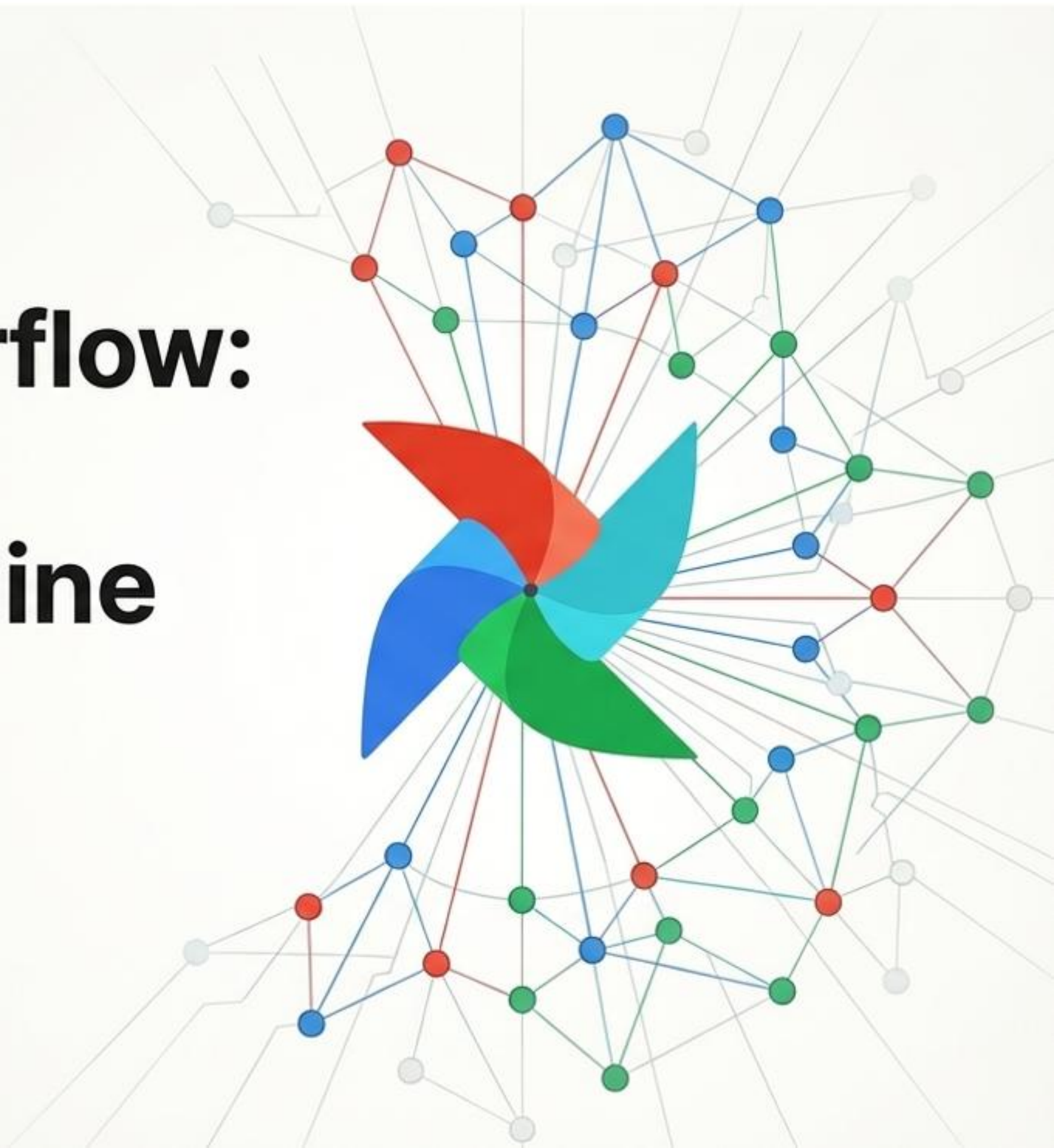


# Master Apache Airflow: Orchestrating the Modern Data Pipeline

From DAG Concepts to  
Enterprise Scale



# The Orchestrator's Journey

**Phase 1**



The Conceptual  
Blueprint  
(Pipelines & DAGs)

**Phase 2**



Building the  
Foundation  
(Architecture &  
Setup)

**Phase 3**



Crafting the  
Workflow  
(DAGs, Operators &  
Context)

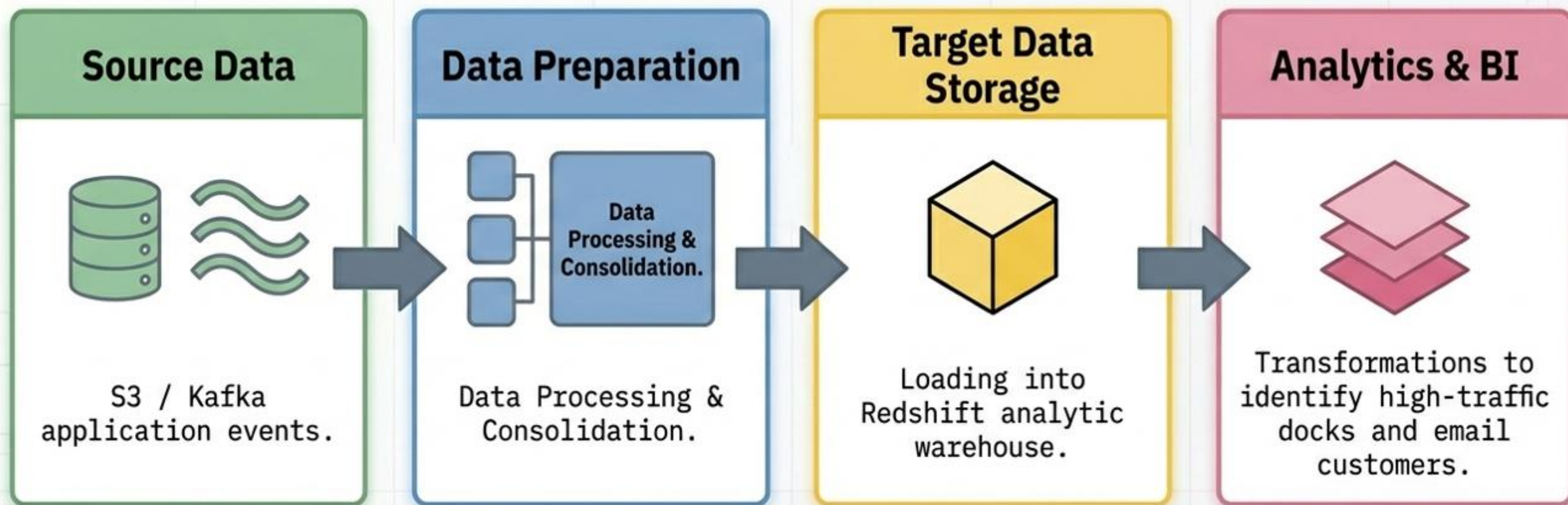
**Phase 4**



Scaling to  
Production  
(Quality, Extensions  
& Monitoring)

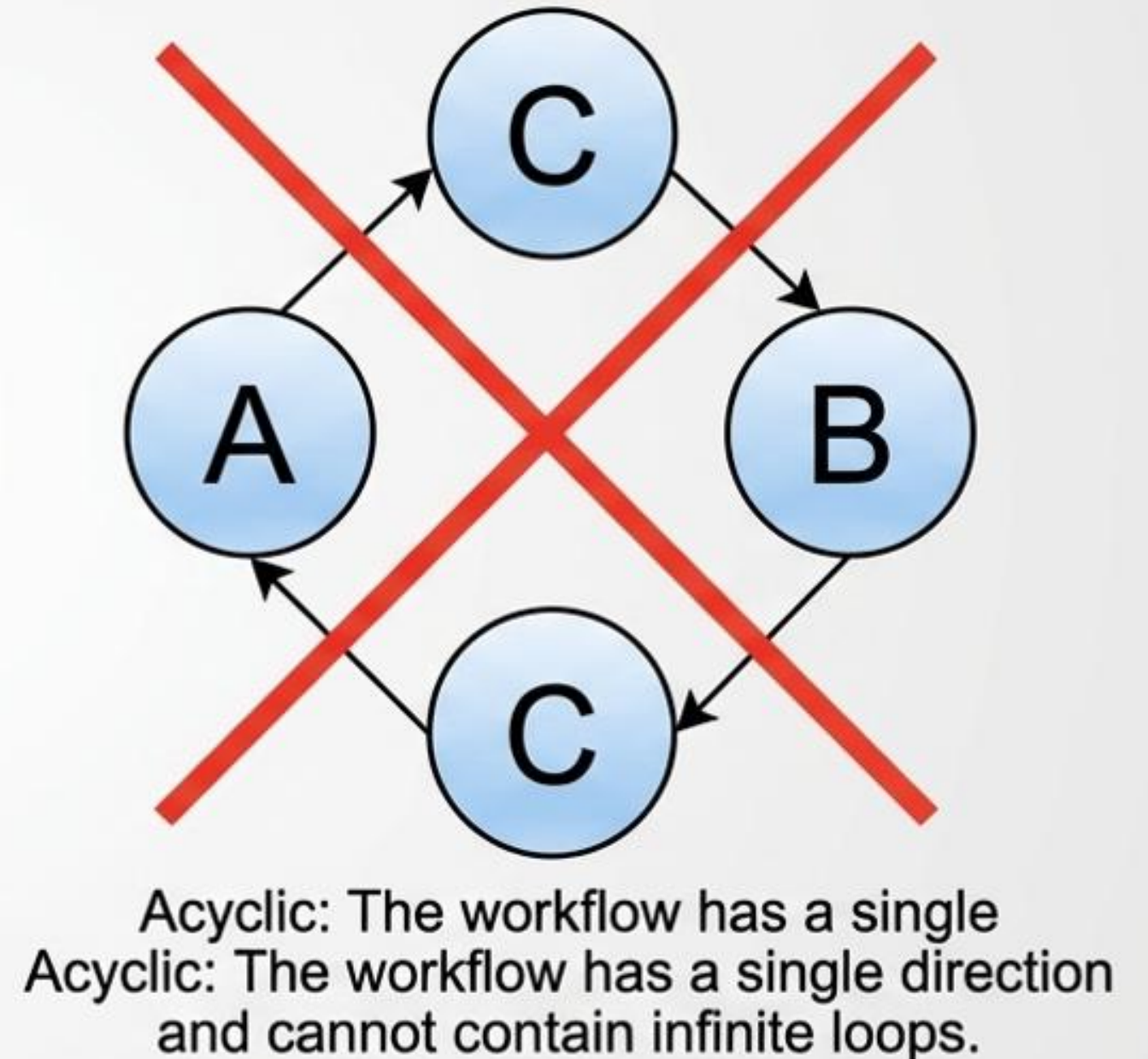
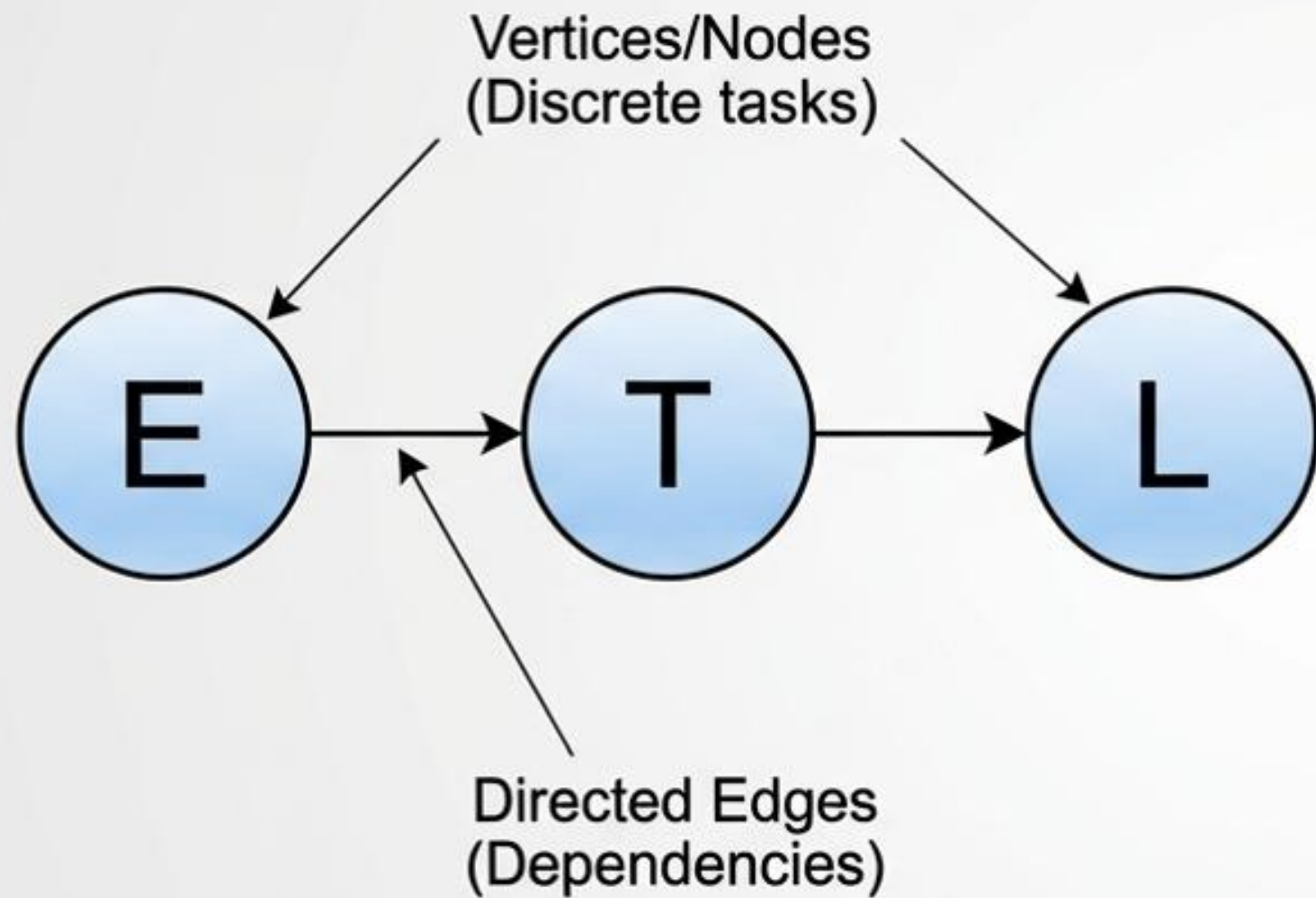
# Defining the Pipeline: The Bikeshare Scenario

**“Definition:** A series of steps in which data is processed.”



Pipelines provide logical guidelines and common terminology to organize everyday data engineering tasks.

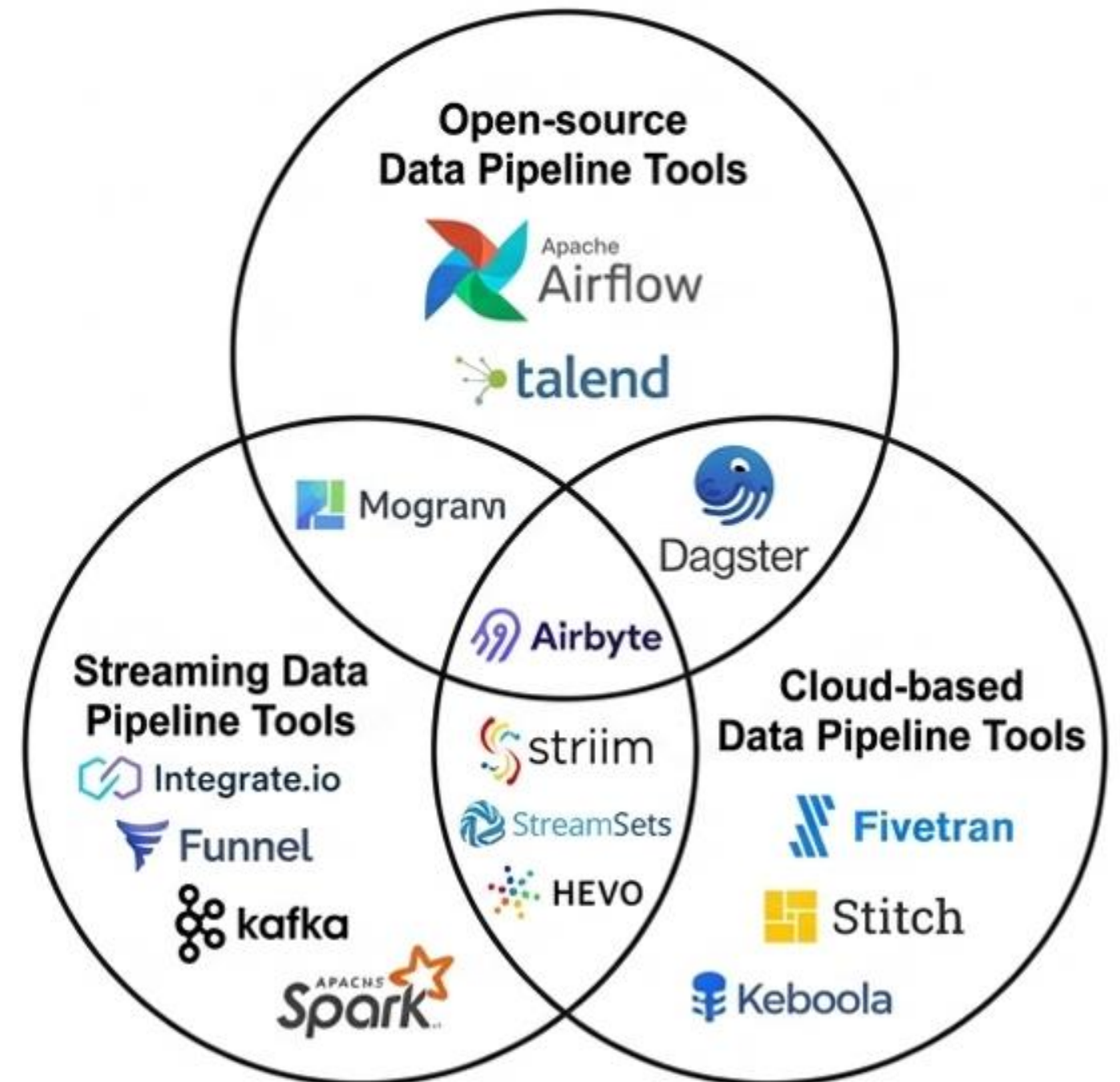
# The Mathematical Soul: Directed Acyclic Graphs (DAGs)



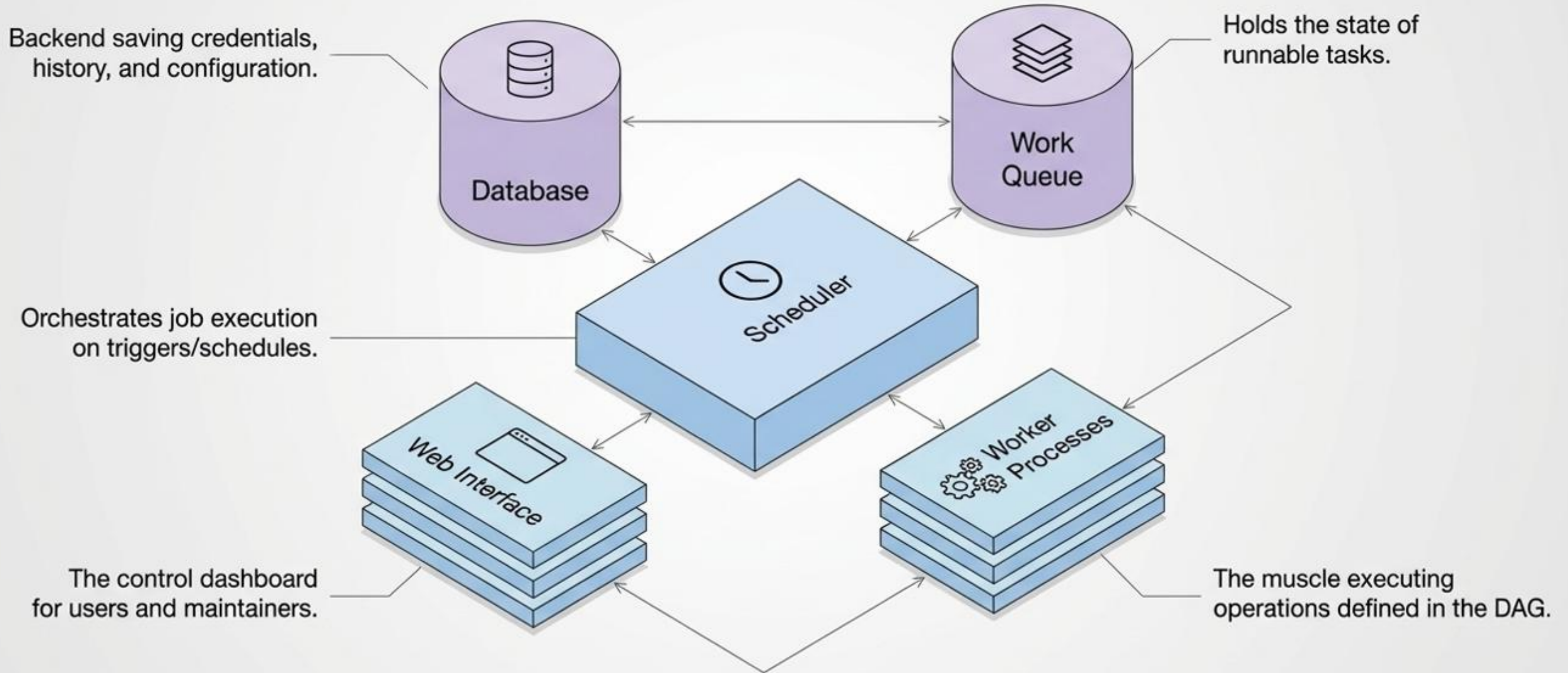
Data pipelines are best expressed as DAGs.

# Enter Apache Airflow

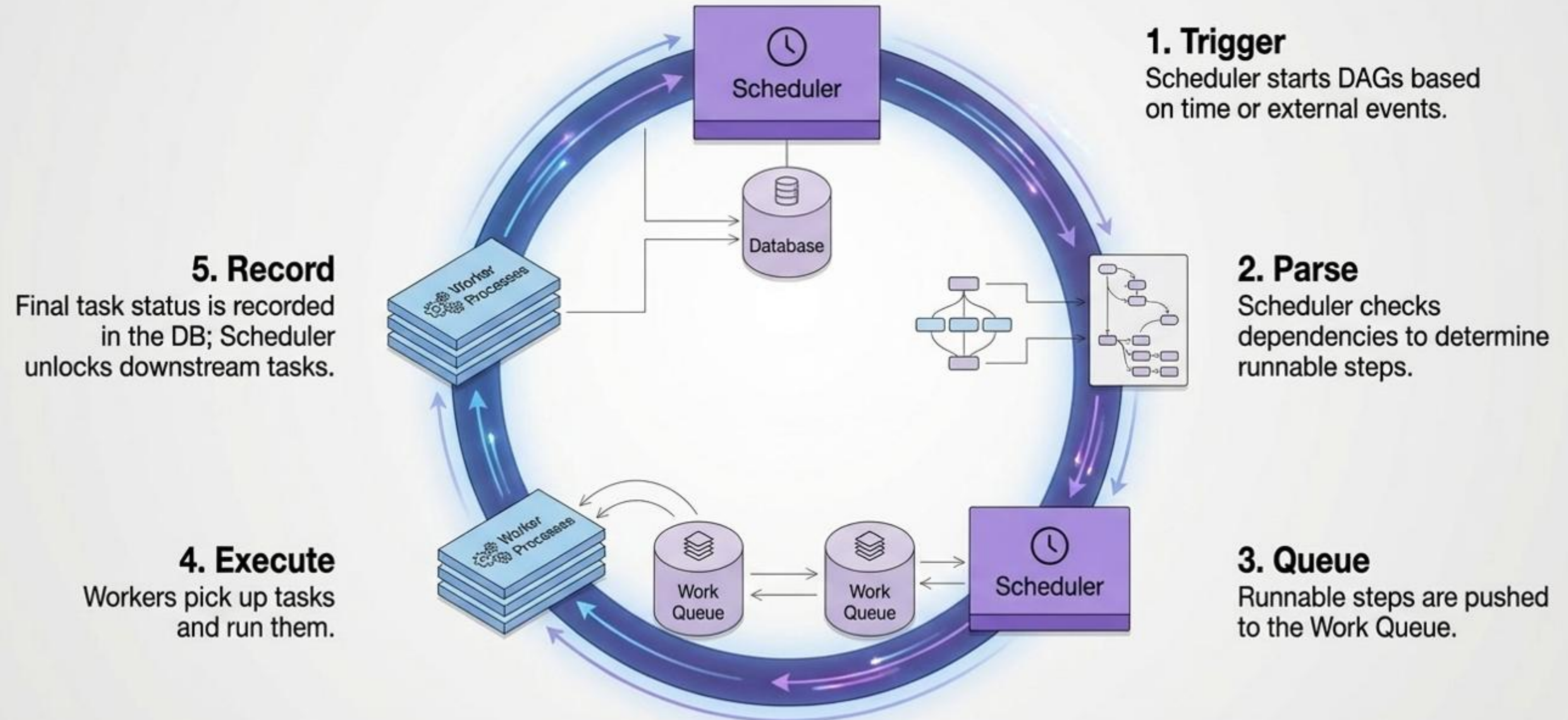
Spec Sheet	
<b>Origins:</b>	Originally developed at Airbnb
<b>Philosophy:</b>	Configuration as Code (Workflows written in Python)
<b>License:</b>	Open source (Apache 2.0)
<b>Capabilities:</b>	Schedules by time/event, integrates with Hadoop, Cloud Services, and databases.



# Anatomy of an Orchestrator



# The Execution Lifecycle



# Practical Setup: Docker & CLI

- **Environment:** Running Airflow locally using Docker (puckel/docker-airflow).
- **Access:** Web UI maps to localhost:8080.
- **CLI Interaction:** Access the container bash shell to run core utilities.

```
docker pull puckel/docker-airflow
```

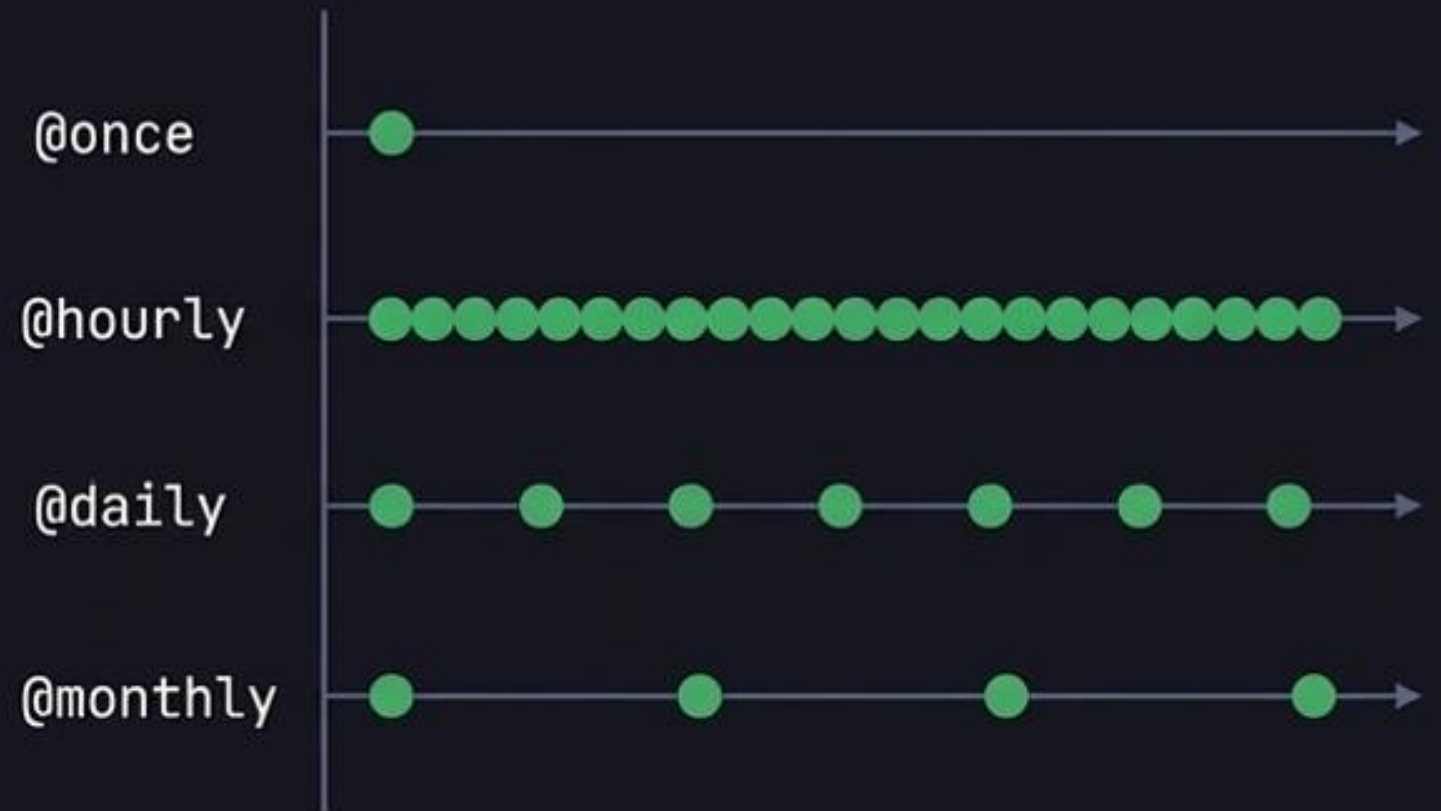
```
docker run -d -p 8080:8080 puckel/docker-airflow  
webservice
```

```
docker exec -it <container> /bin/bash
```

```
airflow list_dags
```

# The Blueprint in Code: DAG Definition & Scheduling

```
divvy_dag = DAG(  
    'divvy',  
    description='Analyzes Divvy Bikeshare Data',  
    start_date=datetime(2019, 2, 4),  
    schedule_interval='@daily'  
)
```



Schedules are defined with explicit Cron expressions or presets.

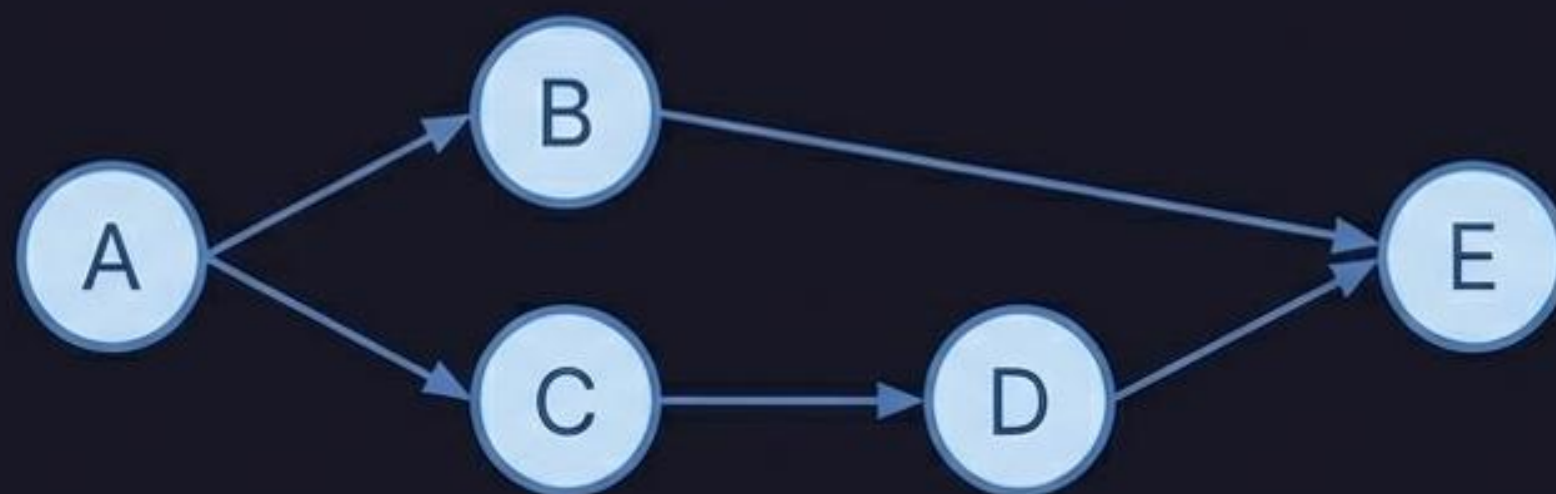
# The Operator Matrix: Atoms of Work

**Concept:** Operators define the atomic steps of work. An instantiated Operator is a Task.

PythonOperator	BashOperator	External Operators
<p>Executes custom Python callables. Ideal for complex, bespoke logic.</p>	<p>Executes bash scripts. Ideal for OS-level file manipulation.</p>	<p>Pre-built integrations for standard systems. Includes PostgresOperator, S3ToRedshiftOperator, and SimpleHttpOperator.</p>

# Directing Traffic: Task Dependencies

**The Rule:** Edges define the exact ordering and dependency between initialized Tasks.



## Method 1: Bitshift Operators (Modern & Concise)

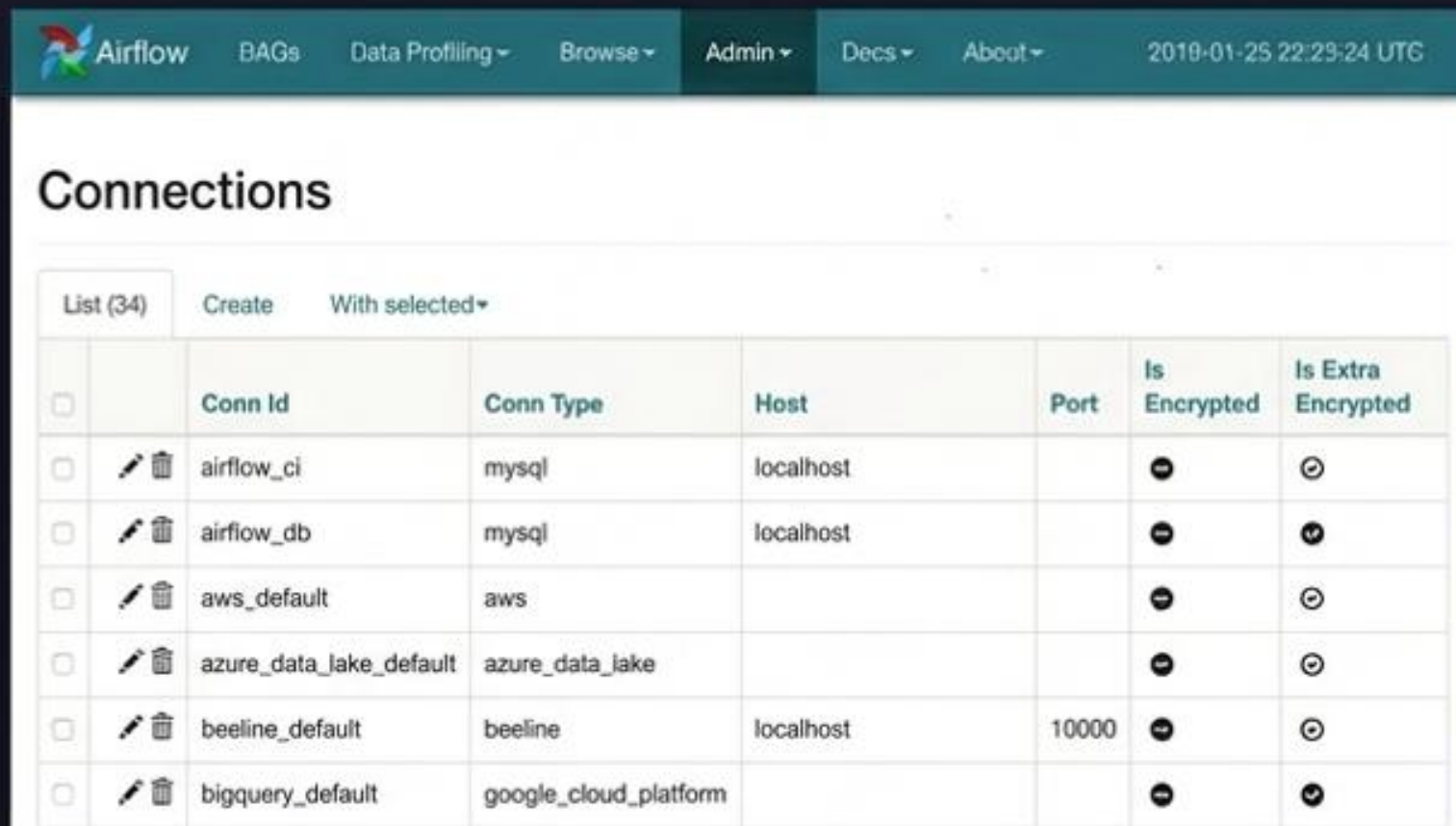
```
a >> b (Task A comes before Task B)  
a << b (Task A comes after Task B)
```

## Method 2: Explicit Setters (Classic)













```
a.set_downstream(b)  
a.set_upstream(b)
```

# Bridging Systems: Connections & Hooks

**Connections:** Managed securely within the Web UI avoiding hardcoded secrets.



The screenshot shows the Airflow web interface with a navigation bar at the top containing 'Airflow', 'BAGs', 'Data Profiling', 'Browse', 'Admin', 'Docs', and 'About'. The date and time '2019-01-25 22:29:24 UTC' are displayed on the right. Below the navigation bar, the page title is 'Connections'. There are buttons for 'List (34)', 'Create', and 'With selected'. A table lists various connections with columns for 'Conn Id', 'Conn Type', 'Host', 'Port', 'Is Encrypted', and 'Is Extra Encrypted'.

		Conn Id	Conn Type	Host	Port	Is Encrypted	Is Extra Encrypted
<input type="checkbox"/>	 	airflow_ci	mysql	localhost		<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	 	airflow_db	mysql	localhost		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/>	 	aws_default	aws			<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	 	azure_data_lake_default	azure_data_lake			<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	 	beeline_default	beeline	localhost	10000	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	 	bigquery_default	google_cloud_platform			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

**Hooks:** The programmatic interface. Integrates directly with systems using predefined connections to execute queries.

```
db_hook = PostgresHook('demo')
df = db_hook.get_pandas_df('SELECT * FROM rides')
```

Hooks provide a consistent API to interact with external systems, abstracting the low-level connection details.

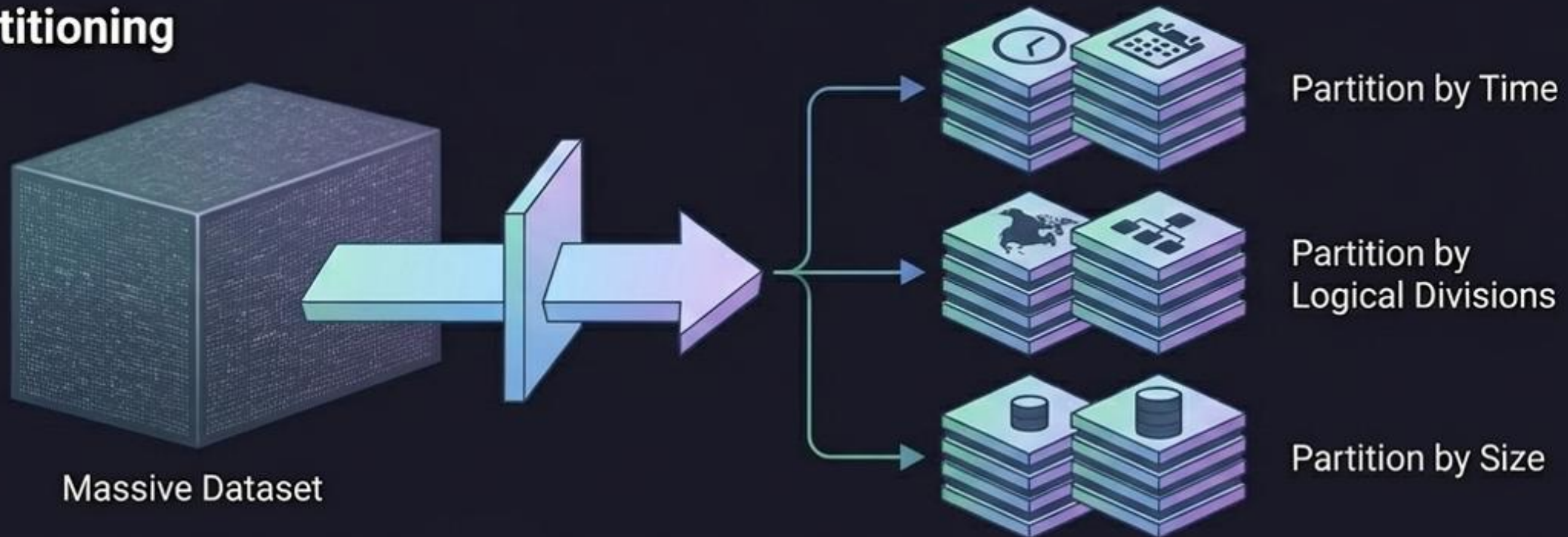
# Runtime Dynamics: Templating & Partitioning

**Jinja Templating:** Leverage 'fill in the blank' context variables to inject runtime specific data dynamically into tasks.

```
Parameter Injection Example

def check_greater_than_zero(*args, **kwargs):
    table = kwargs['params']['table']
    records = redshift_hook.get_records(f'SELECT COUNT(*) FROM {table}')
```

## Data Partitioning



# Trusting the Pipeline: Data Validation

Ensuring data is present, correct, and meaningful to satisfy its intended use.



## Data Consistency

Check 1: Ensure row counts in Redshift match the raw records in S3.



## Data Range

Check 2: Validate all loaded locations maintain a daily visit average greater than 0.



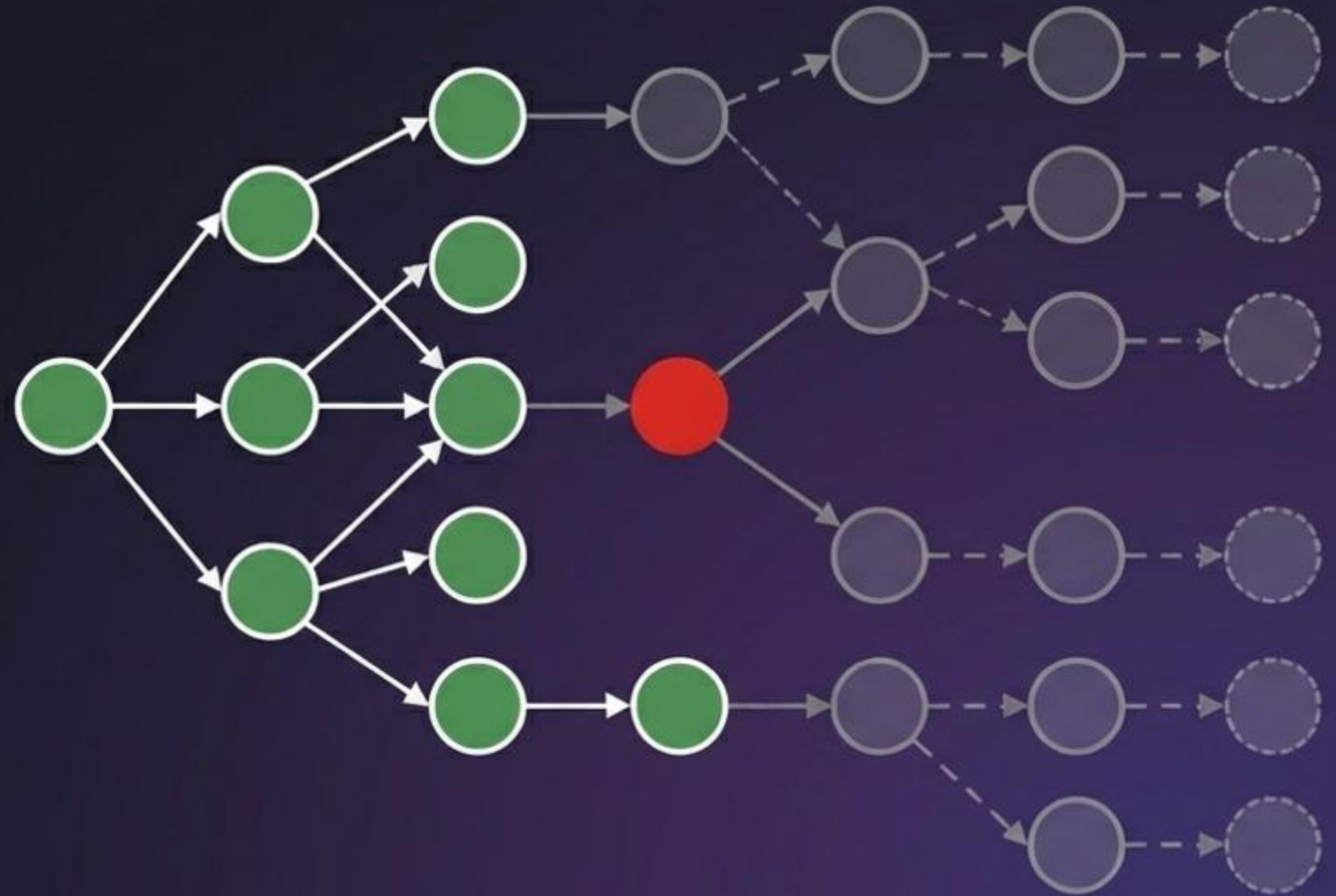
## Data Constraint

Check 3: Confirm output table dimensions match input parameters.

# Transparency & Debugging: Data Lineage

**Lineage:** The discrete steps involving the creation, movement, and calculation of a dataset.

**Visibility:** The Airflow Graph View isolates the exact point of failure cascading downstream.



# Operational Excellence: Task Boundaries & States

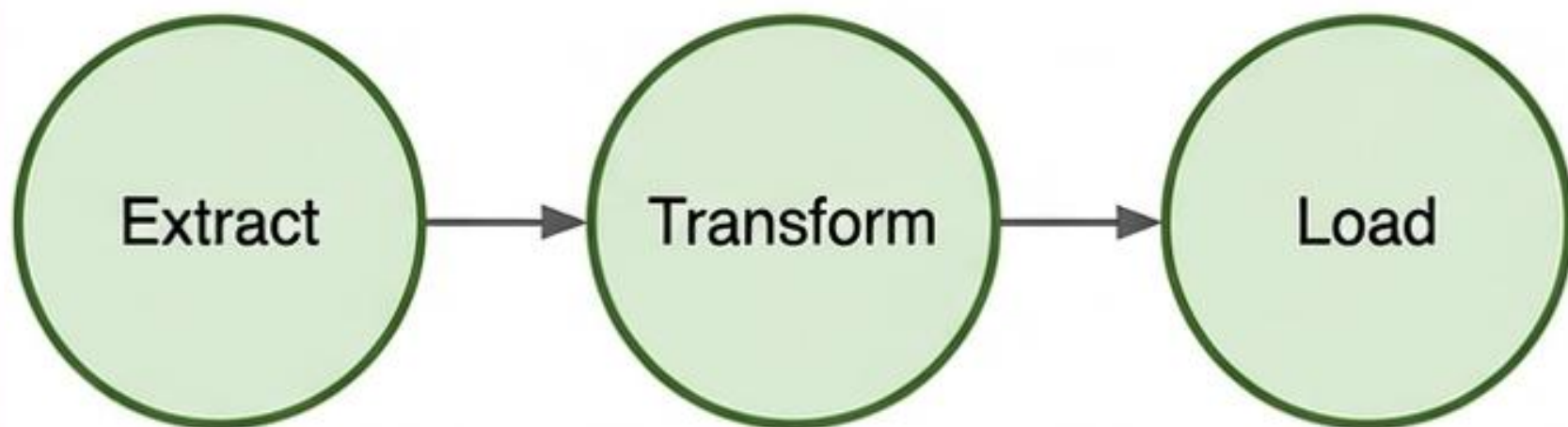
## Good vs. Bad

### Monolithic Script

Extract, Transform, Load, Validate

Anti-pattern: Single point of failure, low parallelism.

### Atomic Tasks



Best Practice: Maximizes parallelism, isolates failures.

**State Management:** Use `max_active_runs` to prevent database locking.

```
max_active_runs=1
```

Past

Future

Upstream

Downstream

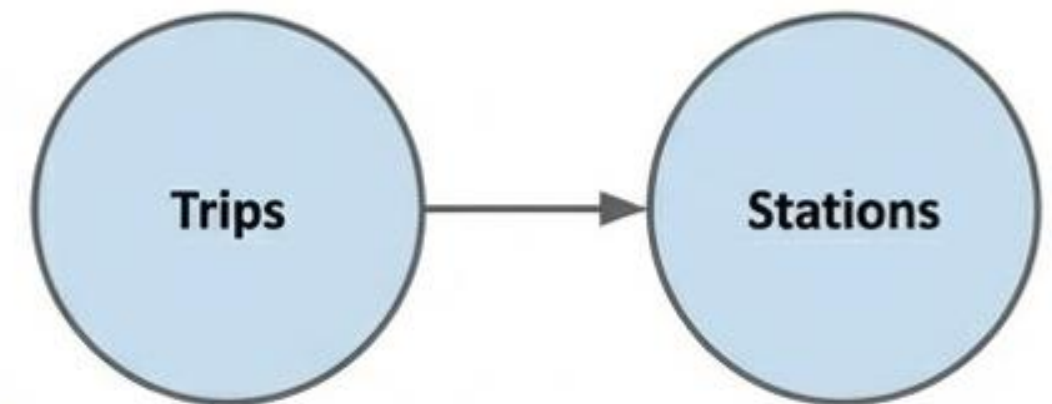
Operators to force re-runs: Manually clear states.

# Extending the Ecosystem: Custom Operators

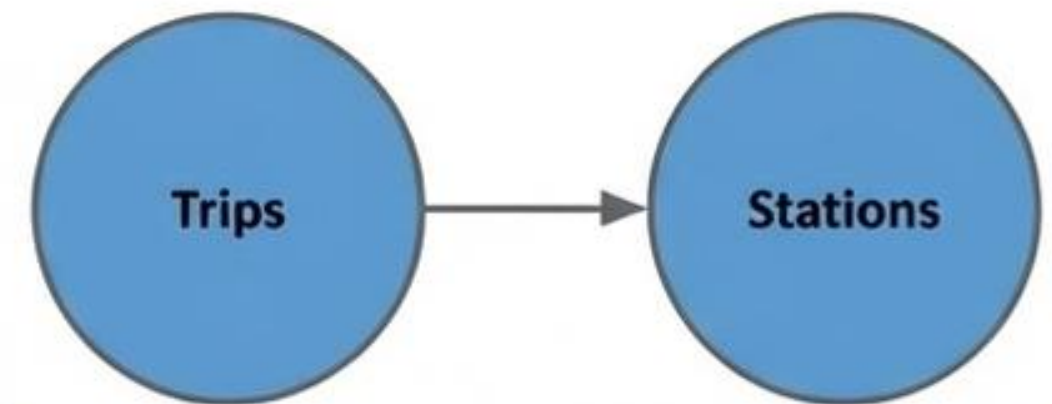
- **Purpose:** Capture frequently used, complex integrations into a reusable, DRY format.
- **Benefits:** Simplifies DAG logic, unifies logging, and enables fine-tuned performance.
- **Community First:** Always check the Open Source Airflow Contrib repository!

```
.  
|-- dags/  
|-- ...  
|-- plugins/  
    |-- custom_operators.py  
|-- ...  
|-- ...  
|-- ...
```

## S3ToRedshiftOperator



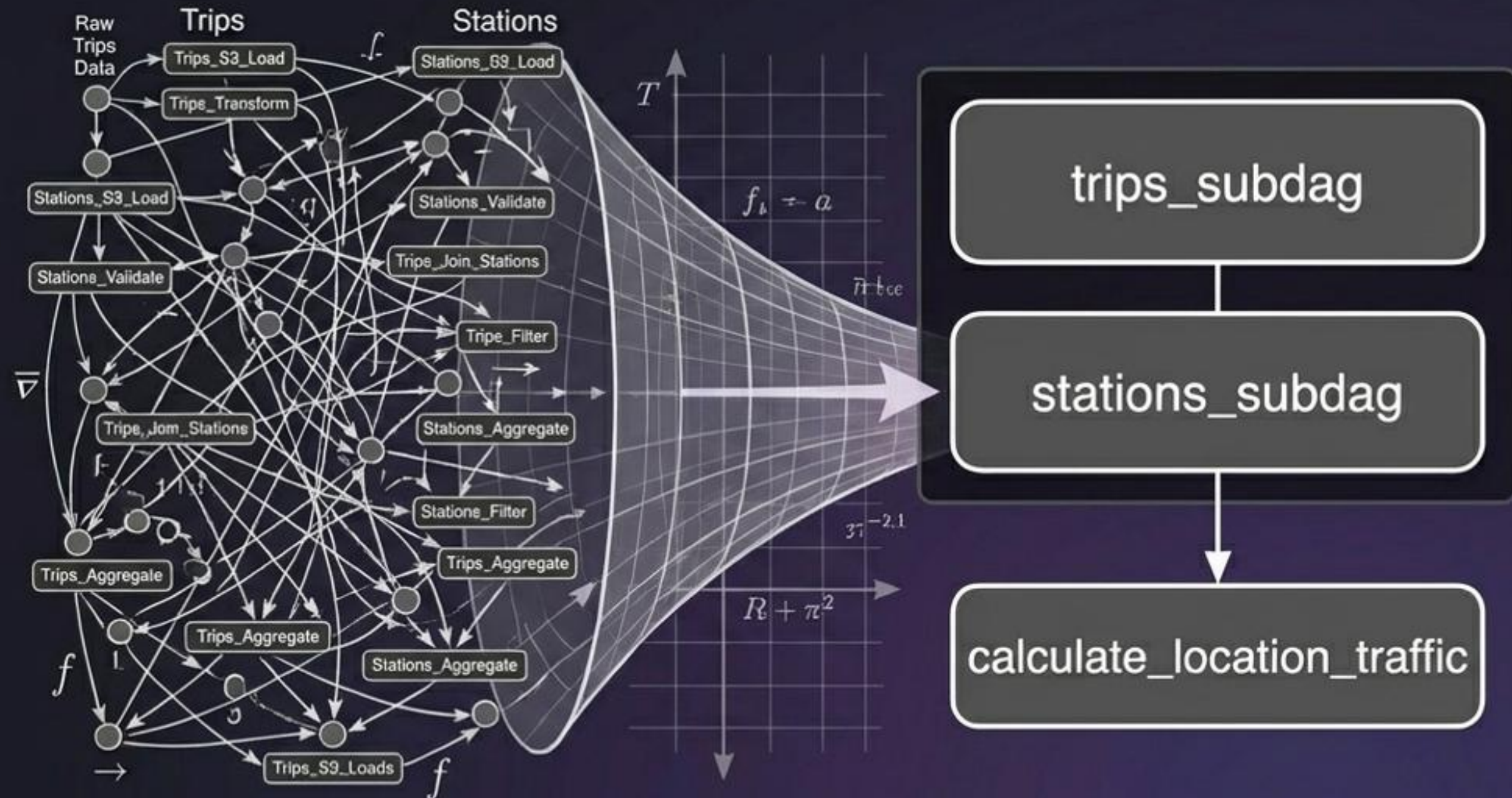
## HasRowsOperator



# Organization at Scale: SubDAGs

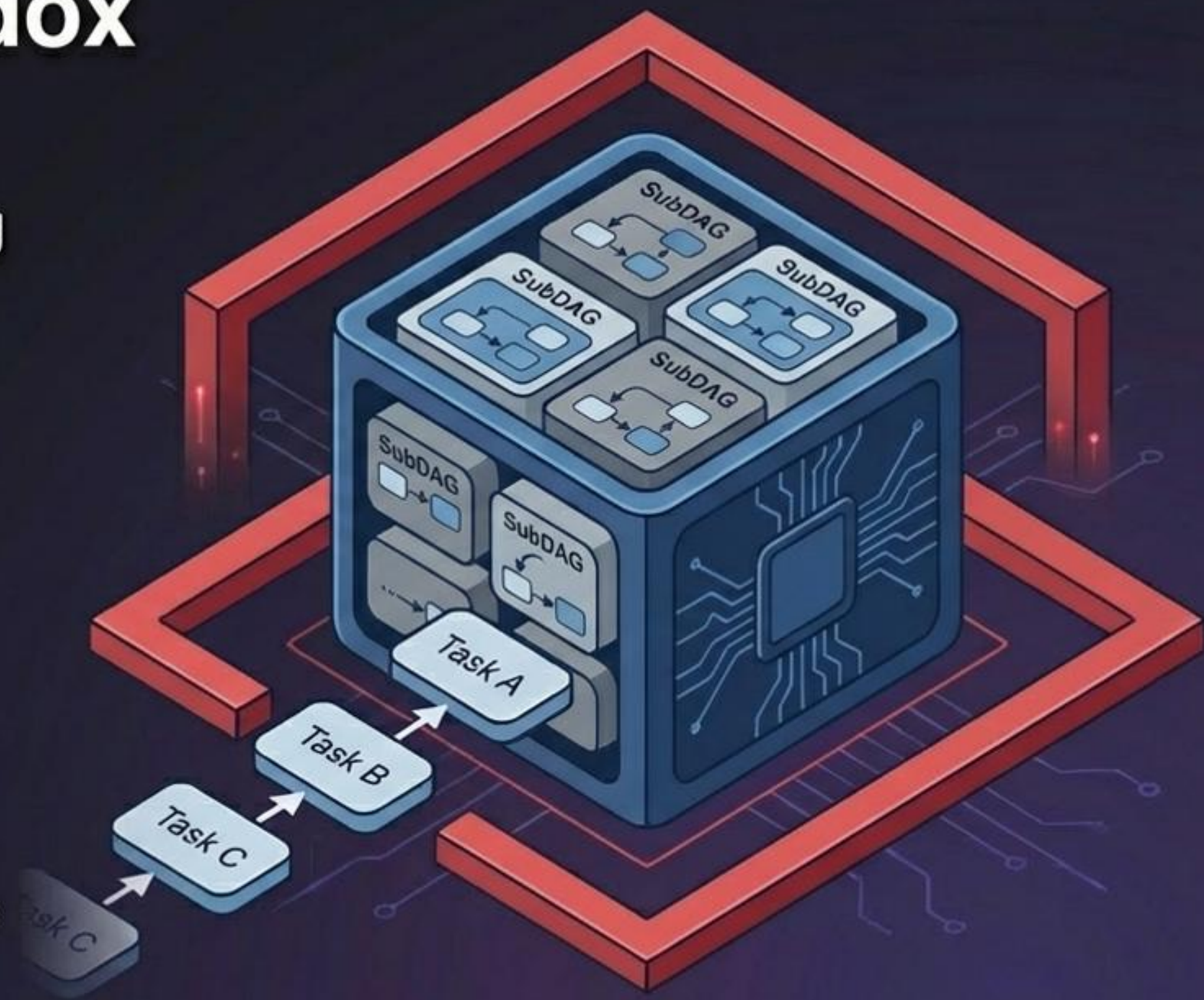
**The Problem:**  
Giant DAGs become impossible to read.

**The Solution:**  
Commonly repeated series of tasks are nested into reusable SubDAGs.



# The SubDAG Paradox

- **Worker Slot Hijacking:** SubDAGs consume worker slots while waiting for their internal tasks to execute, which can lead to rapid deadlocks.
- **Execution Limitations:** They can force workflows into sequential executor limitations, destroying parallelism.
- **Modern Alternative:** The community often relies on Custom Operators or external trigger DAGs to avoid SubDAG traps.



# The Control Tower: SLAs & Telemetry

## Airflow Native Monitoring

Service Level Agreements (SLA): Define strict timeframes by which a DAG must complete. Airflow logs misses in a dedicated UI dashboard.



## Enterprise Telemetry

Enterprise Telemetry: Publish internal metrics to statsd and visualize with Grafana to build holistic alert systems.



# Synthesis: Anatomy of a Production-Grade DAG

```
from airflow.operators.bash import BashOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from custom.operators.has_rows import HasRowsOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'production_dag_s3_to_redshift',
    default_args=default_args,
    description='A production-grade DAG for ELT',
    schedule_interval='@daily', # ← Callout 1
    max_active_runs=1,
    start_date=datetime(2023, 1, 1),
    catchup=False,
    tags=['production'],
)

with dag:
    create_trips_table = PostgresOperator(
        task_id='create_trips_table',
        postgres_conn_id='redshift_default',
        sql='sql/create_trips_table.sql',
    )
    load_trips_from_s3 = BashOperator(
        task_id='load_trips_from_s3_to_redshift', # ← Callout 2
        bash_command='aws s3 cp s3://my-bucket/trips.csv /tmp/trips.csv && psql ...',
    )
    load_stations_from_s3 = BashOperator(
        task_id='load_stations_from_s3_to_redshift',
        bash_command='aws s3 cp s3://my-bucket/stations.csv /tmp/stations.csv && psql ...', # ← Callout 3
    )
    calculate_traffic = PostgresOperator(
        task_id='calculate_location_traffic',
        postgres_conn_id='redshift_default',
        sql='sql/calculate_traffic.sql',
    )

    hook = PostgresHook(postgres_conn_id='redshift_default') # ← Callout 4
    hook.run("DELETE FROM staging WHERE date < {{ execution_date }}")

    # Dependencies
    create_trips_table >> load_trips_from_s3 >> check_trips_data
    create_stations_table >> load_stations_from_s3 >> check_stations_data
    [check_trips_data, check_stations_data] >> calculate_traffic # ← Callout 5
```

1 @daily schedule interval with max\_active\_runs protection.

create\_trips\_table

load\_trips\_from\_s3\_to\_redshift

check\_trips\_data

calculate\_location\_traffic

create\_stations\_table

load\_stations\_from\_s3\_to\_redshift

check\_stations\_data

2 Atomic BashOperator extracting raw S3 data.

3 Custom HasRowsOperator performing Data Quality validation.

4 PostgresHook utilizing Jinja templated variables.

5 Clean Bitshift syntax (>>) enforcing strict task dependencies.

3 CustomBashOperator (>>) extracting raw S3 data.