ChatGPT

Airflow Incremental Examples by Day

This companion document to the seven-day Airflow plan provides concrete examples that build on each other day by day. The goal is to illustrate how the concepts in the plan translate into code and practical use. Each section corresponds to the day with the same number in the learning plan. You can run the snippets locally in a Python environment with Airflow installed and connected to a database (e.g., SQLite for testing).

Day 0 Example – Cron Script vs. Airflow DAG

Before Airflow, it was common to orchestrate data pipelines with ad-hoc scripts scheduled via cron. Imagine a nightly process that extracts user data, transforms it and sends a report:

```
# cron job entry (runs at midnight every day)
0 0 * * * python /usr/local/bin/process_users.py >>/var/log/process_users.log
2>&1
```

The `process_users.py` script might look like this:

```
# process_users.py
import csv
import requests

def extract():
    # download user data from an API
    response = requests.get("https://api.example.com/users")
    return response.json()

def transform(users):
    # filter and shape the data
    return [user for user in users if user.get("active")]

def load(users):
    # write a CSV report
    with open("/tmp/active_users.csv", "w", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=users[0].keys())
        writer.writeheader()
        writer.writerows(users)

if __name__ == "__main__":
    users = extract()
    active_users = transform(users)
    load(active_users)
```

This approach works but lacks visibility and fails silently if anything goes wrong. To convert this pipeline to Airflow, you define a **DAG** with three tasks:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract():
    import requests
    return requests.get("https://api.example.com/users").json()

def transform(ti):
    users = ti.xcom_pull(task_ids="extract")
    return [user for user in users if user.get("active")]

def load(ti):
    import csv
    users = ti.xcom_pull(task_ids="transform")
    with open("/tmp/active_users.csv", "w", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=users[0].keys())
        writer.writeheader()
        writer.writerows(users)

with DAG(
    dag_id="user_etl",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
    catchup=False,
) as dag:
    extract_task = PythonOperator(
        task_id="extract",
        python_callable=extract,
    )
    transform_task = PythonOperator(
        task_id="transform",
        python_callable=transform,
    )
    load_task = PythonOperator(
        task_id="load",
        python_callable=load,
    )
    extract_task >> transform_task >> load_task
```

Airflow will schedule, run and monitor each task individually. If a task fails, you can configure retries, email alerts and view detailed logs in the web UI.

Day 1 Example – Set Up Airflow and Your First DAG

On the first day you install Airflow and build a simple DAG. Install Airflow using pip (pinning a version is recommended):

```
python -m venv venv
source venv/bin/activate
pip install "apache-airflow==2.7.3" --constraint "https://
raw.githubusercontent.com/apache/airflow/constraints-2.7.3/constraints-3.9.txt"

airflow db init
airflow users create \
    --username admin \
    --password admin \
    --firstname Admin \
    --lastname User \
    --role Admin \
    --email admin@example.com

# start the webserver and scheduler in separate terminals
airflow webserver --port 8080
airflow scheduler
```

Once Airflow is running, create a minimal DAG file in `~/airflow/dags/hello_airflow.py`:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

with DAG(
    dag_id="hello_airflow",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
    catchup=False,
) as dag:
    hello = BashOperator(
        task_id="say_hello",
        bash_command="echo 'Hello, Airflow!'"
    )
```

Check the Airflow UI at `http://localhost:8080` to view the DAG, trigger it manually and inspect the task logs. You've now created your first Airflow workflow.

Day 2 Example – DAGs, Tasks and the TaskFlow API

Airflow 2.x introduced the **TaskFlow API**, which allows you to define tasks as Python functions with decorators. Dependencies are inferred from the order in which you call the functions. Here's a DAG that uses TaskFlow to calculate the squares of numbers:

```
from airflow.decorators import dag, task
from datetime import datetime

@dag(schedule_interval="@daily", start_date=datetime(2025, 1, 1), catchup=False)
def square_numbers():
    @task
    def generate_numbers():
        return [1, 2, 3, 4, 5]

    @task
    def square(n: int):
        return n * n

    @task
    def collect(results):
        print(f"Squared numbers: {results}")

    numbers = generate_numbers()
    # dynamic task mapping: create one task per item
    squares = square.expand(n=numbers)
    collect(squares)

square_numbers_dag = square_numbers()
```

In this example, `generate_numbers` produces a list. `square.expand()` dynamically maps the `square` task over each element, creating one task instance per number. The `collect` task then receives a list of results via XCom. This pattern scales gracefully to hundreds of tasks.

You can test individual tasks locally with the CLI:

```
airflow tasks test square_numbers generate_numbers 2025-01-01
airflow tasks test square_numbers square 2025-01-01
```

Day 3 Example – Operators, Hooks and Sensors

Airflow includes a rich library of operators for interacting with systems. Here are some common ones:

BashOperator and PythonOperator

```python
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from datetime import datetime

def print_context(**context):
    print(context)

with DAG(
    dag_id="operators_example",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    t1 = BashOperator(
        task_id="list_files",
        bash_command="ls -l /tmp"
    )
    t2 = PythonOperator(
        task_id="show_context",
        python_callable=print_context
    )
    t1 >> t2
```

Sensors

Sensors wait for a condition to be met before proceeding. For example, a `FileSensor` waits for a file to appear:

```python
from airflow.sensors.filesystem import FileSensor

file_sensor = FileSensor(
    task_id="wait_for_file",
    filepath="/tmp/input/data.csv",
    poke_interval=30,  # check every 30 seconds
    timeout=60 * 60,    # give up after 1 hour
)
```

Hooks and Operators for Databases

Hooks provide reusable interfaces to external services. An operator often wraps a hook to do something specific. For example, to query Postgres:

```
from airflow.providers.postgres.operators.postgres import PostgresOperator

create_table = PostgresOperator(
    task_id="create_customers_table",
    postgres_conn_id="postgres_default",
    sql="""
    CREATE TABLE IF NOT EXISTS customers (
        id SERIAL PRIMARY KEY,
        name TEXT NOT NULL,
        email TEXT
    );
    """
)
```

You can also use hooks directly in a Python task:

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

@task
def get_customer_count():
    hook = PostgresHook(postgres_conn_id="postgres_default")
    records = hook.get_records("SELECT COUNT(*) FROM customers")
    print(f"There are {records[0][0]} customers.")
```

~~~
---

## Day 4 Example – Branching, XComs and Dynamic Workflows

Airflow provides the **BranchPythonOperator** to choose between multiple
downstream paths based on runtime logic.  You can also push and pull data
between tasks using **XComs**.  Here’s an example that checks an API and
branches depending on the result:

```python
from airflow import DAG
from airflow.operators.python import PythonOperator, BranchPythonOperator
from airflow.operators.dummy import DummyOperator
from datetime import datetime
~~~

```
def decide_path():
    import random
    choice = random.choice(["path_a", "path_b"])
    print(f"Branching to {choice}")
    return choice

with DAG(
    dag_id="branching_example",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    start = DummyOperator(task_id="start")
    branch = BranchPythonOperator(
        task_id="branch",
        python_callable=decide_path,
    )
    path_a = DummyOperator(task_id="path_a")
    path_b = DummyOperator(task_id="path_b")
    join = DummyOperator(task_id="join",
trigger_rule="none_failed_min_one_success")
    end = DummyOperator(task_id="end")

    start >> branch >> [path_a, path_b] >> join >> end
```

To demonstrate **XCom**, consider a task that produces a value and another that consumes it:

```
@task
def push_value():
    return "hello xcom"

@task
def pull_and_print(xcom_value):
    print(f"Received: {xcom_value}")

with DAG(
    dag_id="xcom_example",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    message = push_value()
    pull_and_print(message)
```

Airflow automatically passes the return value of `push_value` to `pull_and_print` through XCom. When combined with dynamic task mapping (as seen on Day 2), XCom enables flexible data-driven workflows.

Day 5 Example – Building an ETL Pipeline

Let's build a simple ETL pipeline that extracts data from Postgres, transforms it using pandas and loads it to Amazon S3. Note: you need a valid AWS connection configured in Airflow for the S3 upload to work.

```
from airflow import DAG
from airflow.decorators import task
from airflow.providers.postgres.hooks.postgres import PostgresHook
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
import pandas as pd
from datetime import datetime

@dag(schedule_interval=None, start_date=datetime(2025, 1, 1), catchup=False)
def etl_to_s3():
    @task
    def extract_orders():
        hook = PostgresHook(postgres_conn_id="postgres_default")
        df = hook.get_pandas_df("SELECT * FROM orders WHERE order_date >=
CURRENT_DATE - INTERVAL '1 day'")
        return df.to_json(orient="split")  # serialize DataFrame for XCom

    @task
    def transform_orders(raw_json):
        df = pd.read_json(raw_json, orient="split")
        # simple transformation: calculate total with tax
        df["total_with_tax"] = df["amount"] * 1.1
        return df.to_json(orient="split")

    @task
    def load_to_s3(transformed_json):
        df = pd.read_json(transformed_json, orient="split")
        hook = S3Hook(aws_conn_id="aws_default")
        # convert to CSV
        csv_data = df.to_csv(index=False)
        hook.load_string(
            string_data=csv_data,
            key=f"orders/{datetime.utcnow().strftime('%Y-%m-%d_%H%M%S')}.csv",
            bucket_name="my-data-bucket",
            replace=False,
        )

    raw = extract_orders()
    transformed = transform_orders(raw)
    load_to_s3(transformed)

etl_to_s3_dag = etl_to_s3()
```

This DAG uses hooks to connect to Postgres and S3, tasks decorated with `@task` to handle each stage, and pandas for transformation. The intermediate DataFrames are serialized to JSON for XCom transport.

Day 6 Example – Monitoring, Alerts and CLI Operations

Airflow provides rich monitoring and retry functionality. You can set default arguments for a DAG to handle failures and send notifications:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta
from airflow.utils.email import send_email

def failure_callback(context):
    ti = context.get('task_instance')
    send_email(
        to=['alerts@example.com'],
        subject=f"Task {ti.task_id} failed",
        html_content=f"<p>The task {ti.task_id} failed on {{ context['ds'] }}.</
p>"
    )

default_args = {
    'retries': 2,
    'retry_delay': timedelta(minutes=5),
    'on_failure_callback': failure_callback,
}

with DAG(
    dag_id="monitoring_example",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@hourly",
    default_args=default_args,
    catchup=False,
) as dag:
    def flaky_task():
        import random
        if random.random() < 0.5:
            raise ValueError("Random failure!")
        print("Success")
    t = PythonOperator(
        task_id="flaky",
        python_callable=flaky_task,
        email=['alerts@example.com'],
        email_on_failure=True,
    )
```

In this example, the task will retry twice if it fails, waiting five minutes between attempts. If all retries fail, Airflow sends an alert email via the callback. The `email_on_failure` and `email` arguments provide built-in notifications.

You can monitor DAG runs in the Airflow UI or query the CLI. For example, to list all DAGs and trigger a run:

```
airflow dags list
airflow dags trigger monitoring_example
```

To backfill a DAG over a date range:

```
airflow dags backfill monitoring_example --start-date 2025-01-01 --end-date
2025-01-05
```

Day 7 Example – Scaling and Custom Operators

As your workloads grow, you'll want to scale Airflow beyond the default sequential executor. Airflow supports multiple executors, including Local, Celery, Kubernetes and more. For example, to use the Celery executor, set `executor = CeleryExecutor` in your `airflow.cfg` and configure a message broker like RabbitMQ or Redis. Tasks will then be dispatched to worker processes rather than running on the scheduler.

You can also extend Airflow by writing custom operators. Suppose you need an operator that prints a greeting to a user-defined endpoint via an API. You can create a subclass of `BaseOperator`:

```
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults
import requests

class GreetEndpointOperator(BaseOperator):
    @apply_defaults
    def __init__(self, name: str, endpoint: str, **kwargs):
        super().__init__(**kwargs)
        self.name = name
        self.endpoint = endpoint
    def execute(self, context):
        response = requests.post(self.endpoint, json={"message": f"Hello
{self.name}!"})
        response.raise_for_status()
        self.log.info("Greeting sent successfully")

# using the custom operator in a DAG
from airflow import DAG
from datetime import datetime

with DAG(
    dag_id="custom_operator_example",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    greet = GreetEndpointOperator(
        task_id="greet_api",
        name="Airflow",
        endpoint="https://api.example.com/greet",
    )
```

This custom operator encapsulates the logic to call an external API. By wrapping it in an Airflow operator, you gain scheduling, retries and logging for free. You can package custom operators in Python modules and reuse them across projects.

Finally, consider using Airflow **variables**, **connections** and **secrets backends** to manage configuration and credentials centrally. This practice improves security and portability when deploying pipelines to different environments.

By following these examples alongside the seven-day plan, you can see how Airflow DAGs, tasks, operators, hooks, sensors, branching, XComs, ETL pipelines, monitoring, scaling and custom extensions come together to build robust and maintainable data workflows.
