    tags=["Patterns", "ETL", "Postgres", "Intermediary Storage"],  # add tags in the UI

    params={

        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")

    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params

    template_searchpath=[_SQL_DIR],  # path to the SQL templates

)

def etl_intermediary_storage():


    # ---------------- #

    # Task Definitions #

    # ---------------- #

    # the @task decorator turns any Python function into an Airflow task
    # any @task decorated function that is called inside the @dag decorated
    # function is automatically added to the DAG.
    # if one exists for your use case you can use traditional Airflow operators
    # and mix them with @task decorators. Checkout registry.astronomer.io for available
operators
    # see: https://www.astronomer.io/docs/learn/airflow-decorators for information about
@task
    # see: https://www.astronomer.io/docs/learn/what-is-an-operator for information
about traditional operators

    @task_group
    def tool_setup():

        _create_table_if_not_exists = SQLExecuteQueryOperator(
            task_id="create_table_if_not_exists",
            conn_id=_POSTGRES_CONN_ID,
            database=_POSTGRES_DATABASE,
            sql="create_table_if_not_exists.sql",
            params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE},
        )

        _create_bucket_if_not_exists = S3CreateBucketOperator(
            task_id="create_bucket_if_not_exists",
            bucket_name=_S3_BUCKET,
            aws_conn_id=_AWS_CONN_ID,
        )

        return _create_table_if_not_exists, _create_bucket_if_not_exists

    _tool_setup = tool_setup()

    @task(task_id=_EXTRACT_TASK_ID)
    def extract(**context):
        """
        Extract data from the Open-Meteo API
        Returns:
            dict: The full API response
        """
        url = os.getenv("WEATHER_API_URL")

        coordinates = context["params"]["coordinates"]
        latitude = coordinates["latitude"]
        longitude = coordinates["longitude"]
        dag_run_timestamp = context["ts"]
        dag_id = context["dag"].dag_id
        task_id = context["task"].task_id

        url = url.format(latitude=latitude, longitude=longitude)

        response = requests.get(url).json()

        response_bytes = json.dumps(response).encode("utf-8")

        # Save the data to S3
        hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
        hook.load_bytes(
            bytes_data=response_bytes,
            key=f"{dag_id}/{task_id}/{dag_run_timestamp}.json",
            bucket_name=_S3_BUCKET,
            replace=True,
        )

    @task(task_id=_TRANSFORM_TASK_ID)
    def transform(**context):
        """
        Transform the data
        Args:
            api_response (dict): The full API response
        Returns:
            dict: The transformed data
        """

        dag_run_timestamp = context["ts"]
        dag_id = context["dag"].dag_id
```

```
upstream_task_id = _EXTRACT_TASK_ID
task_id = context[“task”].task_id

# Load the data from S3
hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
response = hook.read_key(
    key=f”{dag_id}/{upstream_task_id}/{dag_run_timestamp}.json”,
    bucket_name=_S3_BUCKET,
)
api_response = json.loads(response)

time = api_response[“hourly”][“time”]

transformed_data = {
    “temperature_2m”: api_response[“hourly”][“temperature_2m”],
    “relative_humidity_2m”: api_response[“hourly”][“relative_humidity_2m”],
    “precipitation_probability”: api_response[“hourly”][
        “precipitation_probability”
    ],
    “timestamp”: time,
    “date”: [
        datetime.strptime(x, “%Y-%m-%dT%H:%M”).date().strftime(“%Y-%m-%d”)
        for x in time
    ],
    “day”: [datetime.strptime(x, “%Y-%m-%dT%H:%M”).day for x in time],
    “month”: [datetime.strptime(x, “%Y-%m-%dT%H:%M”).month for x in time],
    “year”: [datetime.strptime(x, “%Y-%m-%dT%H:%M”).year for x in time],
    “last_updated”: [dag_run_timestamp for i in range(len(time))],
    “latitude”: [api_response[“latitude”] for i in range(len(time))],
    “longitude”: [api_response[“longitude”] for i in range(len(time))],
}

transformed_data_bytes = json.dumps(transformed_data).encode(“utf-8”)

# Save the data to S3
hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
hook.load_bytes(
        bytes_data=transformed_data_bytes,
        key=f"{dag_id}/{task_id}/{dag_run_timestamp}.json",
        bucket_name=_S3_BUCKET,
        replace=True,
    )

@task
def load(**context):
    """
    Load the data to the destination without using a temporary CSV file.
    Args:
        transformed_data (dict): The transformed data
    """
    import csv
    import io

    from airflow.providers.postgres.hooks.postgres import PostgresHook

    dag_run_timestamp = context["ts"]
    dag_id = context["dag"].dag_id
    upstream_task_id = _TRANSFORM_TASK_ID

    # Load the data from S3
    hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
    response = hook.read_key(
        key=f"{dag_id}/{upstream_task_id}/{dag_run_timestamp}.json",
        bucket_name=_S3_BUCKET,
    )
    api_response = json.loads(response)

    # Load the data to Postgres
    hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

    csv_buffer = io.StringIO()
    writer = csv.writer(csv_buffer)
    writer.writerow(api_response.keys())
    rows = zip(*api_response.values())
```

```
            writer.writerows(rows)

            csv_buffer.seek(0)

            with open(f"{_SQL_DIR}/copy_insert.sql") as f:
                sql = f.read()
            sql = sql.replace("{schema}", _POSTGRES_SCHEMA)
            sql = sql.replace("{table}", _POSTGRES_TRANSFORMED_TABLE)

            conn = hook.get_conn()
            cursor = conn.cursor()
            cursor.copy_expert(sql=sql, file=csv_buffer)
            conn.commit()
            cursor.close()
            conn.close()

        _extract = extract()
        _transform = transform()
        _load = load()
        chain(_extract, _transform, _load)
        chain(_tool_setup[0], _load)
        chain(_tool_setup[1], _extract)


    etl_intermediary_storage()
```

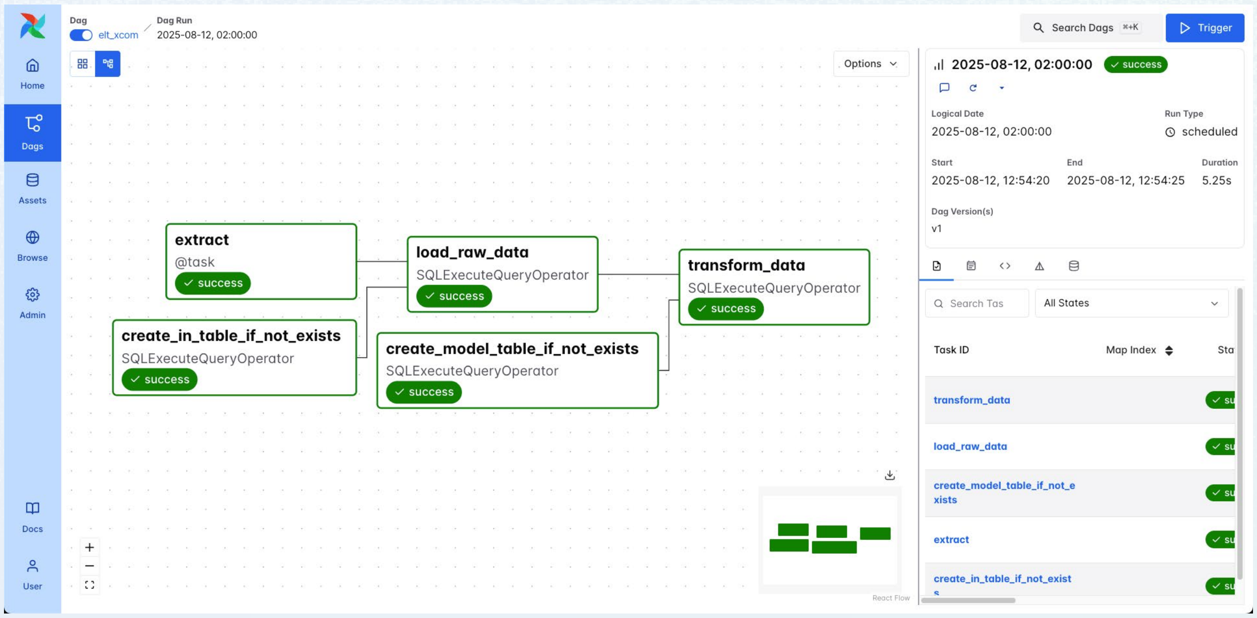
3

ELT pattern DAG using XCom

In this example DAG, the full data extracted from the Open-Meteo API is loaded into a Postgres database and all transformation happens inside the Postgres database using SQL. The data is passed through XCom between the extraction and loading task, which means this setup would need a custom XCom backend in production.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

This is the DAG that is being created:



2025-08-12, 02:00:00 success

Logical Date 2025-08-12, 02:00:00
Run Type scheduled
Start 2025-08-12, 12:54:20
End 2025-08-12, 12:54:25
Duration 5.25s
Dag Version(s) v1

Search Tas | All States

Task ID	Map Index	Sta
transform_data		✓ su
load_raw_data		✓ su
create_model_table_if_not_exists		✓ su
extract		✓ su
create_in_table_if_not_exists		✓ su

```
"""
## Simple ELT DAG loading data from the Open-Meteo API to a Postgres database


This DAG extracts weather data from the Open-Meteo API,
loads it into a Postgres database and transforms it, using an ELT pattern.
It passes the data through XComs between extract and transform.
"""


import os
import json
from datetime import datetime, timedelta


from airflow.sdk import dag, task, chain, Param
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from include.col_orders import WEATHER_COL_ORDER


# ------------------- #
# DAG-level variables #
# ------------------- #


DAG_ID = os.path.basename(__file__).replace(".py", "")


_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_IN_TABLE = os.getenv("POSTGRES_WEATHER_TABLE_IN", f"in_weather_data_{DAG_ID}")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"
```

```
)


_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"




# -------------- #

# DAG definition #

# -------------- #



@dag(

    dag_id=DAG_ID,

    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled

    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options

    max_active_runs=1,  # maximum number of active DAG runs

    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental

    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial

    default_args={

        "owner": "Astro",  # owner of this DAG in the Airflow UI

        "retries": 3,  # tasks retry 3 times before they fail

        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries

    },

    tags=["Patterns", "ELT", "Postgres", "XCom"],  # add tags in the UI

    params={

        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")

    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params
```

```
    template_searchpath=[_SQL_DIR],  # path to the SQL templates
)
def elt_xcom():


    # ---------------- #
    # Task Definitions #
    # ---------------- #
    # the @task decorator turns any Python function into an Airflow task
    # any @task decorated function that is called inside the @dag decorated
    # function is automatically added to the DAG.
    # if one exists for your use case you can use traditional Airflow operators
    # and mix them with @task decorators. Checkout registry.astronomer.io for available
operators
    # see: https://www.astronomer.io/docs/learn/airflow-decorators for information about
@task
    # see: https://www.astronomer.io/docs/learn/what-is-an-operator for information
about traditional operators


    _create_in_table_if_not_exists = SQLExecuteQueryOperator(
        task_id="create_in_table_if_not_exists",
        conn_id=_POSTGRES_CONN_ID,
        database=_POSTGRES_DATABASE,
        sql="create_in_table_if_not_exists.sql",
        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_IN_TABLE},
    )


    _create_model_table_if_not_exists = SQLExecuteQueryOperator(
        task_id="create_model_table_if_not_exists",
        conn_id=_POSTGRES_CONN_ID,
        database=_POSTGRES_DATABASE,
        sql="create_model_table_if_not_exists.sql",
        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE},
    )


    @task
    def extract(**context):
        """
        Extract data from the Open-Meteo API
        Returns:
            dict: The full API response
        """
        import requests

        url = os.getenv("WEATHER_API_URL")

        coordinates = context["params"]["coordinates"]
        latitude = coordinates["latitude"]
        longitude = coordinates["longitude"]

        url = url.format(latitude=latitude, longitude=longitude)

        response = requests.get(url)

        return response.json()

    _extract = extract()

    _load = SQLExecuteQueryOperator(
        task_id="load_raw_data",
        conn_id=_POSTGRES_CONN_ID,
        sql="""
```

```
        INSERT INTO {{ params.schema }}.{{ params.table }} (raw_data)
        VALUES ('{{ ti.xcom_pull(task_ids='extract') | tojson }}'::jsonb);
        """,
        params={
            "schema": _POSTGRES_SCHEMA,
            "table": _POSTGRES_IN_TABLE,
        },
    )

    _transform = SQLExecuteQueryOperator(
        task_id="transform_data",
        conn_id=_POSTGRES_CONN_ID,
        sql="transform.sql",
        params={
            "schema": _POSTGRES_SCHEMA,
            "in_table": _POSTGRES_IN_TABLE,
            "out_table": _POSTGRES_TRANSFORMED_TABLE,
        },
    )

    chain([_create_in_table_if_not_exists, _extract], _load, _transform)
    chain(_create_model_table_if_not_exists, _transform)


elt_xcom()
```

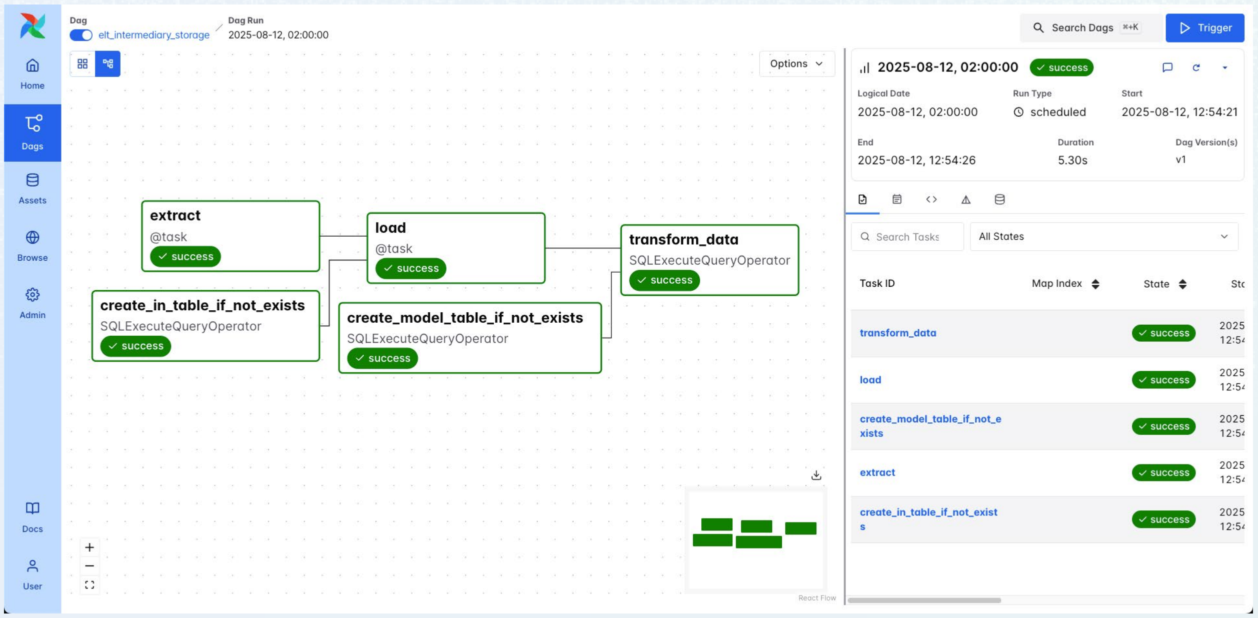
4

ELT pattern DAG using explicit external storage

This DAG accomplishes the same as the above DAG, extracting data from the Open-Meteo API, loading the raw data into a Postgres database and then running a transformation on it, but without passing any data through XCom.

In the companion GitHub repository you can find this DAG **here** and its supporting SQL scripts **here**.

This is the DAG that will be created:



2025-08-12, 02:00:00 ✓ success

Logical Date	Run Type	Start
2025-08-12, 02:00:00	scheduled	2025-08-12, 12:54:21

End	Duration	Dag Version(s)
2025-08-12, 12:54:26	5.30s	v1

Task ID	Map Index	State	Sta
transform_data		✓ success	2025 12:5
load		✓ success	2025 12:5
create_model_table_if_not_exists		✓ success	2025 12:5
extract		✓ success	2025 12:5
create_in_table_if_not_exists		✓ success	2025 12:5

```
"""

## Simple ELT DAG loading data from the Open-Meteo API to a Postgres database


This DAG extracts weather data from the Open-Meteo API,

loads it into a Postgres database and transforms it, using an ELT pattern.

It passes the data through XCom between extract and transform.

"""



import os

import json

from datetime import datetime, timedelta


from airflow.providers.amazon.aws.hooks.s3 import S3Hook

from airflow.sdk import dag, task, chain, Param

from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator

from airflow.providers.postgres.hooks.postgres import PostgresHook


# ------------------- #

# DAG-level variables #

# ------------------- #


DAG_ID = os.path.basename(__file__).replace(".py", "")


_AWS_CONN_ID = os.getenv("MINIO_CONN_ID", "minio_local")

_S3_BUCKET = os.getenv("S3_BUCKET", "open-meteo-etl")


_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
```

```
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")

_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")

_POSTGRES_IN_TABLE = os.getenv("POSTGRES_WEATHER_TABLE_IN", f"in_weather_data_{DAG_ID}")

_POSTGRES_TRANSFORMED_TABLE = os.getenv(

    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"

)


_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"



_EXTRACT_TASK_ID = "extract"



# -------------- #

# DAG definition #

# -------------- #



@dag(

    dag_id=DAG_ID,

    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled

    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options

    max_active_runs=1,  # maximum number of active DAG runs

    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental

    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial

    default_args={

        "owner": "Astro",  # owner of this DAG in the Airflow UI

        "retries": 3,  # tasks retry 3 times before they fail

        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries
    },
    tags=["Patterns", "ELT", "Postgres", "XCom"],  # add tags in the UI
    params={
        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")
    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params
    template_searchpath=[_SQL_DIR],  # path to the SQL templates
)
def elt_intermediary_storage():


    # ---------------- #
    # Task Definitions #
    # ---------------- #
    # the @task decorator turns any Python function into an Airflow task
    # any @task decorated function that is called inside the @dag decorated
    # function is automatically added to the DAG.
    # if one exists for your use case you can use traditional Airflow operators
    # and mix them with @task decorators. Checkout registry.astronomer.io for available
operators
    # see: https://www.astronomer.io/docs/learn/airflow-decorators for information about
@task
    # see: https://www.astronomer.io/docs/learn/what-is-an-operator for information
about traditional operators


    _create_in_table_if_not_exists = SQLExecuteQueryOperator(
        task_id="create_in_table_if_not_exists",
        conn_id=_POSTGRES_CONN_ID,
        database=_POSTGRES_DATABASE,
        sql="create_in_table_if_not_exists.sql",
        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_IN_TABLE},
    )
```

```
_create_model_table_if_not_exists = SQLExecuteQueryOperator(
    task_id="create_model_table_if_not_exists",
    conn_id=_POSTGRES_CONN_ID,
    database=_POSTGRES_DATABASE,
    sql="create_model_table_if_not_exists.sql",
    params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE},
)

@task(task_id=_EXTRACT_TASK_ID)
def extract(**context):
    """
    Extract data from the Open-Meteo API
    Returns:
        dict: The full API response
    """
    import requests

    url = os.getenv("WEATHER_API_URL")

    coordinates = context["params"]["coordinates"]
    latitude = coordinates["latitude"]
    longitude = coordinates["longitude"]
    dag_run_timestamp = context["ts"]
    dag_id = context["dag"].dag_id
    task_id = context["task"].task_id

    url = url.format(latitude=latitude, longitude=longitude)

    response = requests.get(url).json()
```

```
        response_bytes = json.dumps(response).encode("utf-8")

        # Save the data to S3
        hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
        hook.load_bytes(
            bytes_data=response_bytes,
            key=f"{dag_id}/{task_id}/{dag_run_timestamp}.json",
            bucket_name=_S3_BUCKET,
            replace=True,
        )

    _extract = extract()

    @task
    def load(**context):
        """
        Load the data from S3 to Postgres
        """

        dag_run_timestamp = context["ts"]
        dag_id = context["dag"].dag_id
        upstream_task_id = _EXTRACT_TASK_ID

        print(f"{dag_id}/{upstream_task_id}/{dag_run_timestamp}.json")

        s3_hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
        response = s3_hook.read_key(
            key=f"{dag_id}/{upstream_task_id}/{dag_run_timestamp}.json",
            bucket_name=_S3_BUCKET,
        )
        api_response = json.loads(response)

        postgres_hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

        insert_sql = f"""
        INSERT INTO {_POSTGRES_SCHEMA}.{_POSTGRES_IN_TABLE} (raw_data)
        VALUES (%s::jsonb);
        """

        postgres_hook.run(insert_sql, parameters=(json.dumps(api_response),))

    _transform = SQLExecuteQueryOperator(
        task_id="transform_data",
        conn_id=_POSTGRES_CONN_ID,
        sql="transform.sql",
        params={
            "schema": _POSTGRES_SCHEMA,
            "in_table": _POSTGRES_IN_TABLE,
            "out_table": _POSTGRES_TRANSFORMED_TABLE,
        },
    )

    chain([_create_in_table_if_not_exists, _extract], load(), _transform)
    chain(_create_model_table_if_not_exists, _transform)


elt_intermediary_storage()
```

5

ETLT DAG using XCom

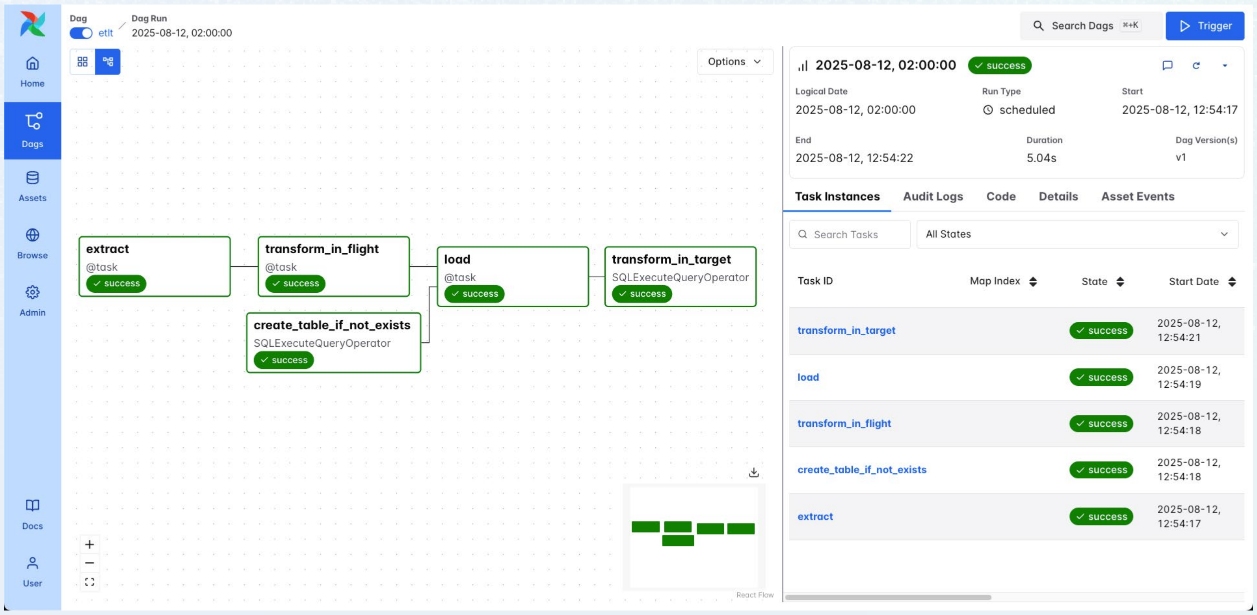
In real world scenarios ETL and ELT are often combined to form ETLT, a pattern where both types of transformations happen, once in flight before loading the data and once more after loading the data to a data warehouse.

This example shows the same ETL steps as the ETL pattern DAG using XCom DAG, with an added additional transformation step that creates a secondary table with aggregated information.

Note that the data is passed through XCom between the extraction and transformation task, as well as between the transformation and loading task, which means this setup would need a custom XCom backend in production.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

This is the DAG that will be created:



2025-08-12, 02:00:00 ✓ success

Logical Date	Run Type	Start
2025-08-12, 02:00:00	scheduled	2025-08-12, 12:54:17

End	Duration	Dag Version(s)
2025-08-12, 12:54:22	5.04s	v1

Task Instances | Audit Logs | Code | Details | Asset Events

Search Tasks | All States

Task ID	Map Index	State	Start Date
transform_in_target		✓ success	2025-08-12, 12:54:21
load		✓ success	2025-08-12, 12:54:19
transform_in_flight		✓ success	2025-08-12, 12:54:18
create_table_if_not_exists		✓ success	2025-08-12, 12:54:18
extract		✓ success	2025-08-12, 12:54:17

```
"""
## Simple ETLT DAG loading data from the Open-Meteo API to a Postgres database

This DAG extracts weather data from the Open-Meteo API, transforms it, and
loads it into a Postgres database, where a second transformation is applied.
Note that it uses XComs to pass data between tasks.
"""

import os
from datetime import datetime, timedelta

from airflow.sdk import dag, task, chain, Param
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from include.col_orders import WEATHER_COL_ORDER

# ------------------- #
# DAG-level variables #
# ------------------- #

DAG_ID = os.path.basename(__file__).replace(".py", "")

_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"
)
```

```
_POSTGRES_SECONDARY_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_SECONDARY", f"secondary_weather_data_{DAG_ID}"
)
_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"



# -------------- #
# DAG definition #
# -------------- #


@dag(
    dag_id=DAG_ID,
    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled
    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options
    max_active_runs=1,  # maximum number of active DAG runs
    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental
    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial
    default_args={
        "owner": "Astro",  # owner of this DAG in the Airflow UI
        "retries": 3,  # tasks retry 3 times before they fail
        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries
    },
    tags=["Patterns", "ETL", "Postgres", "XCom"],  # add tags in the UI
    params={
        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")
    },  # Airflow params can add interactive options on manual runs.
See: https://www.astronomer.io/docs/learn/airflow-params

    template_searchpath=[_SQL_DIR],  # path to the SQL templates

)

def etlt():


    # ---------------- #

    # Task Definitions #

    # ---------------- #

    # the @task decorator turns any Python function into an Airflow task

    # any @task decorated function that is called inside the @dag decorated

    # function is automatically added to the DAG.

    # if one exists for your use case you can use traditional Airflow operators

    # and mix them with @task decorators. Checkout registry.astronomer.io for available
operators

    # see: https://www.astronomer.io/docs/learn/airflow-decorators for information about
@task

    # see: https://www.astronomer.io/docs/learn/what-is-an-operator for information
about traditional operators


    _create_table_if_not_exists = SQLExecuteQueryOperator(

        task_id="create_table_if_not_exists",

        conn_id=_POSTGRES_CONN_ID,

        database=_POSTGRES_DATABASE,

        sql="create_table_if_not_exists.sql",

        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE},

    )


    @task

    def extract(**context):

        """

        Extract data from the Open-Meteo API
        Returns:
            dict: The full API response
        """
        import requests

        url = os.getenv("WEATHER_API_URL")

        coordinates = context["params"]["coordinates"]
        latitude = coordinates["latitude"]
        longitude = coordinates["longitude"]

        url = url.format(latitude=latitude, longitude=longitude)

        response = requests.get(url)

        return response.json()

    @task
    def transform_in_flight(api_response: dict, **context) -> dict:
        """
        Transform the data
        Args:
            api_response (dict): The full API response
        Returns:
            dict: The transformed data
        """

        time = api_response["hourly"]["time"]
        dag_run_timestamp = context["ts"]
        transformed_data = {
            "temperature_2m": api_response["hourly"]["temperature_2m"],
            "relative_humidity_2m": api_response["hourly"]["relative_humidity_2m"],
            "precipitation_probability": api_response["hourly"][
                "precipitation_probability"
            ],
            "timestamp": time,
            "date": [
                datetime.strptime(x, "%Y-%m-%dT%H:%M").date().strftime("%Y-%m-%d")
                for x in time
            ],
            "day": [datetime.strptime(x, "%Y-%m-%dT%H:%M").day for x in time],
            "month": [datetime.strptime(x, "%Y-%m-%dT%H:%M").month for x in time],
            "year": [datetime.strptime(x, "%Y-%m-%dT%H:%M").year for x in time],
            "last_updated": [dag_run_timestamp for i in range(len(time))],
            "latitude": [api_response["latitude"] for i in range(len(time))],
            "longitude": [api_response["longitude"] for i in range(len(time))],
        }

        return transformed_data

    @task
    def load(transformed_data: dict):
        """
        Load the data to the destination without using a temporary CSV file.
        Args:
            transformed_data (dict): The transformed data
        """
        import csv

        import io

        from airflow.providers.postgres.hooks.postgres import PostgresHook


        hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)


        csv_buffer = io.StringIO()

        writer = csv.writer(csv_buffer)

        writer.writerow(WEATHER_COL_ORDER)

        rows = zip(*[transformed_data[col] for col in WEATHER_COL_ORDER])

        writer.writerows(rows)


        csv_buffer.seek(0)


        with open(f"{_SQL_DIR}/copy_insert.sql") as f:

            sql = f.read()

        sql = sql.replace("{schema}", _POSTGRES_SCHEMA)

        sql = sql.replace("{table}", _POSTGRES_TRANSFORMED_TABLE)


        conn = hook.get_conn()

        cursor = conn.cursor()

        cursor.copy_expert(sql=sql, file=csv_buffer)

        conn.commit()

        cursor.close()

        conn.close()


    _transform_in_target = SQLExecuteQueryOperator(

        task_id="transform_in_target",

        conn_id=_POSTGRES_CONN_ID,

        sql="transform.sql",
        params={
            "schema": _POSTGRES_SCHEMA,
            "in_table": _POSTGRES_TRANSFORMED_TABLE,
            "out_table": _POSTGRES_SECONDARY_TABLE,
        },
    )

    _extract = extract()
    _transform = transform_in_flight(api_response=_extract)
    _load = load(transformed_data=_transform)
    chain(_transform, _load)
    chain(_create_table_if_not_exists, _load, _transform_in_target)


etlt()
```

Advanced ETL DAG with data quality checks

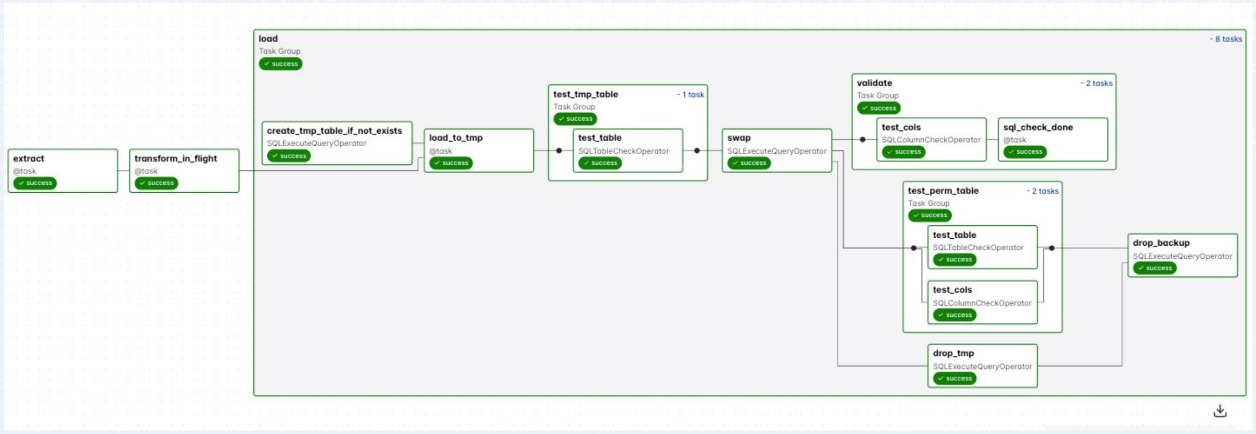
6

This advanced DAG has the same extraction and transforming steps as the ETL pattern DAG using XCom but it loads the data using a temporary table and includes several data quality checks.

In summary this DAG:

1. Extracts weather data from the Open-Meteo API
2. Transforms the data
3. Loads the data into a temporary table in Postgres
4. Performs stopping data quality checks on the temporary table
5. Swaps the temporary table with the permanent table (and creates a backup of the old permanent table)
6. Drops the temporary table
7. Performs stopping data quality checks on the new permanent table
8. Drops the backup table
9. Performs additional warning data quality checks on the new permanent table

This is the DAG that is created:



You can learn more about how to use the SQL check operators to write data quality checks here.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

```
"""

## ETL with Data Quality Checks and Temporary Tables


This DAG will perform the following steps:

1. Extract weather data from the Open-Meteo API

2. Transform the data

3. Load the data into a temporary table in Postgres

4. Perform stopping data quality checks on the temporary table

5. Swap the temporary table with the permanent table (and creating a backup of the old
permanent table)

6. Drop the temporary table

7. Perform stopping data quality checks on the new permanent table

8. Drop the backup table

9. Perform additional warning data quality checks on the new permanent table

"""


import os

from datetime import datetime, timedelta


from airflow.sdk import dag, task, task_group, chain, Param

from airflow.providers.common.sql.operators.sql import (

    SQLColumnCheckOperator,

    SQLExecuteQueryOperator,

    SQLTableCheckOperator,

)

from include.col_orders import WEATHER_COL_ORDER
```

```
DAG_ID = os.path.basename(__file__).replace(".py", "")


_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")

_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")

_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")

_POSTGRES_TMP_TABLE = os.getenv(

    "POSTGRES_WEATHER_TABLE_TMP", f"tmp_weather_data_{DAG_ID}"

)

_POSTGRES_BACKUP_TABLE = os.getenv(

    "POSTGRES_WEATHER_TABLE_BACKUP", f"backup_weather_data_{DAG_ID}"

)

_POSTGRES_TRANSFORMED_TABLE = os.getenv(

    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"

)


_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"




@dag(

    dag_id=DAG_ID,

    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled

    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options

    max_active_runs=1,  # maximum number of active DAG runs

    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental

    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial

    default_args={

        "owner": "Astro",  # owner of this DAG in the Airflow UI

        "retries": 3,  # tasks retry 3 times before they fail
        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries
        "postgres_conn_id": _POSTGRES_CONN_ID,
        "conn_id": _POSTGRES_CONN_ID,
    },
    tags=["Patterns", "ETL", "Postgres", "XCom"],  # add tags in the UI
    params={
        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")
    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params
    template_searchpath=[_SQL_DIR],  # path to the SQL templates
)
def etl_dq_checks_tmp_tables():


    @task
    def extract(**context):
        """
        Extract data from the Open-Meteo API
        Returns:
            dict: The full API response
        """
        import requests


        url = os.getenv("WEATHER_API_URL")


        coordinates = context["params"]["coordinates"]
        latitude = coordinates["latitude"]
        longitude = coordinates["longitude"]


        url = url.format(latitude=latitude, longitude=longitude)
        response = requests.get(url)

        return response.json()

    @task
    def transform_in_flight(api_response: dict, **context) -> dict:
        """
        Transform the data
        Args:
            api_response (dict): The full API response
        Returns:
            dict: The transformed data
        """

        time = api_response["hourly"]["time"]
        dag_run_timestamp = context["ts"]

        transformed_data = {
            "temperature_2m": api_response["hourly"]["temperature_2m"],
            "relative_humidity_2m": api_response["hourly"]["relative_humidity_2m"],
            "precipitation_probability": api_response["hourly"][
                "precipitation_probability"
            ],
            "timestamp": time,
            "date": [
                datetime.strptime(x, "%Y-%m-%dT%H:%M").date().strftime("%Y-%m-%d")
                for x in time
            ],
            "day": [datetime.strptime(x, "%Y-%m-%dT%H:%M").day for x in time],
            "month": [datetime.strptime(x, "%Y-%m-%dT%H:%M").month for x in time],
            "year": [datetime.strptime(x, "%Y-%m-%dT%H:%M").year for x in time],
            “last_updated”: [dag_run_timestamp for i in range(len(time))],
            “latitude”: [api_response[“latitude”] for i in range(len(time))],
            “longitude”: [api_response[“longitude”] for i in range(len(time))],
        }

        return transformed_data

    _extract = extract()
    _transform = transform_in_flight(_extract)

    @task_group
    def load():
        _create_tmp_table_if_not_exists = SQLExecuteQueryOperator(
            task_id=”create_tmp_table_if_not_exists”,
            database=_POSTGRES_DATABASE,
            sql=”create_tmp_table_if_not_exists.sql”,
            params={“schema”: _POSTGRES_SCHEMA, “table”: _POSTGRES_TMP_TABLE},
        )

        @task
        def load_to_tmp(transformed_data: dict):
            “””
            Load the data to the destination without using a temporary CSV file.
            Args:
                transformed_data (dict): The transformed data
            “””
            import csv
            import io

            from airflow.providers.postgres.hooks.postgres import PostgresHook
            hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

            csv_buffer = io.StringIO()
            writer = csv.writer(csv_buffer)
            writer.writerow(WEATHER_COL_ORDER)
            rows = zip(*[transformed_data[col] for col in WEATHER_COL_ORDER])
            writer.writerows(rows)

            csv_buffer.seek(0)

            with open(f"{_SQL_DIR}/copy_insert.sql") as f:
                sql = f.read()
            sql = sql.replace("{schema}", _POSTGRES_SCHEMA)
            sql = sql.replace("{table}", _POSTGRES_TMP_TABLE)

            conn = hook.get_conn()
            cursor = conn.cursor()
            cursor.copy_expert(sql=sql, file=csv_buffer)
            conn.commit()
            cursor.close()
            conn.close()

        _load_to_tmp = load_to_tmp(_transform)

        @task_group
        def test_tmp_table():

            SQLTableCheckOperator(
                task_id="test_table",
                retry_on_failure="True",

                table=f"{_POSTGRES_SCHEMA}.{_POSTGRES_TMP_TABLE}",

                checks={

                    "row_count_check": {"check_statement": "COUNT(*) > 10"},

                },

            )


        swap = SQLExecuteQueryOperator(

            task_id="swap",

            sql="swap.sql",

            params={

                "schema": _POSTGRES_SCHEMA,

                "tmp_table": _POSTGRES_TMP_TABLE,

                "perm_table": _POSTGRES_TRANSFORMED_TABLE,

            },

        )


        drop_tmp = SQLExecuteQueryOperator(

            task_id="drop_tmp",

            sql=f"DROP TABLE IF EXISTS {_POSTGRES_SCHEMA}.{_POSTGRES_TMP_TABLE};",

        )


        @task_group

        def test_perm_table():

            SQLColumnCheckOperator(

                task_id="test_cols",

                retry_on_failure="True",

                table=f"{_POSTGRES_SCHEMA}.{_POSTGRES_TRANSFORMED_TABLE}",

                column_mapping={

                    "month": {"min": {"geq_to": 1}, "max": {"leq_to": 12}},
                    “day”: {“min”: {“geq_to”: 1}, “max”: {“leq_to”: 31}},
                },
                accept_none=”True”,
            )

            SQLTableCheckOperator(
                task_id=”test_table”,
                retry_on_failure=”True”,
                table=f”{_POSTGRES_SCHEMA}.{_POSTGRES_TRANSFORMED_TABLE}”,
                checks={
                    “row_count_check”: {“check_statement”: “COUNT(*) > 10”},
                },
            )

        drop_backup = SQLExecuteQueryOperator(
            task_id=”drop_backup”,
            sql=f”DROP TABLE IF EXISTS {_POSTGRES_SCHEMA}.{_POSTGRES_BACKUP_TABLE};”,
        )

        chain(
            _create_tmp_table_if_not_exists,
            _load_to_tmp,
            test_tmp_table(),
            swap,
            [drop_tmp, test_perm_table()],
            drop_backup,
        )

        @task_group
        def validate():
```

```
            test_cols = SQLColumnCheckOperator(
                task_id="test_cols",
                retry_on_failure="True",
                table=f"{_POSTGRES_SCHEMA}.{_POSTGRES_TRANSFORMED_TABLE}",
                column_mapping={
                    "temperature_2m": {"min": {"geq_to": -10}, "max": {"leq_to": 50}},
                },
                accept_none="True",
            )

            @task(trigger_rule="all_done")
            def sql_check_done():
                return "Additional data quality checks are done!"

            test_cols >> sql_check_done()

        swap >> validate()

    load()


etl_dq_checks_tmp_tables()
```

7

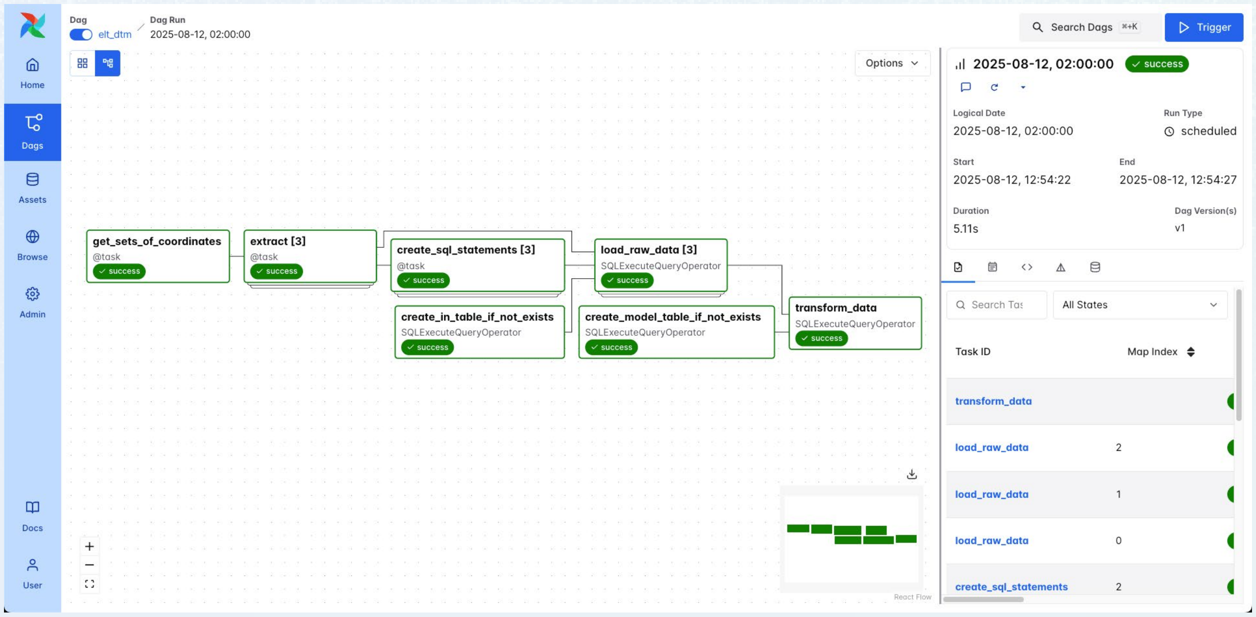
ELT DAG using Dynamic Task Mapping

The DAG below is similar to the ELT pattern DAG using XCom, but instead of just fetching data from one API call including one set of coordinates, it uses dynamic task mapping to run 1 extraction and 1 loading tasks in parallel for each set of coordinates specified at runtime via the DAG configuration params.

The data is passed through XCom between the extraction and loading task, which means this setup would need a custom XCom backend in production.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

This is the DAG that will be created:



2025-08-12, 02:00:00 success

Logical Date 2025-08-12, 02:00:00
Run Type scheduled
Start 2025-08-12, 12:54:22
End 2025-08-12, 12:54:27
Duration 5.11s
Dag Version(s) v1

Search Tas | All States

Task ID	Map Index
transform_data	
load_raw_data	2
load_raw_data	1
load_raw_data	0
create_sql_statements	2

```
"""
## ELT DAG extracting data from 3 API calls in parallel using Dynamic Tasks

This DAG extracts weather data from the Open-Meteo API, loads it into a
ostgres database and transforms it, using an ELT pattern.
It extracts data from 3 different locations in parallel using dynamic task mapping.
"""

import json
import os
from datetime import datetime, timedelta

from airflow.sdk import dag, task, chain, Param
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator

# ------------------- #
# DAG-level variables #
# ------------------- #

DAG_ID = os.path.basename(__file__).replace(".py", "")

_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_IN_TABLE = os.getenv("POSTGRES_WEATHER_TABLE_IN", f"in_weather_data_{DAG_ID}")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{DAG_ID}"
)
```

```
_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/pattern_dags/{DAG_ID}"


# -------------- #
# DAG definition #
# -------------- #


@dag(
    dag_id=DAG_ID,
    start_date=datetime(2025, 8, 1),  # date after which the DAG can be scheduled
    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options
    max_active_runs=1,  # maximum number of active DAG runs
    max_consecutive_failed_dag_runs=5,  # auto-pauses the DAG after 5 consecutive failed
runs, experimental
    doc_md=__doc__,  # add DAG Docs in the UI, see https://www.astronomer.io/docs/learn/
custom-airflow-ui-docs-tutorial
    default_args={
        "owner": "Astro",  # owner of this DAG in the Airflow UI
        "retries": 3,  # tasks retry 3 times before they fail
        "retry_delay": timedelta(seconds=30),  # tasks wait 30s in between retries
    },
    tags=["Patterns", "ELT", "Postgres", "XCom"],  # add tags in the UI
    params={
        "set_of_coordinates": Param(
            {
                "Location 1": {"latitude": 46.9480, "longitude": 7.4474},
                "Location 2": {"latitude": 38.4272, "longitude": 14.9524},
                "Location 3": {"latitude": 46.9480, "longitude": -2.9916},
            },
            type="object",
        )
    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params
    template_searchpath=[_SQL_DIR],  # path to the SQL templates
)
def elt_dtm():


    # ---------------- #
    # Task Definitions #
    # ---------------- #
    # the @task decorator turns any Python function into an Airflow task
    # any @task decorated function that is called inside the @dag decorated
    # function is automatically added to the DAG.
    # if one exists for your use case you can use traditional Airflow operators
    # and mix them with @task decorators. Checkout registry.astronomer.io for available
operators
    # see: https://www.astronomer.io/docs/learn/airflow-decorators for information about
@task
    # see: https://www.astronomer.io/docs/learn/what-is-an-operator for information
about traditional operators


    _create_in_table_if_not_exists = SQLExecuteQueryOperator(
        task_id="create_in_table_if_not_exists",
        conn_id=_POSTGRES_CONN_ID,
        database=_POSTGRES_DATABASE,
        sql="create_in_table_if_not_exists.sql",
        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_IN_TABLE},
    )


    _create_model_table_if_not_exists = SQLExecuteQueryOperator(
        task_id="create_model_table_if_not_exists",
        conn_id=_POSTGRES_CONN_ID,

        database=_POSTGRES_DATABASE,

        sql="create_model_table_if_not_exists.sql",

        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE},

    )


    @task

    def get_sets_of_coordinates(**context):

        """

        Get the set of coordinates from the params

        """

        coordinate_dict = context["params"]["set_of_coordinates"]

        list_of_coordinates = []

        for k, v in coordinate_dict.items():

            list_of_coordinates.append(v)


        return list_of_coordinates


    _get_sets_of_coordinates = get_sets_of_coordinates()


    @task(map_index_template="{{ my_custom_map_index }}")

    def extract(coordinates: dict):

        """

        Extract data from the Open-Meteo API

        Returns:

            dict: The full API response

        """

        import requests


        url = os.getenv("WEATHER_API_URL")
```

```
        latitude = coordinates["latitude"]

        longitude = coordinates["longitude"]


        url = url.format(latitude=latitude, longitude=longitude)


        response = requests.get(url)


        # optional custom map index (2.9+)

        from airflow.sdk import get_current_context


        context = get_current_context()

        context["my_custom_map_index"] = (

            "Coordinates: " + str(latitude) + ", " + str(longitude)

        )


        return response.json()


    _extract = extract.expand(coordinates=_get_sets_of_coordinates)


    @task

    def create_sql_statements(data, schema, table):

        """

        Create SQL statements to load the data

        """

        data_json = json.dumps(data)


        return f"""

            INSERT INTO {schema}.{table} (raw_data)

            VALUES ('{data_json}'::jsonb);

            """
    _create_sql_statements = create_sql_statements.partial(
        schema=_POSTGRES_SCHEMA, table=_POSTGRES_IN_TABLE
    ).expand(data=_extract)

    _load = SQLExecuteQueryOperator.partial(
        task_id="load_raw_data",
        conn_id=_POSTGRES_CONN_ID,
    ).expand(sql=_create_sql_statements)

    _transform = SQLExecuteQueryOperator(
        task_id="transform_data",
        conn_id=_POSTGRES_CONN_ID,
        sql="transform.sql",
        params={
            "schema": _POSTGRES_SCHEMA,
            "in_table": _POSTGRES_IN_TABLE,
            "out_table": _POSTGRES_TRANSFORMED_TABLE,
        },
    )

    chain([_create_in_table_if_not_exists, _extract], _load, _transform)
    chain(_create_model_table_if_not_exists, _transform)


elt_dtm()
```

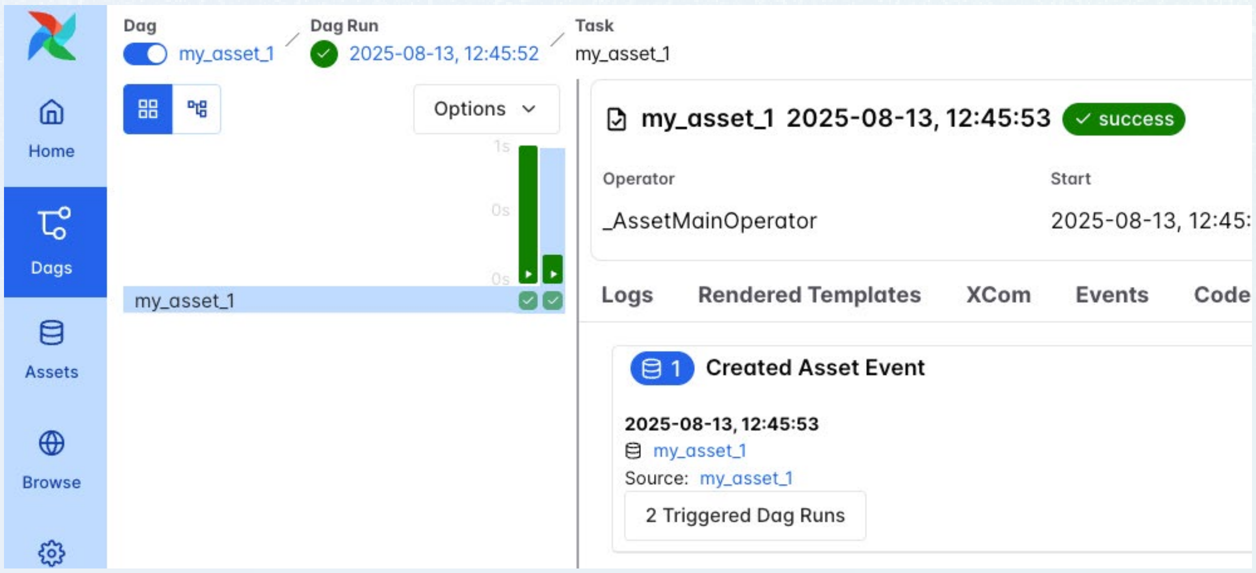
PART TWO

Asset Code Examples

Airflow 3 introduced an additional second way to write your pipeline: the asset-centric approach. You can use the `@asset` decorator as a shorthand to create one DAG containing one task that updates one Airflow Asset object.

```
from airflow.sdk import asset


@asset(schedule=None, tags=["basic_asset_example"])
def my_asset_1():
    print("Add any Python Code!")
```

The code above creates one DAG with the DAG ID `my_asset_1` containing one task with the task ID `my_asset_1` that produces updates to the Airflow Asset `my_asset_1`.



my_asset_1 2025-08-13, 12:45:53 ✓ success

Operator: _AssetMainOperator

Start: 2025-08-13, 12:45:

1 Created Asset Event

2025-08-13, 12:45:53
my_asset_1
Source: my_asset_1
2 Triggered Dag Runs

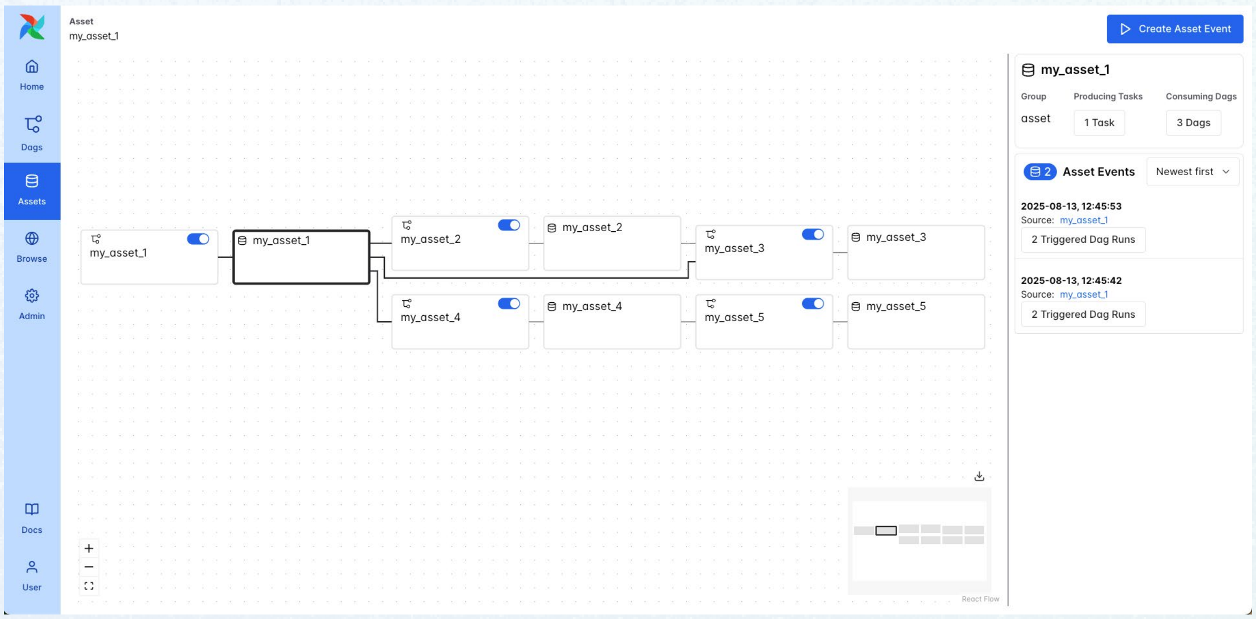
Assets can run on a schedule like regular DAGs, including data-aware schedules, i.e. an `@asset` can be scheduled based on updates to Assets. See the **Asset and data-aware scheduling guide** for more information.

It is also possible to push data to XCom from assets and retrieve them using a cross-DAG XCom pull, as well as adjust small amounts of `Metadata` to the extra dictionary of the Asset event.

The following code snippet shows several assets that depend on each other, push to and pull from XCom and use the extra dictionary of the Asset event. You can find the example **here on GitHub**.

```
from airflow.sdk import asset, Metadata, Asset


@asset(schedule=None, tags=["basic_asset_example"])

def my_asset_1():

    print("Add any Python Code!")


@asset(schedule=[my_asset_1], tags=["basic_asset_example"])

def my_asset_2():

    # you can return data from an @asset decorated function

    # to push it to XCom

    return {"a": 1, "b": 2}


@asset(schedule=[my_asset_1, my_asset_2], tags=["basic_asset_example"])

def my_asset_3(context):

    # you can pull the data from XCom via the context

    # note that this is a cross-dag xcom pull

    data = context["ti"].xcom_pull(

        dag_id="my_asset_2",

        task_ids=["my_asset_2"],

        key="return_value",

        include_prior_dates=True,

    )[0]


    print(data)


@asset(schedule=[my_asset_1], tags=["basic_asset_example"])

def my_asset_4(context):
    # you can also attache Metadata to the Asset Event
    yield Metadata(Asset("my_asset_4"), {"a": 1, "b": 2})


@asset(schedule=[my_asset_4], tags=["basic_asset_example"])
def my_asset_5(context):
    # and retrieve the Metadata from the Asset Event
    metadata = context["triggering_asset_events"][Asset("my_asset_4")][0].extra
    print(metadata)
```

In the Airflow 3 UI you can view asset dependencies by selecting a graph from the list shown under the `Assets` button. The following asset graph is created by the example code above.



my_asset_1

Group	Producing Tasks	Consuming Dags
asset	1 Task	3 Dags

2 Asset Events — Newest first

2025-08-13, 12:45:53
Source: my_asset_1
2 Triggered Dag Runs

2025-08-13, 12:45:42
Source: my_asset_1
2 Triggered Dag Runs

8

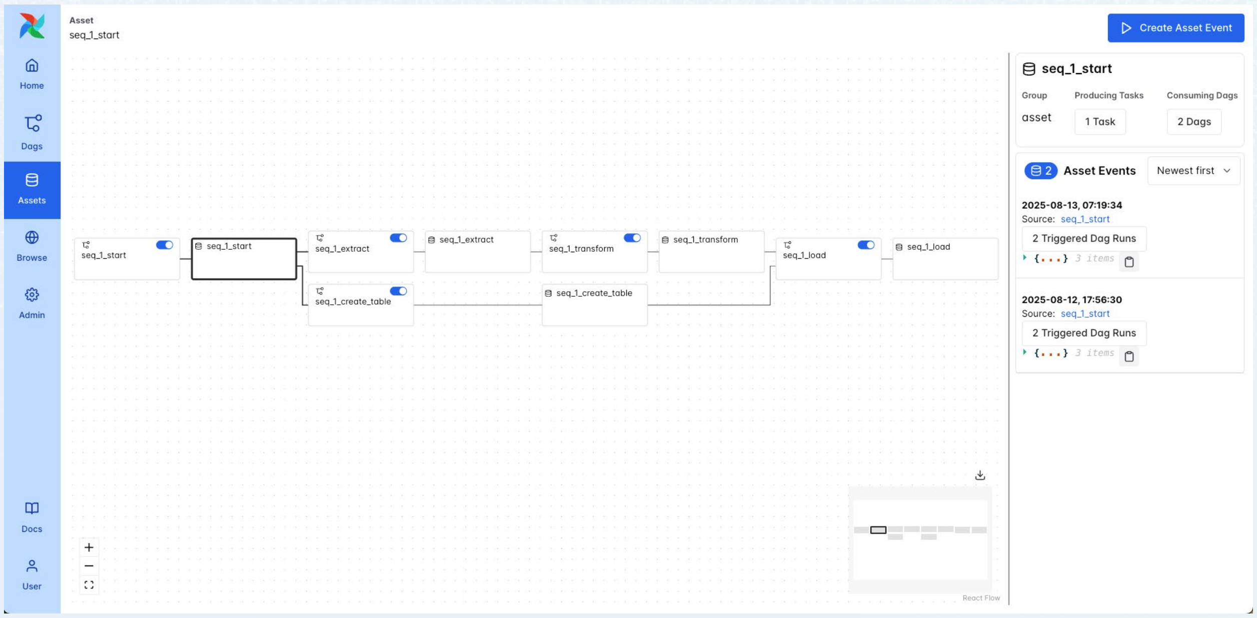
Asset sequence ETL with XCom

The sequence of assets below mirrors the ETL pattern DAG using XCom using the `@asset` decorator instead.

The sequence shows an ETL pattern to extract data from the Open-Meteo API, transform it and then load it into a Postgres database. The data is passed through XCom between the extraction and transformation task, as well as between the transformation and loading task, which means this setup would need a custom XCom backend in production.

In the companion GitHub repository you can find this DAG here and its supporting SQL scripts here.

This is the asset graph that is created:



seq_1_start

Group	Producing Tasks	Consuming Dags
asset	1 Task	2 Dags

2 Asset Events — Newest first

2025-08-13, 07:19:34
Source: seq_1_start
2 Triggered Dag Runs
{...} 3 items

2025-08-12, 17:56:30
Source: seq_1_start
2 Triggered Dag Runs
{...} 3 items

```
"""
## Simple asset sequence creating 4 DAGs to perform an ETL pattern

These 4 asset decorated functions created 4 DAGs each containing one task
following a similar pattern as the regular etl_xcom DAG.

Note that passing the data between assets via XCom necessitates a Cross-DAG xcom pull
and other information like DAG params and timestamps are passed via the metadata
asset events.
"""

import os
from datetime import datetime

from airflow.sdk import Param, asset, Metadata, Asset
from airflow.providers.postgres.hooks.postgres import PostgresHook
from include.col_orders import WEATHER_COL_ORDER

# ------------------- #
# DAG-level variables #
# ------------------- #

SEQUENCE_NAME = os.path.basename(__file__).replace(".py", "")

_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{SEQUENCE_NAME}"
)

_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/asset_sequences/{SEQUENCE_NAME}"
```

```
# First asset definition

@asset(

    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options

    tags=["assets", "seq_1", "XCom"],  # add tags in the UI

    params={

        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")

    },

)

def seq_1_start(context):

    yield Metadata(Asset("seq_1_start"), {

            "latitude": context["params"]["coordinates"]["latitude"],

            "longitude": context["params"]["coordinates"]["longitude"],

            "ts": context["ts"],

        },

    )



@asset(

    schedule=[seq_1_start],

    tags=["assets", "seq_1", "XCom"],

)

def seq_1_create_table(context):

    from jinja2 import Template


    hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)


    with open(f"{_SQL_DIR}/create_table_if_not_exists.sql", "r") as f:

        sql_template = f.read()


    template = Template(sql_template)

    sql = template.render(

        params={"schema": _POSTGRES_SCHEMA, "table": _POSTGRES_TRANSFORMED_TABLE}

    )


    hook.run(sql)


    # You can attach metadata to an asset event. This is useful for passing small
amounts of

    # data specific to a sequence.

    # See: https://www.astronomer.io/docs/learn/airflow-datasets/#attaching-informa-
tion-to-an-asset-event
```

```
@asset(
    schedule=[seq_1_start],
    tags=["assets", "seq_1", "XCom"],
    params={
        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")
    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params
)
def seq_1_extract(context):
    """
    Extract data from the Open-Meteo API
    Returns:
        dict: The full API response
    """
    import requests

    url = os.getenv("WEATHER_API_URL")

    # if the DAG is run manually use the time stamp and lat/long of the manual run
    if str(context["dag_run"].run_type) == "DagRunType.MANUAL":
        latitude = context["params"]["coordinates"]["latitude"]
        longitude = context["params"]["coordinates"]["longitude"]
        ts = context["ts"]
    # if the DAG is run as part of the asset sequence use the lat/long and timestamp of
the upstream asset event
    else:
        # You can retrieve metadata from the triggering asset event.
        # See: https://www.astronomer.io/docs/learn/airflow-datasets/#retrieving-as-
set-information-in-a-downstream-task
        metadata_upstream = context["triggering_asset_events"][
            Asset("seq_1_start")
        ][0].extra

        latitude = metadata_upstream["latitude"]
        longitude = metadata_upstream["longitude"]
        ts = metadata_upstream["ts"]

    url = url.format(latitude=latitude, longitude=longitude)
    response = requests.get(url)

    # attach the timestamp of the DAG run to the asset event
    yield Metadata(Asset("seq_1_extract"), {"ts": ts})

    # returning a value from an @asset pushes it to XCom
    return response.json()


@asset(
    schedule=[seq_1_extract],
    tags=["assets", "seq_1", "XCom"],
)
def seq_1_transform(context) -> dict:
    """
    Transform the data
    Args:
        api_response (dict): The full API response
    Returns:
        dict: The transformed data
    """

    # if the DAG is run manually use the time stamp of the manual run
    if str(context["dag_run"].run_type) == "DagRunType.MANUAL":
        dag_run_timestamp = context["ts"]
    # if the DAG is run as part of the asset sequence use the timestamp of the upstream 
asset event
    else:
        # You can retrieve metadata from the triggering asset event.
        # See: https://www.astronomer.io/docs/learn/airflow-datasets/#retrieving-as-
set-information-in-a-downstream-task
        metadata_upstream = context["triggering_asset_events"][Asset("seq_1_extract")][
            0
        ].extra

        dag_run_timestamp = metadata_upstream["ts"]

    # To retrieve data pushed by an upstream asset perform a cross-dag xcom pull
    api_response = context["ti"].xcom_pull(
        dag_id="seq_1_extract",
        task_ids=["seq_1_extract"],
        key="return_value",
        include_prior_dates=True,
    )[0]

    time = api_response["hourly"]["time"]

    transformed_data = {
        "temperature_2m": api_response["hourly"]["temperature_2m"],
        "relative_humidity_2m": api_response["hourly"]["relative_humidity_2m"],
        "precipitation_probability": api_response["hourly"][
            "precipitation_probability"
        ],
        "timestamp": time,
        "date": [
            datetime.strptime(x, "%Y-%m-%dT%H:%M").date().strftime("%Y-%m-%d")
            for x in time
        ],
        "day": [datetime.strptime(x, "%Y-%m-%dT%H:%M").day for x in time],
        "month": [datetime.strptime(x, "%Y-%m-%dT%H:%M").month for x in time],
        "year": [datetime.strptime(x, "%Y-%m-%dT%H:%M").year for x in time],
        "last_updated": [dag_run_timestamp for i in range(len(time))],
        "latitude": [api_response["latitude"] for i in range(len(time))],
        "longitude": [api_response["longitude"] for i in range(len(time))],
    }

    return transformed_data


@asset(
    schedule=[seq_1_create_table, seq_1_transform],
    tags=["assets", "seq_1", "XCom"],
)
def seq_1_load(context):
    """
    Load the data to the destination without using a temporary CSV file.
    Args:
        transformed_data (dict): The transformed data
    """
```

```
import csv
import io
from airflow.providers.postgres.hooks.postgres import PostgresHook

transformed_data = context["ti"].xcom_pull(
    dag_id="seq_1_transform",
    task_ids=["seq_1_transform"],
    key="return_value",
    include_prior_dates=True,
)[0]

hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

csv_buffer = io.StringIO()
writer = csv.writer(csv_buffer)
writer.writerow(WEATHER_COL_ORDER)
rows = zip(*[transformed_data[col] for col in WEATHER_COL_ORDER])
writer.writerows(rows)

csv_buffer.seek(0)

with open(f"{_SQL_DIR}/copy_insert.sql") as f:
    sql = f.read()
sql = sql.replace("{schema}", _POSTGRES_SCHEMA)
sql = sql.replace("{table}", _POSTGRES_TRANSFORMED_TABLE)

conn = hook.get_conn()
cursor = conn.cursor()
cursor.copy_expert(sql=sql, file=csv_buffer)
conn.commit()
cursor.close()
conn.close()
```

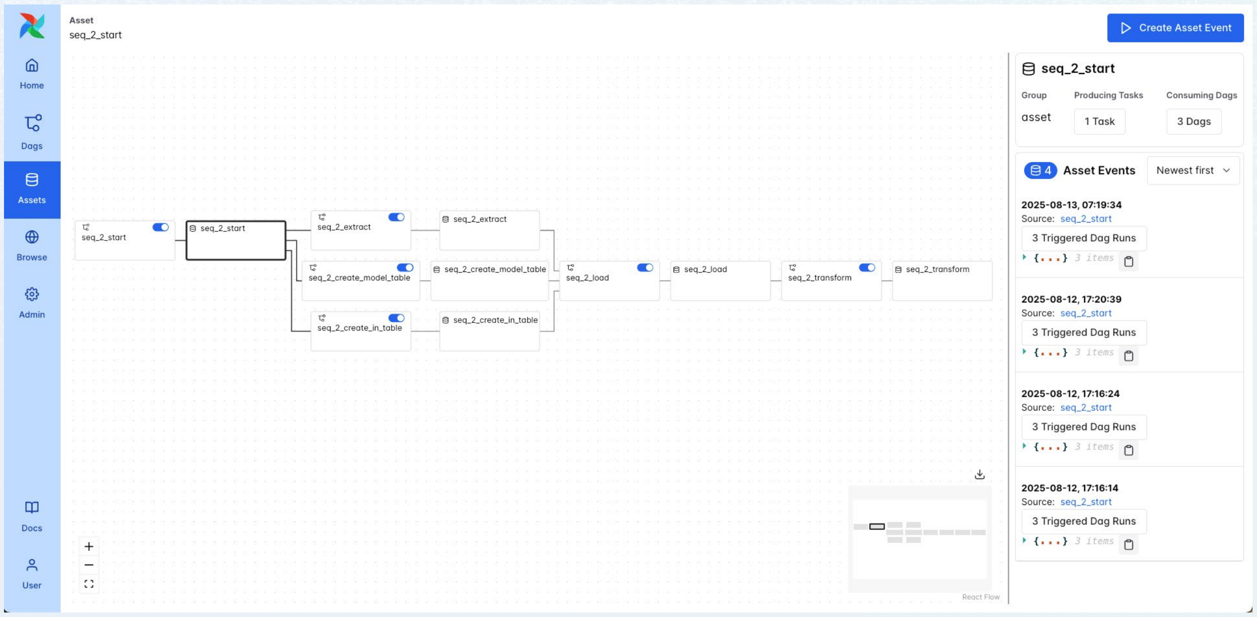
9

Asset sequence ETL using explicit external storage

This asset sequence accomplishes the same as the ELT pattern DAG using explicit external storage, extracting data from the Open-Meteo API, loading the raw data into a Postgres database and then running a transformation on it, but without passing any data through XCom.

In the companion GitHub repository you can find this DAG **here** and its supporting SQL scripts **here**.

This is the asset graph that is created:



seq_2_start

Group	Producing Tasks	Consuming Dags
asset	1 Task	3 Dags

4 Asset Events — Newest first

2025-08-13, 07:19:34
Source: seq_2_start
3 Triggered Dag Runs
{...} 3 items

2025-08-12, 17:20:39
Source: seq_2_start
3 Triggered Dag Runs
{...} 3 items

2025-08-12, 17:16:24
Source: seq_2_start
3 Triggered Dag Runs
{...} 3 items

2025-08-12, 17:16:14
Source: seq_2_start
3 Triggered Dag Runs
{...} 3 items

```
"""
## Simple asset sequence creating 4 DAGs to perform an ELT pattern with intermediary
storage

These 4 asset decorated functions created 4 DAGs each containing one task
following a similar pattern as the regular elt_intermediary_storage DAG.

Note that passing the data between assets via XCom necessitates a Cross-DAG xcom pull
and other information like DAG params and timestamps are passed via the metadata
asset events.
"""

import os
import json

from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.sdk import Param, asset, Metadata, Asset
from airflow.providers.postgres.hooks.postgres import PostgresHook

# ------------------- #
# DAG-level variables #
# ------------------- #

SEQUENCE_NAME = os.path.basename(__file__).replace(".py", "")

_AWS_CONN_ID = os.getenv("MINIO_CONN_ID", "minio_local")
_S3_BUCKET = os.getenv("S3_BUCKET", "open-meteo-etl")

_POSTGRES_CONN_ID = os.getenv("POSTGRES_CONN_ID", "postgres_default")
_POSTGRES_DATABASE = os.getenv("POSTGRES_DATABASE", "postgres")
_POSTGRES_SCHEMA = os.getenv("POSTGRES_SCHEMA", "public")
_POSTGRES_IN_TABLE = os.getenv("POSTGRES_WEATHER_TABLE_IN", f"in_weather_data_{SEQUENCE_
NAME}")
_POSTGRES_TRANSFORMED_TABLE = os.getenv(
    "POSTGRES_WEATHER_TABLE_TRANSFORMED", f"model_weather_data_{SEQUENCE_NAME}"
)


_SQL_DIR = f"{os.getenv('AIRFLOW_HOME')}/include/sql/asset_sequences/{SEQUENCE_NAME}"


_INTERMEDIARY_STORAGE_KEY = "seq_2_extract"




@asset(

    schedule="@daily",  # see: https://www.astronomer.io/docs/learn/scheduling-in-air-
flow for options

    tags=["assets", "seq_2", "XCom"],  # add tags in the UI

    params={

        "coordinates": Param({"latitude": 46.9481, "longitude": 7.4474}, type="object")

    },  # Airflow params can add interactive options on manual runs. See: https://www.
astronomer.io/docs/learn/airflow-params

)

def seq_2_start(context):

    yield Metadata(

        Asset("seq_2_start"),

        {

            "latitude": context["params"]["coordinates"]["latitude"],

            "longitude": context["params"]["coordinates"]["longitude"],

            "ts": context["ts"],

        },

    )




@asset(

    schedule=[seq_2_start],

    tags=["assets", "seq_2", "XCom"],

)

def seq_2_create_in_table(context):

    from jinja2 import Template


    hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)


    with open(f"{_SQL_DIR}/create_in_table_if_not_exists.sql", "r") as f:
            sql_template = f.read()

        template = Template(sql_template)
        sql = template.render(
            params={“schema”: _POSTGRES_SCHEMA, “table”: _POSTGRES_IN_TABLE}
        )

        hook.run(sql)


@asset(
    schedule=[seq_2_start],
    tags=[“assets”, “seq_2”, “XCom”],
)
def seq_2_create_model_table(context):
    from jinja2 import Template

    hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

    with open(f”{_SQL_DIR}/create_model_table_if_not_exists.sql”, “r”) as f:
        sql_template = f.read()

    template = Template(sql_template)
    sql = template.render(
        params={“schema”: _POSTGRES_SCHEMA, “table”: _POSTGRES_TRANSFORMED_TABLE}
    )

    hook.run(sql)


@asset(
    schedule=[seq_2_start],
    tags=[“assets”, “seq_2”, “XCom”],
    params={
        “coordinates”: Param({“latitude”: 46.9481, “longitude”: 7.4474}, type=”object”)
    },
)
```

```
def seq_2_extract(context):
    """
    Extract data from the Open-Meteo API
    Returns:
        dict: The full API response
    """
    import requests

    # if the DAG is run manually use the time stamp and lat/long of the manual run
    if str(context["dag_run"].run_type) == "DagRunType.MANUAL":
        latitude = context["params"]["coordinates"]["latitude"]
        longitude = context["params"]["coordinates"]["longitude"]
        dag_run_timestamp = context["ts"]
    # if the DAG is run as part of the asset sequence use the lat/long and timestamp of
the upstream asset event
    else:
        # You can retrieve metadata from the triggering asset event.
        # See: https://www.astronomer.io/docs/learn/airflow-datasets/#retrieving-as-
set-information-in-a-downstream-task
        metadata_upstream = context["triggering_asset_events"][Asset("seq_2_start")][
            0
        ].extra

        latitude = metadata_upstream["latitude"]
        longitude = metadata_upstream["longitude"]
        dag_run_timestamp = metadata_upstream["ts"]

    url = os.getenv("WEATHER_API_URL")

    url = url.format(latitude=latitude, longitude=longitude)

    response = requests.get(url).json()

    response_bytes = json.dumps(response).encode("utf-8")

    # Save the data to S3
    hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
    hook.load_bytes(
        bytes_data=response_bytes,
        key=f"{_INTERMEDIARY_STORAGE_KEY}/{dag_run_timestamp}.json",
        bucket_name=_S3_BUCKET,
        replace=True,
    )

    yield Metadata(Asset("seq_2_extract"), {"ts": dag_run_timestamp})


@asset(
    schedule=[seq_2_create_in_table, seq_2_create_model_table, seq_2_extract],
    tags=["assets", "seq_2", "XCom"],
)
def seq_2_load(context):
    """
    Load the data from S3 to Postgres
    """

    # if the DAG is run manually use the time stamp of the manual run
    if str(context["dag_run"].run_type) == "DagRunType.MANUAL":
        dag_run_timestamp = context["ts"]
    # if the DAG is run as part of the asset sequence use the timestamp of the upstream 
asset event
    else:
        # You can retrieve metadata from the triggering asset event.
        # See: https://www.astronomer.io/docs/learn/airflow-datasets/#retrieving-as-
set-information-in-a-downstream-task
        metadata_upstream = context["triggering_asset_events"][Asset("seq_2_extract")][
            0
        ].extra

        dag_run_timestamp = metadata_upstream["ts"]

    s3_hook = S3Hook(aws_conn_id=_AWS_CONN_ID)
    response = s3_hook.read_key(
        key=f"{_INTERMEDIARY_STORAGE_KEY}/{dag_run_timestamp}.json",
        bucket_name=_S3_BUCKET,
    )
    api_response = json.loads(response)

    postgres_hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

    insert_sql = f"""
    INSERT INTO {_POSTGRES_SCHEMA}.{_POSTGRES_IN_TABLE} (raw_data)
    VALUES (%s::jsonb);
    """

    postgres_hook.run(insert_sql, parameters=(json.dumps(api_response),))


@asset(
    schedule=[seq_2_load],
    tags=["assets", "seq_2", "XCom"],
)
def seq_2_transform(context):
    from jinja2 import Template

    hook = PostgresHook(postgres_conn_id=_POSTGRES_CONN_ID)

    with open(f"{_SQL_DIR}/transform.sql", "r") as f:
        sql_template = f.read()

    template = Template(sql_template)
    sql = template.render(
        params={
            "schema": _POSTGRES_SCHEMA,
            "in_table": _POSTGRES_IN_TABLE,
            "out_table": _POSTGRES_TRANSFORMED_TABLE,
        }
    )

    hook.run(sql)
```