Lecture 3: Apache Airflow DAG Evolution

This directory contains a series of Apache Airflow DAG examples that demonstrate the progressive evolution of a data pipeline. Each example builds upon the previous one, introducing new concepts and best practices.

Overview

All examples follow a similar pattern:

1. **Fetch Events**: Download event data from an API endpoint
2. **Calculate Stats**: Process the events and generate statistics grouped by date and user
3. **Send Stats** (later examples): Email the statistics to a recipient

Step-by-Step Explanation

01_unscheduled.py - Basic Unscheduled DAG

Purpose: Introduces the most basic DAG structure without any scheduling.

Key Features:

- Creates a DAG with `schedule_interval=None` (manual trigger only)
- Uses `BashOperator` to fetch events from an API
- Uses `PythonOperator` to calculate statistics
- Fixed file paths (`/data/events.json`, `/data/stats.csv`)

Concepts Introduced:

- Basic DAG definition
- Task dependencies using `>>` operator
- Bash and Python operators

02_daily_schedule.py - Daily Scheduled DAG

Purpose: Adds scheduling to run the DAG automatically.

Key Changes:

- `schedule_interval="@daily"` - Runs once per day
- Added `end_date` to limit execution window (Jan 1-5, 2019)
- DAG will automatically create DAG runs for each day in the date range

Concepts Introduced:

- Scheduled intervals (`@daily`, `@hourly`, `@weekly`, etc.)
- Start and end dates for DAG execution windows

03_with_end_date.py - Using datetime Module

Purpose: Demonstrates using `datetime` module with aliasing.

Key Changes:

- Uses `import datetime as dt` instead of `from datetime import datetime`
- Functionally identical to example 02, but shows different import style

Concepts Introduced:

- Alternative datetime import patterns
- Code style variations

04_time_delta.py - Custom Schedule Interval

Purpose: Shows how to use custom time intervals instead of preset schedules.

Key Changes:

- `schedule_interval=dt.timedelta(days=3)` - Runs every 3 days instead of daily
- Demonstrates flexibility in scheduling beyond preset intervals

Concepts Introduced:

- Custom schedule intervals using `timedelta`
- Flexible scheduling options

05_query_with_dates.py - Hardcoded Date Parameters

Purpose: Introduces date-based API querying with hardcoded dates.

Key Changes:

- API call includes `start_date` and `end_date` query parameters
- Dates are hardcoded: `start_date=2019-01-01&end_date=2019-01-02`
- Problem: Only works for one specific date range

Concepts Introduced:

- Parameterized API calls
- Date-based data extraction
- **Problem**: Hardcoded dates don't work for scheduled runs

06_templated_query.py - Dynamic Date Templating

Purpose: Uses Airflow templating to dynamically insert execution dates into API queries.

Key Changes:

- Uses Jinja2 templating: `{{execution_date.strftime('%Y-%m-%d')}}`

- Uses `{{next_execution_date.strftime('%Y-%m-%d')}}` for end date
- Each DAG run automatically uses its own execution date

Concepts Introduced:

- Airflow templating with Jinja2
- `execution_date` and `next_execution_date` variables
- Dynamic date handling in scheduled DAGs

Note: `execution_date` is the start of the data interval, `next_execution_date` is the end.

07_templated_query_ds.py - Simplified Date Strings

Purpose: Simplifies date templating using Airflow's built-in date string macros.

Key Changes:

- Uses `{{ds}}` instead of `{{execution_date.strftime('%Y-%m-%d')}}`
- Uses `{{next_ds}}` instead of `{{next_execution_date.strftime('%Y-%m-%d')}}`
- Schedule changed back to `timedelta(days=3)` to show it works with any interval
- Much cleaner and more readable

Concepts Introduced:

- Airflow date string macros (`ds`, `next_ds`)
- Simplified templating syntax
- `ds` = execution date as YYYY-MM-DD string
- `next_ds` = next execution date as YYYY-MM-DD string

08_templated_path.py - Templated File Paths

Purpose: Extends templating to file paths, enabling date-partitioned data storage.

Key Changes:

- Output file path includes date: `/data/events/{{ds}}.json`
- Python function uses `templates_dict` to access templated values
- Function signature changed to `_calculate_stats(**context)`
- Output stats also date-partitioned: `/data/stats/{{ds}}.csv`
- Each DAG run processes and stores data for its specific date

Concepts Introduced:

- Templating in file paths
- `templates_dict` for passing templated values to Python functions
- Context object (`**context`) to access template variables
- Date-partitioned data storage (best practice for data pipelines)

Benefits:

- Prevents overwriting data from different runs
- Enables reprocessing of specific dates
- Better data organization

09_no_catchup.py - Disabling Backfill

Purpose: Demonstrates how to prevent Airflow from backfilling missed DAG runs.

Key Changes:

- Added `catchup=False` to DAG definition
- When DAG is first enabled, it only runs for the current/latest interval
- Previous intervals are skipped

Concepts Introduced:

- `catchup` parameter
- Backfill behavior control
- When to use: Prevents unnecessary processing of old data when DAG is first deployed

Use Cases:

- Real-time pipelines where old data isn't needed
- Cost optimization (avoid processing historical data)
- Testing new DAGs without triggering many historical runs

10_non_atomic_send.py - Non-Atomic Email Operation

Purpose: Demonstrates a problematic pattern where email sending is part of the stats calculation task.

Key Changes:

- Added `_email_stats()` function
- Email sending happens inside `_calculate_stats()` function
- `catchup=True` (explicitly set, though it's the default)

Problems with This Approach:

- **Non-atomic**: If email fails, the entire task fails, even though stats were calculated
- **No retry granularity**: Can't retry just the email without recalculating stats
- **Mixed concerns**: Data processing and notification are coupled
- **Inefficient**: Stats calculation must succeed before email can be sent

Concepts Introduced:

- Task atomicity
- Separation of concerns
- What NOT to do (anti-pattern)

11_atomic_send.py - Atomic Email Operation

Purpose: Demonstrates the correct pattern with separate tasks for calculation and notification.

Key Changes:

- Split email sending into a separate task: `send_stats`
- `calculate_stats` only calculates and saves stats
- `send_stats` reads the saved CSV and sends email
- Task dependency: `fetch_events >> calculate_stats >> send_stats`

Benefits of This Approach:

- **Atomic operations**: Each task has a single responsibility
- **Independent retries**: Can retry email without recalculating stats
- **Better monitoring**: Can see exactly which step failed
- **Efficiency**: Stats are saved once, email can be retried multiple times
- **Separation of concerns**: Data processing and notification are decoupled

Concepts Introduced:

- Task separation and atomicity
- Proper task dependencies
- Best practices for error handling and retries
- Reading data from previous tasks

Key Concepts Summary

Scheduling

- `schedule_interval=None`: Manual trigger only
- `schedule_interval="@daily"`: Preset intervals
- `schedule_interval=timedelta(days=N)`: Custom intervals

Templating

- `{{ds}}`: Execution date as YYYY-MM-DD string
- `{{next_ds}}`: Next execution date as YYYY-MM-DD string
- `{{execution_date}}`: Full datetime object
- `{{next_execution_date}}`: Next execution datetime object
- `templates_dict`: Pass templated values to Python functions

Catchup Behavior

- `catchup=True` (default): Backfills all missed intervals
- `catchup=False`: Only runs for current/latest interval

Task Design Principles

- **Atomicity**: Each task should do one thing

- **Separation of concerns**: Separate data processing from notifications
- **Idempotency**: Tasks should be safe to rerun
- **Date partitioning**: Store data with date in path for better organization

Common Patterns

Date-Partitioned Data Storage

```
# Good: Date in path
"/data/events/{{ds}}.json"
"/data/stats/{{ds}}.csv"

# Bad: Fixed path (overwrites previous runs)
"/data/events.json"
"/data/stats.csv"
```

Task Dependencies

```
# Sequential
task1 >> task2 >> task3

# Parallel then sequential
[task1, task2] >> task3
```

Accessing Templated Values in Python

```
def my_function(**context):
    value = context["templates_dict"]["my_key"]
    # or
    ds = context["ds"]  # execution date string
```

Progression Summary

1. **01-04**: Basic scheduling and interval configuration
2. **05-07**: Dynamic date handling in API queries
3. **08**: Date-partitioned file storage
4. **09**: Catchup control
5. **10-11**: Task design patterns (anti-pattern vs. best practice)

Each example builds upon previous concepts, creating a comprehensive learning path for Apache Airflow DAG development.

12-14: Binance Price Tracking Exercise

Overview

This exercise demonstrates a real-world data pipeline that:

1. **Fetches** Bitcoin price data from Binance API every minute
2. **Aggregates** minute-level data into hourly averages
3. **Aggregates** hourly data into daily averages
4. **Saves** all data to CSV files for analysis

Files

- `12_binance_fetch_minute.py`: Fetches price every minute from Binance API
- `13_binance_calculate_hourly.py`: Calculates hourly averages (runs every hour)
- `14_binance_calculate_daily.py`: Calculates daily averages (runs every 24 hours)

API Endpoint

The DAGs fetch data from:

```
https://api.binance.com/api/v3/avgPrice?symbol=BTCUSDT
```

Response Format:

```
{
    "mins": 5,
    "price": "68285.81006621",
    "closeTime": 1771317380403
}
```

Data Flow

```
Minute Fetch (every 1 min)
    ↓
/data/binance/raw/{date}/daily_raw.csv
    ↓
Hourly Aggregation (every 1 hour)
    ↓
/data/binance/hourly/{date}/hourly_avg.csv
    ↓
Daily Aggregation (every 24 hours)
    ↓
/data/binance/daily/daily_avg.csv
```

Key Features

1. **Multi-Level Aggregation**: Demonstrates hierarchical data processing

2. **Date Partitioning**: Data organized by date for efficient storage
3. **Error Handling**: Retry logic for API failures
4. **Real-Time Processing**: Continuous data collection and aggregation
5. **CSV Output**: All data saved in CSV format for easy analysis

Running the Exercise

1. **Place DAGs in Airflow DAGs folder**

2. **Enable all three DAGs** in Airflow UI:

   - `binance_fetch_minute` (runs every minute)
   - `binance_calculate_hourly` (runs every hour)
   - `binance_calculate_daily` (runs daily)

3. **Let it run for at least 24 hours** to collect meaningful data

4. **Check output files**:

   - Raw minute data: `/data/binance/raw/{date}/daily_raw.csv`
   - Hourly averages: `/data/binance/hourly/{date}/hourly_avg.csv`
   - Daily averages: `/data/binance/daily/daily_avg.csv`

Output CSV Structure

Minute Data (`daily_raw.csv`):

```
mins,price,closeTime,timestamp,fetch_time,price_float
5,68285.81006621,1771317380403,2024-01-15T10:30:00,2024-01-15
10:30:00,68285.81
```

Hourly Averages (`hourly_avg.csv`):

```
date,hour,avg_price,min_price,max_price,first_price,last_price,data_points
,calculated_at
2024-01-15,10,68250.25,68100.00,68400.00,68200.00,68300.00,60,2024-01-15
10:59:59
```

Daily Averages (`daily_avg.csv`):

```
date,avg_price,min_price,max_price,opening_price,closing_price,price_chang
e,price_change_pct,total_data_points,hours_with_data,calculated_at
2024-01-
15,68250.25,67500.00,69000.00,68000.00,68500.00,500.00,0.74,1440,24,2024-
01-16 00:00:00
```

Concepts Demonstrated

- **Scheduled Data Collection**: Continuous API polling
- **Data Aggregation**: Multi-level time-based aggregation
- **Date Partitioning**: Organized data storage by date
- **Error Handling**: Retry mechanisms for network failures
- **Data Persistence**: CSV file storage for historical analysis
- **Pipeline Orchestration**: Multiple DAGs working together

Why Airflow vs. Regular Scripts

With Airflow:

- ✅ Automatic scheduling (runs every minute/hour/day without manual intervention)
- ✅ Built-in retry logic if API fails
- ✅ Web UI monitoring of all runs
- ✅ Historical execution tracking
- ✅ Can run continuously for days/weeks
- ✅ Automatic data partitioning by date

Without Airflow (Regular Script):

- ❌ Must manually run script every minute (or use cron, which is harder to monitor)
- ❌ No built-in retry mechanism
- ❌ No visibility into execution history
- ❌ Hard to track which data has been collected
- ❌ Manual date partitioning logic required

Dependencies

Make sure to install required packages:

```
pip install requests pandas
```

Or use the `requirements.txt` file:

```
pip install -r requirements.txt
```

Events API

The DAGs in this lecture fetch events from a Flask API running at `http://events_api:5000/events`.

Running the Events API

To run the events API locally:

1. Install dependencies:

```
pip install -r requirements.txt
```

2. Start the Flask server:

```
python events_api.py
```

The API will be available at `http://localhost:5000/events`.

API Endpoints

GET `/events`

Retrieves events data. Supports optional query parameters for date filtering.

Query Parameters:

- `start_date` (optional): Start date in `YYYY-MM-DD` format
- `end_date` (optional): End date in `YYYY-MM-DD` format

Example Requests:

```
# Get all events (default: Jan 1-10, 2019)
curl http://localhost:5000/events

# Get events for a specific date range
curl "http://localhost:5000/events?start_date=2019-01-01&end_date=2019-01-02"
```

Response: Returns a JSON array of events, each containing:

- `date`: Event date in `YYYY-MM-DD` format
- `user`: User identifier
- `event_type`: Type of event (click, view, purchase, login)
- `value`: Numeric value associated with the event

Example Response:

```
[
  {
    "date": "2019-01-01",
    "user": "alice",
    "event_type": "click",
    "value": 42
  },
```

```
  {
    "date": "2019-01-01",
    "user": "bob",
    "event_type": "view",
    "value": 15
  }
]
```

GET `/health`

Health check endpoint that returns the API status.

Response:

```
{
  "status": "healthy"
}
```

Docker Usage

If you're running Airflow in Docker and need the events API accessible at `http://events_api:5000`, you can:

1. Add the events API service to your `docker-compose.yml`:

   ```
   events_api:
     build: ./docker/events-api  # if you have a Dockerfile
     # or
     image: python:3.8-slim
     command: python -m flask run --host=0.0.0.0 --port=5000
     volumes:
       - ./events_api.py:/app/events_api.py
     ports:
       - "5000:5000"
   ```

2. Or run it as a separate container and ensure it's on the same Docker network as your Airflow services.

Task-by-Task Explanation

This section explains each task in the DAGs, how they work, and why running them in Airflow is different from running them as regular Python scripts.

Task 1: `fetch_events` (BashOperator)

What It Does

The `fetch_events` task downloads event data from the events API and saves it to a local file.

How It Works

In Airflow (BashOperator):

```
fetch_events = BashOperator(
    task_id="fetch_events",
    bash_command=(
        "mkdir -p /data/events && "
        "curl -o /data/events/{{ds}}.json "
        "http://events_api:5000/events?"
        "start_date={{ds}}&"
        "end_date={{next_ds}}"
    ),
    dag=dag,
)
```

As a Regular Script:

```
# You would run this manually:
mkdir -p /data/events
curl -o /data/events/2019-01-01.json \
  "http://events_api:5000/events?start_date=2019-01-01&end_date=2019-01-
02"
```

Key Differences: Why Airflow is Better

1. **Automatic Scheduling**
    - **Airflow**: Runs automatically based on `schedule_interval` (e.g., daily at midnight)
    - **Regular Script**: You must manually run it or set up cron jobs, which are harder to monitor and manage
2. **Dynamic Date Handling**
    - **Airflow**: Uses templating (`{{ds}}`, `{{next_ds}}`) to automatically use the correct date for each DAG run
    - **Regular Script**: You must manually change dates or write complex date logic in your script
3. **Task Dependencies**
    - **Airflow**: Automatically ensures `calculate_stats` only runs after `fetch_events` completes successfully
    - **Regular Script**: You must manually check if the previous step succeeded before running the next
4. **Error Handling & Retries**

- **Airflow**: Built-in retry mechanism - if the API is temporarily down, Airflow can automatically retry the task
- **Regular Script**: If the curl command fails, you must manually detect and handle the error

5. **Monitoring & Visibility**

- **Airflow**: Web UI shows task status, logs, execution time, and history
- **Regular Script**: You must check logs manually or build your own monitoring

6. **Idempotency**

- **Airflow**: Each DAG run is tracked separately - you can see which dates have been processed
- **Regular Script**: Hard to track what has been processed without building your own tracking system

7. **Backfill Support**

- **Airflow**: Can automatically backfill missed dates when you enable a DAG
- **Regular Script**: You must manually run the script for each missed date

---

Task 2: `calculate_stats` (PythonOperator)

What It Does

The `calculate_stats` task reads the downloaded events JSON file, groups events by date and user, counts occurrences, and saves the statistics to a CSV file.

How It Works

In Airflow (PythonOperator):

```
def _calculate_stats(**context):
    """Calculates event statistics."""
    input_path = context["templates_dict"]["input_path"]
    output_path = context["templates_dict"]["output_path"]

    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()

    Path(output_path).parent.mkdir(exist_ok=True)
    stats.to_csv(output_path, index=False)

calculate_stats = PythonOperator(
    task_id="calculate_stats",
    python_callable=_calculate_stats,
    templates_dict={
        "input_path": "/data/events/{{ds}}.json",
        "output_path": "/data/stats/{{ds}}.csv",
    },
    dag=dag,
)
```

As a Regular Script:

```
# You would run this manually:
from pathlib import Path
import pandas as pd

input_path = "/data/events/2019-01-01.json"
output_path = "/data/stats/2019-01-01.csv"

events = pd.read_json(input_path)
stats = events.groupby(["date", "user"]).size().reset_index()

Path(output_path).parent.mkdir(exist_ok=True)
stats.to_csv(output_path, index=False)
```

Key Differences: Why Airflow is Better

1. **Context-Aware Execution**
   - **Airflow**: Receives execution context automatically (`**context`) with templated values, execution date, and DAG run information
   - **Regular Script**: You must manually pass dates and paths, or hardcode them
2. **Date-Partitioned Processing**
   - **Airflow**: Each DAG run processes data for its specific date automatically via templating
   - **Regular Script**: You must manually change file paths and dates for each run
3. **Task Isolation**
   - **Airflow**: Each task runs in isolation - if `calculate_stats` fails, `fetch_events` doesn't need to rerun (unless you configure it to)
   - **Regular Script**: If processing fails, you might need to re-download data unnecessarily
4. **Automatic Dependency Management**
   - **Airflow**: Knows it needs `fetch_events` to complete first - won't start until dependencies are met
   - **Regular Script**: You must manually ensure the input file exists before processing
5. **Retry Logic**
   - **Airflow**: Can retry just the calculation if it fails (e.g., due to memory issues), without re-fetching data
   - **Regular Script**: If processing fails, you might rerun the entire pipeline unnecessarily
6. **Logging & Debugging**
   - **Airflow**: Centralized logging in the web UI, easy to see what went wrong

- **Regular Script**: Logs scattered across console output or files you manage yourself

7. **Parallel Execution**
   - **Airflow**: Can process multiple dates in parallel (if configured) without conflicts
   - **Regular Script**: Running multiple instances risks file conflicts unless you carefully manage locking

---

Task 3: `send_stats` (PythonOperator) - Advanced Examples Only

What It Does

The `send_stats` task reads the calculated statistics CSV file and sends it via email (or simulates sending).

How It Works

In Airflow (PythonOperator):

```
def _send_stats(email, **context):
    stats = pd.read_csv(context["templates_dict"]["stats_path"])
    email_stats(stats, email=email)

send_stats = PythonOperator(
    task_id="send_stats",
    python_callable=_send_stats,
    op_kwargs={"email": "user@example.com"},
    templates_dict={"stats_path": "/data/stats/{{ds}}.csv"},
    dag=dag,
)
```

As a Regular Script:

```
# You would run this manually:
import pandas as pd

stats_path = "/data/stats/2019-01-01.csv"
email = "user@example.com"

stats = pd.read_csv(stats_path)
email_stats(stats, email=email)
```

Key Differences: Why Airflow is Better

1. **Atomic Task Separation** (Best Practice)
   - **Airflow**: `send_stats` is a separate task from `calculate_stats`
   - **Benefit**: If email fails, you can retry just the email without recalculating stats

- **Regular Script**: Often combined with calculation, so failures require full reprocessing

2. **Independent Retries**
   - **Airflow**: Can retry email sending multiple times without touching the stats calculation
   - **Regular Script**: Email retry might require recalculating stats (inefficient)
3. **Clear Failure Points**
   - **Airflow**: Web UI clearly shows if email failed vs. calculation failed
   - **Regular Script**: Harder to identify which part of a combined script failed
4. **Separation of Concerns**
   - **Airflow**: Data processing (`calculate_stats`) and notification (`send_stats`) are decoupled
   - **Regular Script**: Often mixed together, making code harder to maintain and test
5. **Conditional Execution**
   - **Airflow**: Can skip email sending based on conditions (e.g., only send if stats exceed threshold)
   - **Regular Script**: Requires complex conditional logic in a single script
6. **Notification Flexibility**
   - **Airflow**: Easy to add multiple notification tasks (email, Slack, PagerDuty) without modifying calculation logic
   - **Regular Script**: Adding new notifications requires modifying the main script

---

Summary: Airflow vs. Regular Scripts

What Airflow Provides That Regular Scripts Don't

Feature	Airflow	Regular Scripts
Scheduling	Built-in scheduler with cron-like syntax	Requires cron or external scheduler
Dependencies	Automatic task dependency management	Manual dependency checking
Retries	Built-in retry mechanism per task	Must implement yourself
Monitoring	Web UI with task status, logs, history	Manual log checking
Backfill	Automatic backfill of missed dates	Manual execution for each date
Templating	Dynamic date/path templating	Hardcoded or manual date logic
Parallelism	Built-in support for parallel task execution	Complex to implement safely
Idempotency	Each run tracked separately	Must build tracking yourself
Error Isolation	Tasks fail independently	Entire script fails together
Date Partitioning	Automatic via templating	Manual path management
Task Reusability	Tasks can be reused across DAGs	Code duplication

Feature	Airflow	Regular Scripts
Audit Trail	Complete execution history	Must build logging yourself

When to Use Airflow vs. Regular Scripts

Use Airflow when:

- You have scheduled, recurring data pipelines
- Tasks have dependencies on each other
- You need to process data for multiple dates/periods
- You need monitoring, alerting, and retry capabilities
- Multiple people need to understand and maintain the pipeline
- You need to track what has been processed

Use Regular Scripts when:

- One-time data processing tasks
- Simple, linear scripts with no dependencies
- No scheduling needed
- Personal/small-scale projects
- Quick prototypes or experiments

Example: The Same Pipeline in Both Approaches

Regular Script Approach:

```
# manual_pipeline.py
import subprocess
from datetime import datetime, timedelta
import pandas as pd
from pathlib import Path

# Hardcoded date - must change manually
date = "2019-01-01"
next_date = "2019-01-02"

# Step 1: Fetch events
subprocess.run([
    "curl", "-o", f"/data/events/{date}.json",
    f"http://events_api:5000/events?start_date={date}&end_date=
{next_date}"
])

# Step 2: Calculate stats
events = pd.read_json(f"/data/events/{date}.json")
stats = events.groupby(["date", "user"]).size().reset_index()
Path(f"/data/stats/{date}.csv").parent.mkdir(exist_ok=True)
stats.to_csv(f"/data/stats/{date}.csv", index=False)

# Step 3: Send email
```

```
stats = pd.read_csv(f"/data/stats/{date}.csv")
email_stats(stats, email="user@example.com")

# Problems:
# - Must manually change date for each run
# - No automatic retries
# - No dependency checking
# - No monitoring
# - Hard to backfill multiple dates
```

Airflow Approach:

```
# DAG automatically handles:
# - Scheduling (runs daily automatically)
# - Date templating ({{ds}} changes automatically)
# - Dependencies (calculate_stats waits for fetch_events)
# - Retries (automatic retry on failure)
# - Monitoring (visible in web UI)
# - Backfill (can process multiple dates automatically)
```

The Airflow approach provides **production-grade orchestration** with minimal code, while the regular script approach requires you to build all these features yourself.