# Modernize a decade old pipeline with Airflow 2.0

Dima Suvorov, Kuntal Basu, Stas Bytsko and QP Hou

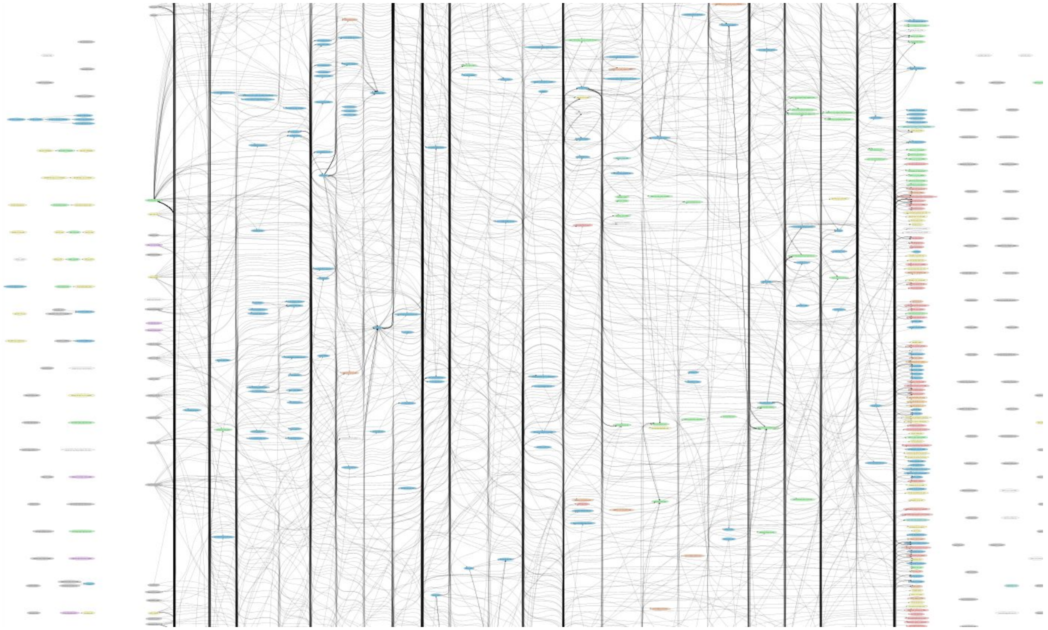
SCRIBD

# Talk Overview

- Migration overview - Stas
- Custom Trigger rules
- Migration to Airflow 2.0 - QP
- Running Backfill at scale - Kuntal
- Self-Service Backfill UI plugin - Dima
- Fixing bugs in Backfill code
- Databricks clusters cost optimization

# Migration overview

- Compute + Storage => ☁
  - AWS & Databricks
- Improve security and compliance
- Custom scheduler -> Airflow
- Mono-DAG
  - 1.4K tasks
  - Nestedness: up to 22 layers deep



https://tech.scribd.com/blog/2020/modernizing-an-old-data-pipeline.html

# Custom Trigger Rules

# Example

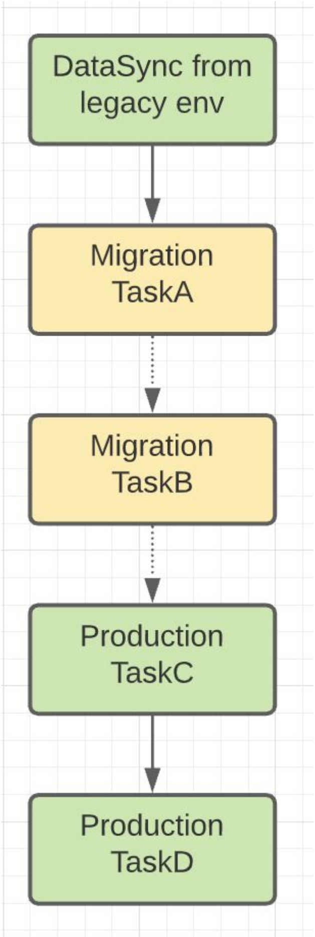
Migrated DAG gradually. DAG served 2 purposes:

- Run Production tasks
- Validate not-yet migrated tasks

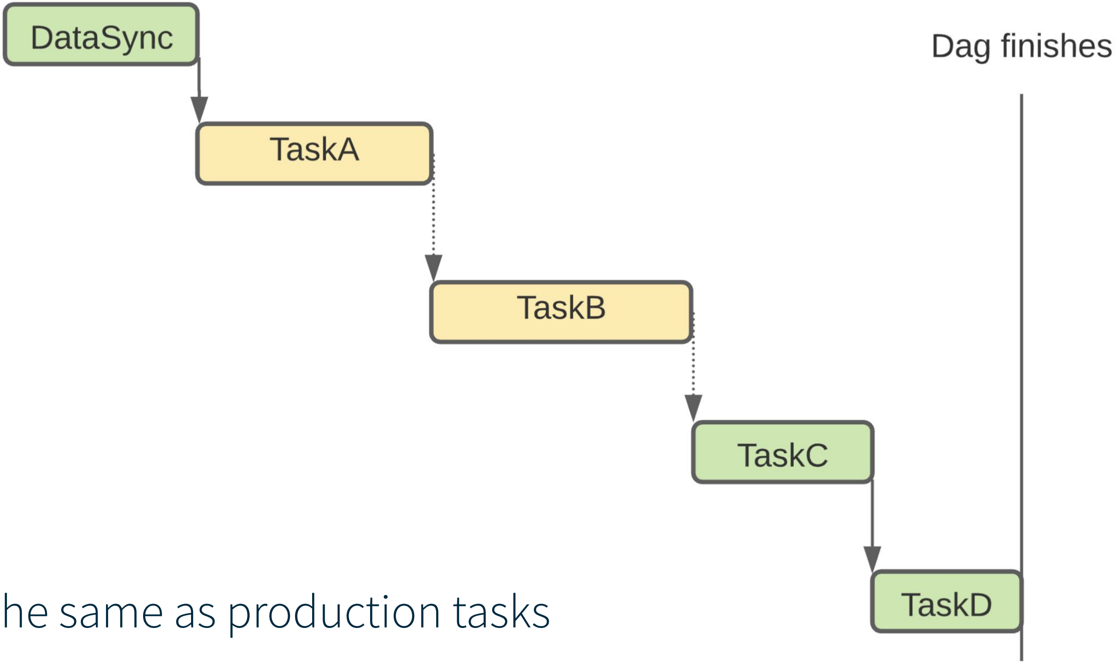
Components:

1. DataSync - All tasks need input data from legacy env
2. Production Operators - generates business value
3. Migration Operators - unreliable, under test
   - Output written to separate database and validated against synced data produced by task in legacy env

Dependencies on Migration Operators - not real, only to simplify migration for teams

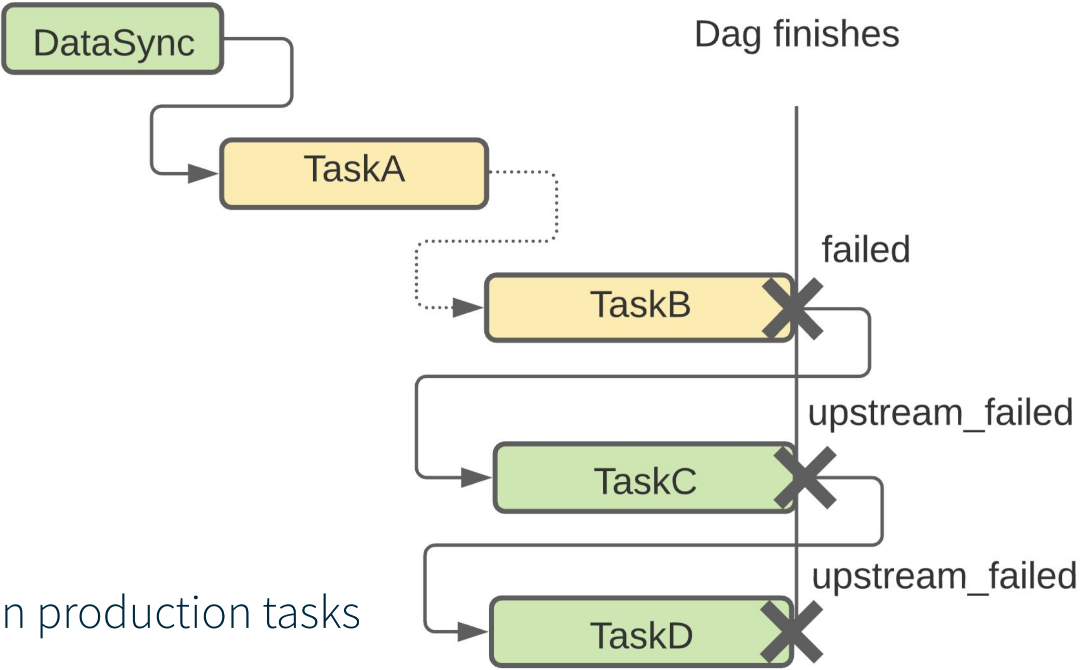


# Naïve approach



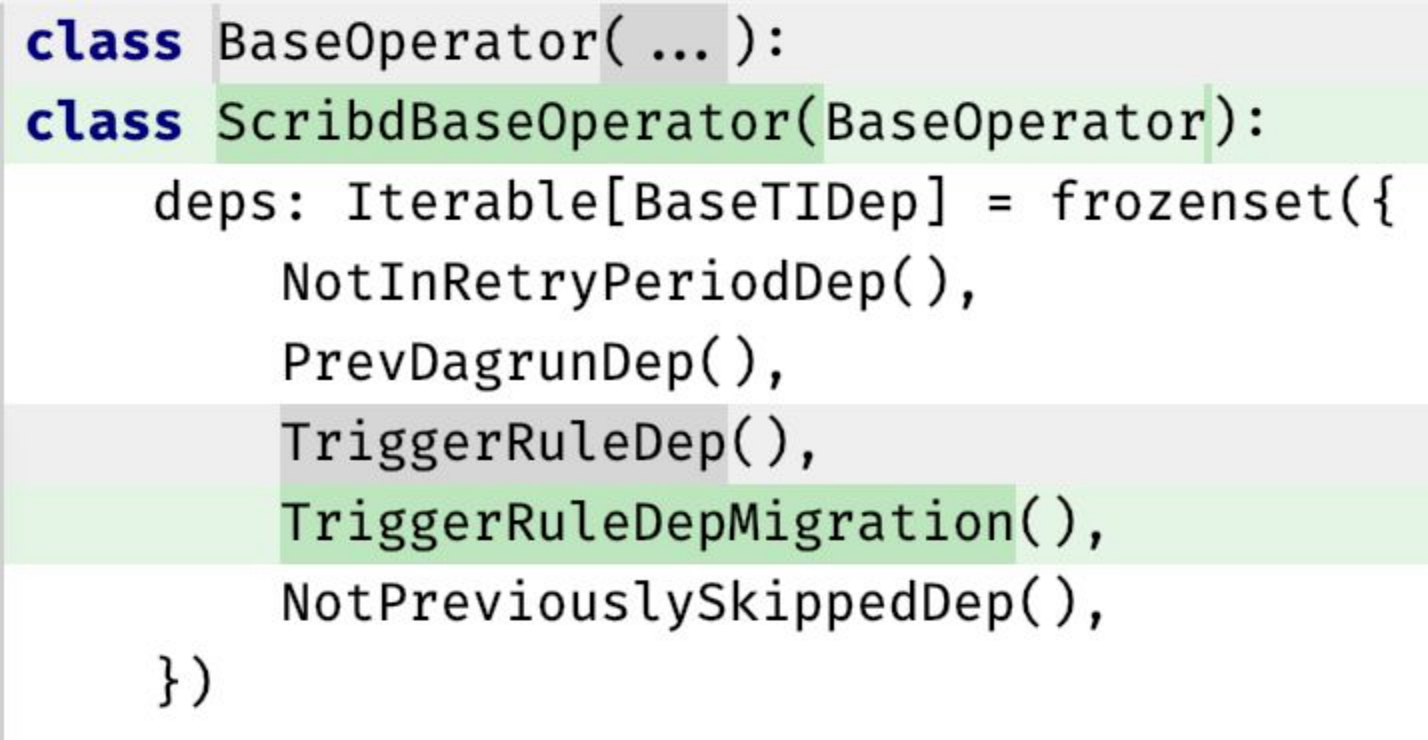
Pretend our migration tasks are the same as production tasks

# Naïve approach problem



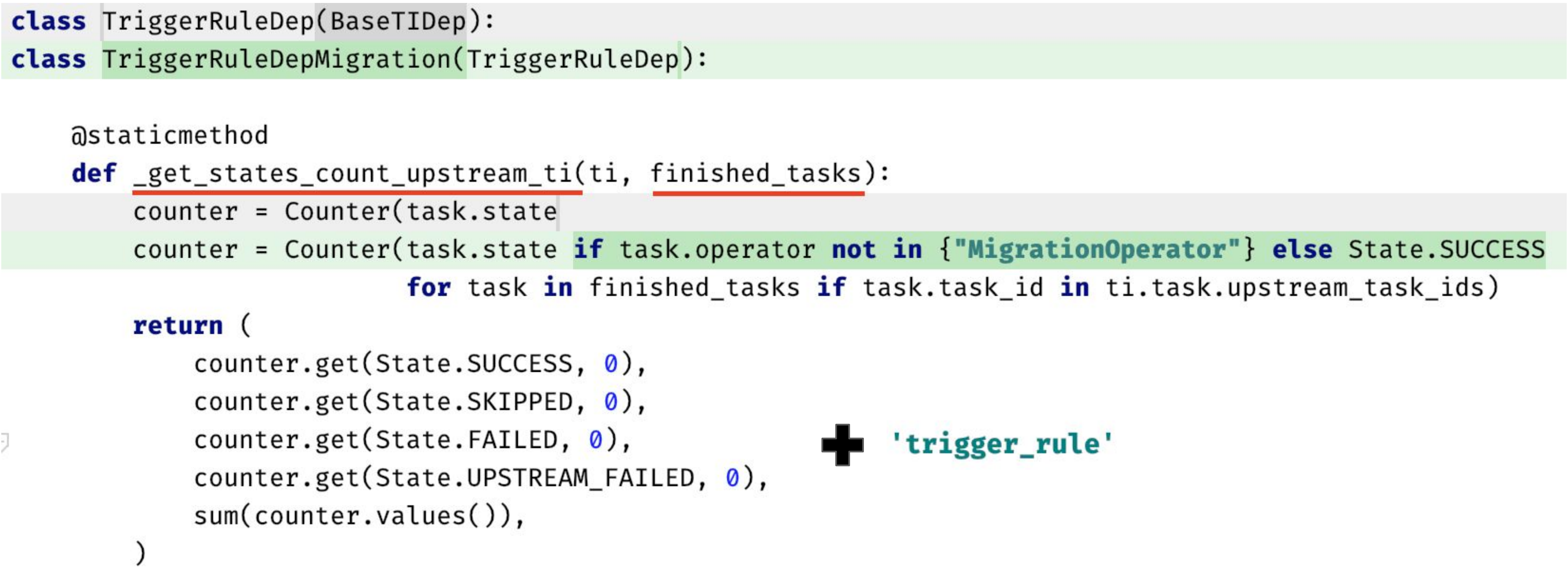
Migration tasks can fail, bringing down production tasks

# Custom TriggerRuleDep



```
class BaseOperator( ... ):
class ScribdBaseOperator(BaseOperator):
    deps: Iterable[BaseTIDep] = frozenset({
        NotInRetryPeriodDep(),
        PrevDagrunDep(),
        TriggerRuleDep(),
        TriggerRuleDepMigration(),
        NotPreviouslySkippedDep(),
    })
```

# Custom TriggerRuleDep

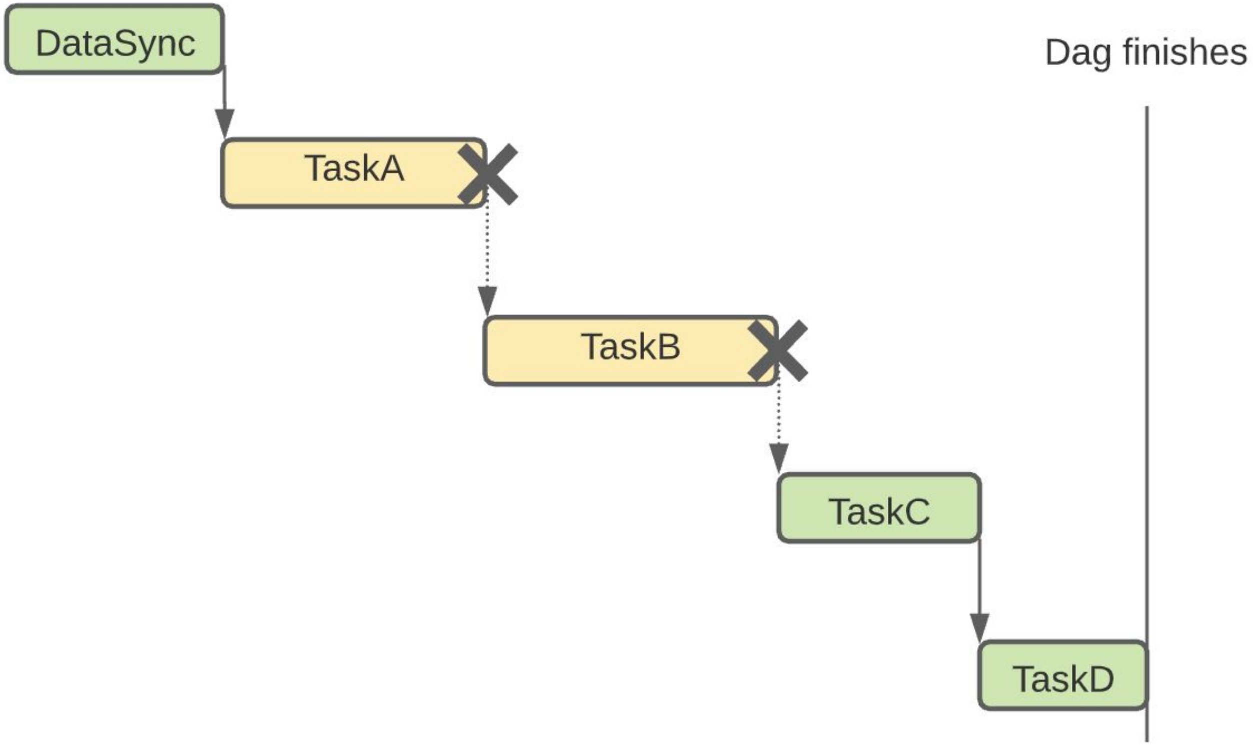


```
class TriggerRuleDep(BaseTIDep):
class TriggerRuleDepMigration(TriggerRuleDep):

    @staticmethod
    def _get_states_count_upstream_ti(ti, finished_tasks):
        counter = Counter(task.state
        counter = Counter(task.state if task.operator not in {"MigrationOperator"} else State.SUCCESS
                          for task in finished_tasks if task.task_id in ti.task.upstream_task_ids)
        return (
            counter.get(State.SUCCESS, 0),
            counter.get(State.SKIPPED, 0),
            counter.get(State.FAILED, 0),
            counter.get(State.UPSTREAM_FAILED, 0),
            sum(counter.values()),
        )
```

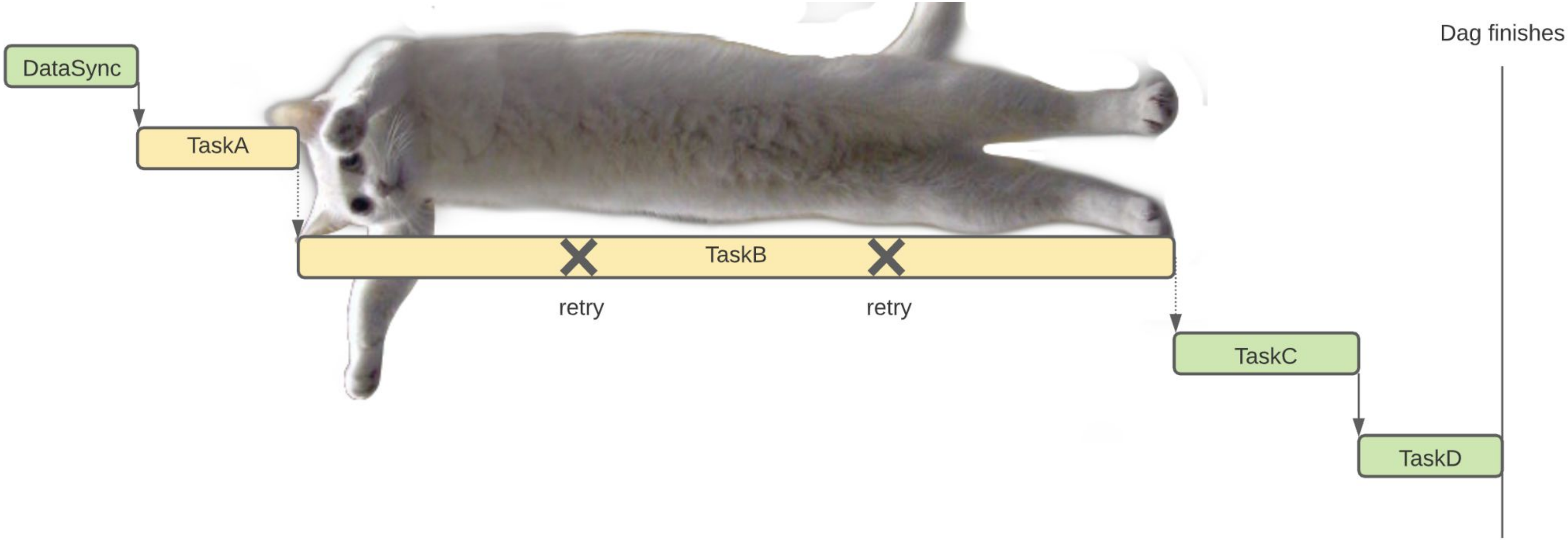
➕ 'trigger_rule'

Always treat Migration tasks as successful

# Naïve approach solution - DAG

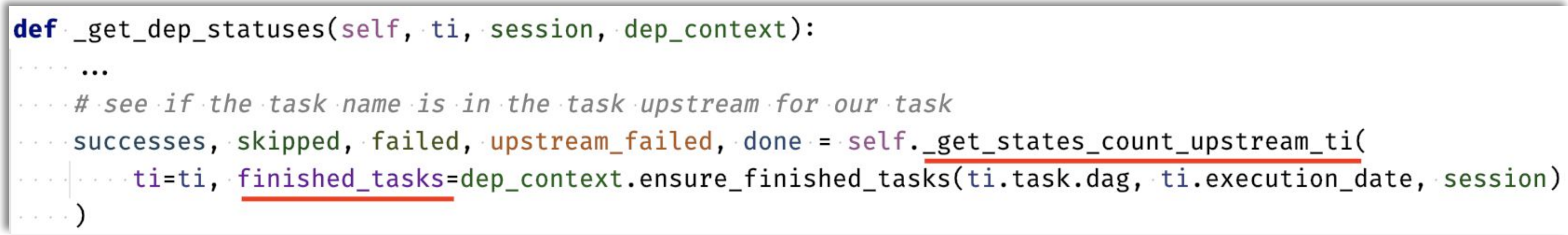


# Perf. Problem

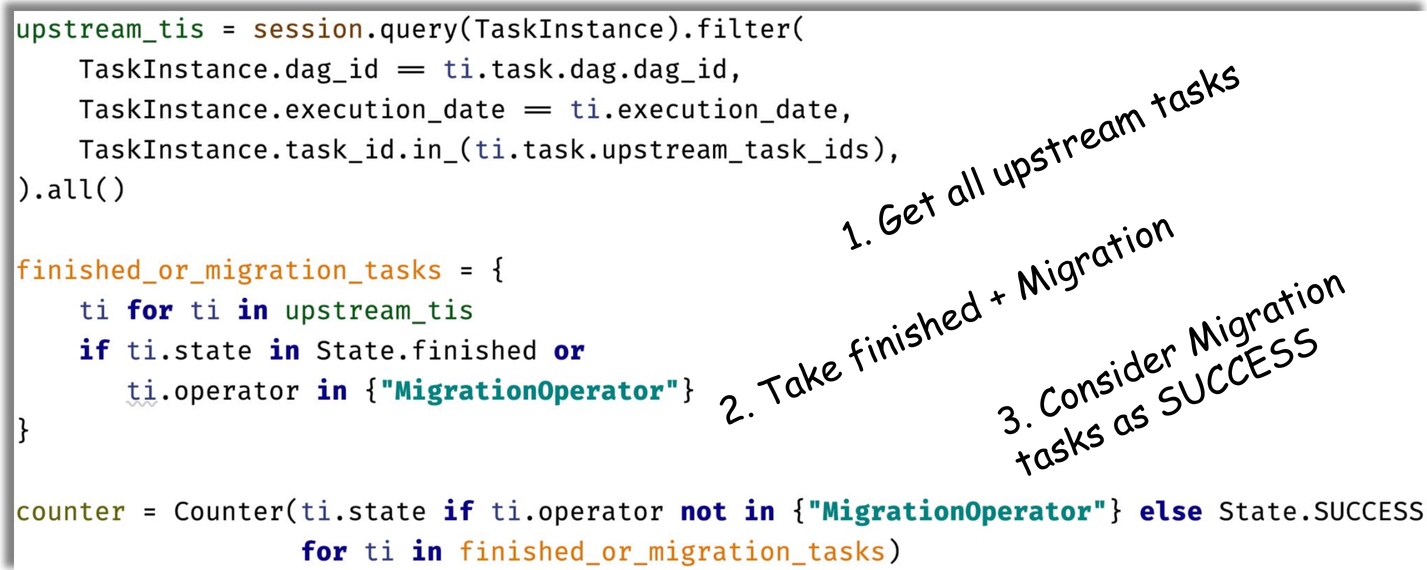


Unoptimized task can hold up all of the production downstreams

# Perf. Optimization



```python
def _get_dep_statuses(self, ti, session, dep_context):
    ...
    # see if the task name is in the task upstream for our task
    successes, skipped, failed, upstream_failed, done = self._get_states_count_upstream_ti(
        ti=ti, finished_tasks=dep_context.ensure_finished_tasks(ti.task.dag, ti.execution_date, session)
    )
```



```python
upstream_tis = session.query(TaskInstance).filter(
    TaskInstance.dag_id == ti.task.dag.dag_id,
    TaskInstance.execution_date == ti.execution_date,
    TaskInstance.task_id.in_(ti.task.upstream_task_ids),
).all()

finished_or_migration_tasks = {
    ti for ti in upstream_tis
    if ti.state in State.finished or
       ti.operator in {"MigrationOperator"}
}

counter = Counter(ti.state if ti.operator not in {"MigrationOperator"} else State.SUCCESS
                  for ti in finished_or_migration_tasks)
```

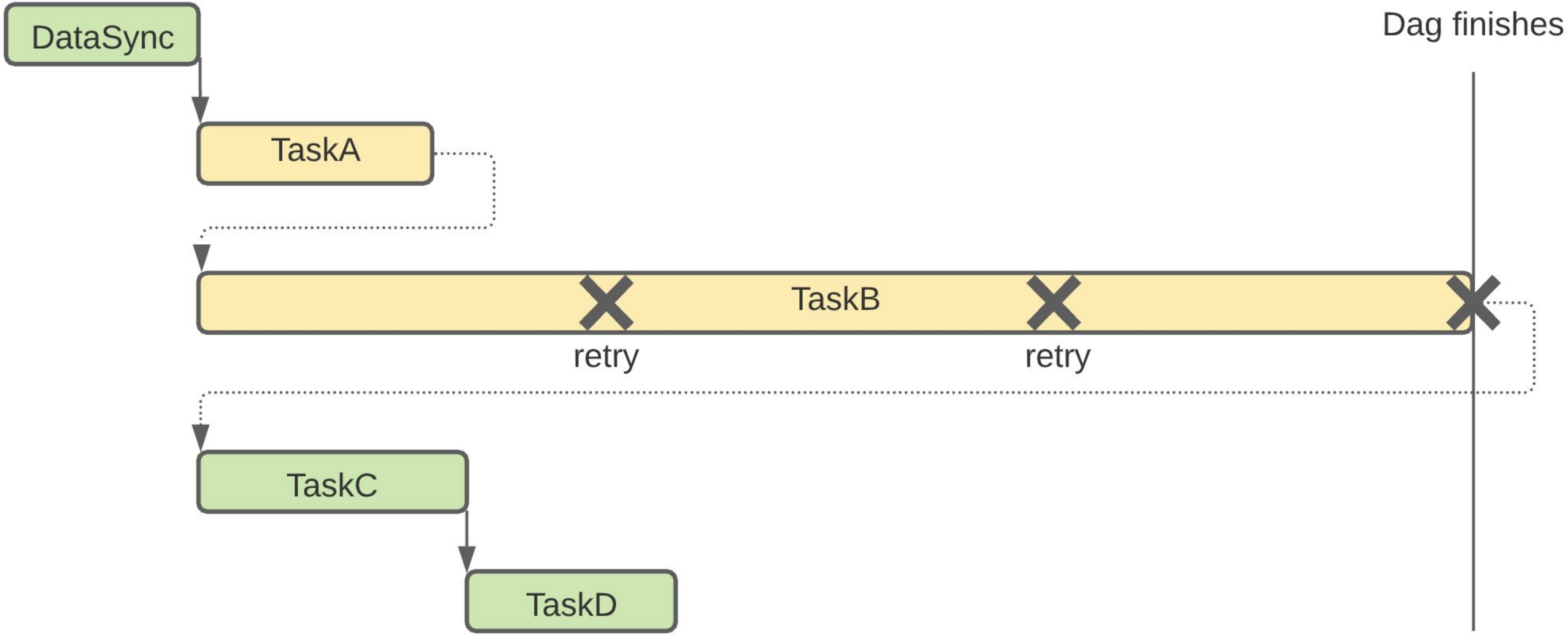
1. Get all upstream tasks

2. Take finished + Migration

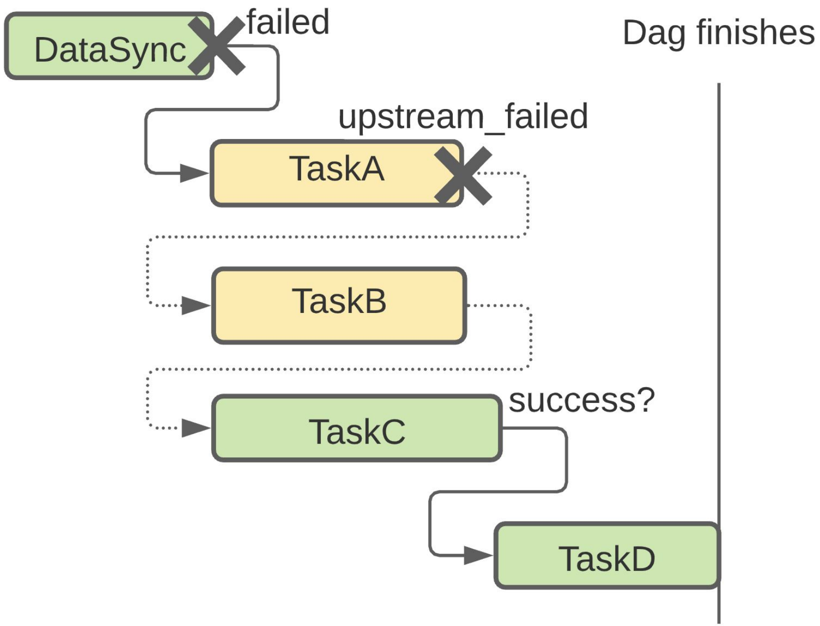
3. Consider Migration tasks as SUCCESS

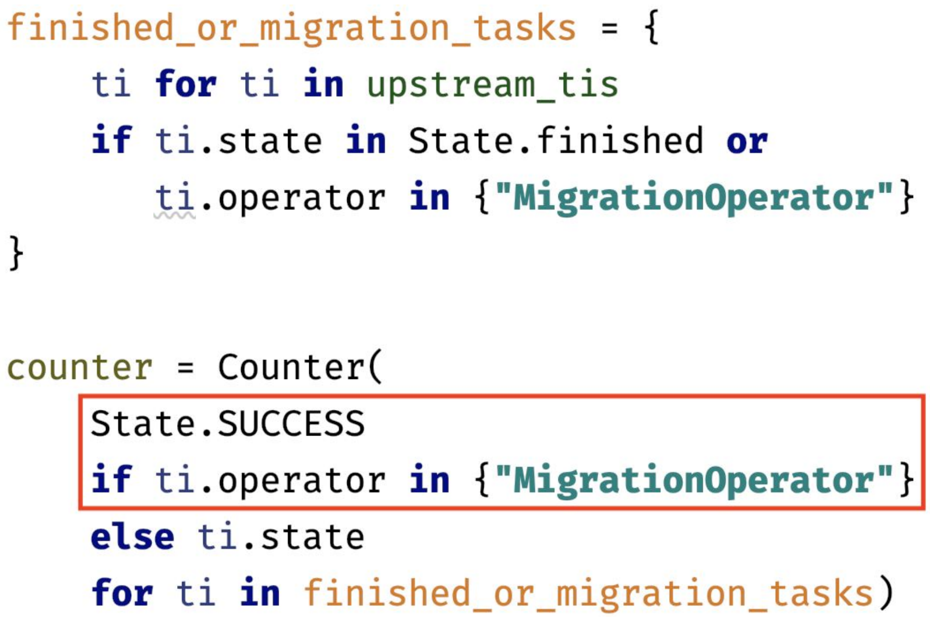
inline and rewrite

# Perf. Optimization - DAG



# Last problem: UPSTREAM_FAILED not propagated

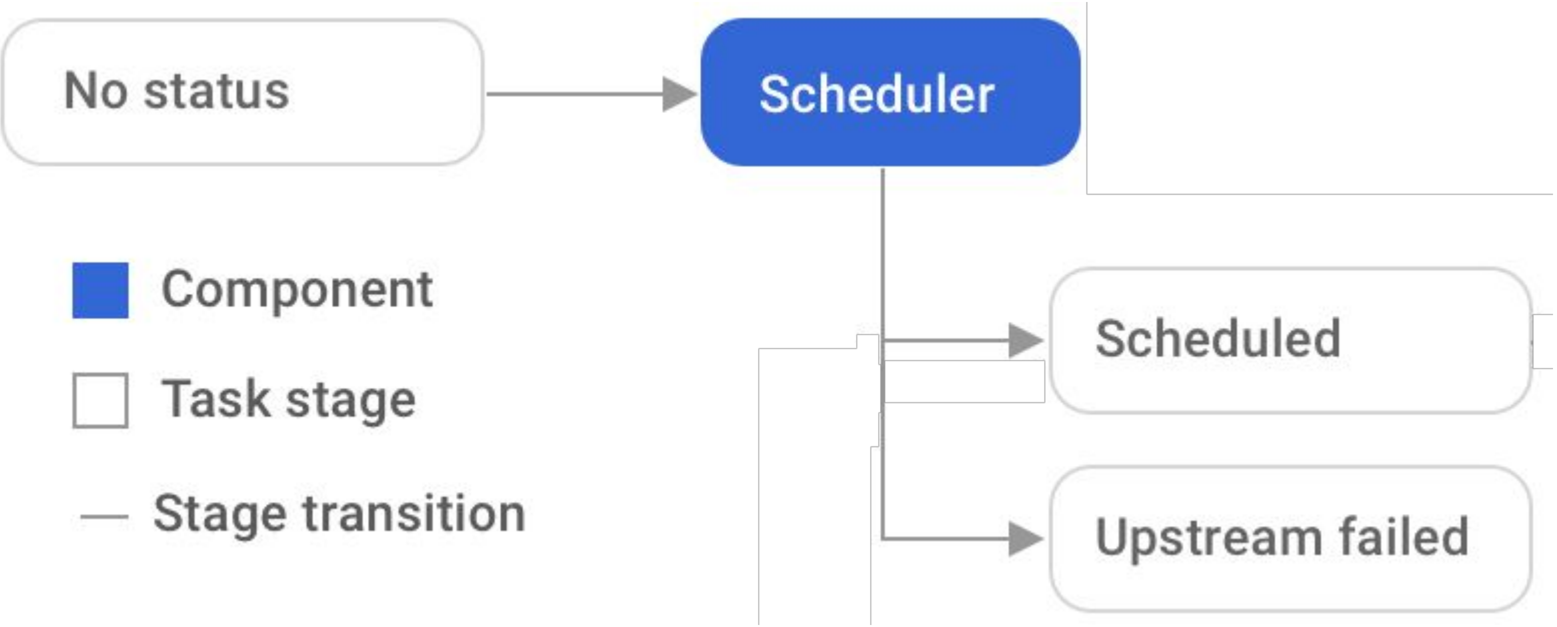




```
finished_or_migration_tasks = {
    ti for ti in upstream_tis
    if ti.state in State.finished or
       ti.operator in {"MigrationOperator"}
}

counter = Counter(
    State.SUCCESS
    if ti.operator in {"MigrationOperator"}
    else ti.state
    for ti in finished_or_migration_tasks)
```

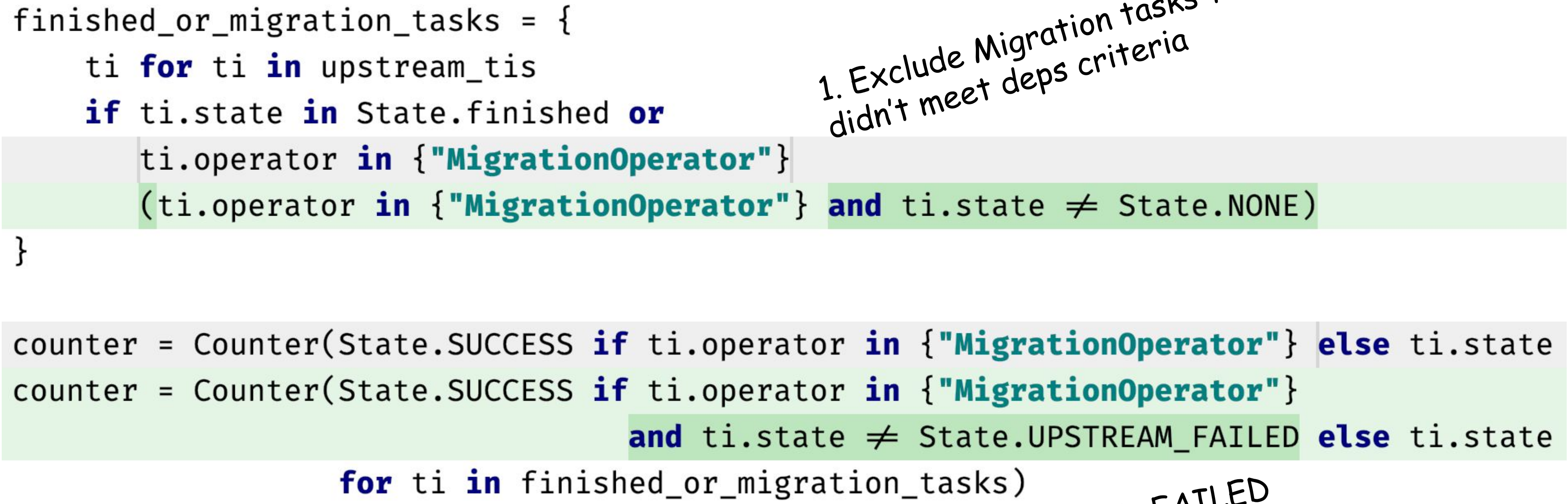
If DataSync fails, all tasks have to take this into account and stop - State.UPSTREAM_FAILED has to be propagated

# Task lifecycle refresher



https://airflow.apache.org/docs/apache-airflow/stable/concepts/tasks.html#task-instances

# Propagate UPSTREAM_FAILED

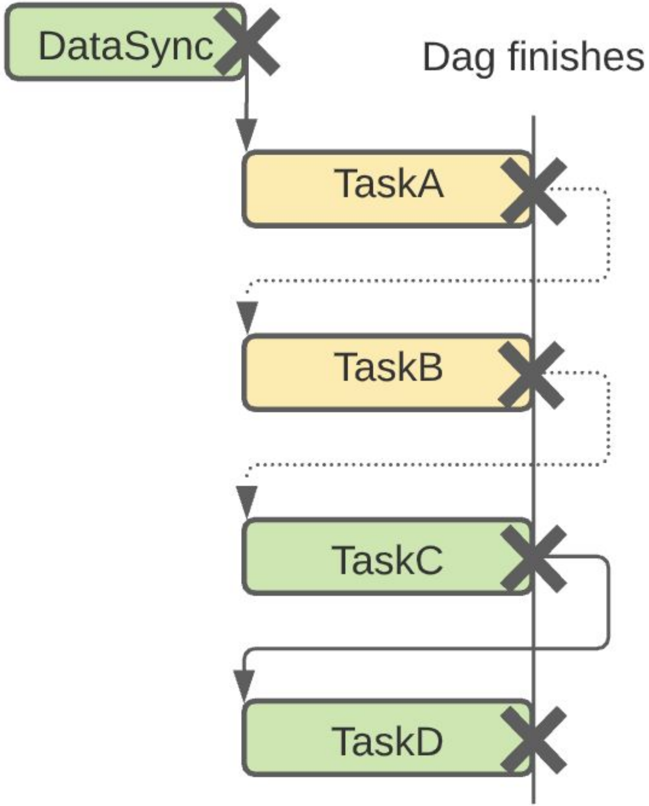


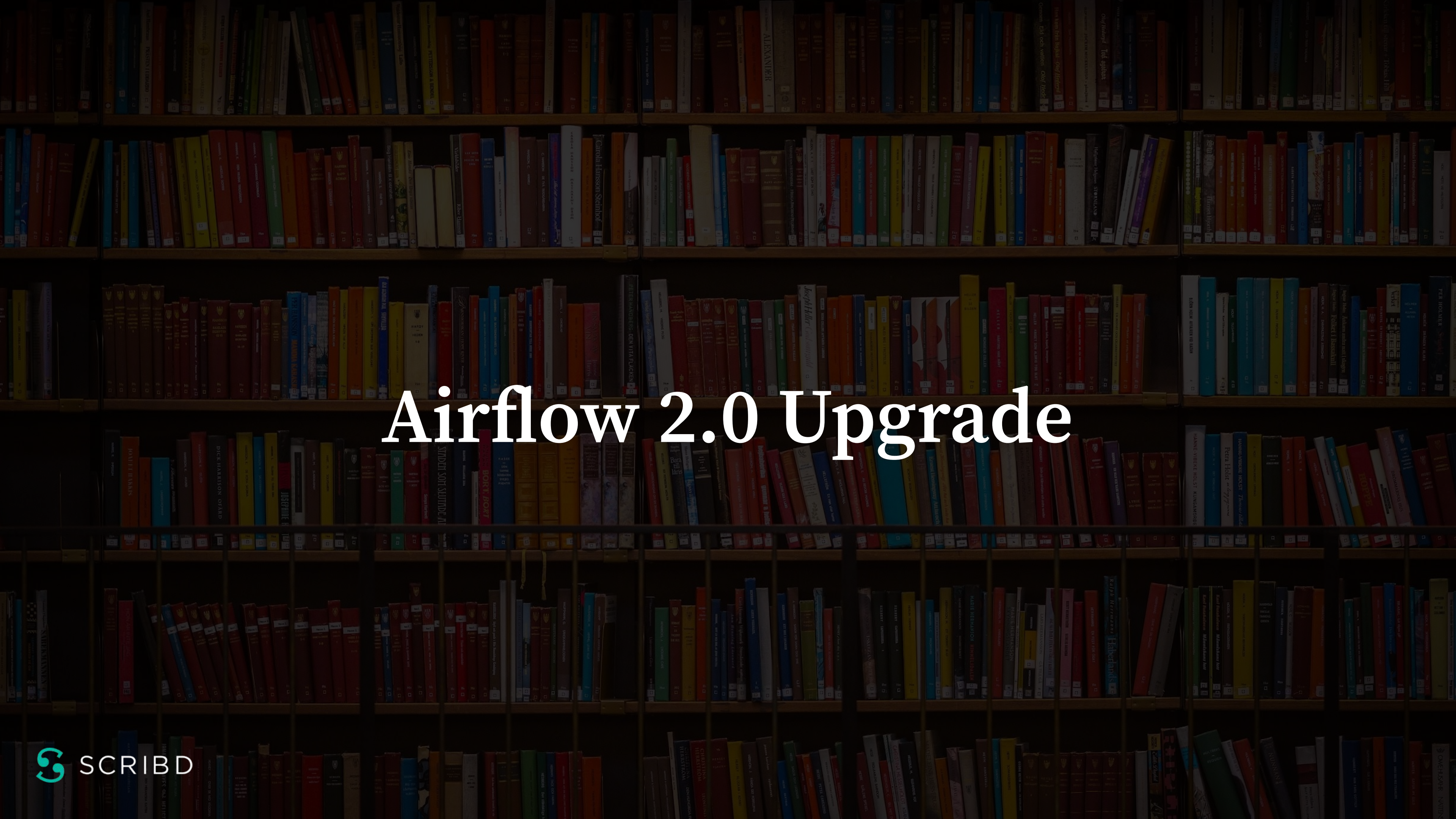
```
finished_or_migration_tasks = {
    ti for ti in upstream_tis
    if ti.state in State.finished or
       ti.operator in {"MigrationOperator"}
       (ti.operator in {"MigrationOperator"} and ti.state ≠ State.NONE)
}

counter = Counter(State.SUCCESS if ti.operator in {"MigrationOperator"} else ti.state
counter = Counter(State.SUCCESS if ti.operator in {"MigrationOperator"}
                                    and ti.state ≠ State.UPSTREAM_FAILED else ti.state
                  for ti in finished_or_migration_tasks)
```

1. Exclude Migration tasks that didn't meet deps criteria

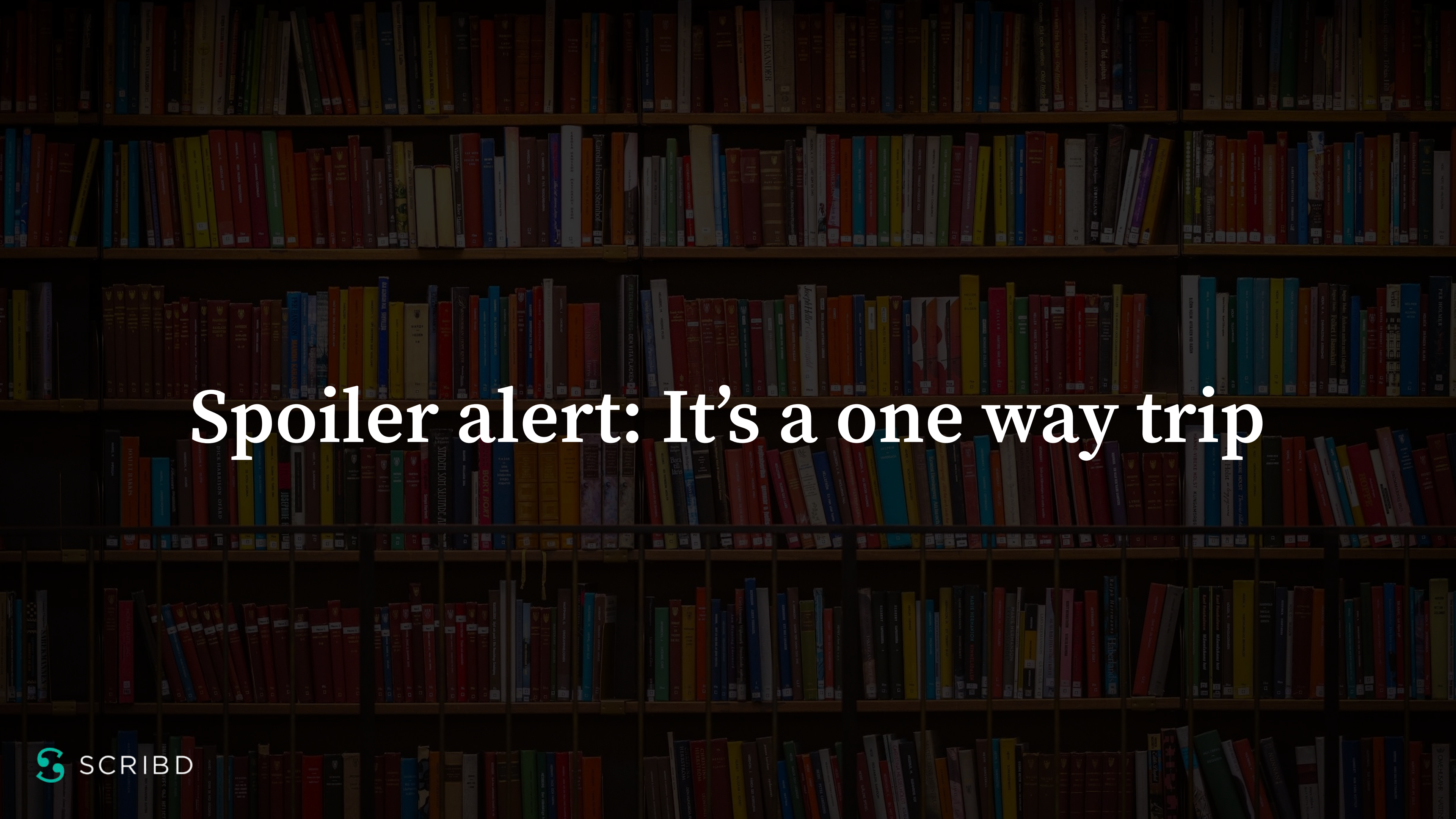
2. Propagate UPSTREAM_FAILED Status

# Propagate UPSTREAM_FAILED - DAG



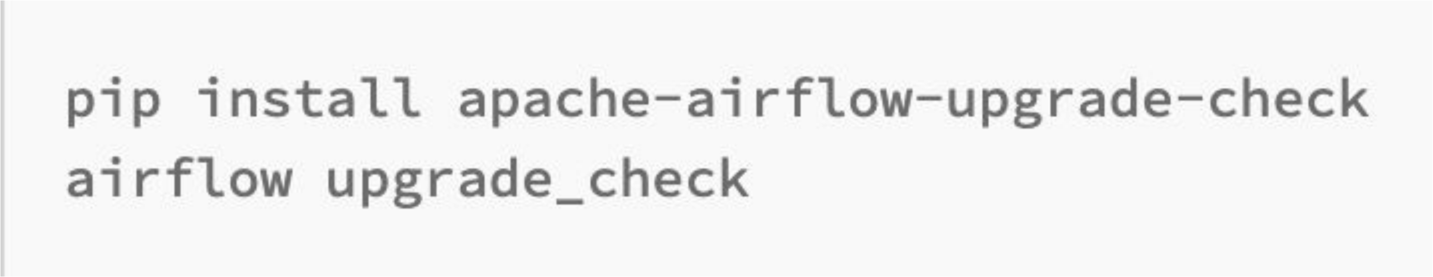


# Airflow 2.0 Upgrade



# Spoiler alert: It’s a one way trip

# Airflow upgrade check is your friend

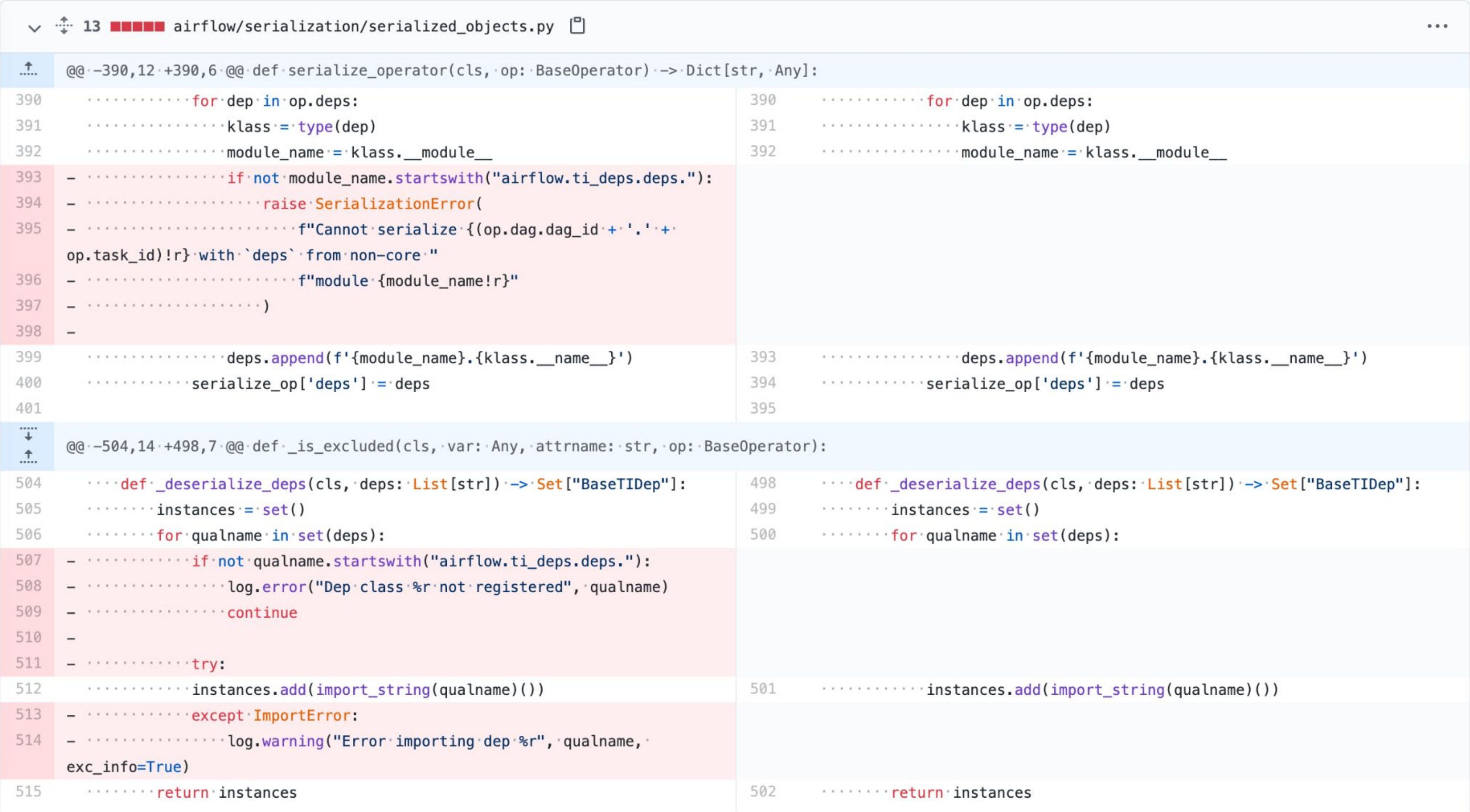


```
pip install apache-airflow-upgrade-check
airflow upgrade_check
```

# Airflow 2.0 upgrade - MySQL (Aurora RDS)

- MySQL 5.6 not supported by Airflow 2.0
  - Missing JSON column types
- MySQL 5.7 kind of works
- MySQL 8 not supported by Aurora RDS
  - Required for scheduler HA

# Airflow 2.0 upgrade - Trigger rules



13 airflow/serialization/serialized_objects.py

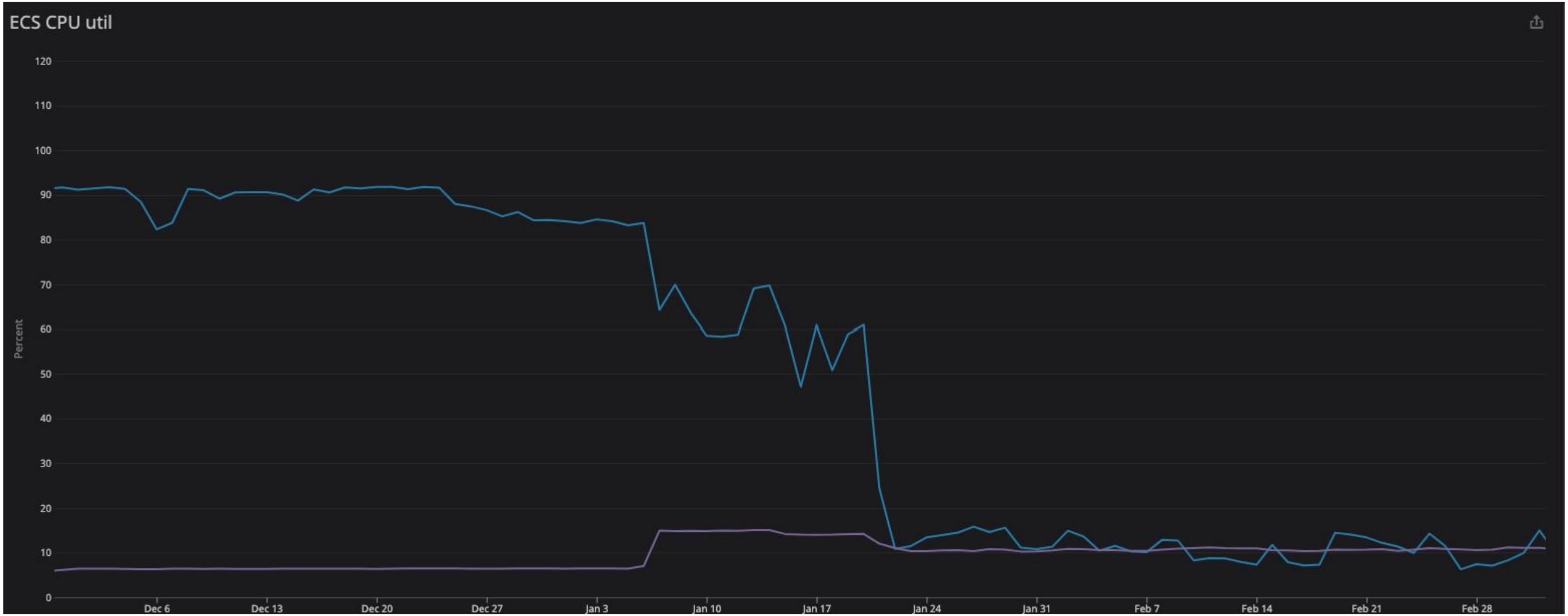
```
@@ -390,12 +390,6 @@ def serialize_operator(cls, op: BaseOperator) -> Dict[str, Any]:
            for dep in op.deps:
                klass = type(dep)
                module_name = klass.__module__
-               if not module_name.startswith("airflow.ti_deps.deps."):
-                   raise SerializationError(
-                       f"Cannot serialize {(op.dag.dag_id + '.' + op.task_id)!r} with `deps` from non-core "
-                       f"module {module_name!r}"
-                   )
-
                deps.append(f'{module_name}.{klass.__name__}')
            serialize_op['deps'] = deps

@@ -504,14 +498,7 @@ def _is_excluded(cls, var: Any, attrname: str, op: BaseOperator):
    def _deserialize_deps(cls, deps: List[str]) -> Set["BaseTIDep"]:
        instances = set()
        for qualname in set(deps):
-           if not qualname.startswith("airflow.ti_deps.deps."):
-               log.error("Dep class %r not registered", qualname)
-               continue
-
-           try:
            instances.add(import_string(qualname)())
-           except ImportError:
-               log.warning("Error importing dep %r", qualname, exc_info=True)
        return instances
```

```
            for dep in op.deps:
                klass = type(dep)
                module_name = klass.__module__
                deps.append(f'{module_name}.{klass.__name__}')
            serialize_op['deps'] = deps

    def _deserialize_deps(cls, deps: List[str]) -> Set["BaseTIDep"]:
        instances = set()
        for qualname in set(deps):
            instances.add(import_string(qualname)())
        return instances
```

# Airflow 2.0 upgrade - Performance improvement

- Faster Web UI
- Faster scheduler
- Scheduler sharding

# Scheduler CPU usage after 2.0 upgrade



# Running Backfill at Scale

SCRIBD

# Running Backfill at scale

### Our goal

1. Backfill data for 14 years
2. Our intended DAG concurrency (i.e. how many version of single DAG we can run concurrently) was 150, we settled later to 100

### Limitless Limits

People say “Sky is the limit”, but to reach the sky there is a small matter of gravity that we have to overcome. Exactly that happened to us. Let us talk about our gravitational boundaries.

# Airflow limits

## AIRFLOW _CORE_ PARALLELISM

The amount of parallelism as a setting to the executor. This defines the max number of task instances that should run simultaneously. Default value is 32. We override that in our backfill execution commands to 100.

## AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG

The maximum number of active DAG runs per DAG. It maps to max_active_runs attribute in the DAG definition. Default value is 16. We override it to 100.

## AIRFLOW__CORE__DAG_CONCURRENCY

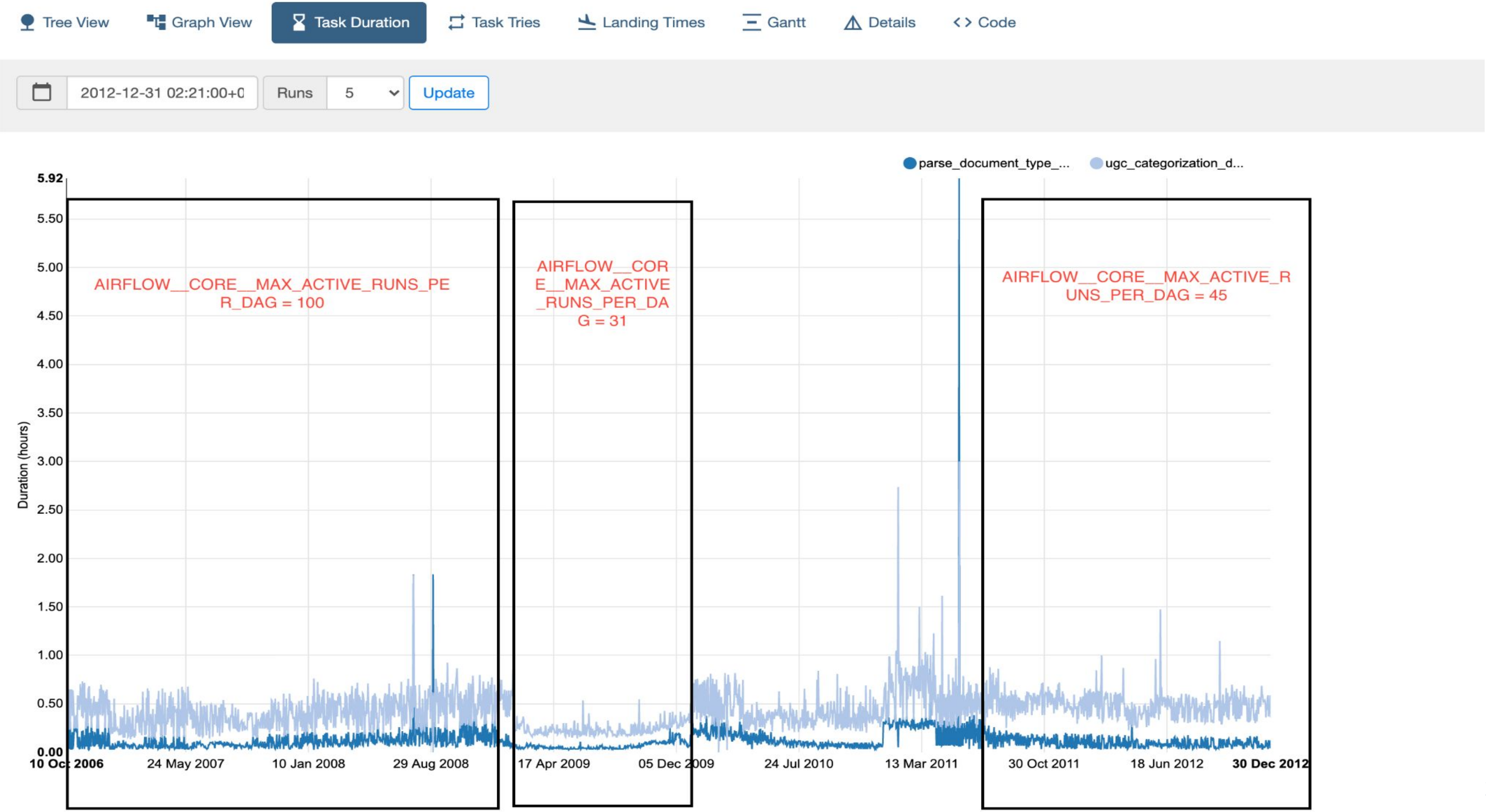
The number of task instances allowed to run concurrently by the scheduler in one DAG. It maps to concurrency attribute in the DAG definition. We override it to 100.

# Databricks and AWS limits

Apart from Airflow limits, we got restricted by Databricks and AWS account limits while working on the backfill. Here are some examples:

1. AWS account limit of 1000 TB of total GP2 EBS volume size. We increased it to 1500 TB while at the same time reduced our EBS volume size per machine by almost 60%.
2. Databricks API limit. We were getting “429 Too Many Requests” errors from Databricks.
3. Databricks Node creation limit at 200 nodes per minute.  We worked with Databricks to get these limits lifted for our account.

# How much is too much



AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG = 100

AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG = 31

AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG = 45

# Self-Service Backfill UI plugin

SCRIBD

# Self-Service Backfill UI plugin

Why?

- Switch from Legacy in-house system to Airflow
- Increased load on Airflow Admins
- Give back the ability to run self-serviced backfills to our engineers
- Web UI based backfill trigger is still being discussed by the community

# Self-Service Backfill UI plugin

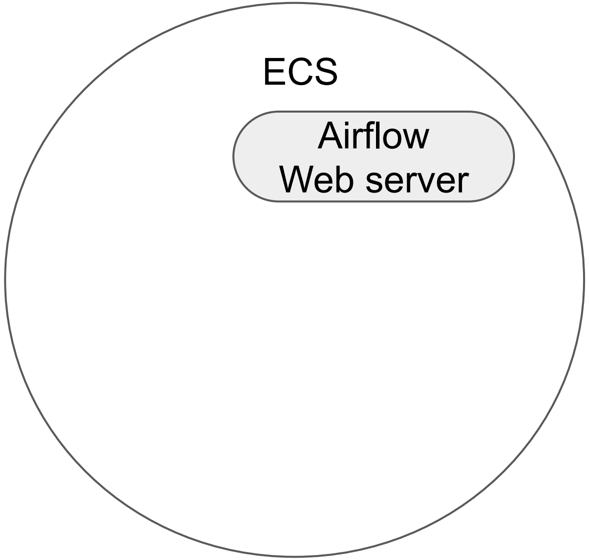
Considered approaches:

- Feed all tasks to the scheduler
- New type of Job in the Web Server
- Use built-in Backfill functionality

# Self-Service Backfill UI plugin

How?

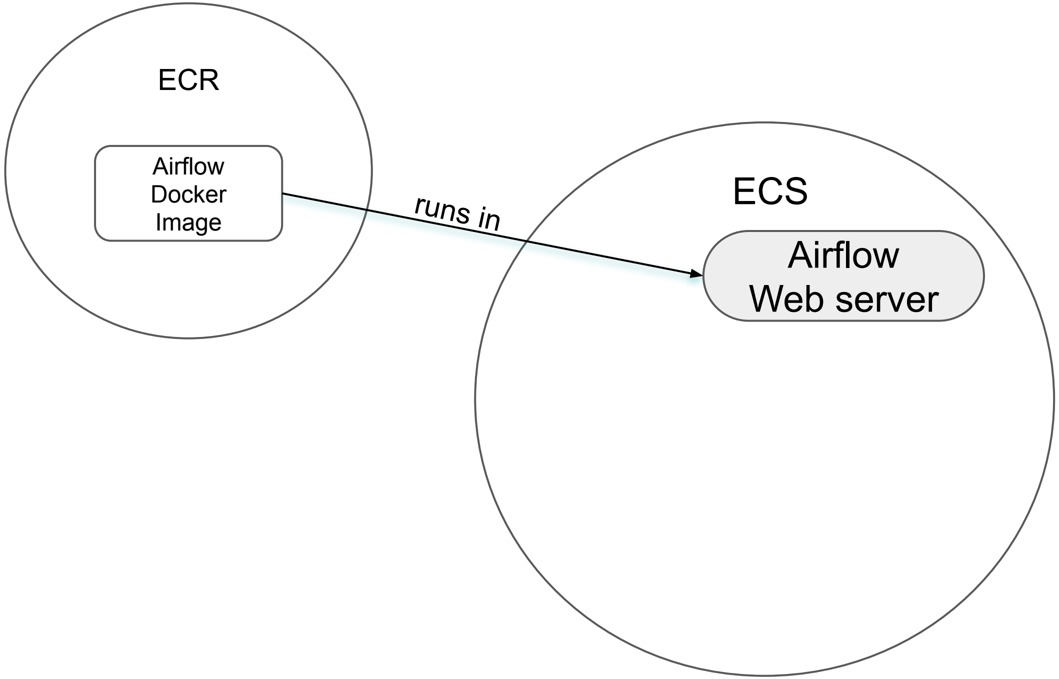
- AWS Elastic Container Service



# Self-Service Backfill UI plugin

How?

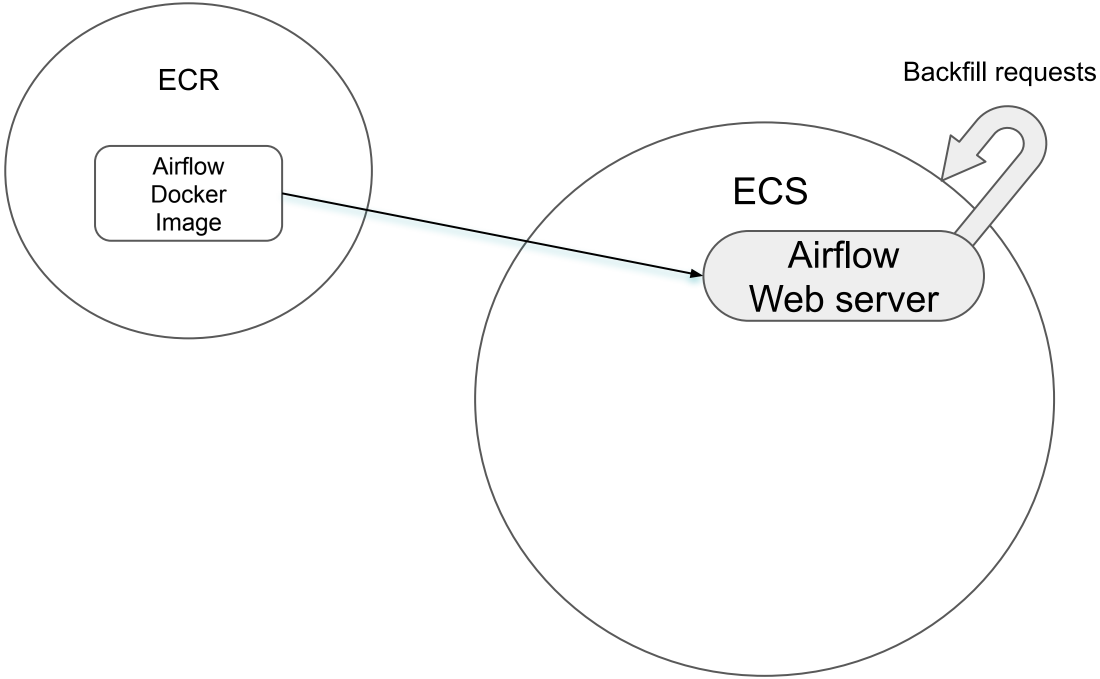
- AWS Elastic Container Service
- AWS Elastic Container Registry



# Self-Service Backfill UI plugin

How?

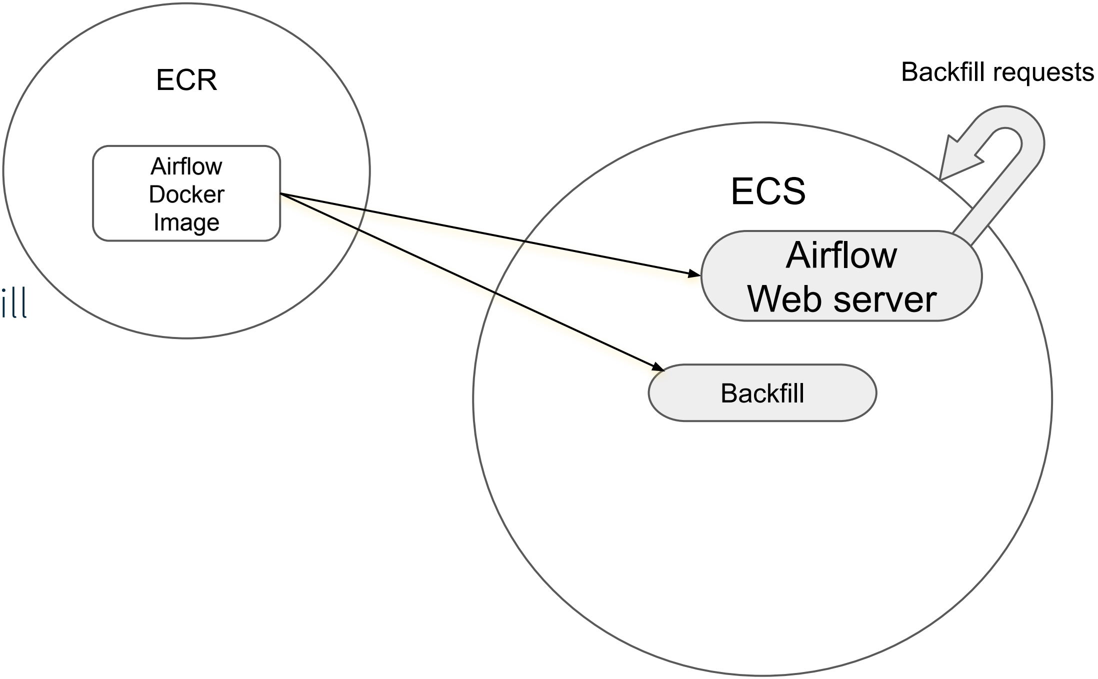
- AWS Elastic Container Service
- AWS Elastic Container Registry
- ECS container to ECS calls



# Self-Service Backfill UI plugin

How?

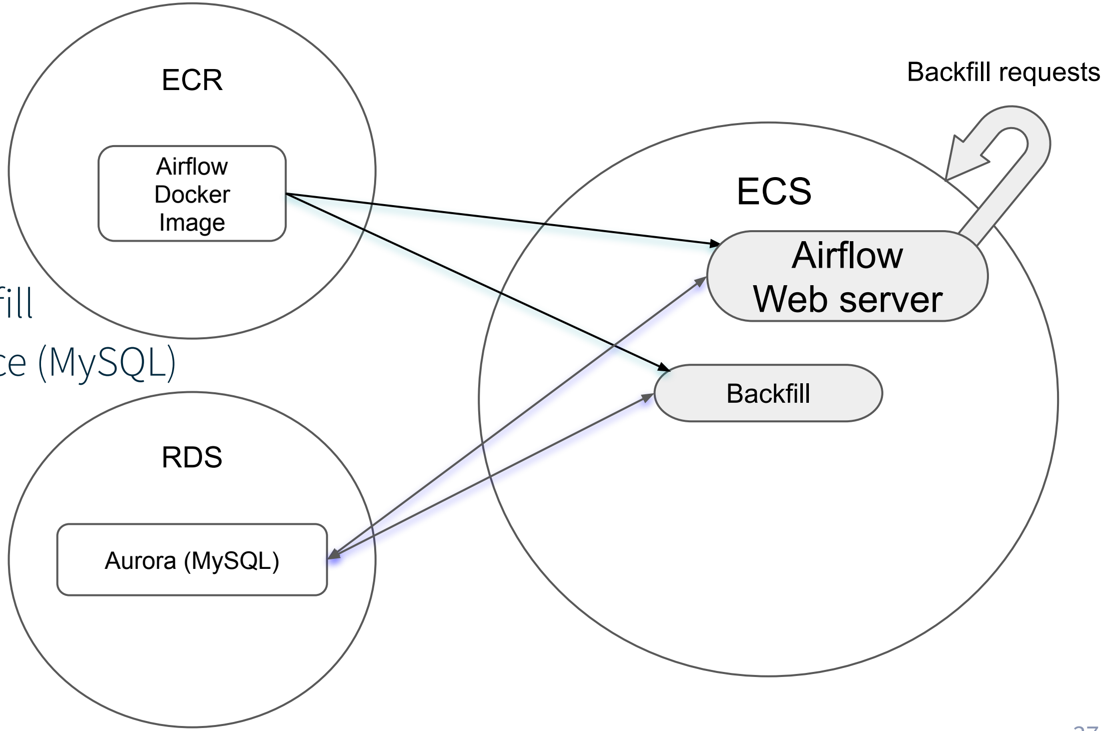
- AWS Elastic Container Service
- AWS Elastic Container Registry
- ECS container to ECS calls
- New ECS container for each Backfill



# Self-Service Backfill UI plugin

How?

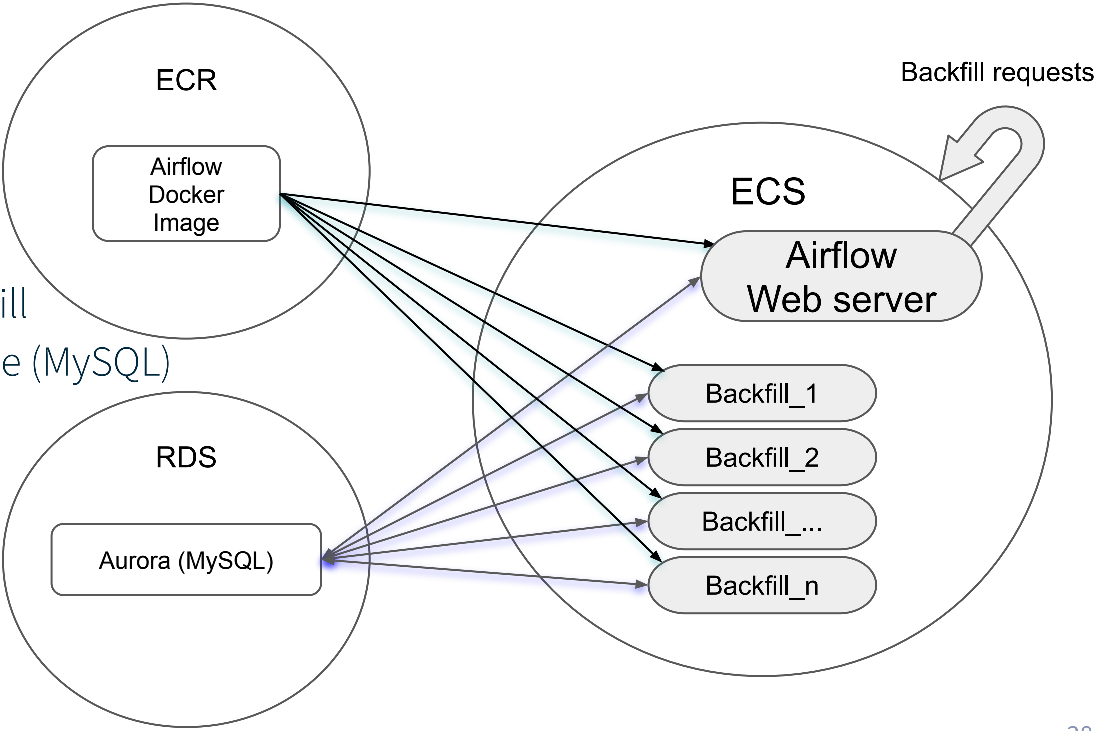
- AWS Elastic Container Service
- AWS Elastic Container Registry
- ECS container to ECS calls
- New ECS container for each Backfill
- Aurora Relational Database Service (MySQL)

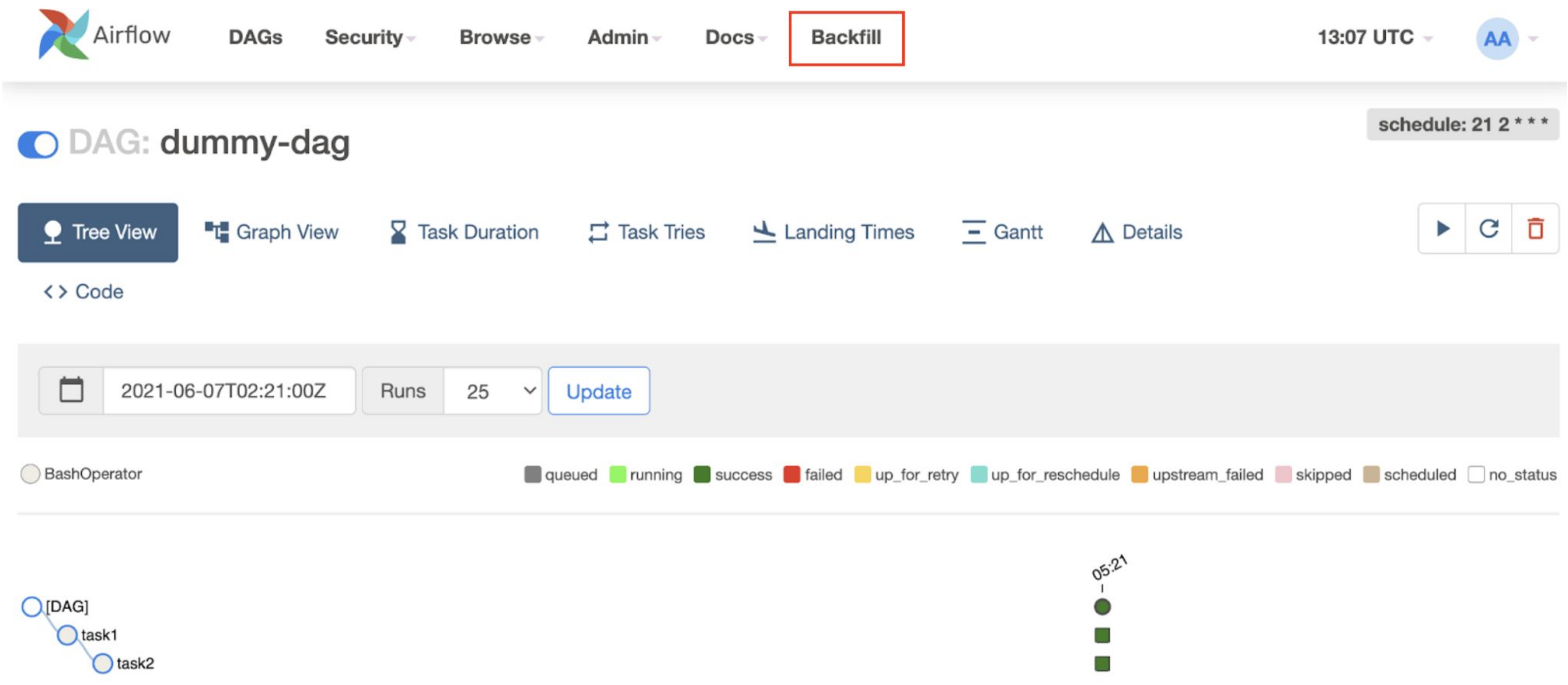


# Self-Service Backfill UI plugin

How?

- AWS Elastic Container Service
- AWS Elastic Container Registry
- ECS container to ECS calls
- New ECS container for each Backfill
- Aurora Relational Database Service (MySQL)



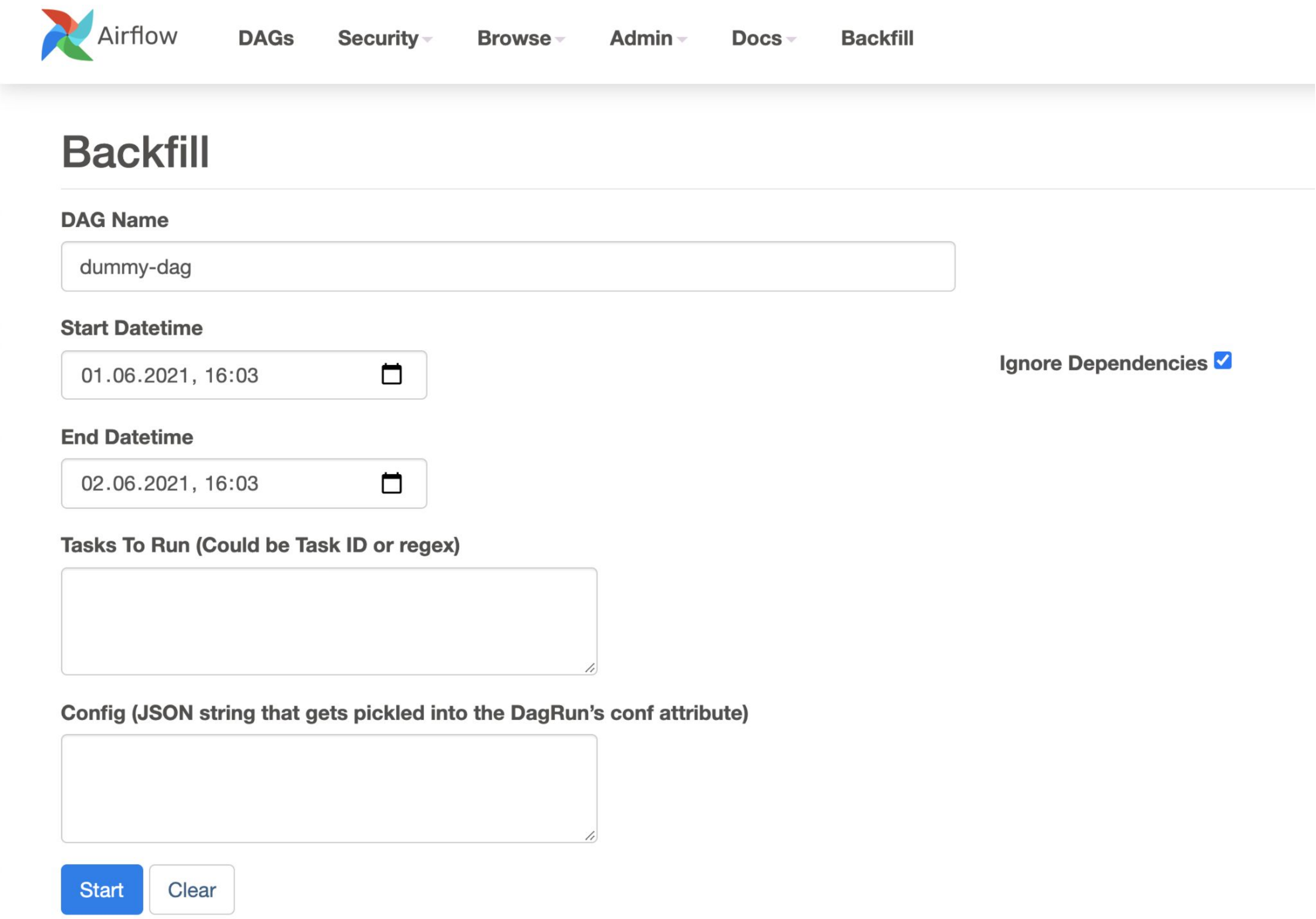


DAG: dummy-dag

schedule: 21 2 * * *

2021-06-07T02:21:00Z

Runs 25



# Backfill

**DAG Name**

dummy-dag

**Start Datetime**

01.06.2021, 16:03

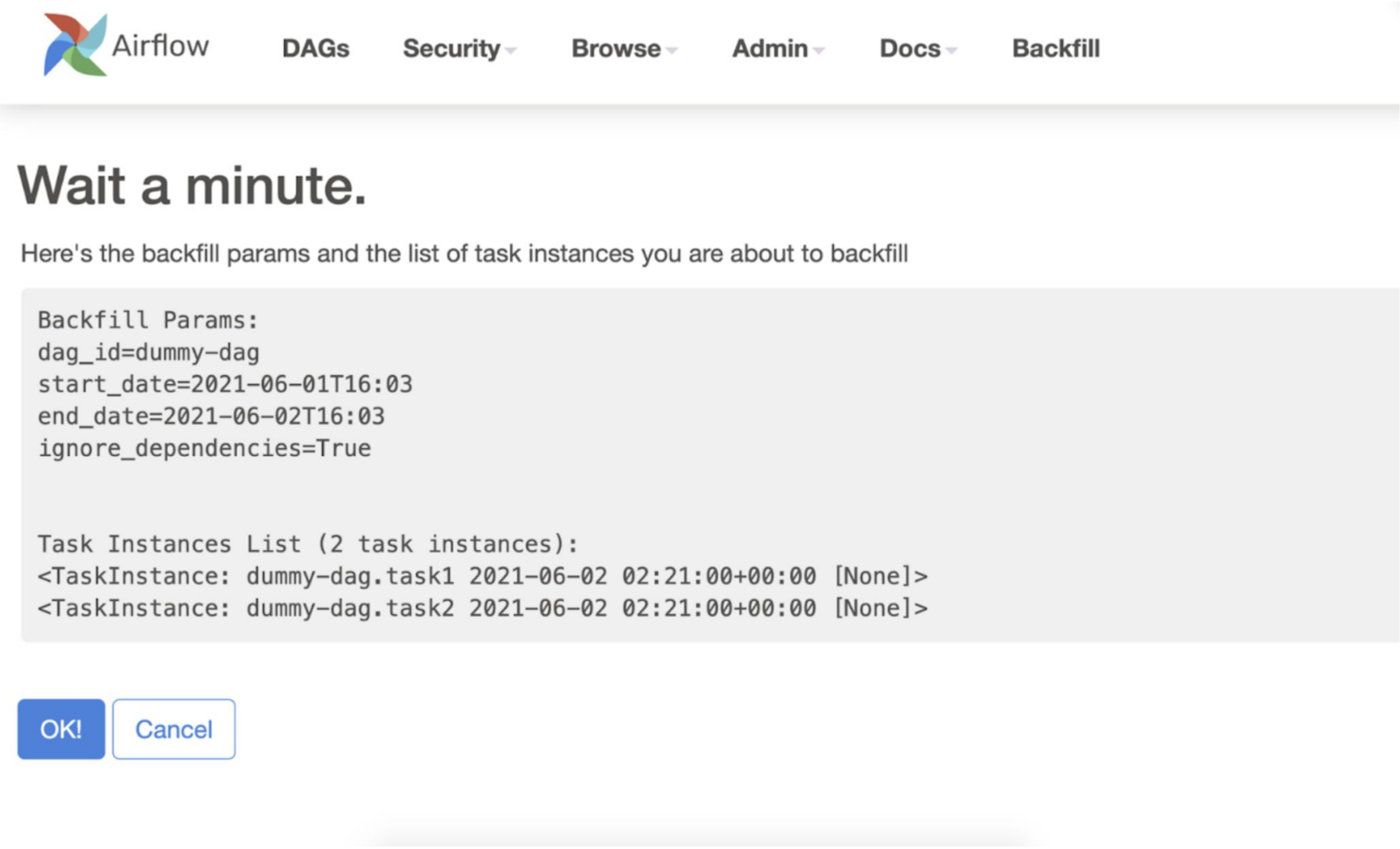
**End Datetime**

02.06.2021, 16:03

**Ignore Dependencies** ☑

**Tasks To Run (Could be Task ID or regex)**

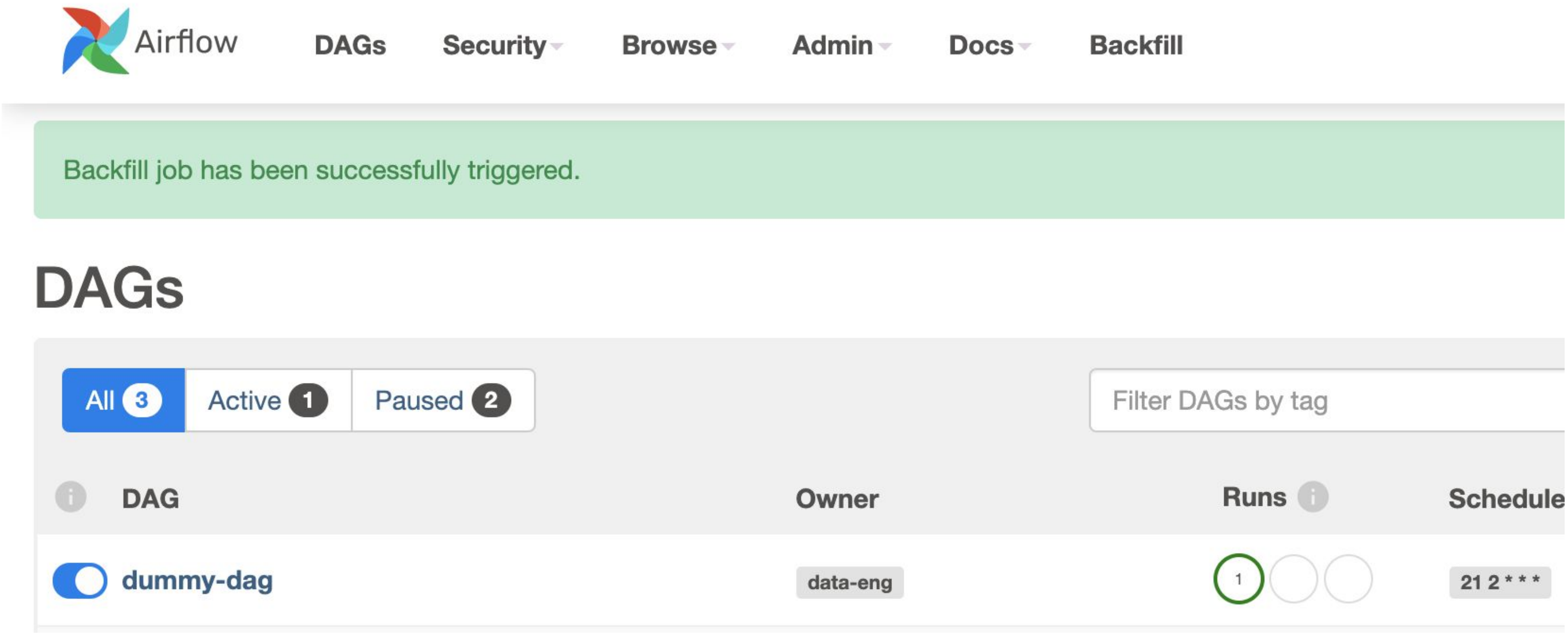
**Config (JSON string that gets pickled into the DagRun's conf attribute)**



# Wait a minute.

Here's the backfill params and the list of task instances you are about to backfill

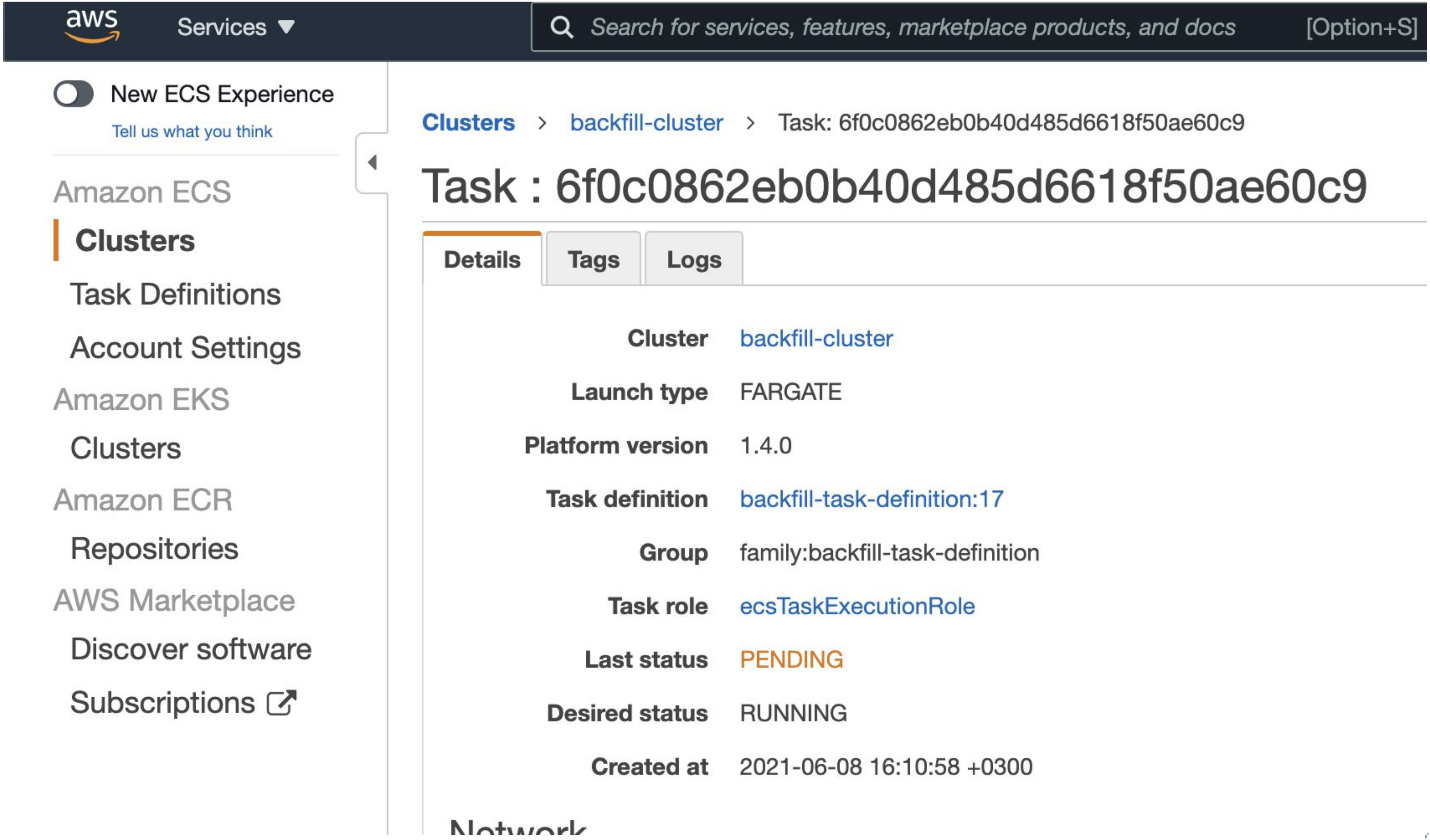
```
Backfill Params:
dag_id=dummy-dag
start_date=2021-06-01T16:03
end_date=2021-06-02T16:03
ignore_dependencies=True


Task Instances List (2 task instances):
<TaskInstance: dummy-dag.task1 2021-06-02 02:21:00+00:00 [None]>
<TaskInstance: dummy-dag.task2 2021-06-02 02:21:00+00:00 [None]>
```



Backfill job has been successfully triggered.

# DAGs

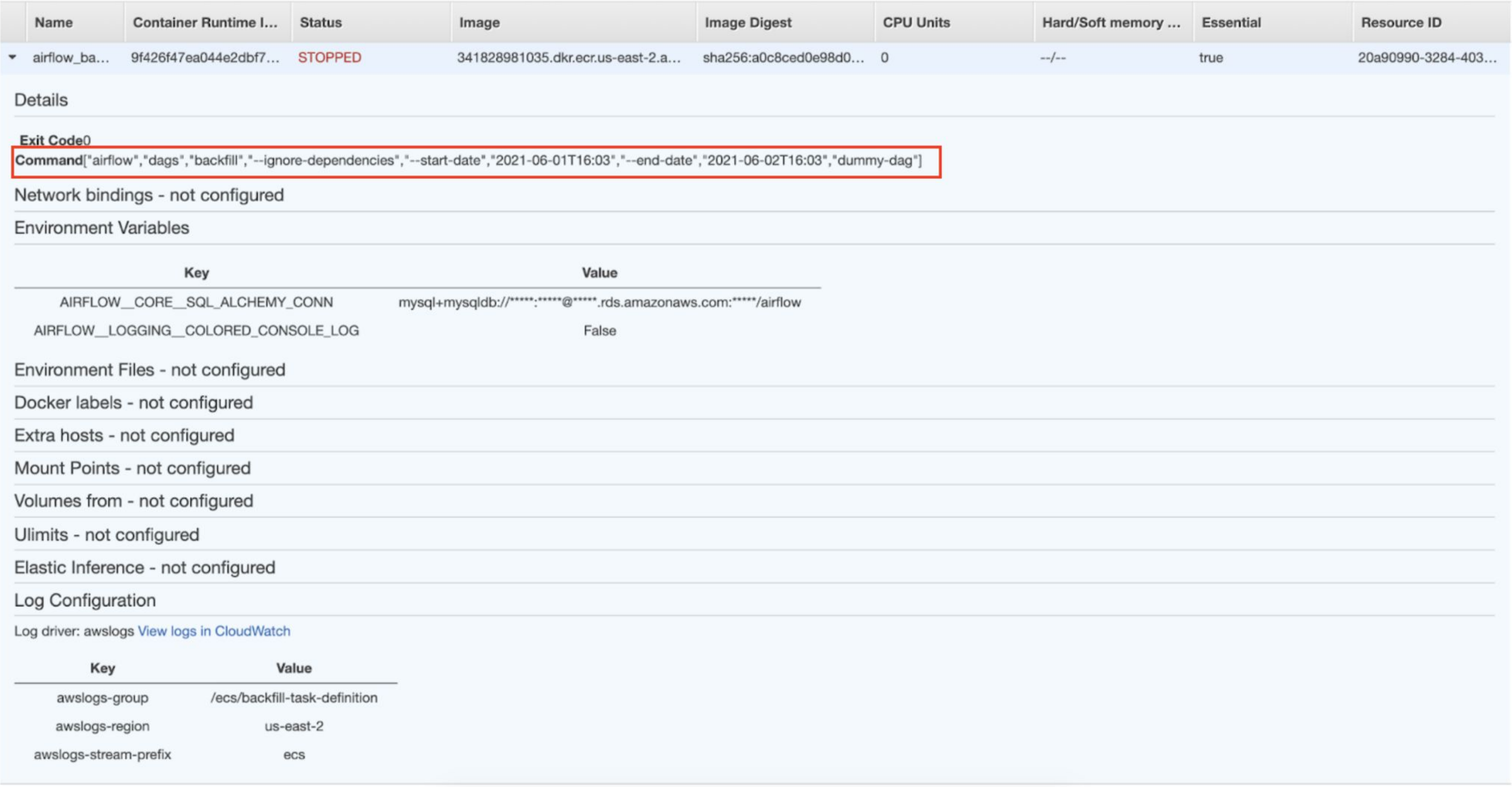
| DAG | Owner | Runs | Schedule |
|---|---|---|---|
| dummy-dag | data-eng | 1 | 21 2 * * * |



# Task : 6f0c0862eb0b40d485d6618f50ae60c9

| | |
|---|---|
| Cluster | backfill-cluster |
| Launch type | FARGATE |
| Platform version | 1.4.0 |
| Task definition | backfill-task-definition:17 |
| Group | family:backfill-task-definition |
| Task role | ecsTaskExecutionRole |
| Last status | PENDING |
| Desired status | RUNNING |
| Created at | 2021-06-08 16:10:58 +0300 |

## Network



| | Name | Container Runtime I... | Status | Image | Image Digest | CPU Units | Hard/Soft memory ... | Essential | Resource ID |
|---|---|---|---|---|---|---|---|---|---|
| ▾ | airflow_ba... | 9f426f47ea044e2dbf7... | STOPPED | 341828981035.dkr.ecr.us-east-2.a... | sha256:a0c8ced0e98d0... | 0 | --/-- | true | 20a90990-3284-403... |

Details

**Exit Code**0

**Command**["airflow","dags","backfill","--ignore-dependencies","--start-date","2021-06-01T16:03","--end-date","2021-06-02T16:03","dummy-dag"]

Network bindings - not configured

Environment Variables

| Key | Value |
|---|---|
| AIRFLOW__CORE__SQL_ALCHEMY_CONN | mysql+mysqldb://*****:*****@*****.rds.amazonaws.com:*****/airflow |
| AIRFLOW__LOGGING__COLORED_CONSOLE_LOG | False |

Environment Files - not configured

Docker labels - not configured

Extra hosts - not configured

Mount Points - not configured

Volumes from - not configured

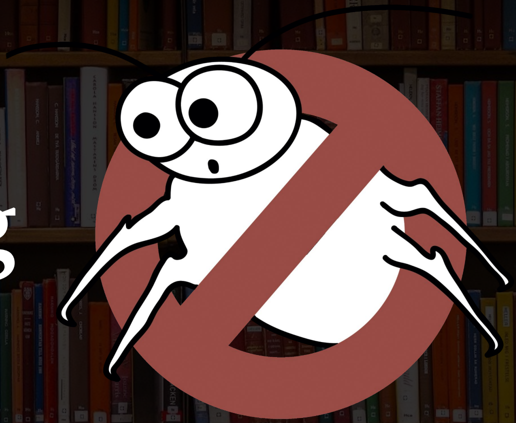
Ulimits - not configured

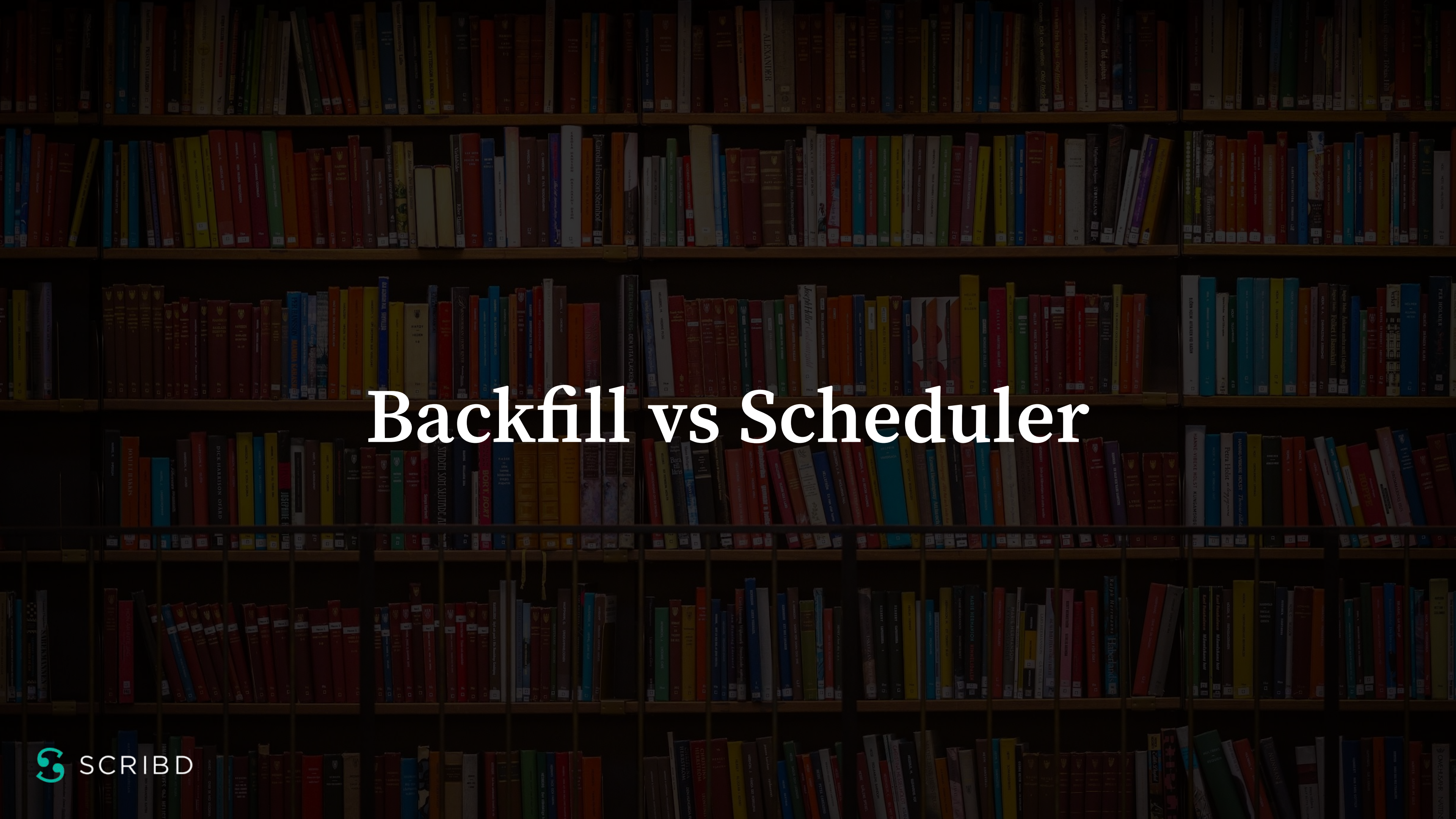
Elastic Inference - not configured

Log Configuration

Log driver: awslogs View logs in CloudWatch

| Key | Value |
|---|---|
| awslogs-group | /ecs/backfill-task-definition |
| awslogs-region | us-east-2 |
| awslogs-stream-prefix | ecs |

# Bug  Busters



# Backfill vs Scheduler

SCRIBD

# Backfill vs Scheduler

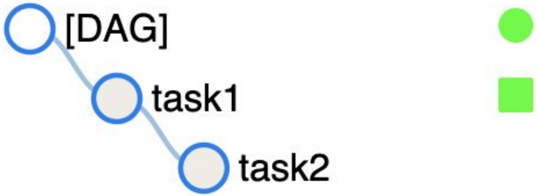
3 types of jobs in Airflow:

- SchedulerJob
- BackfillJob
- LocalTaskJob

# Backfill vs Scheduler

Example:

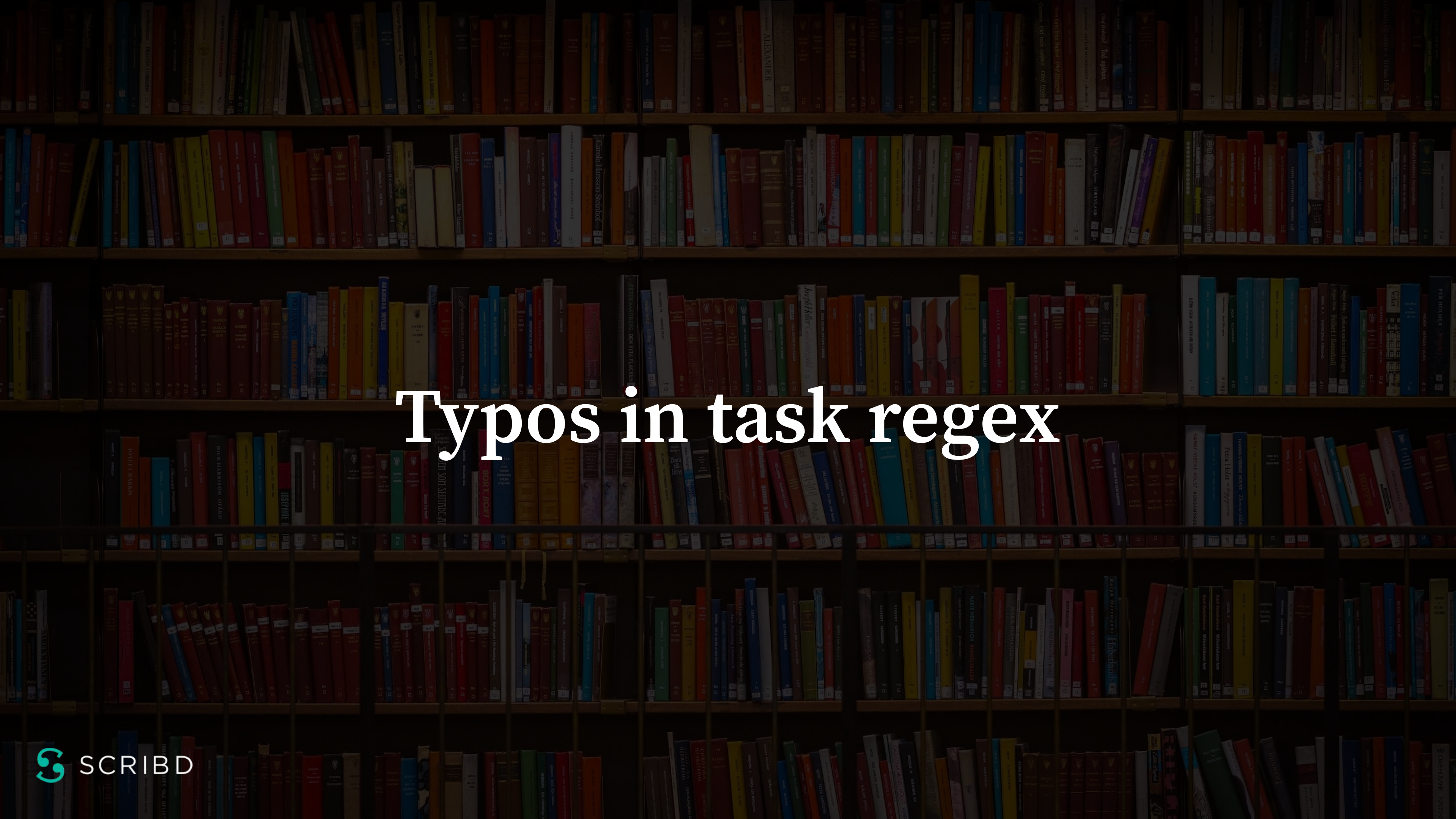
1. SchedulerJob creates a DagRun and starts Task instances
2. SchedulerJob starts “task1”



3. BackfillJob started for a single task - “task1”
4. BackfillJob overwrites scheduler’s DagRun
5. SchedulerJob forgets about “task2” and it never gets triggered

# Backfill vs Scheduler

Upstream fix PR under review: https://github.com/apache/airflow/pull/16089



# Typos in task regex

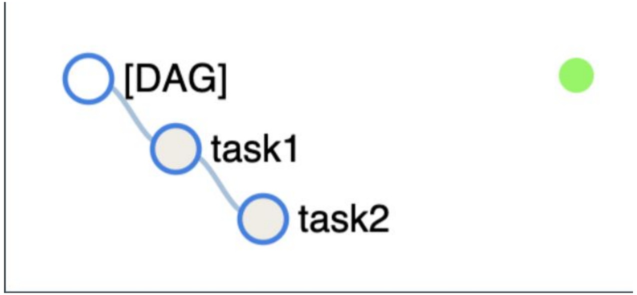
# Typo in task regex

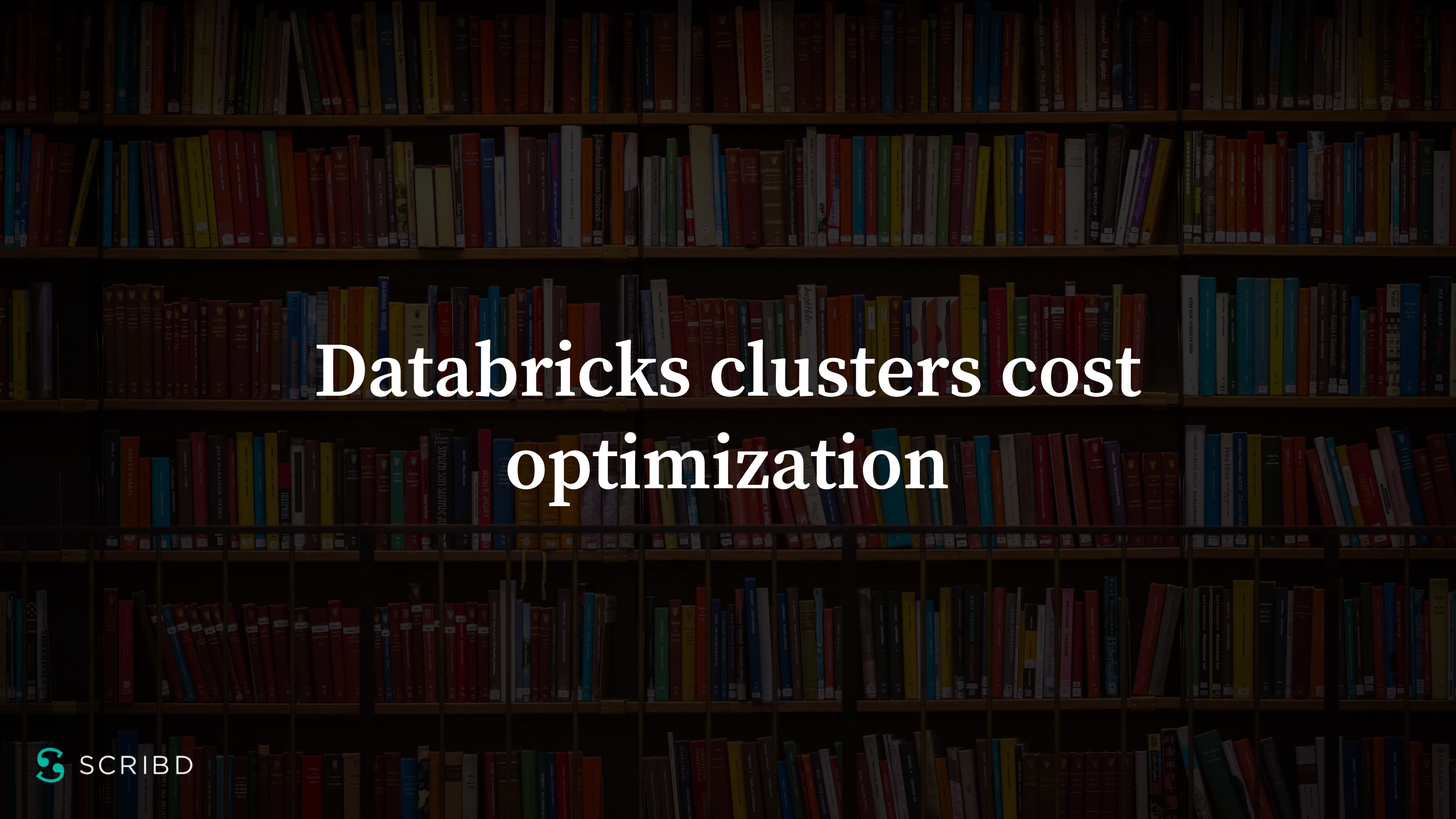
From the Backfill CLI command help output:

`-t TASK_REGEX, --task-regex TASK_REGEX The regex to filter specific task_ids to backfill (optional)`

If you made a typo and typed `--task-regex task3`

You will get:





# Databricks clusters cost optimization

# Databricks clusters cost optimization

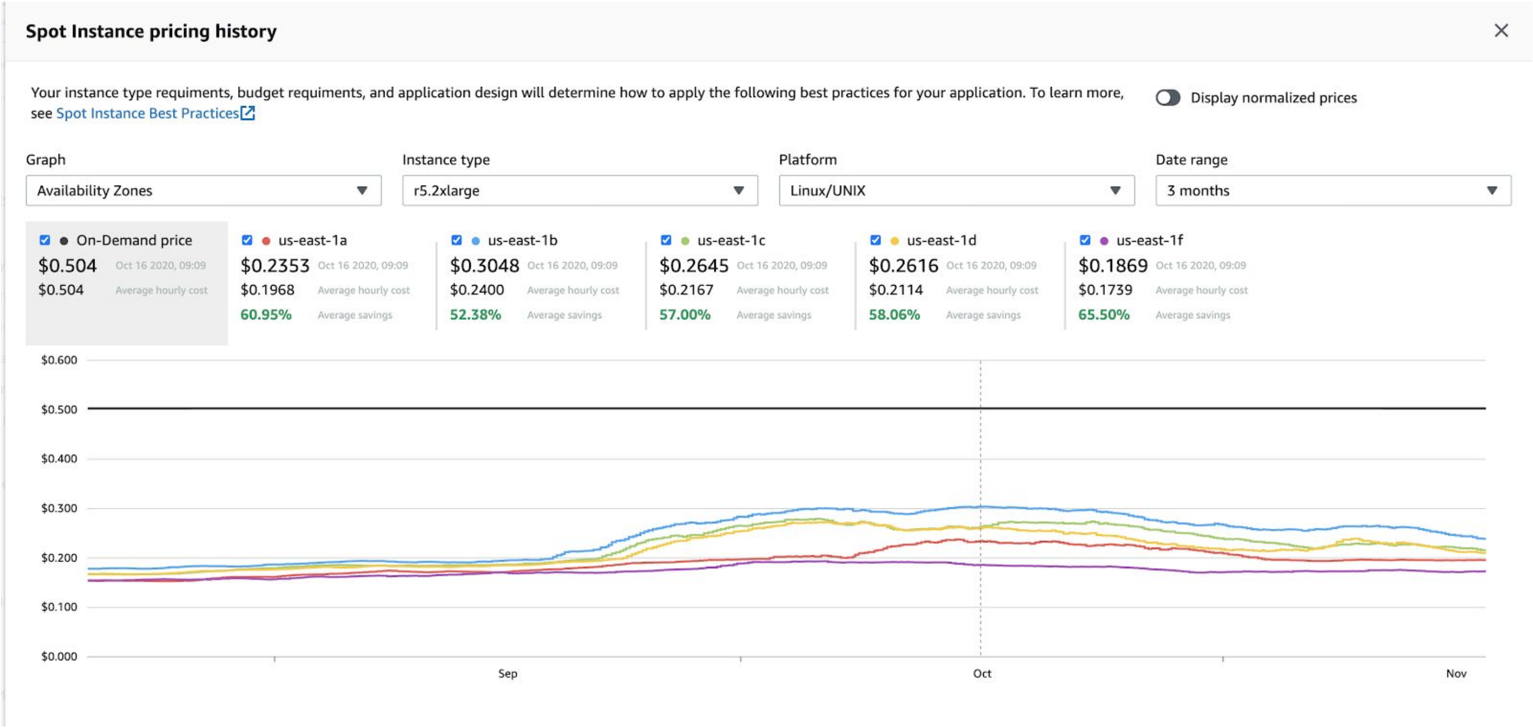
Why?

- AWS has limited number of instances for each AZ

  “We currently do not have sufficient capacity in the Availability Zone you requested”

# Databricks clusters cost optimization

EC2 spot prices across availability zones:



**Spot Instance pricing history**

Your instance type requiments, budget requiments, and application design will determine how to apply the following best practices for your application. To learn more, see Spot Instance Best Practices

Graph: Availability Zones

Instance type: r5.2xlarge

Platform: Linux/UNIX

Date range: 3 months

| ☑ ● On-Demand price | ☑ ● us-east-1a | ☑ ● us-east-1b | ☑ ● us-east-1c | ☑ ● us-east-1d | ☑ ● us-east-1f |
|---|---|---|---|---|---|
| $0.504 Oct 16 2020, 09:09 | $0.2353 Oct 16 2020, 09:09 | $0.3048 Oct 16 2020, 09:09 | $0.2645 Oct 16 2020, 09:09 | $0.2616 Oct 16 2020, 09:09 | $0.1869 Oct 16 2020, 09:09 |
| $0.504 Average hourly cost | $0.1968 Average hourly cost | $0.2400 Average hourly cost | $0.2167 Average hourly cost | $0.2114 Average hourly cost | $0.1739 Average hourly cost |
| | 60.95% Average savings | 52.38% Average savings | 57.00% Average savings | 58.06% Average savings | 65.50% Average savings |

# Databricks clusters cost optimization

How?

- Custom Airflow Databricks operator
- AWS “Describe Spot Price History” API
- Take the cheapest AZ in AWS region
- Fallback to the next cheapest AZ

# Databricks clusters cost optimization

Gain:

- 10-20% cost saving
- Reduce chances of running into AWS instance limit

Learn More:

https://tech.scribd.com/blog/2020/optimize-databricks-cluster-configuration.html

# Q&A

tech.scribd.com

(we’re hiring)