Sriram Vamsi Ilapakurthy –Senior
Software Engineer

Exploring DAG Design Patterns

What We'll Cover Today

- Motivation
- Introduction to DAGs in Airflow
- Task best practices
- Organize tasks
- DAG flexibility
- Parallelism

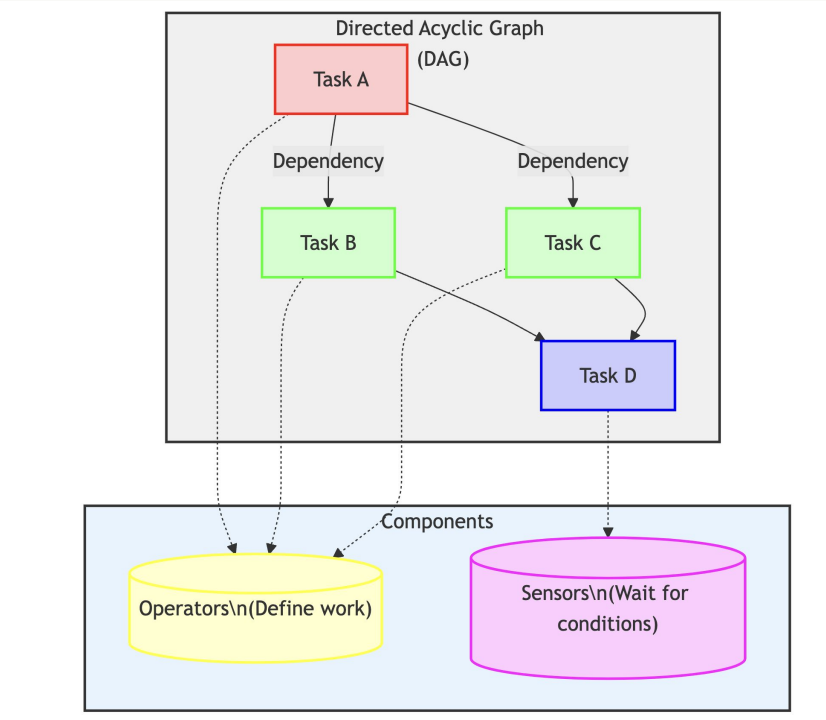
- My presentation, comments and opinions are provided in my personal capacity and not as a representative of Walmart. They do not reflect the views of Walmart and are not endorsed by Walmart

Motivation

- Maintainability
- Efficiency - Resource/cost
- Flexibility - Reuse

Introduction to DAGs in Airflow

- DAG
- Key components
  - Tasks
    - Operators
    - Sensors
  - Dependencies



Task Best Practices

Keep tasks small & focussed

```python
def process_and_load_data():
    # Extract data
    data = extract_data()
    # Clean data
    cleaned_data = clean_data(data)
    # Transform data
    transformed_data = transform_data(cleaned_data)
    # Load data
    load_data(transformed_data)

task = PythonOperator(
    task_id='process_and_load_data',
    python_callable=process_and_load_data,
    dag=dag,
)
```

Idempotency

- Tasks should produce the same results regardless of how many times they're run.

```python
def upsert_data(**kwargs):
    data = get_data_from_source()

    for record in data:

        db.upsert(record)  # Updates if exists, inserts if not
```

Atomicity

- They should complete entirely or not at all, treat it like a transaction.

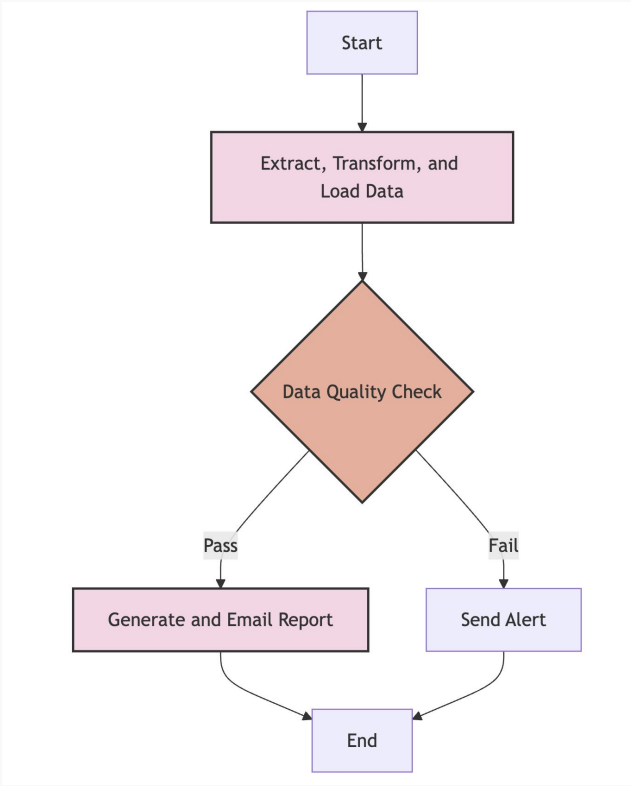
```
def atomic_task():
    try:
        print("Performing task operations...")
    except Exception as e:
        print(f"Error occurred: {e}")
        raise
```

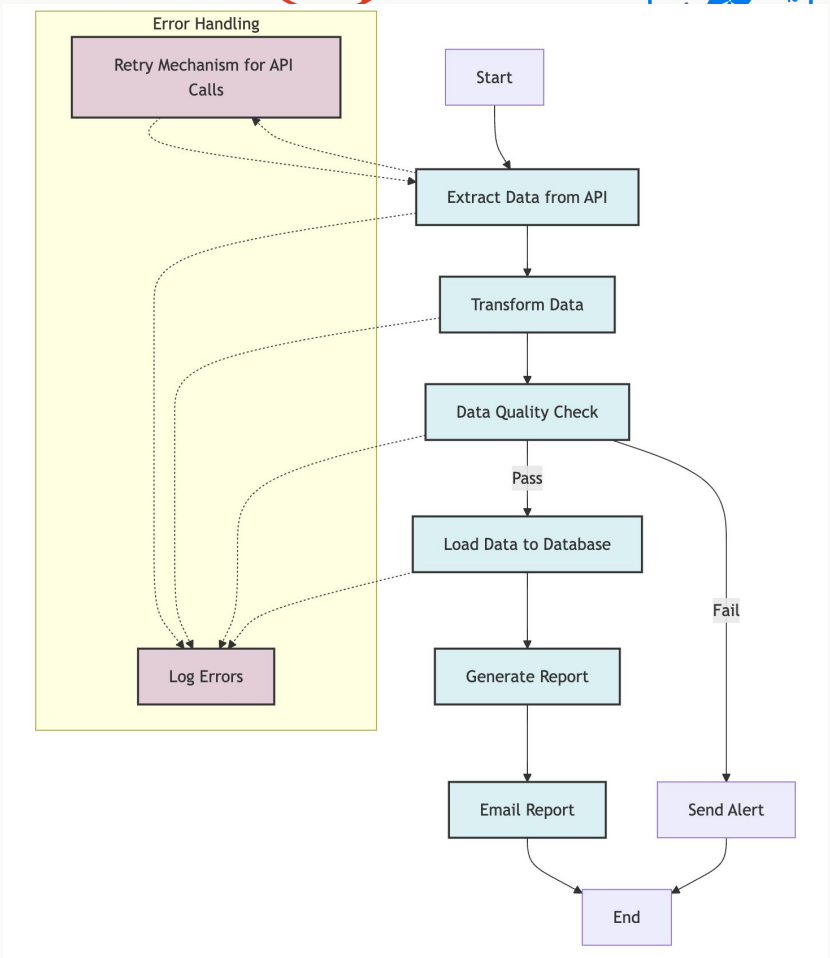
Retries and error handling

- Proper error handling is vital for robust DAGs.

```
def unreliable_task():
    import random
    if random.choice([True, False]):
        raise Exception("Random failure!")

retry_task = PythonOperator(
    task_id='retry_task',
    python_callable=unreliable_task,
    retries=3,
    retry_delay=timedelta(minutes=5),
)
```

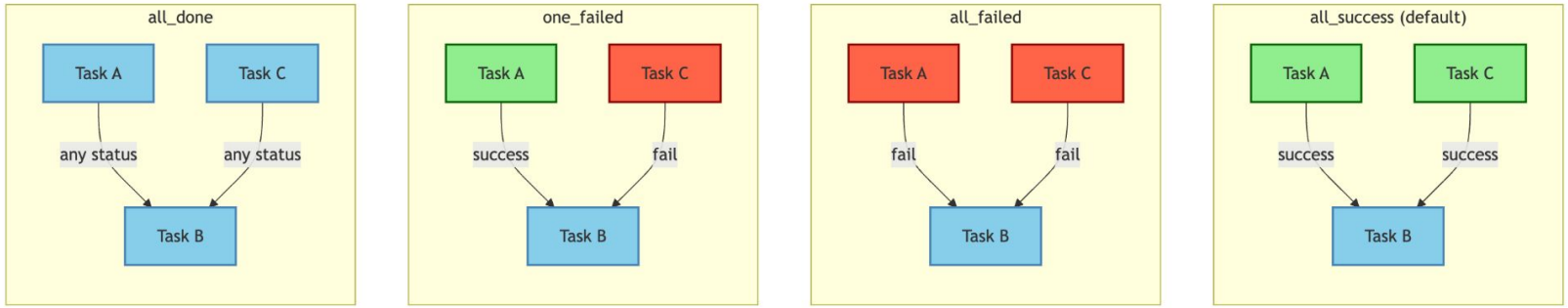
Bad vs Good Example





Organize tasks

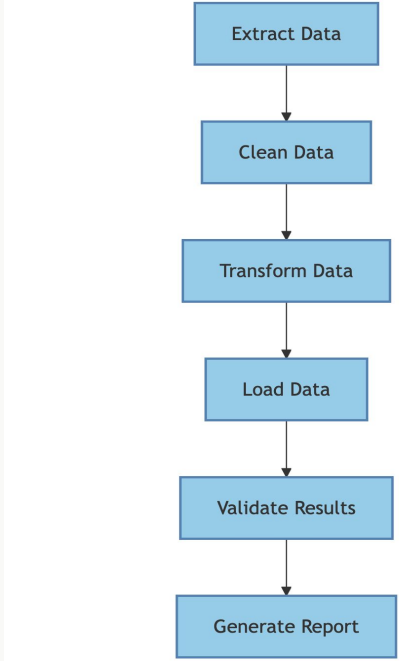
Trigger Rules



Linear Workflow Pattern

- Tasks are executed sequentially, with each task depending on the previous one.
- Pros:
  - Clear dependency chain
  - Easy to track progress and identify bottlenecks
- Cons:
  - Limited parallelism, potentially slower execution for complex workflows
  - If one task fails, the entire workflow stops

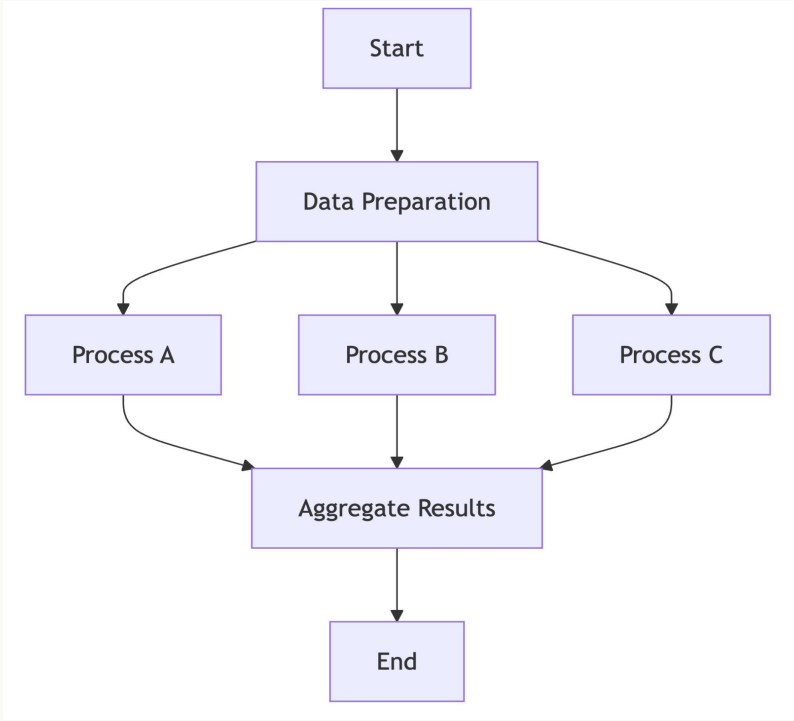
```
task_1 >> task_2 >> task_3
```



Fan-Out/Fan-In Pattern

- Tasks fan out for parallel processing and then converge for final processing.
- For processing multiple datasets or data partitions in parallel

```
task_1 >> [task_2, task_3, task_4]
```



Branching and Conditional Execution

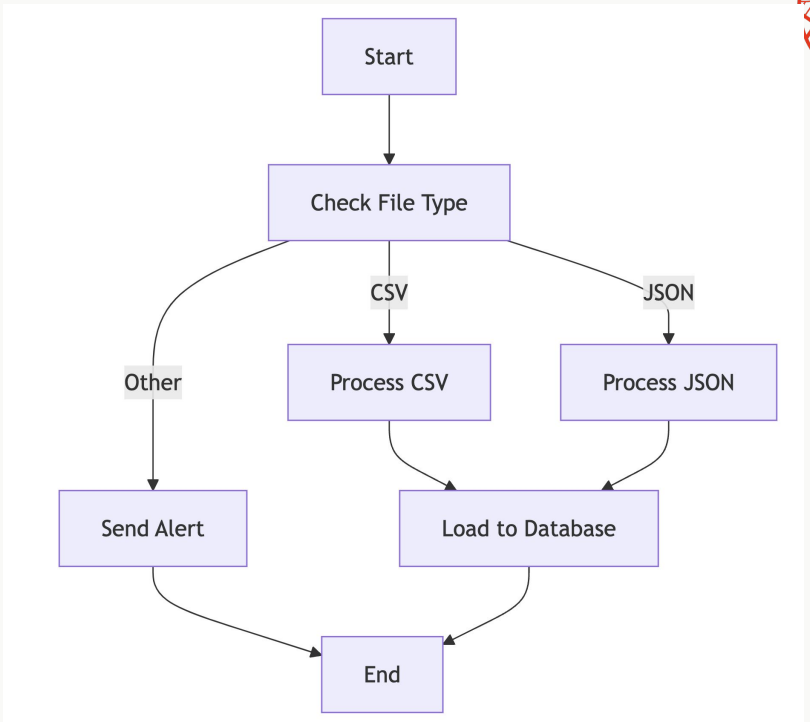
- Dynamically choose which tasks to execute based on runtime conditions.

- Pros:
  - Allows for dynamic and flexible workflows
  - Reduces the need for multiple similar DAGs
- Cons:
  - May require careful testing to ensure all branches work correctly

Branching and Conditional Execution

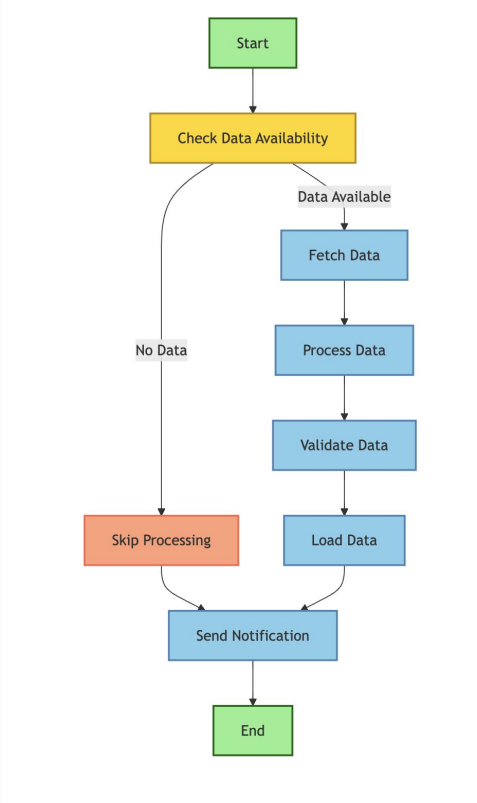
```
BranchPythonOperator(
      task_id='data_quality_check',
      python_callable=data_quality_check,
    )
```

- Trigger rules



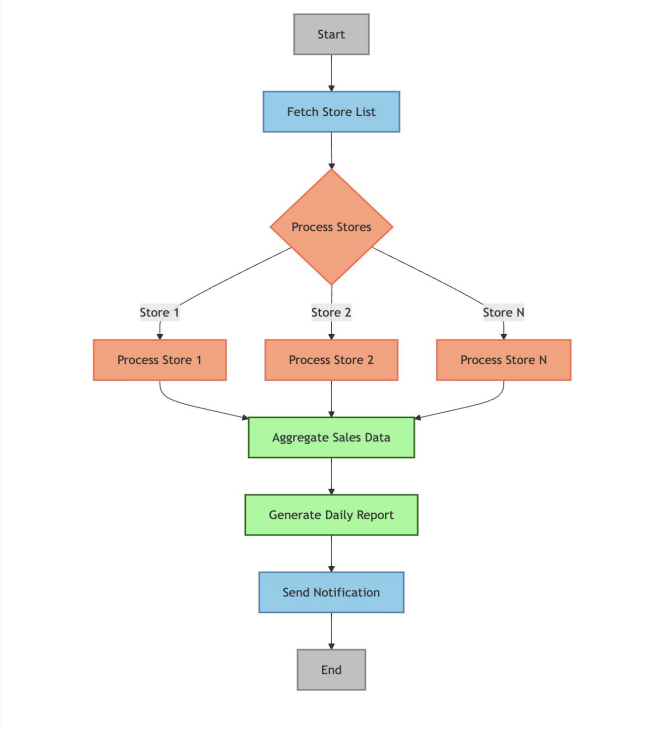
Branching and Conditional Execution

```
ShortCircuitOperator(
        task_id='check_data_availability',

python_callable=check_data_availability,
    )
```



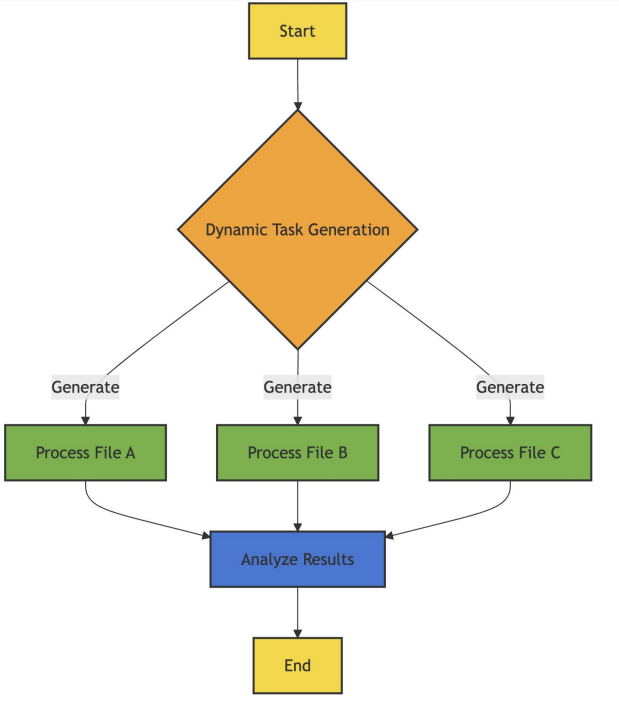
Dynamic Task Generation

- You need to create many similar tasks dynamically based on data or configuration.
- Dynamically generating tasks based on API results
- Parallelism



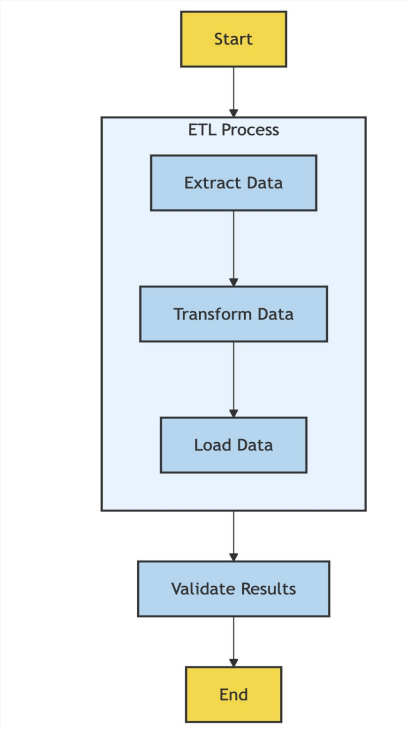
Dynamic Task Generation Example

```
   # List of files to process
files_to_process = ['file_A.csv',
'file_B.csv', 'file_C.csv']


# Dynamically create tasks for each file
process_tasks = []
for file in files_to_process:
     task = PythonOperator(
           task_id=f'process_{file}',
           python_callable=process_file,
           op_kwargs={'filename': file}
        )
     process_tasks.append(task)
```



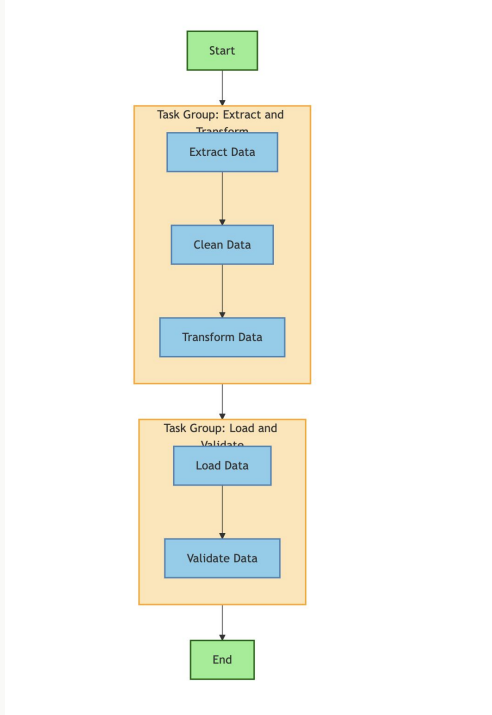
Task Groups

- Organize complex DAGs into logical groups
- Improve DAG readability and maintainability
- Simplify dependency management between groups of tasks

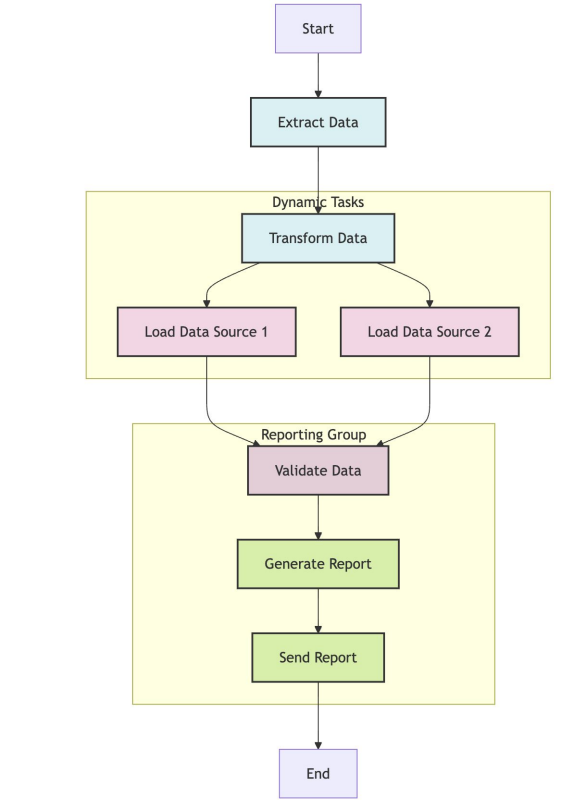


Task Groups

```
with TaskGroup('extract_and_transform') as
extract_transform_group:
        extract >> clean >> transform

with TaskGroup('load_and_validate') as
load_validate_group:
        load >> validate


start >> extract_transform_group >>
load_validate_group >> end
```



Combined dynamic tasks + task groups



Configure DAGs

Configuring DAGs for Flexibility and Scalability

- Leverage DAG parameters for dynamic execution
- Implement cross-DAG dependencies
- Generate DAGs dynamically for complex workflows

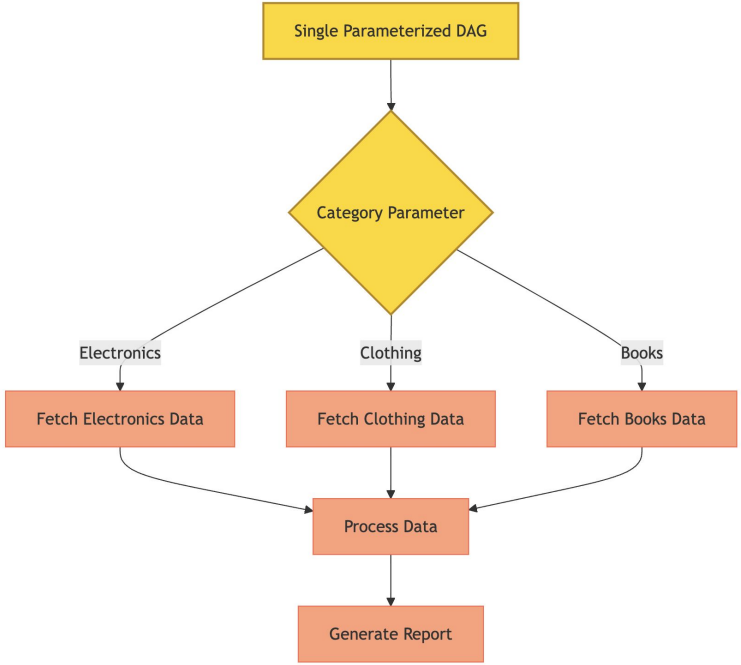
DAG Params

```
@dag(

start_date=datetime(2023, 6, 1),
schedule=None,
catchup=False,
params={ "greeting": "Hello!",
         "multiplier": Param( default=3, type="integer", ),
         "repeat_count": Param( default=5, type="integer",),
},
)

@task

def display_parameters(params: dict):
      param1 = params["param1"]
      param2 = params["param2"]

      print(param1 * 3) # Multiply string (param1) by 3
      print(f"Parameter 2: {param2}") # Display param2
```

DAG Params

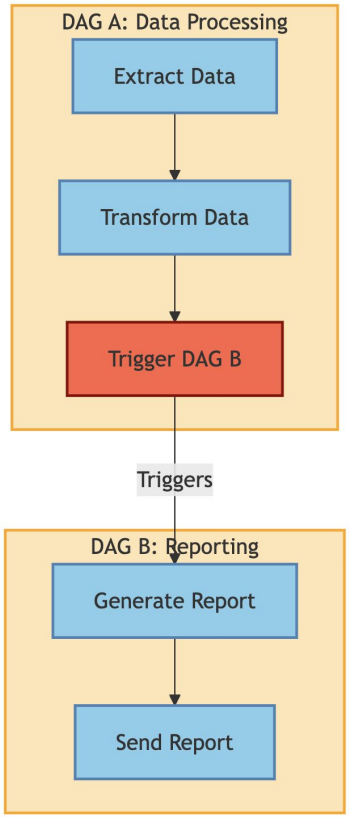
```
 categories = {
     'electronics':
 'https://api.example.com/electronics',
     'clothing':
 'https://api.example.com/clothing',
     'books': 'https://api.example.com/books'
 }
```

```
fetch_task = PythonOperator(
    task_id=f'fetch_{category["name"]}_data',
    python_callable=fetch_data,
    op_kwargs={'category': category['name']},
)
```



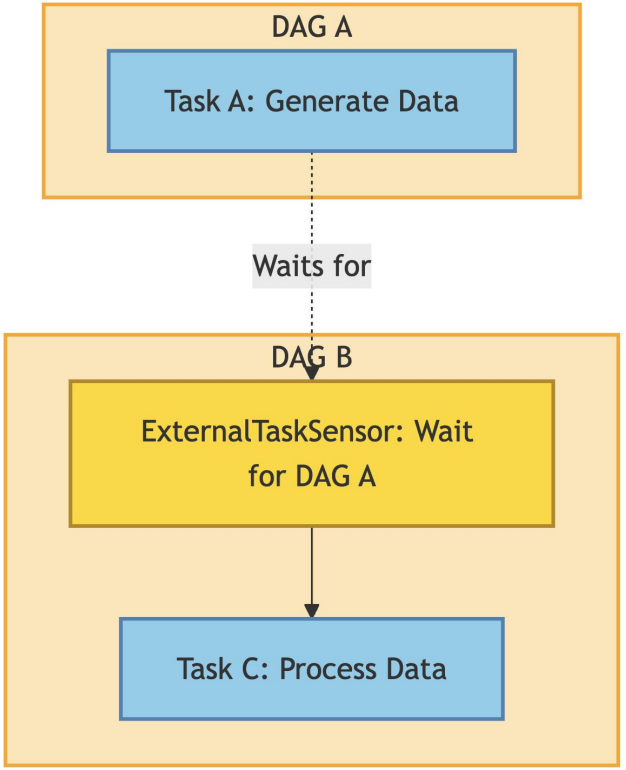
Cross DAG triggering

```python
trigger_dag_b = TriggerDagRunOperator(
        task_id='trigger_dag_b',
        trigger_dag_id='dag_b_reporting',
        conf={'triggered_by': 'dag_a'},
    )
```



Cross DAG Sensor

```
wait_for_dag_a = ExternalTaskSensor(
    task_id='wait_for_dag_a',
    external_dag_id='dag_a_generate',
    external_task_id='generate_data',
    timeout=3600,  # Timeout after 1 hour
    poke_interval=60,  # Check every 60 seconds
    mode='poke'
     )
```

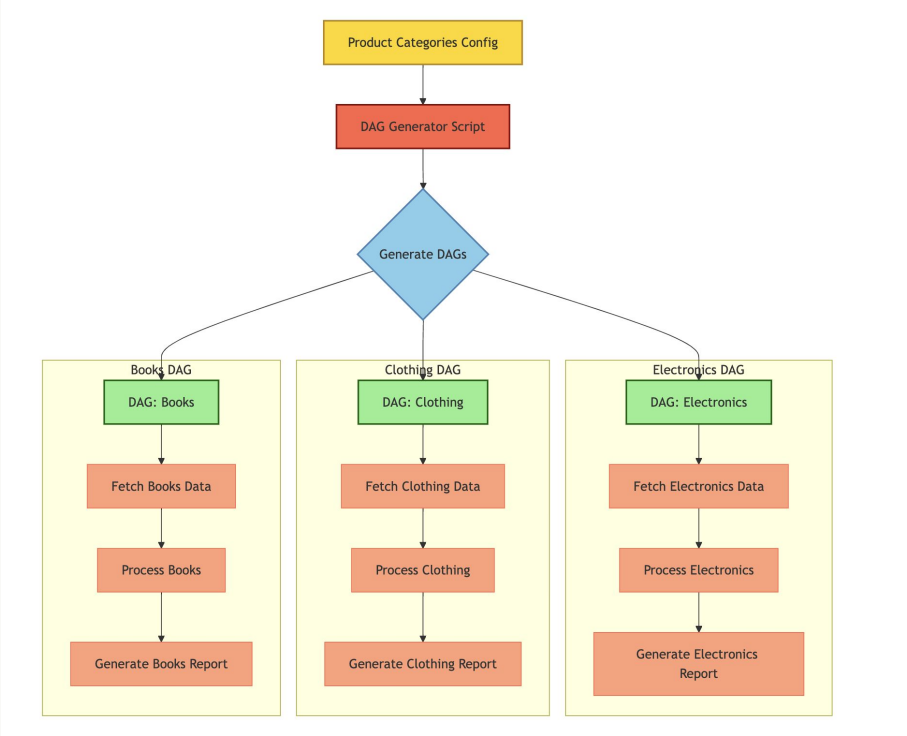


Dynamic DAG Generation

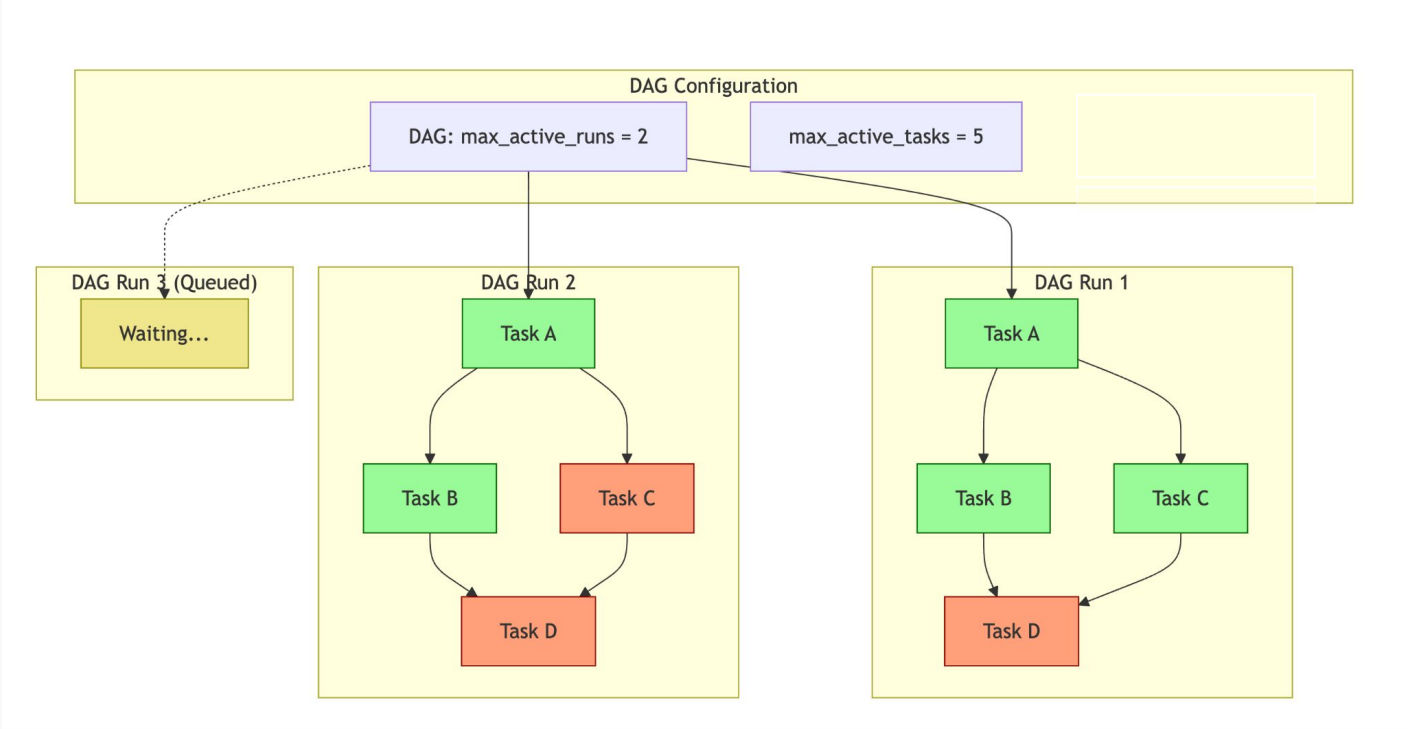
- DAGs share common code
- Needs to run at different schedules
- Generate using common template

Dynamic DAG Generation

```yaml
categories:
  - name: electronics
    schedule: '0 1 * * *'
    api_endpoint:
'https://api.example.com/electronics'
  - name: clothing
    schedule: '0 2 * * *'
    api_endpoint:
'https://api.example.com/clothing'
  - name: books
    schedule: '0 3 * * *'
    api_endpoint:
'https://api.example.com/books'
```



DAG Concurrency






- Task best practices
- Organize tasks
- DAG flexibility

Questions?