
Apache Airflow 2.3 and beyond:
What comes next?
Ash Berlin-Taylor

Who Am I?

PMC member,
Apache Airflow project

Director of Airflow Engineering,
Astronomer.io

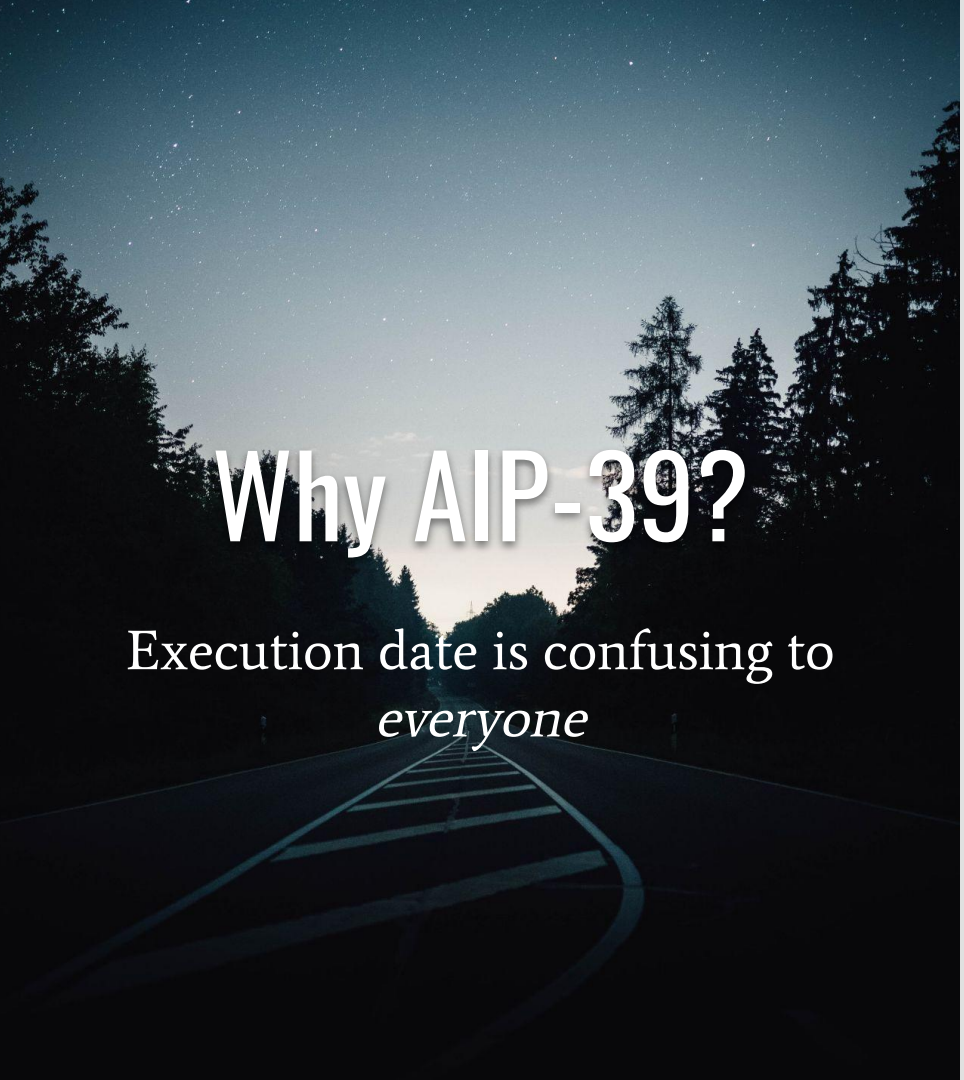

How did we get here?

We build our computer (systems) the way we build our cities: over time, without a plan, on top of ruins

— Ellen Ulman


Airflow 2.2

AIP-39: Run DAGs on customizable schedules



Why AIP-39?

Execution date is confusing to *everyone*

Replace it with a new term all together: "data interval"

Allow overlapping schedule and "data" interval

Fully customizable timetable (user provided class)

👋 execution_date

The concept of "execution_date" was confusing to every new user

So we removed it (well deprecated it)

In its place we now have:

logical_date (aka execution_date)

data_interval_start (same value as execution_date for built in)

data_interval_end (next_execution_date)

AIP-40: Any operator can "defer" itself

Why AIP-40

Reducing resource usage for big clusters

Generalisation of Smart Sensors

Many "cloud" operators follow a "setup ⇒ poll" loop

Wasteful using a whole executor slot

Deferrable Tasks

Allows tasks or sensors to free up worker resources when waiting for external systems/events.

`airflow triggerer`: new daemon process that runs asyncio event loop

```
with DAG(id="process_images",
         timetable=solar.Timetable('dusk_nautical', 'Australia/Melbourne')):
    @task
    def prepare():
        pod_bay.doors.open()

    @task
    def capture_images():
        ...

    @task
    def finalize():
        pod_bay.doors.close()

    prepare() >> solar.TimeSensorAsync('dusk_astronomical') >> capture_images() >> finalize()
```

Roadmap: *A* possible future

Making DAGs a joy to write

Airflow should be the go to orchestrator for *every* data workflow job

Airflow should be easier to operate confidently

Roadmap Concepts

- Making DAGs a joy to write
- Airflow should be the go to orchestrator for every data workflow job
- Airflow should be easier to operate confidently


The near future

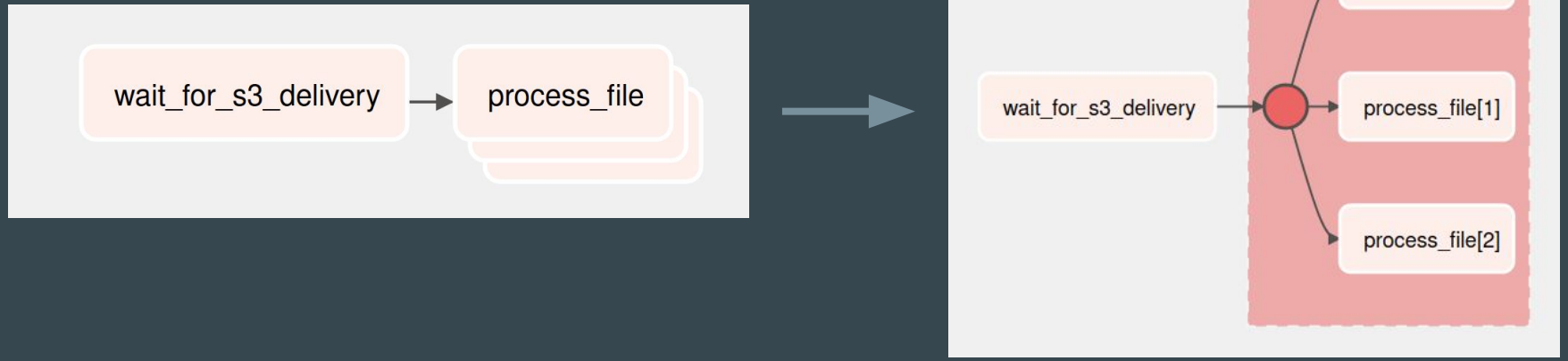

Dynamic DAGs

```python
@task
def get_files_from_s3():
    return [...]

my_files = get_files_from_s3()
s3_delete_files = MyFileProcessOperator.partial(
    aws_conn_id="my-aws-conn-id",
    bucket="my-bucket"
).map(key=my_files)
```

Mapped tasks

- Mapped tasks are a "template" that is expanded Just In Time
- Replaced with *n* new Task Instances
- Can map over: XCom, Variables, or static literals


wait_for_s3_delivery
process_file
process_file
wait_for_s3_delivery
process_file[0]
process_file[1]
process_file[2]

```python
@dag
def my_dag(markets: list[str], campaigns: dict[str, list[int]]):
    @task
    def ingest(market):
        ...
    @task
    def calculate_roi(market, campaign):
        ...
    @task
    def aggregate_rois(market, campaign_rois):
        total = 0
        n = 0
        for campaign_roi in campaign_rois:
            n +=1
            total += campaign_roi
        return campaign_roi/total

    data = ingest.map(markets)
    rois = calculate_roi.map(market, data)
    stats = aggregate_rois(market, rois)
```

`airflowctl`: CLI over the API

Untrusted workers

DAG/task lifecycle hooks and easier notifications

```
task = MyOperator(
    task_id = "something",
    on_failure_callback=send_slack_message(
        channels=['#data-ops'],
        mentions=['@ash'],
    ),
)


@task(on_failure_callback=[send_slack_message(), send_email]
def my_task():
    ...
```


A better cross-DAG story

Event triggered DAGs

New concept: a Data object

```
result = Data("mycompany/vendor_a/summary")

@dag(schedule_interal="@daily")
def summarizer():
    cluster = EmrCreateJobFlowOperator(task_id="create_job_flow",
job_flow_overrides=...)
    EmrRunStepsOperatorAsync(task_id="summarize", job_flow_id=cluster.output,
        steps={
            "Name": "calculate_pi",
            "ActionOnFailure": "CONTINUE",
            "HadoopJarStep": {
                "Jar": "command-runner.jar",
                "Args": ["s3://example-spark-airflow/summarize-table.py",
"{{data_interval_start}}", "{{data_interval_end}}", "{{results.tablename}}"],
            },
        },
        outlets=[result])
dag1 = summarizer()
```

```python
result = DataRef("mycompany/vendor_a/summary")

@dag(schedule_on=result)
def consumer():
    @task
    def get_result(data_obj):
        S3Hook.get_file(data_obj.resolve())

    get_result(result)

dag2 = consumer()
```

Looking further ahead

DAG versioning

Make the UI accurate if DAG structure changes over time

Make the "version" of DAG used for a single DagRun consistent.

Easier DAG deployment

Streaming

Better support for Machine Learning

Of course we're hiring
https://www.astronomer.io/careers