**Airflow Plugins**

Airflow was built with the intention of allowing its users to extend and customize its functionality through plugins. The most common types of user-created plugins for Airflow are Operators and Hooks. These plugins make DAGs reusable and simpler to maintain.

To create custom operator, follow the steps:

1. Identify Operators that perform similar functions and can be consolidated
2. Define a new Operator in the plugins folder
3. Replace the original Operators with your new custom one, re-parameterize, and instantiate them.

**Airflow Contrib**

Airflow has a rich and vibrant open source community. This community is constantly adding new functionality and extending the capabilities of Airflow. As an Airflow user, you should always check [**Airflow contrib**](https://github.com/apache/airflow/tree/master/airflow/contrib) before building your own airflow plugins, to see if what you need already exists.

Operators and hooks for common data tools like Apache Spark and Cassandra, as well as vendor specific integrations for Amazon Web Services, Azure, and Google Cloud Platform can be found in Airflow contrib. If the functionality exists and its not quite what you want, that’s a great opportunity to add that functionality through an open source contribution.

[**Check out Airflow Contrib**](https://github.com/apache/airflow/tree/master/airflow/contrib)

### Task Boundaries

DAG tasks should be designed such that they are:

* Atomic and have a single purpose
* Maximize parallelism
* Make failure states obvious

Every task in your dag should perform **only one job.**

“Write programs that do one thing and do it well.” - Ken Thompson’s Unix Philosophy

##### Benefits of Task Boundaries

* Re-visitable: Task boundaries are useful for you if you revisit a pipeline you wrote after a 6 month absence. You'll have a much easier time understanding how it works and the lineage of the data if the boundaries between tasks are clear and well defined. This is true in the code itself, and within the Airflow UI.
* Tasks that do just one thing are often more easily parallelized. This parallelization can offer a significant speedup in the execution of our DAGs.

**SubDAGs**

Commonly repeated series of tasks within DAGs can be captured as reusable SubDAGs. Benefits include:

* Decrease the amount of code we need to write and maintain to create a new DAG
* Easier to understand the high level goals of a DAG
* Bug fixes, speedups, and other enhancements can be made more quickly and distributed to all DAGs that use that SubDAG

**Drawbacks of Using SubDAGs**

* Limit the visibility within the Airflow UI
* Abstraction makes understanding what the DAG is doing more difficult
* Encourages premature optimization

**Common Questions**

**Can Airflow nest subDAGs?** - Yes, you can nest subDAGs. However, you should have a really good reason to do so because it makes it much harder to understand what's going on in the code. Generally, subDAGs are not necessary at all, let alone subDAGs within subDAGs.

### Pipeline Monitoring

Airflow can surface metrics and emails to help you stay on top of pipeline issues.

#### SLAs

Airflow DAGs may optionally specify an SLA, or “Service Level Agreement”, which is defined as **a time by which a DAG must complete.** For time-sensitive applications these features are critical for developing trust amongst your pipeline customers and ensuring that data is delivered while it is still meaningful. Slipping SLAs can also be **early indicators of performance problems**, or a need to scale up the size of your Airflow cluster

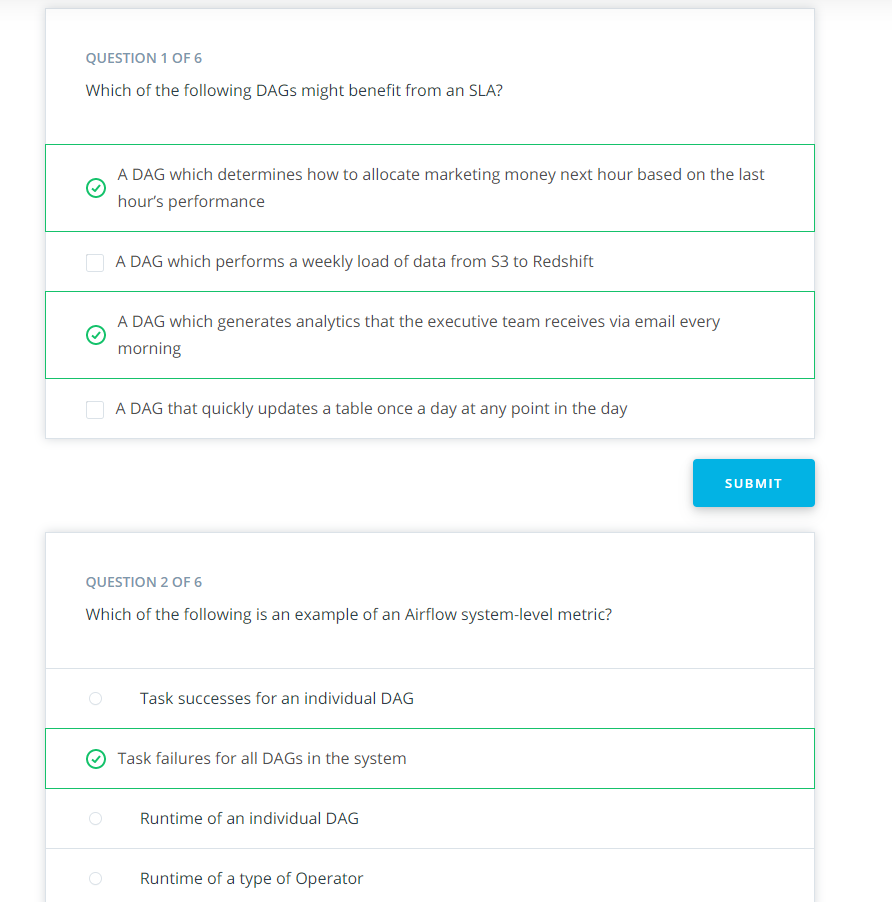
#### Emails and Alerts

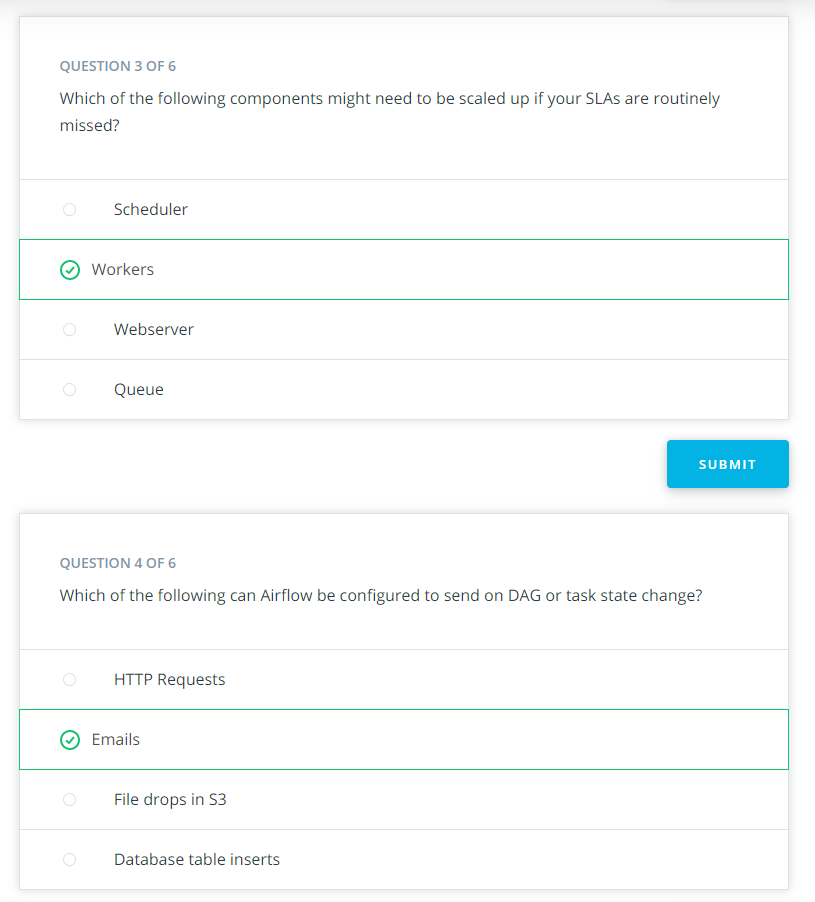
Airflow can be configured to send emails on DAG and task state changes. These state changes may include successes, failures, or retries. Failure emails can allow you to easily trigger alerts. It is common for alerting systems like PagerDuty to accept emails as a source of alerts. If a mission-critical data pipeline fails, you will need to know as soon as possible to get online and get it fixed.

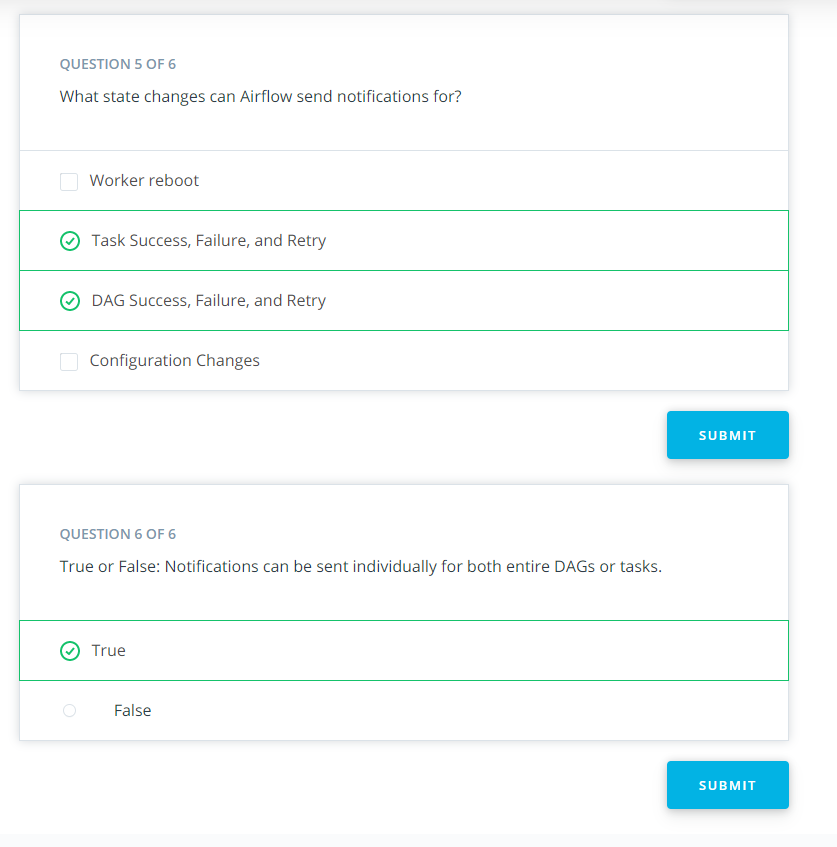
#### Metrics

Airflow comes out of the box with the ability to send system metrics using a metrics aggregator called statsd. Statsd can be coupled with metrics visualization tools like [Grafana](https://grafana.com/) to provide you and your team high level insights into the overall performance of your DAGs, jobs, and tasks. These systems can be integrated into your alerting system, such as pagerduty, so that you can ensure problems are dealt with immediately. These Airflow system-level metrics allow you and your team to stay ahead of issues before they even occur by watching long-term trends.

NEXT







## Here is the Solution for Exercise 4: Building a Full Pipeline

### This is the solution code for exercise4.py

**import** datetime

**from** airflow **import** DAG

**from** airflow.operators **import** (

FactsCalculatorOperator,

HasRowsOperator,

S3ToRedshiftOperator

)

*#*

*# The following DAG performs the following functions:*

*#*

*# 1. Loads Trip data from S3 to RedShift*

*# 2. Performs a data quality check on the Trips table in RedShift*

*# 3. Uses the FactsCalculatorOperator to create a Facts table in Redshift*

*# a. \*\*NOTE\*\*: to complete this step you must complete the FactsCalcuatorOperator*

*# skeleton defined in plugins/operators/facts\_calculator.py*

*#*

dag = DAG("lesson3.exercise4", start\_date=datetime.datetime.utcnow())

*#*

*# The following code will load trips data from S3 to RedShift. Use the s3\_key*

*# "data-pipelines/divvy/unpartitioned/divvy\_trips\_2018.csv"*

*# and the s3\_bucket "udacity-dend"*

*#*

copy\_trips\_task = S3ToRedshiftOperator(

task\_id="load\_trips\_from\_s3\_to\_redshift",

dag=dag,

table="trips",

redshift\_conn\_id="redshift",

aws\_credentials\_id="aws\_credentials",

s3\_bucket="udacity-dend",

s3\_key="data-pipelines/divvy/unpartitioned/divvy\_trips\_2018.csv"

)

*#*

*# Data quality check on the Trips table*

*#*

check\_trips = HasRowsOperator(

task\_id="check\_trips\_data",

dag=dag,

redshift\_conn\_id="redshift",

table="trips"

)

*#*

*# We use the FactsCalculatorOperator to create a Facts table in RedShift. The fact column is*

*# `tripduration` and the groupby\_column is `bikeid`*

*#*

calculate\_facts = FactsCalculatorOperator(

task\_id="calculate\_facts\_trips",

dag=dag,

redshift\_conn\_id="redshift",

origin\_table="trips",

destination\_table="trips\_facts",

fact\_column="tripduration",

groupby\_column="bikeid"

)

*#*

*# Task ordering for the DAG tasks*

*#*

copy\_trips\_task >> check\_trips

check\_trips >> calculate\_facts

### This is the solution code for the Custom Operator: facts\_calculator

**import** logging

**from** airflow.hooks.postgres\_hook **import** PostgresHook

**from** airflow.models **import** BaseOperator

**from** airflow.utils.decorators **import** apply\_defaults

**class** **FactsCalculatorOperator**(BaseOperator):

facts\_sql\_template = """

DROP TABLE IF EXISTS {destination\_table};

CREATE TABLE {destination\_table} AS

SELECT

{groupby\_column},

MAX({fact\_column}) AS max\_{fact\_column},

MIN({fact\_column}) AS min\_{fact\_column},

AVG({fact\_column}) AS average\_{fact\_column}

FROM {origin\_table}

GROUP BY {groupby\_column};

"""

@apply\_defaults

**def** **\_\_init\_\_**(self,

redshift\_conn\_id="",

origin\_table="",

destination\_table="",

fact\_column="",

groupby\_column="",

\*args, \*\*kwargs):

super(FactsCalculatorOperator, self).\_\_init\_\_(\*args, \*\*kwargs)

self.redshift\_conn\_id = redshift\_conn\_id

self.origin\_table = origin\_table

self.destination\_table = destination\_table

self.fact\_column = fact\_column

self.groupby\_column = groupby\_column

**def** **execute**(self, context):

redshift = PostgresHook(postgres\_conn\_id=self.redshift\_conn\_id)

facts\_sql = FactsCalculatorOperator.facts\_sql\_template.format(

origin\_table=self.origin\_table,

destination\_table=self.destination\_table,

fact\_column=self.fact\_column,

groupby\_column=self.groupby\_column

)

redshift.run(facts\_sql)

NEXT

## Other Pipeline Orchestrators

Here are some resources to explore other data pipeline orchestrators.

1. This [**Github link**](https://github.com/pditommaso/awesome-pipeline) contains perhaps way too many examples, but it shows a nice list of other pipeline orchestrators.
2. You can also check out these pages to see how Airflow's components can be generalized to the elements of other pipeline orchestrators.
   * [**Quora page**](https://www.quora.com/Which-is-a-better-data-pipeline-scheduling-platform-Airflow-or-Luigi)
   * [**Github link**](https://xunnanxu.github.io/2018/04/13/Workflow-Processing-Engine-Overview-2018-Airflow-vs-Azkaban-vs-Conductor-vs-Oozie-vs-Amazon-Step-Functions/)
   * [**Medium post**](https://medium.com/@cyrusv/luigi-vs-airflow-vs-zope-wfmc-comparison-of-open-source-workflow-engines-de5209e6dac1)