

# Bringing data pipelines to production, with Airflow

Jake Roach

ASTRONOMER



## Jake Roach



Buffalo, NY



Data Engineer and Instructor



150 rounds of golf with my wife!

ASTRONOMER

ASTRONOMER

Growing a data team that's manually **building and running data pipelines**, creating analyses, and developing models, but without a tool to centrally manage all these moving parts



Develop and run  
workflows as code  
in a centralized  
platform



Unlock tools to  
supercharge data  
pipelines



Monitor and  
manage data  
pipeline execution

ASTRONOMER



# Apache Airflow

- Programmatically define data pipelines
  - Dynamic
  - Extensible
  - Flexible
- DAGs
- Schedule, retry, alert
- Monitor data workflows

DAG (directed-acyclic graph)



# Extracting, transforming, and loading market data

- Pull data from the Polygon API
- Flatten the JSON response
- Transform the flattened response
- Load the transformed DataFrame to a Postgres database

```
market_data_etl.py

def extract():
    # Pull data from an API for today's date
    market_date = "2023-12-08"
    raw_response = requests.get(f"https://...{market_date}")
    ...
    return raw_dataset

def transform(raw_dataset):
    # Transform the raw dataset
    ...

...

# Execute the ETL
raw_dataset = extract()
cleaned_dataset = transform(raw_dataset)
load(cleaned_dataset)
```

```
> python3 market_data_etl.py
```

## Homegrown data pipelines require additional features to be production-grade

- Schedule
- Policy on retries
- Store sensitive credentials
- Visibility into execution details
- Lots of custom code

# Airflow makes data pipelines production-ready

- Scheduled to run daily each morning
- Retry on failure
- Securely store and retrieve sensitive credentials
- Persist execution details before run

Use traditional Airflow operators to build a DAG

ASTRONOMER



**extract\_market\_data**

PythonOperator



**flatten\_market\_data**

PythonOperator



**transform\_market\_data**

PythonOperator



**load\_market\_data**

PythonOperator





Provide a DAG  
ID, start date,  
schedule interval

Programmatically  
define tasks

Set dependencies  
between tasks

```
dags/market_etl__traditional.py

# Instantiate the DAG
with DAG(
    dag_id="market_etl__traditional",
    start_date=datetime(2023, 8, 1),
    end_date=datetime(2023, 8, 31),
    schedule="0 9 * * 1-5"
    ...
) as dag:
    # Create tasks using traditional operators
    extract_market_data = PythonOperator(
        dag=dag,
        task_id="extract_market_data",
        python_callable=extract_market_data__callable
    )

    ...

    # Set dependencies between tasks
    extract_market_data >> ... >> load_market_data
```

```
with DAG(
    ...
    default_args={
        "retries": 3,
        "retry_delay": timedelta(minutes=1)
    }
) as dag:
```

```
polygon_api_key = Variable.get("POLYGON_API_KEY")
```

```
flatten_market_data = PythonOperator(
    dag=dag,
    task_id="flatten_market_data",
    python_callable=flatten_market_data__callable,
    op_kwargs={"raw_dataset": "{{ ti.xcom_pull(task_ids='extract_market_data') }}" }
)
```

- Configure custom retry policy
- Securely retrieve variables in code
- Share data between tasks with XComs



## DAG: market\_etl\_\_traditional

Schedule: 0 9 \* \* 1-5

Next Run: 2023-08-03, 09:00:00



Grid



Graph



Calendar



Task Duration



Task Tries



Landing Times



Gantt



Details



Code



Audit Log



12/02/2023, 07:54:18 PM

25

All Run Types

All Run States

Clear Filters

Auto-refresh

Press **shift** + **/** for Shortcuts

deferred

failed

queued

removed

restarting

running

scheduled

skipped

success

up\_for\_reschedule

up\_for\_retry

upstream\_failed

no\_status



DAG

market\_etl\_\_traditional

Run

2023-08-03, 09:00:00 UTC

Task

extract\_market\_data

Clear task

Mark state as...

Filter Tasks

Details

Graph

Gantt

Code

Logs

More Details

Rendered Template

XCom

List Instances, all runs

## Task Instance Notes:

Add Note

## Task Instance Details

Status

success

Task ID

extract\_market\_data

Run ID

scheduled\_\_2023-08-02T09:00:00+00:00

Operator

PythonOperator

# Airflow makes writing production-ready data pipelines a breeze

- Schedule DAG runs
- Visible failures with options to retry
- Ability to securely interact with source systems and destinations
- Single “pane of glass” into pipeline execution details

**TaskFlow API makes getting started with Airflow even easier**

# The TaskFlow API makes writing DAGs more intuitive for data teams

**@dag**  
**@task**

DAGs and tasks  
defined as  
functions



Useful when  
passing information  
between tasks



Makes Airflow more  
accessible to data  
scientists and  
analysts

Same parameters as  
traditional DAG  
definition

Intuitive process to  
create tasks

Easy to share data  
between tasks and  
set dependencies

# ASTRONOMER

```
dags/market_etl__taskflow_api.py

@dag(
    start_date=datetime(2023, 8, 1),
    end_date=datetime(2023, 8, 31),
    schedule="0 9 * * 1-5",
    ...
)
def market_etl__taskflow_api():
    @task()
    def extract_market_data():
        ...
        return raw_dataset

    @task()
    def flatten_market_data(raw_dataset, **context):
        ...

        raw_data = extract_market_data()
        flattened_data = flatten_market_data(raw_data)
        transformed_data = transform_market_data(flattened_data)
        load_market_data(transformed_data)

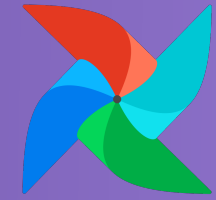
market_etl__taskflow_api()
```

# The TaskFlow API helps data teams provide immediate business value with Airflow

- Easier to port Python ETL logic to Airflow DAGs
- Make Airflow more accessible
- Maintaining the best functionality of DAGs and tasks
- Astro Python SDK extends this functionality

# Extracting, transforming, and loading market data with Airflow

- Extracted, transformed, and loaded market data
- Schedule, visibility into pipeline execution, securely interact with data assets
- Use both traditional Airflow operators and TaskFlow API



**extract\_market\_data**

PythonOperator

**flatten\_market\_data**

...

**transform\_market\_data**

@task

**load\_market\_data**

@task



# Airflow reduces the code needed to integrate your data stack

- TaskFlow API
- Variables and Connections
- Jinja templating at run-time

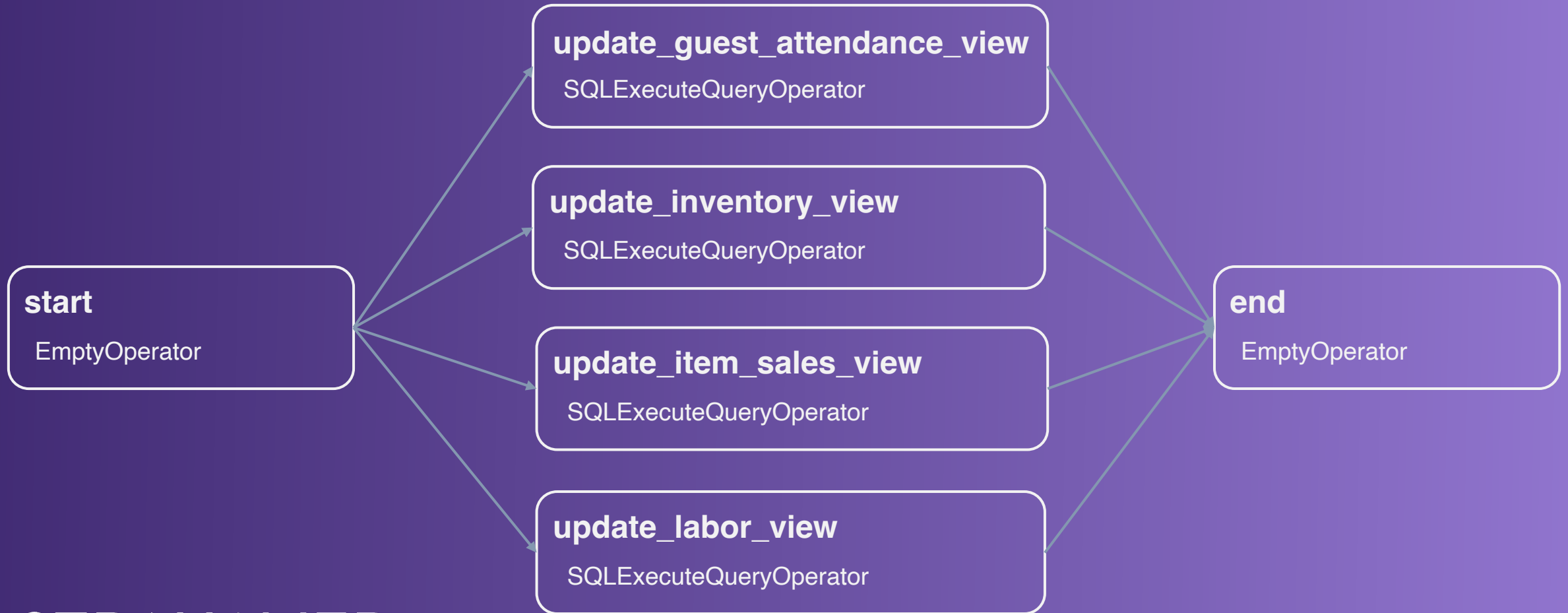
Ever-growing number of pre-built operators to help orchestrate workflows



ASTRONOMER



# Automating analytics workflows





dags/daily\_operations\_view\_update.py

...

# Define tasks using SQLExecuteQueryOperator

```
update_guest_attendance_view = SQLExecuteQueryOperator(  
    dag=dag,  
    task_id="update_guest_attendance_view",  
    postgres_conn_id="postgres_daily_operational_conn",  
    sql="""  
    CREATE OR REPLACE VIEW admissions_by_entrance AS (  
        SELECT  
            ...  
        WHERE admission_date = '{{ ds }}'  
        ...  
    );""")
```

...

# Set dependencies between tasks

```
start >> [update_guest_attendance_view, update_labor_view, ...] >> end
```

## SQLExecuteQuery- Operator

} Easily connect to data  
sources and  
destinations

} Use Jinja templating  
to render runtime

} Allow view updates to  
run in parallel

Add a templated  
path to store SQL  
files



Provide path to SQL  
file in operator call



Create connection  
with Astro CLI



```
with DAG(  
    ...  
    template_searchpath="include/sql"  
    ...  
) as dag:  
  
    ...  
  
    update_inventory_view = SQLExecuteQueryOperator(  
        dag=dag,  
        task_id="update_inventory_view",  
        postgres_conn_id="postgres_daily_operational_conn",  
        sql="update_inventory_view.sql"  
    )  
  
    ...
```

```
> astro dev run connections add ... <connection-name>
```

# Using pre-built operators streamlines data pipeline data development



Reduces custom  
code to build a  
DAG



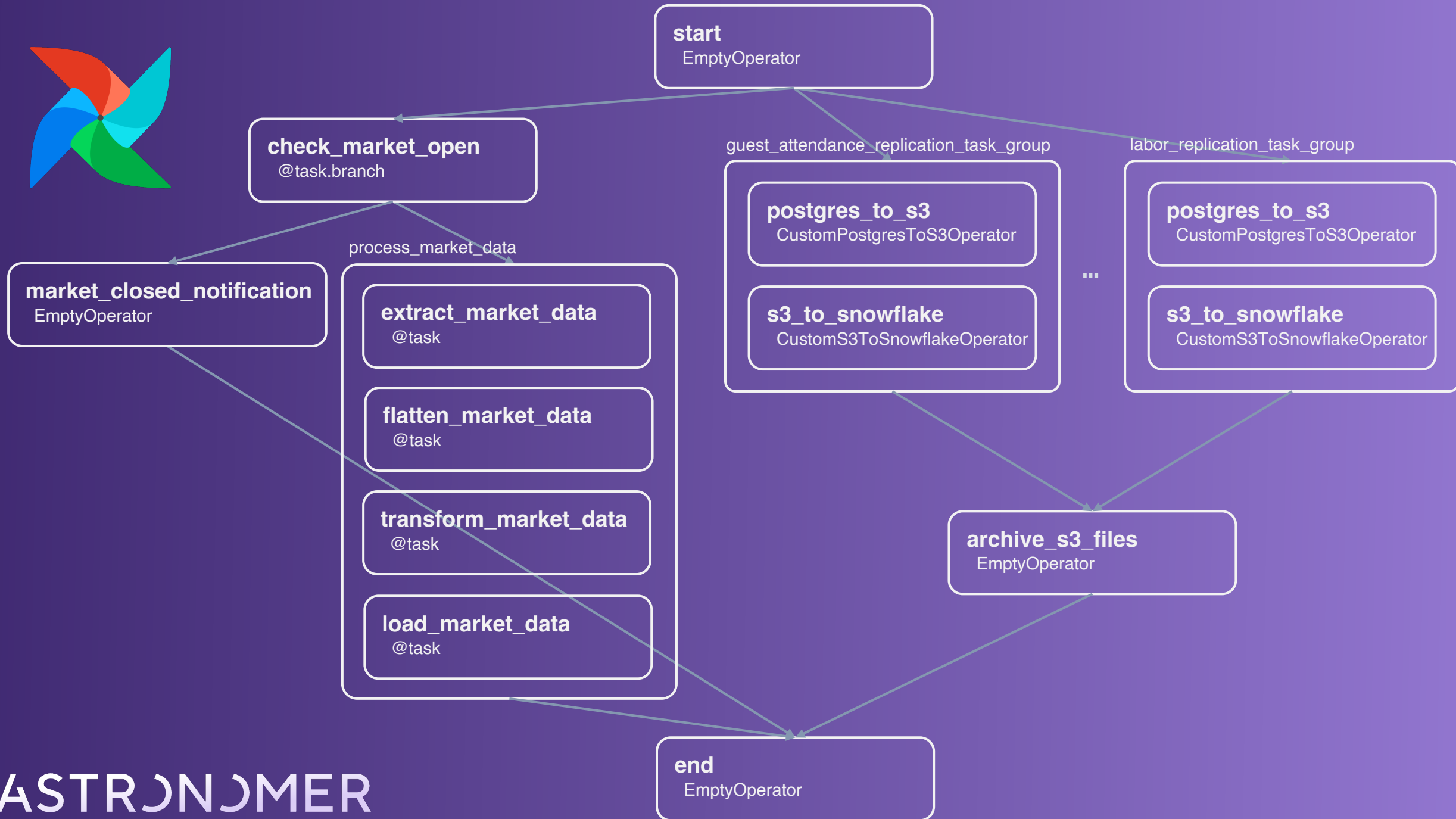
Workflows are  
easier to read and  
troubleshoot



Leverage Jinja  
template engine

# Airflow makes complex data pipeline development simple

- Pulling from disparate sources
- Mix of traditional operators and TaskFlow API
- Task groups and branching logic
- Custom operators



# Task groups help to delineate similar tasks within a DAG

```
@task_group(group_id="process_market_data")
def process_market_data():
    # Add tasks to task group
    ...
```

```
with TaskGroup(
    group_id="process_market_data"
) as task_group:
    # Add tasks to task group
    ...
```

process\_market\_data

**extract\_market\_data**  
@task

**flatten\_market\_data**  
@task

**transform\_market\_data**  
@task

**load\_market\_data**  
@task



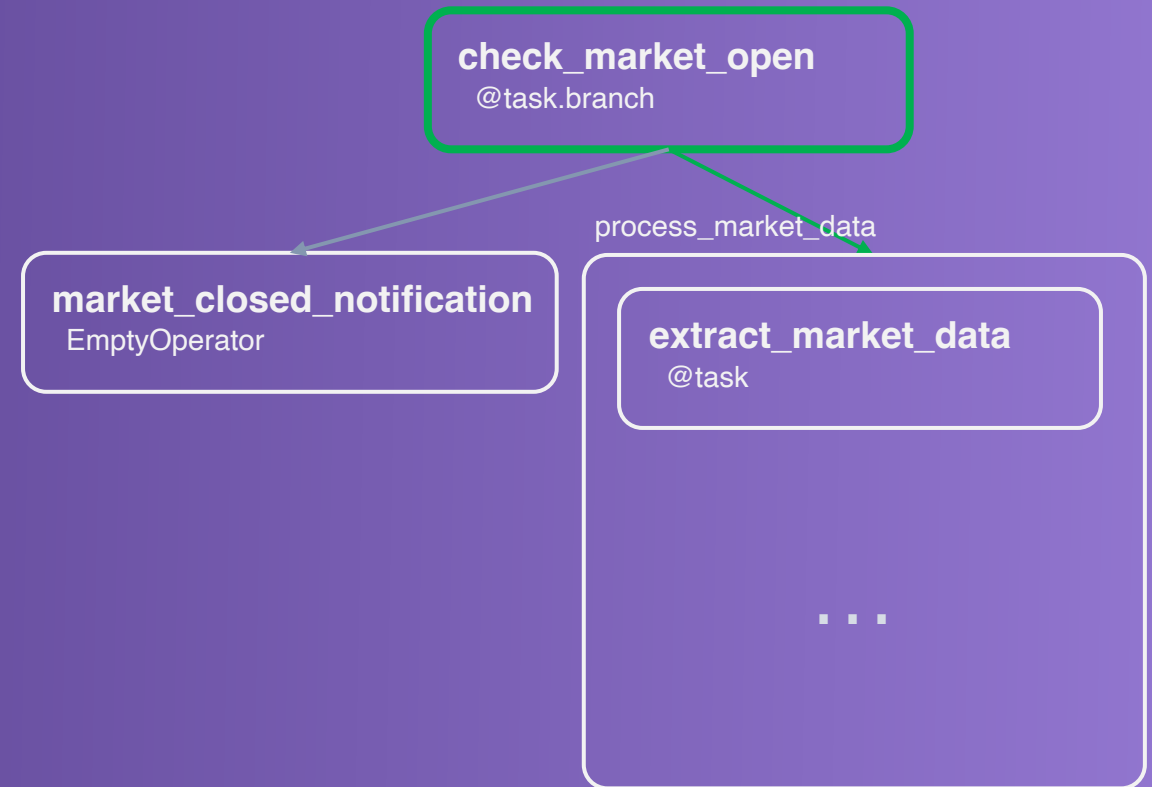
# Branching allows for complex logic to be implemented in pipelines

```
@task.branch
def check_market_open(**context):
    ...

    # Branch between two options
    if is_weekend or is_market_holiday:
        return ["market_closed_notification"]

    return ["process_market_data.<task-name>"]

...
```



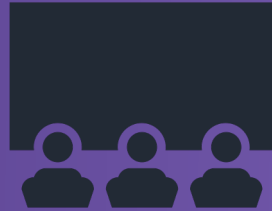
# Custom operators leverage Airflow's extensibility meet to your data team's needs

- Implement custom logic
- Extends existing functionality
- Easy to reuse across DAGs

# Airflow helps to simplify developing and maintaining data workflows of all types



Programmatically  
define data  
pipelines



Single-pane of  
glass into  
orchestration  
environment



Leverage tools to  
supercharge your  
data pipelines

ASTRONOMER

# Astronomer provides out-of-the-box tools to run Airflow in production

- The leading managed service provider for Airflow
- Easy to create and maintain Airflow environments, without managing the infrastructure
- Astro CLI, Cloud IDE , Astro SDK

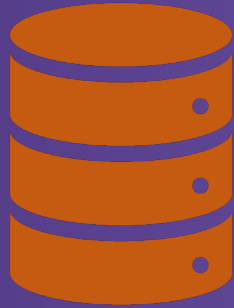


+



ASTRONOMER

# Obstacles



Creating a complex  
data with a limited  
stack



Enable Airflow to  
make the biggest  
impact

ASTRONOMER

# Appendix



<https://github.com/jroachgolf84/astronomer-panel-interview>



<https://airflow.apache.org>

<https://airflow.apache.org/docs/apache-airflow/stable/index.html>



<https://www.astronomer.io>

ASTRONOMER

ASTRONOMER