From Repetition to Refactor: Smarter DAG Design in Airflow 3

Steven Woods
Director, Software Engineering
Zelis

3.0

Agenda: From Repetition to Refactor

Initial DAG Design
- Common anti-patterns: hardcoded logic, duplication, rigid sequencing

Refactor Strategy
- Applying D.R.Y. principles
- Task factories, parameterization, dynamic task mapping

Designing for Flexibility
- Modular DAGs for batch, streaming, and ad-hoc workflows

Pros & Cons of Refactor
- Benefits: scalability, maintainability, observability
- Trade-offs: complexity, learning curve

Q&A

Initial DAG Design

🐾 **Project:** Animal image processor
Description:
Develop an automated and scalable workflow that retrieves animal images from the internet and processes them for downstream use.

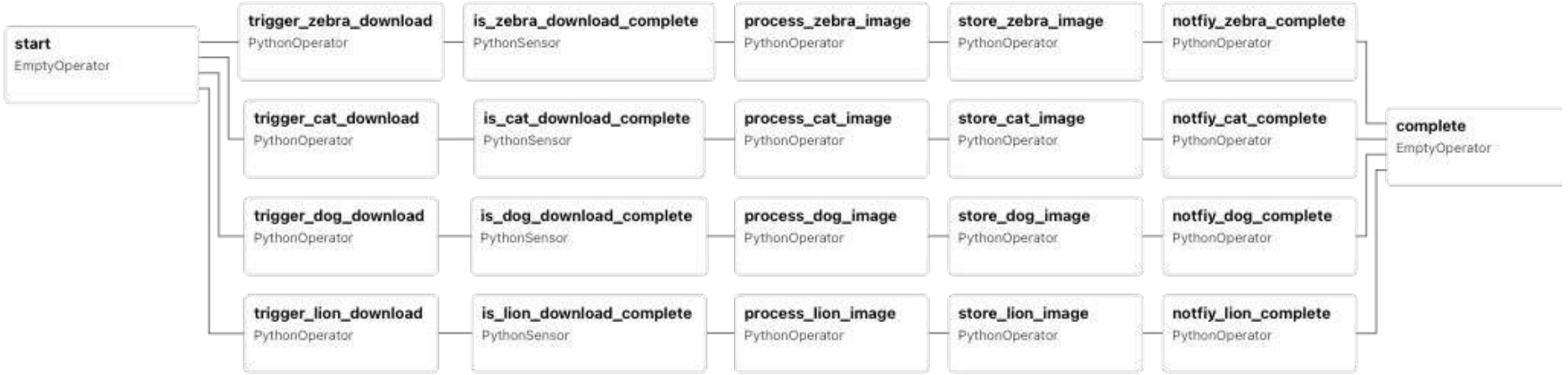
🎯 **Objective:**

- Fetch a new animal image daily from a designated internet source
- Target animals: cat, zebra, dog, lion
- Store the image in an object storage service (e.g., Amazon S3)
- Resize to standardized dimensions & convert image

Initial DAG Design

📋 Tasks

- start – Workflow entry point
- trigger_(*animal*)_download – Trigger Lambda/external process to fetch animal photos
- is_(*animal*)_download_complete – Sensor to check if download finished (file present in S3)
- process_(*animal*)_image – Resize and convert the image
- store_(*animal*)_image – Save image metadata to the database
- notify_(*animal*)_complete – Send completion notification email
- complete – Workflow end point



Initial DAG Design

I'll call this design **"repetitive task rows."** It has several scalability and management issues:

- More rows = more code (PRs, deployments)
- Hard to run concurrently or isolate failures
- Doesn't scale (4 animals → 10, 20, 100+)
- Graph becomes unreadable
- Code harder to maintain
- One failing task row can break the whole DAG

Initial DAG Design

Each new animal adds 5+ new tasks (download, check, process, store, notify)

Code grows linearly with every animal → high maintenance overhead

Hard to isolate and retry failures for a single animal

Increases DAG execution time and complexity

DAG visual quickly clutters with repetitive task rows

Initial DAG Design

- Failure in cat or dog tasks → entire DAG run fails
- With DAG concurrency = 1:
  Next DAG run blocked until current completes
  Persistent failures = major processing slowdown

Refactor Strategy

Recommended solution: Split into **2 DAGs**:

DAG 1: Trigger

- Runs on a schedule (e.g., @daily)
- Reads configuration from Airflow variables
- Passes parameters to the processing DAG
- Simple to enable/disable

DAG 2: Processor

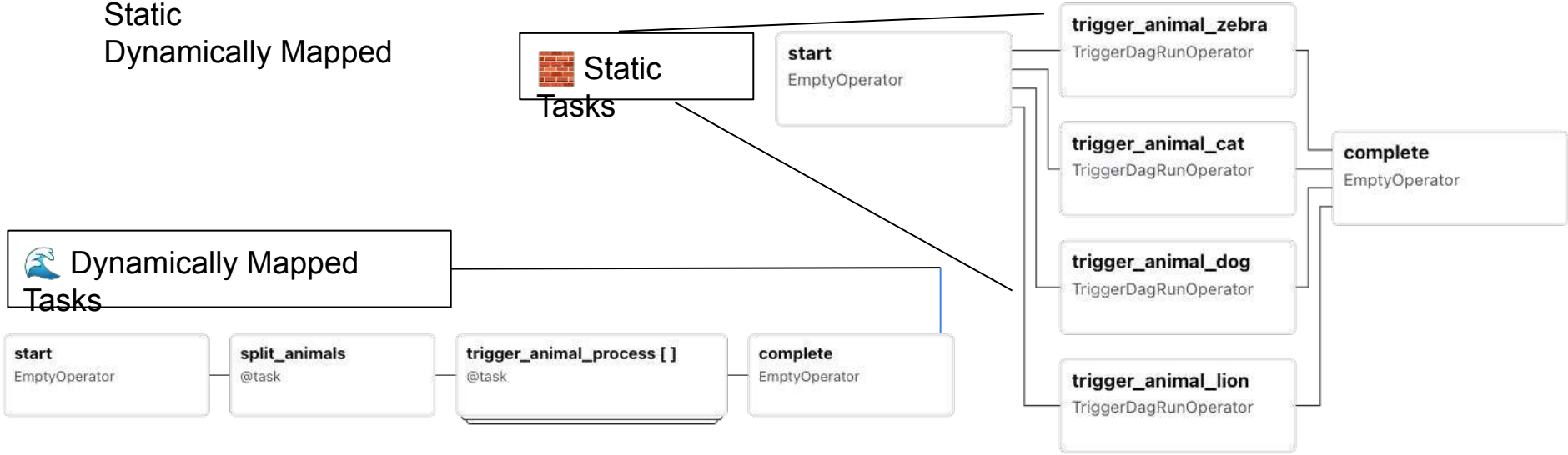
- Accepts parameters (e.g., { ”animal": "zebra" })
- Has no schedule – runs only when triggered
- Can be triggered by:
  - DAG 1 (Trigger DAG)
  - Manual runs with params
  - External processes
- Supports ad-hoc requests and flexible processing

```
Variable:
[
    {”animal": "zebra"},
    {”animal": "cat"},
    {”animal": "dog"},
    {”animal": "bear"}
]
```

Refactor Strategy

DAG 1: Trigger

- Two execution modes:
  Group processing → TriggerDagRunOperator(wait_for_completion=True)
  Individual processing → TriggerDagRunOperator(wait_for_completion=False)
- Supports running **multiple configurations** as needed
  Single
  Static
  Dynamically Mapped



Refactor Strategy

DAG 2: Processor

- Responsible for processing configs (accepts parameters, e.g., { "animal": "zebra" })
- No schedule – runs only when triggered
- Can be started by:
  - DAG 1 (Trigger DAG)
  - Manual runs with parameters
  - External processes
- Supports ad-hoc requests
- Easy to extend – add new configs via Airflow variables or code (no new tasks/operators required)
- Flexible & dynamic – avoids hardcoding (e.g., per-animal rules), focuses on runtime config processing

Designing for Flexibility

The design supports a diverse range of workflows:

📦 Batch

- All animal downloads run on a fixed nightly schedule
- Processes the full set of animals in one batch (e.g., zebra, lion, cat, dog)
- Best for use cases where freshness isn't critical and daily updates are sufficient

🔄 Streaming

- A data pipeline publishes animal IDs (e.g., {"animal": "zebra"}, {"animal": "cat"}) to Kafka/Kinesis. Each event tells your DAG which animal photo to download.

⚡ Ad-hoc

- A simple website allows users to select an animal (e.g., 🦓, 🐶, 🐱)
- When triggered, the site calls Airflow's API to start the download + processing DAG
- No fixed schedule — runs only when requested

Pros & Cons of Refactor

✅ Pros of Updated Design

Scalable – Add or remove animals by simply updating a variable (no new code required)

Optimized execution – Integrated use of DAG concurrency, task concurrency, and parallelism improves workload control and performance

Separation of responsibility – Each DAG focuses on its own role, making the codebase cleaner and easier to manage

No code changes needed – No PRs, reviews, or deployments for adding configs

Simplified troubleshooting – Issues with one animal don't block others; use Grid View to quickly spot and fix failures

Flexible execution – Run configurations manually, externally triggered, in bulk, or one at a time

⚠️ Cons of Updated Design

Split codebase – Logic is separated across two DAGs (Trigger & Processor)

Limited validation – New "animals" are added via variables without code review

Fragmented visibility – No single DAG view of the entire run; execution is split across DAGs

Questions?

linkedin.com/in/stevengwoods