DBUG: Using Airflow to build Business solutions

sahaj software | Inspiring Brilliance

The Use Case

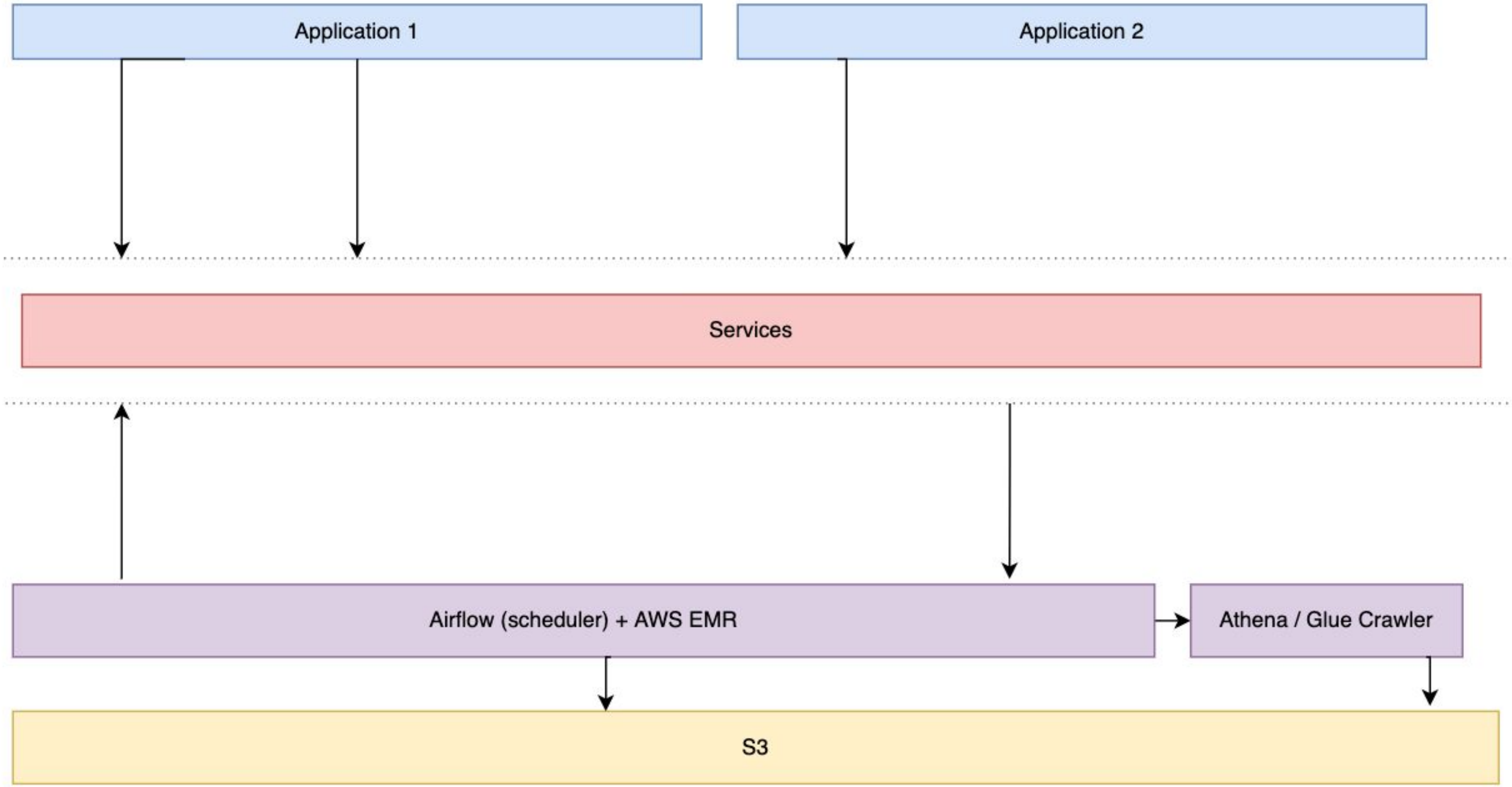
An End-to-End Data Platform, with Airflow at its core.

- Receive raw data from multiple sources
- Run several data science models on the datasets
- Prepare insights and serve them to users on-demand
- 200+ TB of data (per region)

Why Airflow?

- Support for Spark and S3
- Extensible with custom operators and plugins
- Stable
- Strong community support

Simplified System Architecture



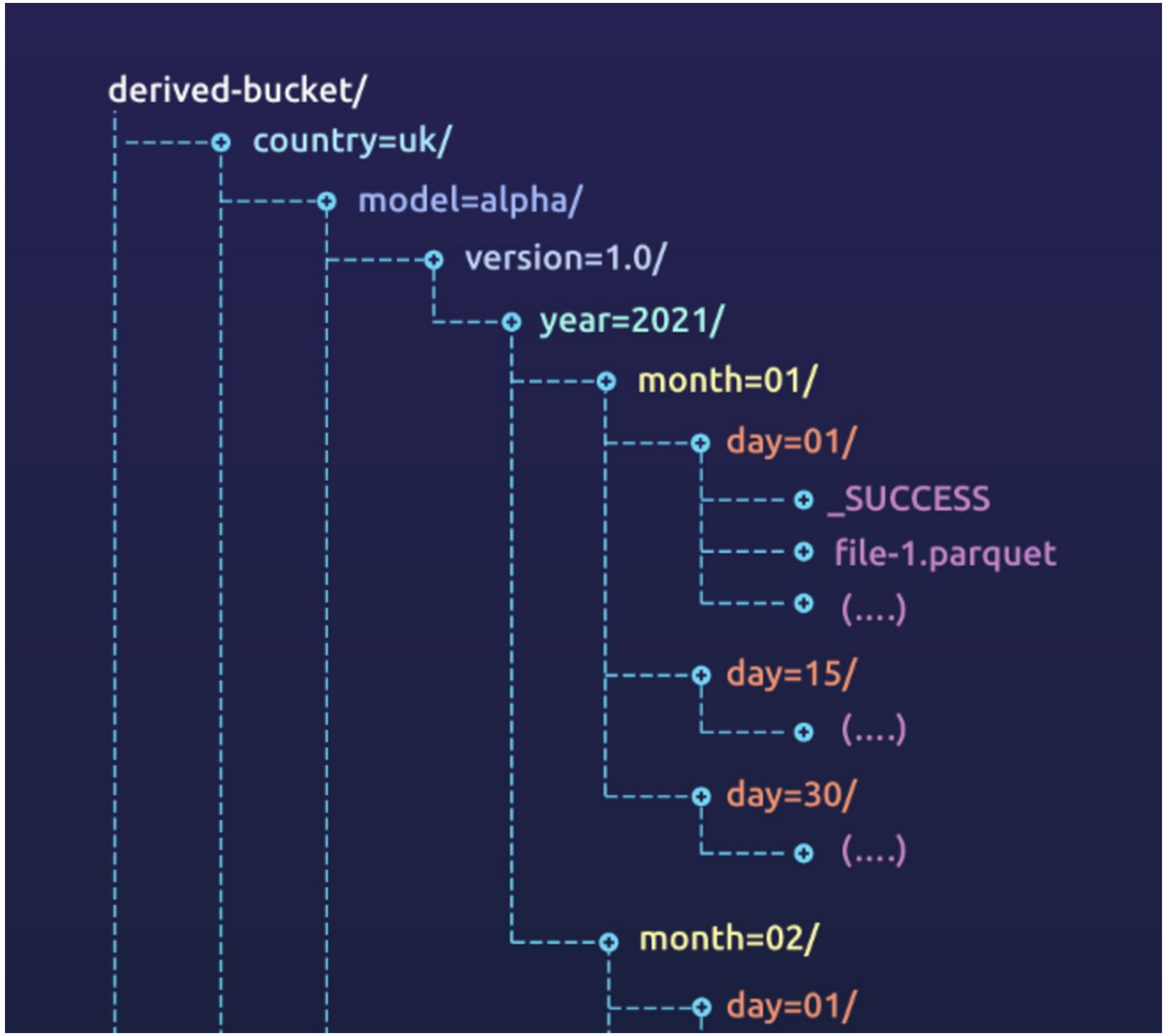
Design Principles

Separate upstream data from any insights we process and produce

https://medium.com/inspiredbrilliance/data-storage-patterns-versioning-and-partitions-a8ce1fd82765

Design Principles

Versioned model outputs



https://medium.com/inspiredbrilliance/data-storage-patterns-versioning-and-partitions-a8ce1fd82765

Design Principles

Amazon EMR

One cluster per job

Spawn clusters when required. Terminate when done. Helps isolate resource constraint problems.

AWS Spot instances

Over-Provision clusters and use Spot instances for all nodes.

Job Failure alerts

Monitor the job once triggered, and raise an alert on failure.

Product Evolution

Initial stages - Synchronous operator

- Custom code using Airflow plugins
    - Start cluster and send commands
    - Wait for _SUCCESS marker on HDFS (busy waiting)
    - Copy results to S3

Handle frequent failures

- Input data missing
- Expected output data already present
- EMR too slow / expensive to report at runtime
- Need to verify requirements before spinning up a cluster

Solution: Validation steps using AWS APIs in the DAGs

Long-running tasks

- Model complexity and Data volumes $\propto$ Job execution time
- Not safe to kill Airflow while a task is running
- Need to wait for deployment windows

Solution: Split into smaller operators. Change DAG schedules to have higher delays.

(Very) Long-running tasks

- Local executors have a fixed size for task pools
- Busy waiting takes up a slot for the entire duration
- Domino effect - Tasks remain ‘Scheduled’ forever

Solution: Async operators

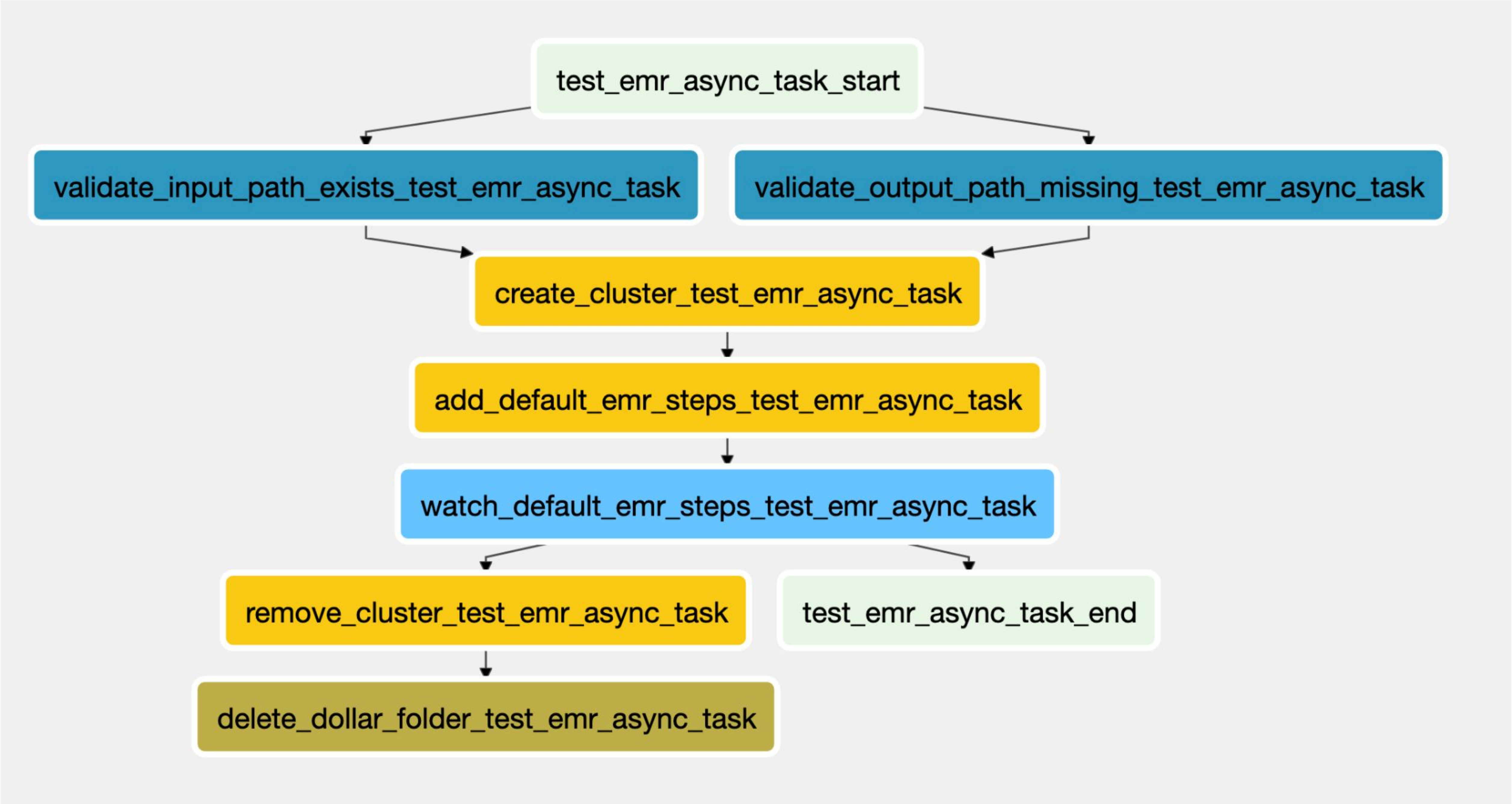
- Move away from custom operators to AWS provided ones
- Operators to start/stop clusters, add steps, watch for step completion
- Sensors: No more busy waiting 🎉

Code quality issues

- 30+ DAGs at this point
- Each DAG is a standalone .py file
- Almost all Spark jobs now follow the same structure
- Duplicate code across DAGs leading to bugs and other issues

Solution: We now have a pattern that works. Build an Abstraction that can be reused everywhere

End structure



EmrTask Abstraction

Specified per DAG

- Job name
- Input / Output / Config argos
- Cluster instance types + node counts
- Optional Spark config overrides
- Max Execution time

Automatically handled

- Prepare all operators with links
- Resolve custom macros in templates
- Choose appropriate validations
- Find Spark artifact and prepare commands
- Setup clusters consistently with debugging and observability

An example

```python
"""
DAG Documentation here
"""
from airflow import DAG
from airflow.models.baseoperator import chain

from lib.model_name.jobs import DAG_MACROS
from utils import tags
from utils.build_dags import EmrTask
from utils.common import dag_id_from_file_name
from utils.dag_helpers import default_args

dag_id = dag_id_from_file_name(full_file_path=__file__)
dag = DAG(
    dag_id=dag_id,
    default_args=default_args(2022, 8, 23),
    tags=[tags.wip, tags.retry_upon_input_failure],
    schedule_interval=None,
    user_defined_macros=EmrTask.get_macros(DAG_MACROS)
)

with dag:
    chain(*spark_job().all)

dag.doc_md = __doc__
```

Testing DAGs

Unit Tests for DAGs

- Verify the order of execution of tasks
- Verify the right args are passed to the Spark jobs
- Verify other operators (Athena crawlers, service updates etc.)
- Verify toggles (Airflow vars) based behaviour
- Verify Jinja templates are evaluated correctly

Unit Tests for DAGs

Verifying Jinja Templates

- Used extensively to identify S3 paths
  - Paths with run date, version number
  - Last ‘n’ previous outputs
  - Latest available dataset (with max age)
  - Paths based on feature toggles

Challenge: Evaluating Jinjas in tests

Unit Tests for DAGs

Evaluating Jinjas in tests

- Need to load a DAG and render the TaskInstance to evaluate the Jinja
- Once rendered, we can retrieve evaluated values of task attributes

Approach:

- Create a DagBag object
- Retrieve DAG from DagBag
- Retrieve and render TaskInstance in DAG

Challenges:

- Slow tests
- Running in parallel
- Varying output with Airflow variables

High level Tests

- Integration tests
  - Verify that all referenced Spark artifacts are available
  - Verify all glue crawlers exist
  - Verify DAGs and tasks referenced by ExternalTaskSensors actually exist
- QoL tests
  - Verify all DAGs have documentation

Current Challenges

Current Challenges

- Failures out of Airflow’s control (Ex: Spot terminations)
- Understand system health
  - Do I have enough data to run this job today?
- Understanding lineage
  - Which models will be affected if I make a change in this one?

Thanks!

Dakshin K
Solution Consultant, Sahaj

+91 7012984445
dakshinamoorthyk@sahaj.ai
Chennai, IN

Bonus - Deployment Strategies