May 23–27, 2022

AIRFLOW SUMMIT

Airflow Summit 2022

Apache Airflow at Scale

John Jackson
Principal Product Manager
Amazon Web Services (AWS)

What We'll Cover Today

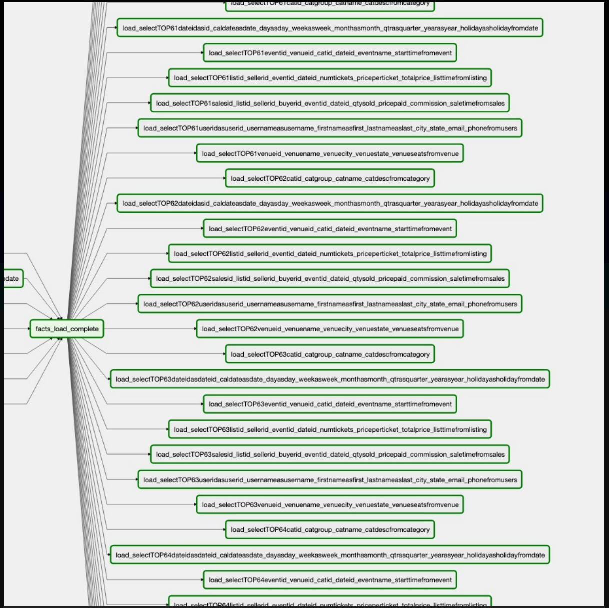
- Introduction
- What is Apache Airflow at Scale
    - Understanding Considerations
    - Scheduler Loops and Configurations
- Scaling Workloads
    - Containers
    - Pools and priority
- Scaling DAGs
    - Dynamic DAGs/DAG Factories
    - CI/CD
    - DAG Access Control
- Multiple Environments
    - How to split up workloads (users/downstream access/priority)
    - Central Governance: Creation and Monitoring
    - Example: Distributing workloads across Airflow clusters
- Q&A

Introduction

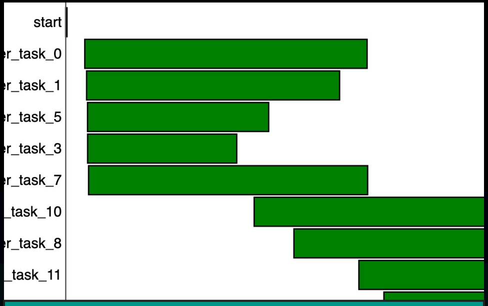
John Jackson

- Product Manager for Amazon Managed Workflows for Apache Airflow (MWAA)
- 2+ years with Amazon Web Services
- Part of the Airflow Summit 2022 Organizing Committee
- Software Developer/Solution Architect/Product Manager for over 25 years
- Based in Vancouver, Canada
- https://github.com/john-jac

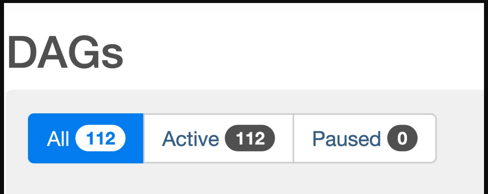
What is Apache Airflow at Scale?



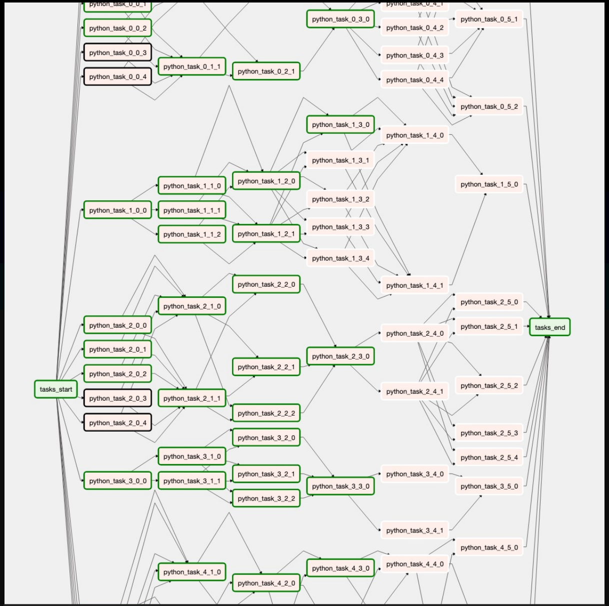
Large number of concurrent tasks



Long running tasks



Large number of workflow definitions



Large or complicated workflows

Considerations

What things affect your ability to scale

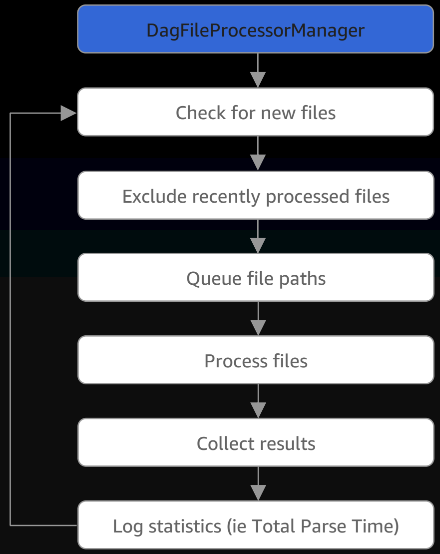
DAGs are parsed continuously, whether active or not

DAG objects are analyzed by the Scheduler to see which tasks should be queued next

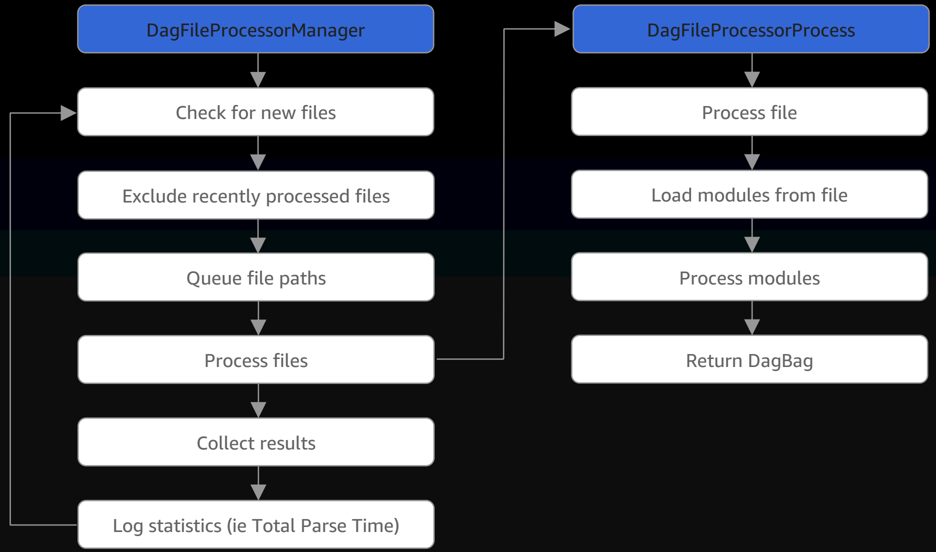
Typically there is a fixed amount of compute for these operations (number of Schedulers, Workers, Web Servers, plus DB size, network capacity, etc.)

Airflow Summit 2022

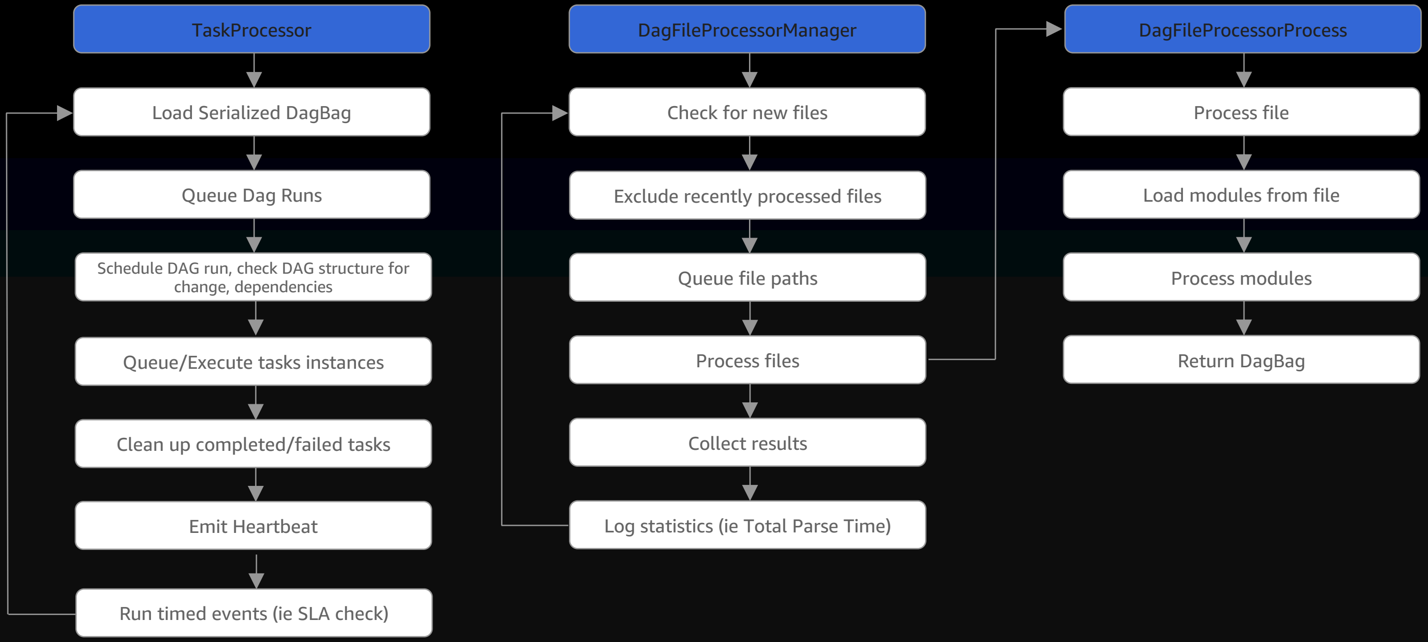
Scheduler Loops and Configurations



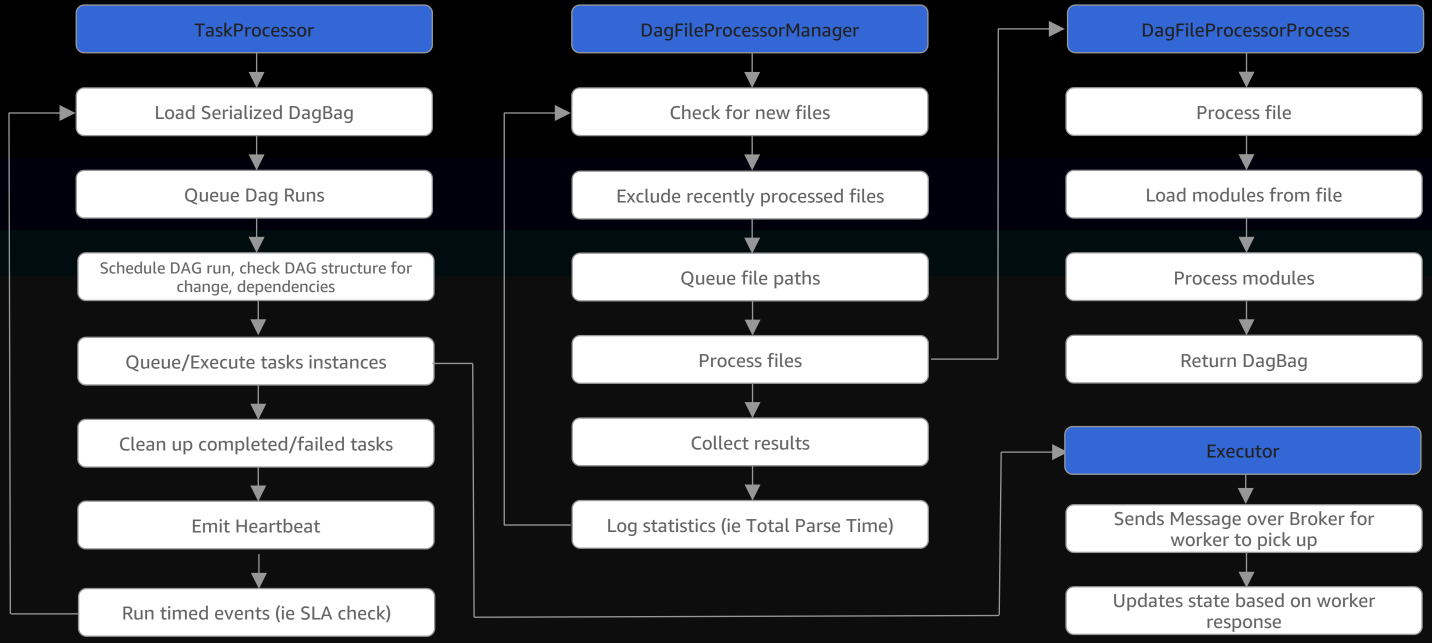
Scheduler Loops and Configurations



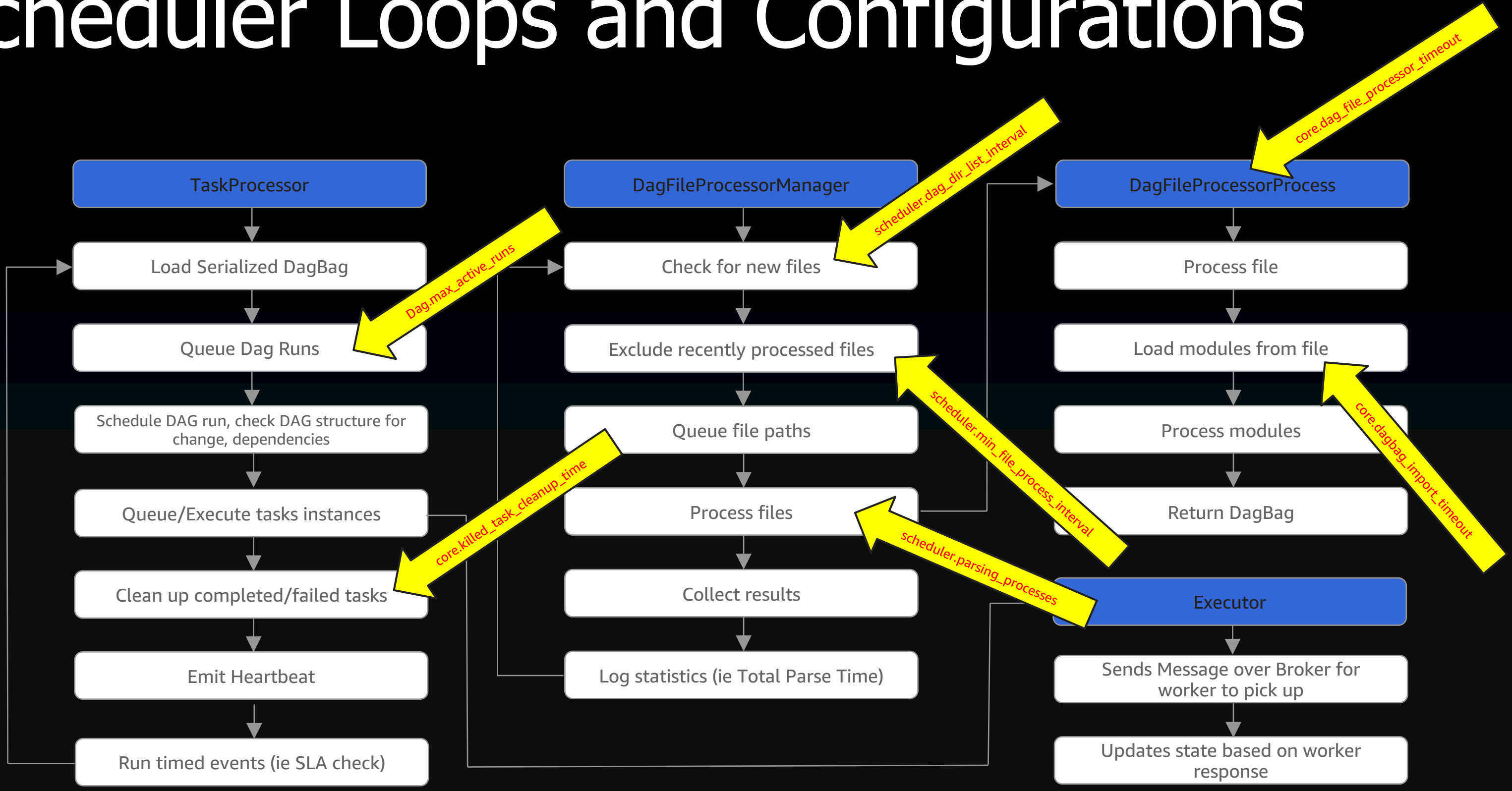
Scheduler Loops and Configurations



Scheduler Loops and Configurations



Scheduler Loops and Configurations



Configuration Options

Some key options when running at scale

dag_dir_list_interval - How often to scan the DAGs directory for new files. Default is 5 minutes (300 seconds).

min_file_process_interval - Number of seconds after which a DAG file is re-parsed. The DAG file is parsed every min_file_process_interval number of seconds. Default is 30 seconds.

parsing_processes - The scheduler can run multiple processes in parallel to parse DAG files. This defines how many processes will run. Default is 2.

dag_file_processor_timeout - How long before timing out the processing of a dag file. Default is 50 seconds.

dagbag_import_timeout - How long before timing out a python file import. Default is 30 seconds.

https://airflow.apache.org/docs/apache-airflow/stable/concepts/scheduler.html#scheduler-ha-tunables

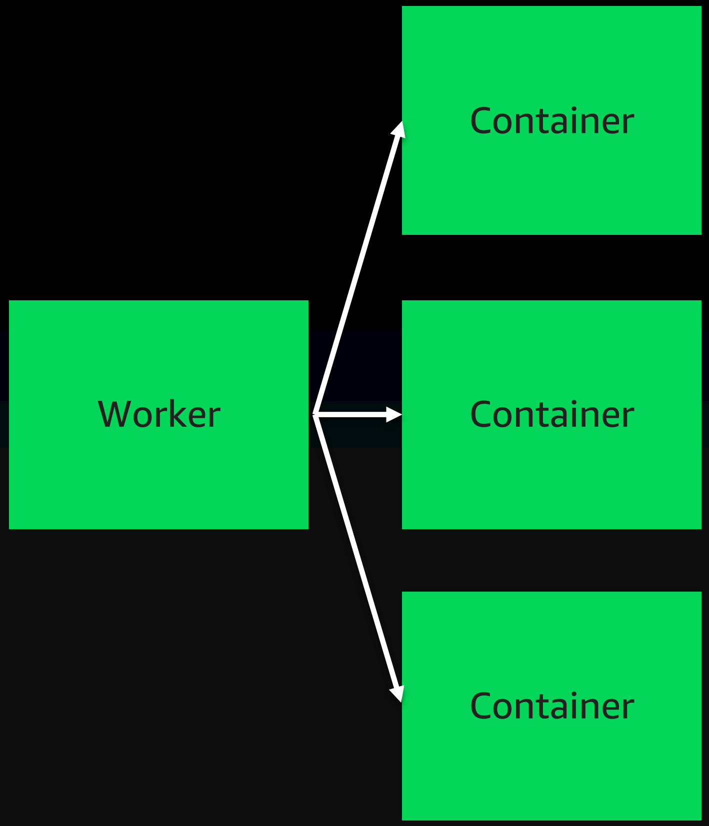
Scaling Workloads

Getting the most work out of your Airflow cluster

Containers

Offloading work

- Using Kubernetes, Docker, ECS, EKS, EMR, Batch, etc Operators
- Using Airflow as an Orchestrator, not for doing the actual processing
- Similar philosophy around ETL—use Airflow to orchestrate the overall ETL set of jobs, but use a dedicated ETL service (Spark, Kafka, Hive) to execute the actual ETL or ELT (Snowflake, Redshift, other analytics databases) to perform the transform after load



Pools, Priority, and Parallelism

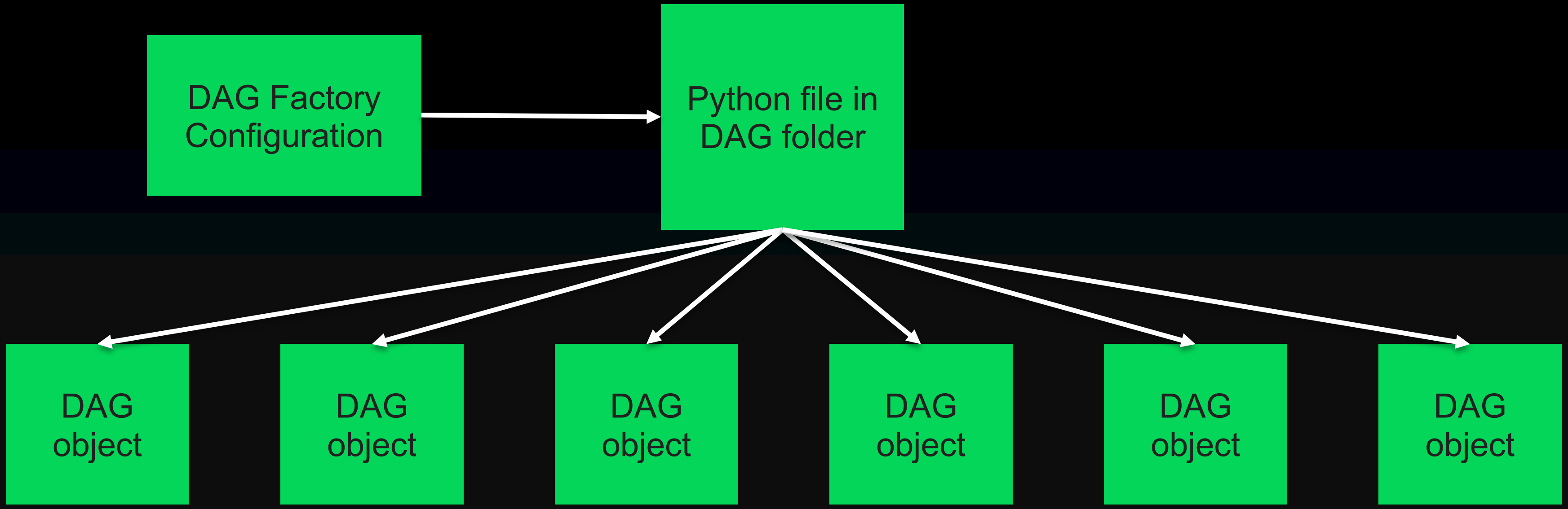
Control which tasks run, when, and how many

- Airflow **pools** can be used to limit the execution parallelism on arbitrary sets of tasks. Typically this is done to limit downstream impact, for example putting all database tasks in an "RDS" pool that has a limit based upon the connection limit of the DB
- The **priority_weight** of a task defines priorities in the executor queue. In a given pool, as slots free up, queued tasks start running based on the Priority Weights of the task and its descendants.
- **Parallelism** at the system level defines the maximum number of task instances that can run concurrently in Airflow regardless of scheduler count and worker count. Generally, this value is reflective of the number of task instances with the running state in the metadata database.
- **Concurrency** is the maximum number of task instances allowed to run concurrently in each DAG, and is configurable at the DAG level with *max_active_tasks*, which is defaulted as *max_active_tasks_per_dag*.
- A **deferrable operator** is able to suspend itself and free up the worker when it knows it has to wait, and hand off the job of resuming it to d a Trigger

Scaling DAGs

More workflows, less code

Dynamic DAGs/DAG Factories



Airflow Summit 2022

Dynamic DAG Example 1

```
@dag(
        dag_id=f"{DAG_ID}_listing",
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_listing():
        t = PostgresOperator(task_id="query_listing",
                sql="select * from listing;",
                postgres_conn_id=POSTGRES_CONN_ID)
update_table_listing_dag = update_table_listing()

@dag(
        dag_id=f"{DAG_ID}_sales",
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_sales():
        t = PostgresOperator(task_id="query_sales",
                sql="select * from sales;",
                postgres_conn_id=POSTGRES_CONN_ID)
update_table_sales_dag = update_table_sales()

@dag(
        dag_id=f"{DAG_ID}_accounts",
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_accounts():
        t = PostgresOperator(task_id="query_accounts",
                sql="select * from accounts;",
                postgres_conn_id=POSTGRES_CONN_ID)
update_table_accounts_dag = update_table_accounts()
```

Dynamic DAG Example 1

```python
@dag(
        dag_id=f"{DAG_ID}_listing",
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_listing():
        t = PostgresOperator(task_id="query_listing",
                sql="select * from listing;",
                postgres_conn_id=POSTGRES_CONN_ID)
update_table_listing_dag = update_table_listing()

@dag(
        dag_id=f"{DAG_ID}_sales",
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_sales():
        t = PostgresOperator(task_id="query_sales",
                sql="select * from sales;",
                postgres_conn_id=POSTGRES_CONN_ID)
update_table_sales_dag = update_table_sales()

@dag(
        dag_id=f"{DAG_ID}_accounts",
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_accounts():
        t = PostgresOperator(task_id="query_accounts",
                sql="select * from accounts;",
                postgres_conn_id=POSTGRES_CONN_ID)
update_table_accounts_dag = update_table_accounts()
```

```python
def get_sources():
        pg_request = "SELECT * FROM information_schema.tables \
                WHERE table_schema = 'public'"
        pg_hook = PostgresHook(postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
        connection = pg_hook.get_conn()
        cursor = connection.cursor()
        cursor.execute(pg_request)
        sources = cursor.fetchall()
        for source in sources:
                print("Source: {0}".format(source))
        return sources
SOURCES = get_sources()
```

```python
for source in SOURCES:
        dag_id=f"{DAG_ID}_{source[2]}"
        @dag(
                dag_id=dag_id,
                schedule_interval="@hourly",
                start_date=datetime(2022, 1, 1),
                catchup=False,
        )
        def update_table_dag(sql=""):
                t = PostgresOperator(task_id="query_table",
                        sql=f"select * from {source[2]}",
                        postgres_conn_id=POSTGRES_CONN_ID)
        globals()[dag_id] = update_table_dag()
```

Dynamic DAG Example 2

Less Parsing Overhead

```
TABLE_LIST_FILE_PATH="/usr/local/airflow/dags"

@task()
def get_sources():
        pg_request = "SELECT * FROM information_schema.tables \
                WHERE table_schema = 'public'"
        pg_hook = PostgresHook(
                postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
        connection = pg_hook.get_conn()
        cursor = connection.cursor()
        cursor.execute(pg_request)
        sources = cursor.fetchall()
        jsonStr = json.dumps(sources)
        with open(TABLE_LIST_FILE_PATH, 'w') as f:
                f.write(jsonStr)

        return sources

@dag(
        dag_id=f"{DAG_ID}_get_sources",
        schedule_interval="55 * * * *",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_list():
        t1 = get_sources()
update_table_list_dag = update_table_list()
```

```
with open(TABLE_LIST_FILE_PATH) as f:
        jsonStr = f.readlines()
        sources = json.loads(jsonStr)
        for source in sources:
                dag_id=f"{DAG_ID}_{source[2]}"
                @dag(
                        dag_id=dag_id,
                        schedule_interval="0 * * * *",
                        start_date=datetime(2022, 1, 1),
                        catchup=False,
                )
                def update_table_dag(sql=""):
                        t = PostgresOperator(task_id="query_table",
                                sql=f"select * from {source[2]}",
                                postgres_conn_id=POSTGRES_CONN_ID)
                globals()[dag_id] = update_table_dag()
```

Dynamic DAG Example 2

Less Parsing Overhead

```
TABLE_LIST_FILE_PATH="/usr/local/airflow/dags"

@task()
def get_sources():
        pg_request = "SELECT * FROM information_schema.tables \
                WHERE table_schema = 'public'"
        pg_hook = PostgresHook(
                postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
        connection = pg_hook.get_conn()
        cursor = connection.cursor()
        cursor.execute(pg_request)
        sources = cursor.fetchall()
        jsonStr = json.dumps(sources)
        with open(TABLE_LIST_FILE_PATH, 'w') as f:
                f.write(jsonStr)

        return sources

@dag(
        dag_id=f"{DAG_ID}_get_sources",
        schedule_interval="55 * * * *",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_list():
        t1 = get_sources()
update_table_list_dag = update_table_list()
```

```
with open(TABLE_LIST_FILE_PATH) as f:
        jsonStr = f.readlines()
        sources = json.loads(jsonStr)
        for source in sources:
                dag_id=f"{DAG_ID}_{source[2]}"
                @dag(
                        dag_id=dag_id,
                        schedule_interval="0 * * * *",
                        start_date=datetime(2022, 1, 1),
                        catchup=False,
                )
                def update_table_dag(sql=""):
                        t = PostgresOperator(task_id="query_table",
                                sql=f"select * from {source[2]}",
                                postgres_conn_id=POSTGRES_CONN_ID)
                globals()[dag_id] = update_table_dag()
```

Dynamic DAG Example 2

Less Parsing Overhead

```
TABLE_LIST_FILE_PATH="/usr/local/airflow/dags"

@task()
def get_sources():
        pg_request = "SELECT * FROM information_schema.tables \
                WHERE table_schema = 'public'"
        pg_hook = PostgresHook(
                postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
        connection = pg_hook.get_conn()
        cursor = connection.cursor()
        cursor.execute(pg_request)
        sources = cursor.fetchall()
        jsonStr = json.dumps(sources)
        with open(TABLE_LIST_FILE_PATH, 'w') as f:
                f.write(jsonStr)

        return sources

@dag(
        dag_id=f"{DAG_ID}_get_sources",
        schedule_interval="55 * * * *",
        start_date=datetime(2022, 1, 1),
        catchup=False,
)
def update_table_list():
        t1 = get_sources()
update_table_list_dag = update_table_list()
```

```
with open(TABLE_LIST_FILE_PATH) as f:
        jsonStr = f.readlines()
        sources = json.loads(jsonStr)
        for source in sources:
                dag_id=f"{DAG_ID}_{source[2]}"
                @dag(
                        dag_id=dag_id,
                        schedule_interval="0 * * * *",
                        start_date=datetime(2022, 1, 1),
                        catchup=False,
                )
                def update_table_dag(sql=""):
                        t = PostgresOperator(task_id="query_table",
                                sql=f"select * from {source[2]}",
                                postgres_conn_id=POSTGRES_CONN_ID)
                globals()[dag_id] = update_table_dag()
```

CI/CD

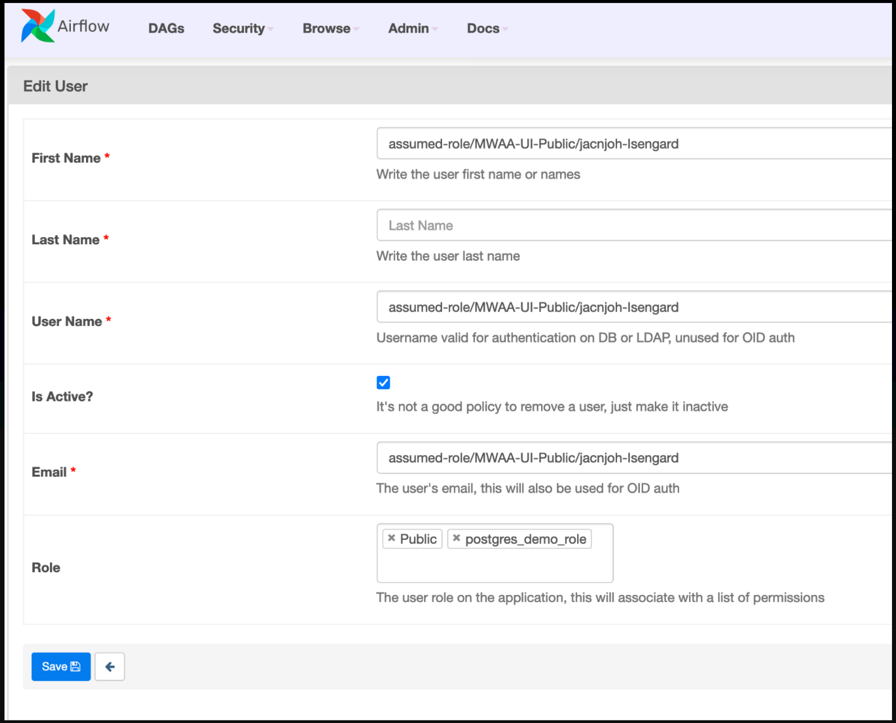
Continuous Integration and Deployment

- Add controls to verify integrity, scope, and usage of DAGs before deploying
- Update configurations
- Test using staging environment
- Automate anything that has to be done more than once

DAG Access Control

"Cooperative Multitenancy"

- Using multiple RBAC roles
- Airflow is not (yet) multi-tenant (AIP-1)
- DAG Factory can limit what a DAG can do
- Multiple clusters may be a better alternative for true multitenancy



Multiple Airflow Clusters

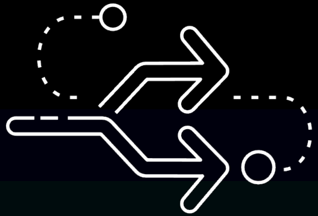
Level-up your Isolation and Resilience

Splitting Workloads Across Environments

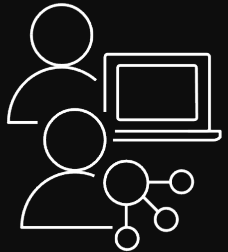
How to decide what runs where

Stage

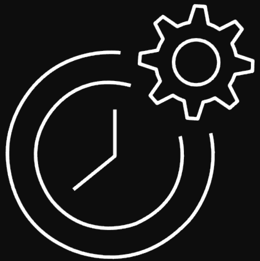
Airflow Downstream Access



Cross-environment



User Groups/Teams



Workload Priority/Schedule/Pattern

Centralized Creation and Monitoring

Create

Terraform, Helm, CloudFormation, GitLab, Kubernetes, CDK, Docker Compose, …

Log

Datadog, S3, Prometheus, CloudWatch, …

Monitor

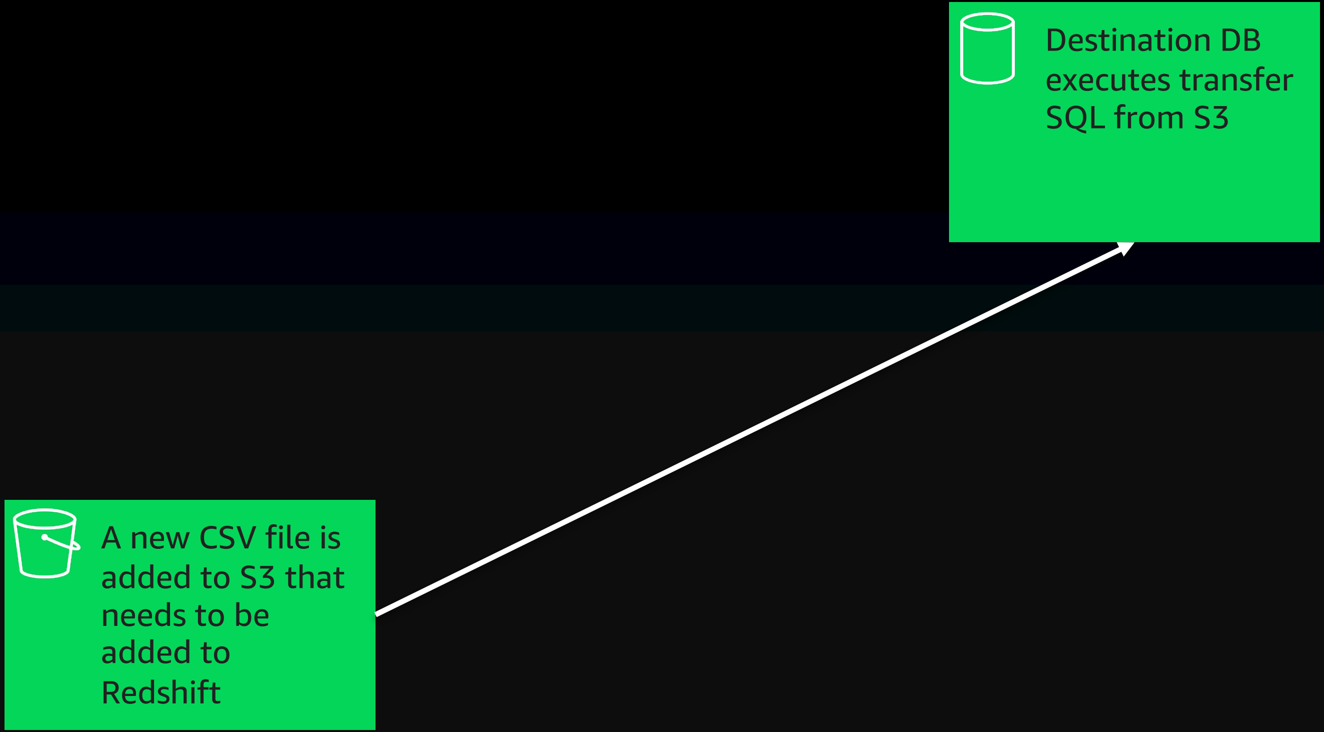
StatsD, Grafana, CloudWatch, DataDog, …

Alarm

SLAs, Callbacks, Email, Prometheus, EventBridge, …

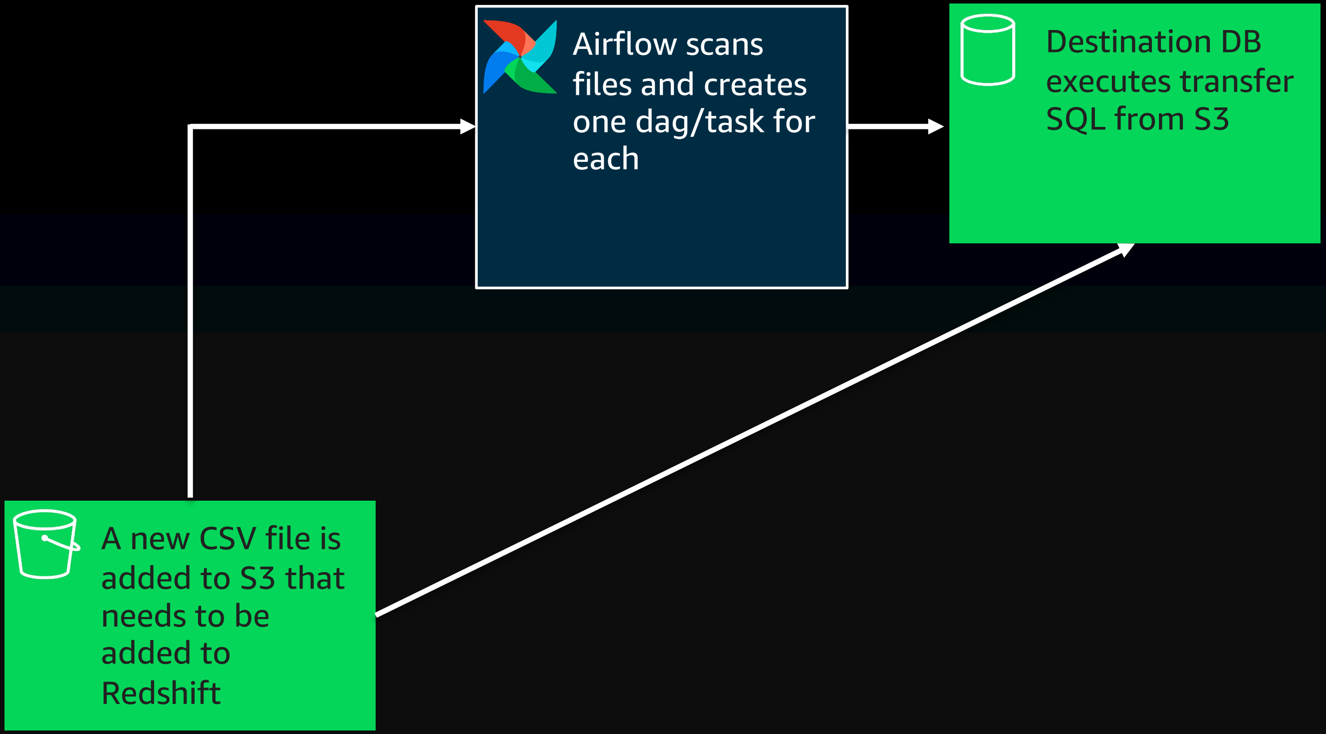
Airflow
Summit 2022

Distributing Dags Cross-Environment

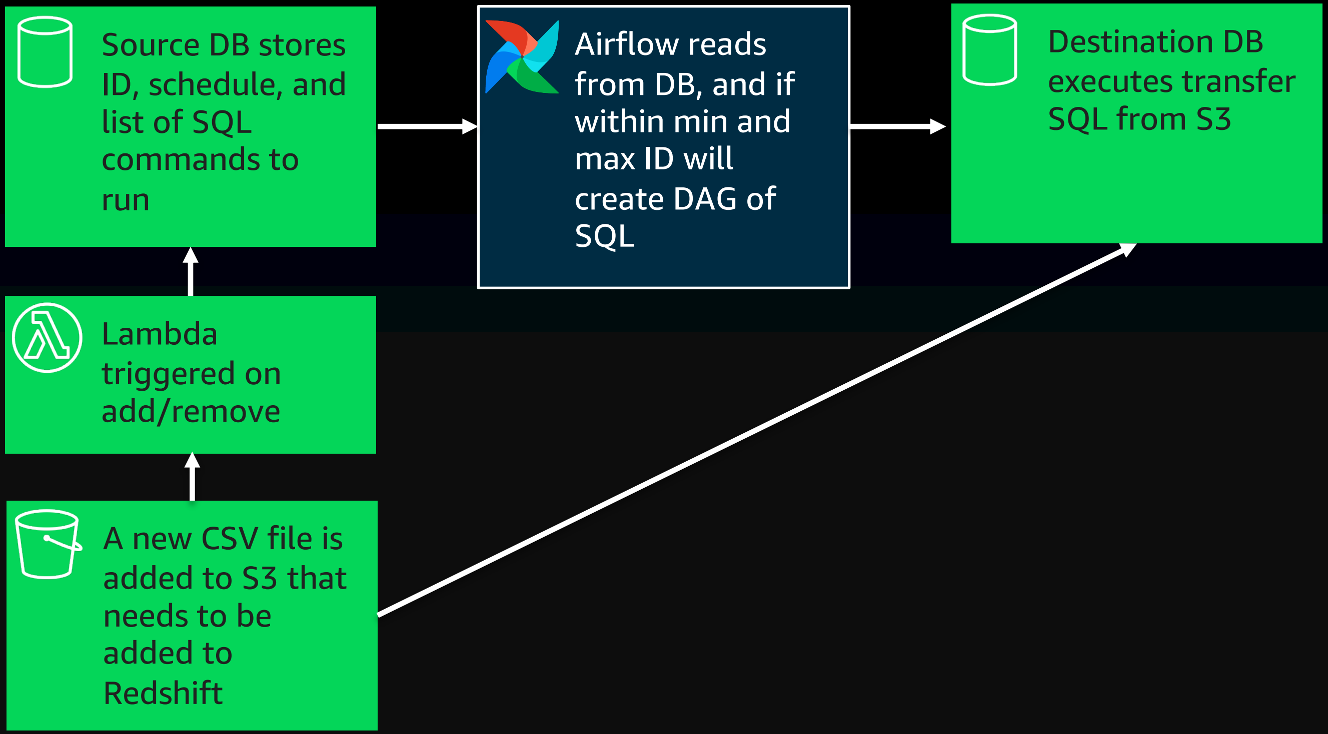


Airflow Summit 2022

Distributing Dags Cross-Environment

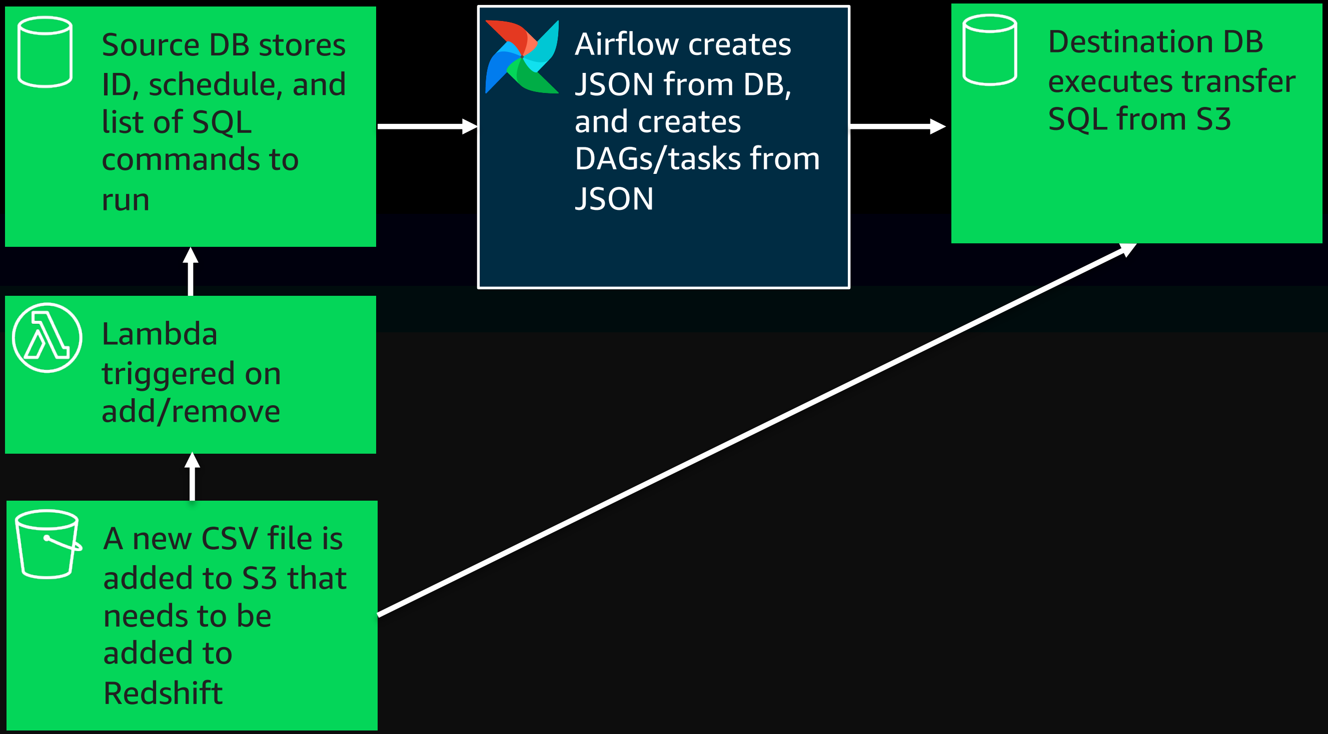


Distributing Dags Cross-Environment



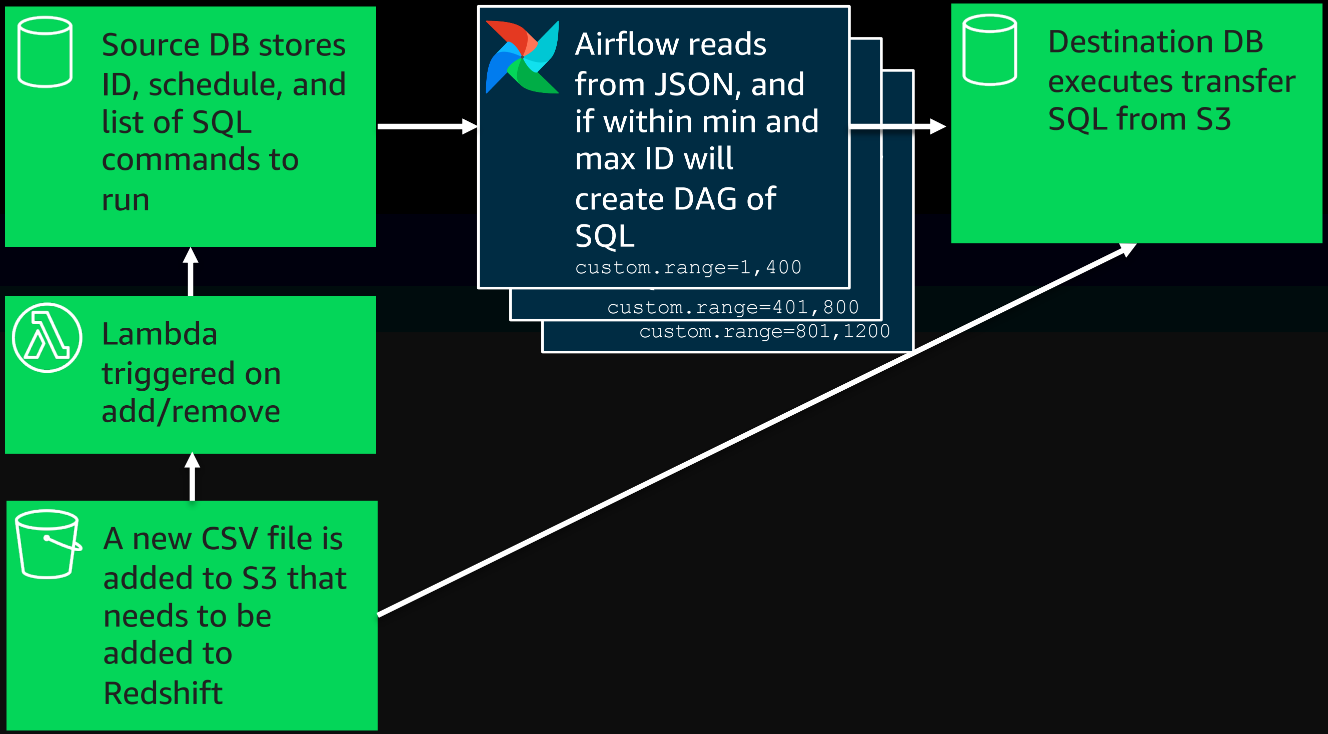
Airflow Summit 2022

Distributing Dags Cross-Environment



Airflow Summit 2022

Distributing Dags Cross-Environment



Dynamic DAG Example 3

Cross-environment

```
with open(TABLE_LIST_FILE_PATH) as f:
        jsonStr = f.readlines()
        sources = json.loads(jsonStr)

        range = os.getenv('AIRFLOW__CUSTOM__RANGE',default='0,0').split(',')
        min = int(range[0])
        max = int(range[1])

        for i in range(min,max+1):
                source = sources[i]
                dag_id=f"{DAG_ID}_{source[2]}"
                @dag(
                        dag_id=dag_id,
                        schedule_interval=”0 * * * *",
                        start_date=datetime(2022, 1, 1),
                        catchup=False,
                )
                def update_table_dag(sql=""):
                        t = PostgresOperator(task_id="query_table",
                                sql=f"select * from {source[2]}",
                                postgres_conn_id=POSTGRES_CONN_ID)
                globals()[dag_id] = update_table_dag()
```

Resources

For more information

- Airflow Slack Group: https://apache-airflow.slack.com/
- Airflow Documentation: https://airflow.apache.org/docs/apache-airflow/stable/index.html
- Airflow GitHub: https://github.com/apache/airflow
- AWS Blogs: https://aws.amazon.com/managed-workflows-for-apache-airflow/resources/

Thank you!

@JohnJacksonPM