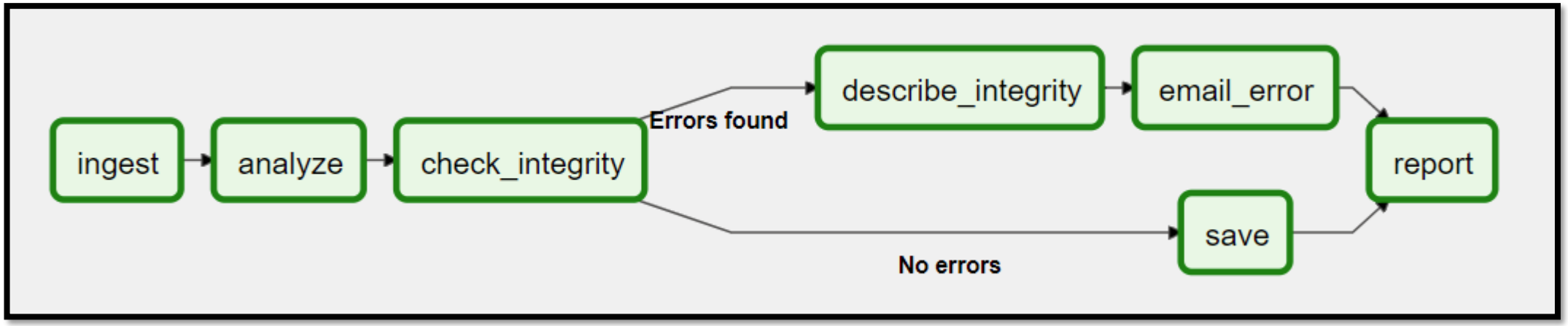
What is Airflow?

Apache Airflow

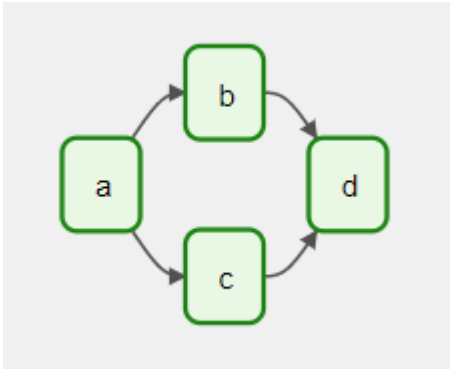
- Airflow is a platform, **built with Python**, that lets you build and run workflows, **programmatically using Python.**
  - A workflow is represented as a DAG (a Directed Acyclic Graph)
    - DAG contains individual tasks, arranged with dependencies
- Directed Acyclic Graph is a specialized version of a Directed Graph which contains no cycles (i.e., no loops)



DAG

- A *DAG* (Directed Acyclic Graph) is the core concept of Airflow, collecting Tasks together, organized with dependencies and relationships to say how they should run.

- This simple DAG defines four Tasks - A, B, C, and D - and dictates:
  - the order in which they have to run, and
  - which tasks depend on what other tasks.
- It will also say how often to run the DAG - maybe “every 5 minutes starting tomorrow”, or “every day since January 1st, 2020”.



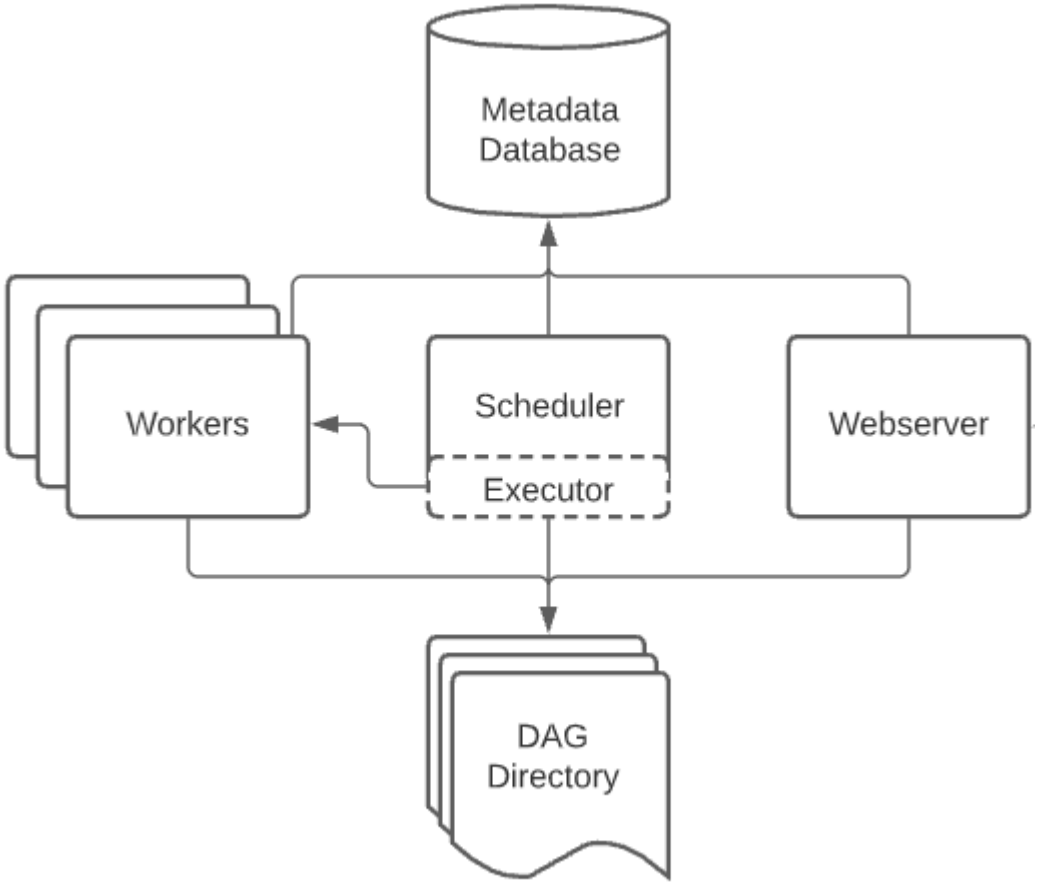
Note:

- The DAG itself doesn’t care about *what* is happening inside the tasks; it is merely concerned with *how* to execute them - the order to run them in, how many times to retry them, if they have timeouts, and so on

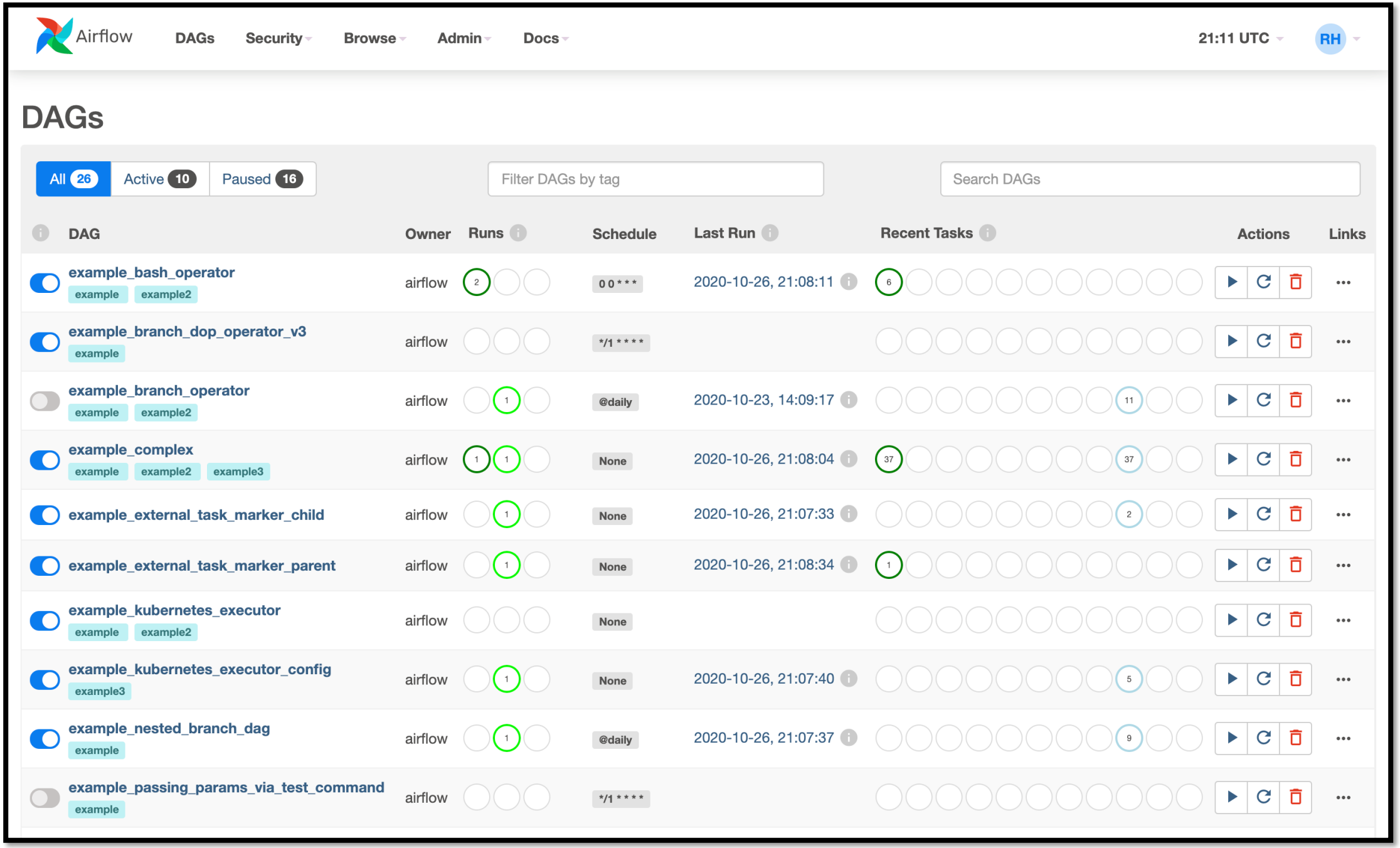
Core Components

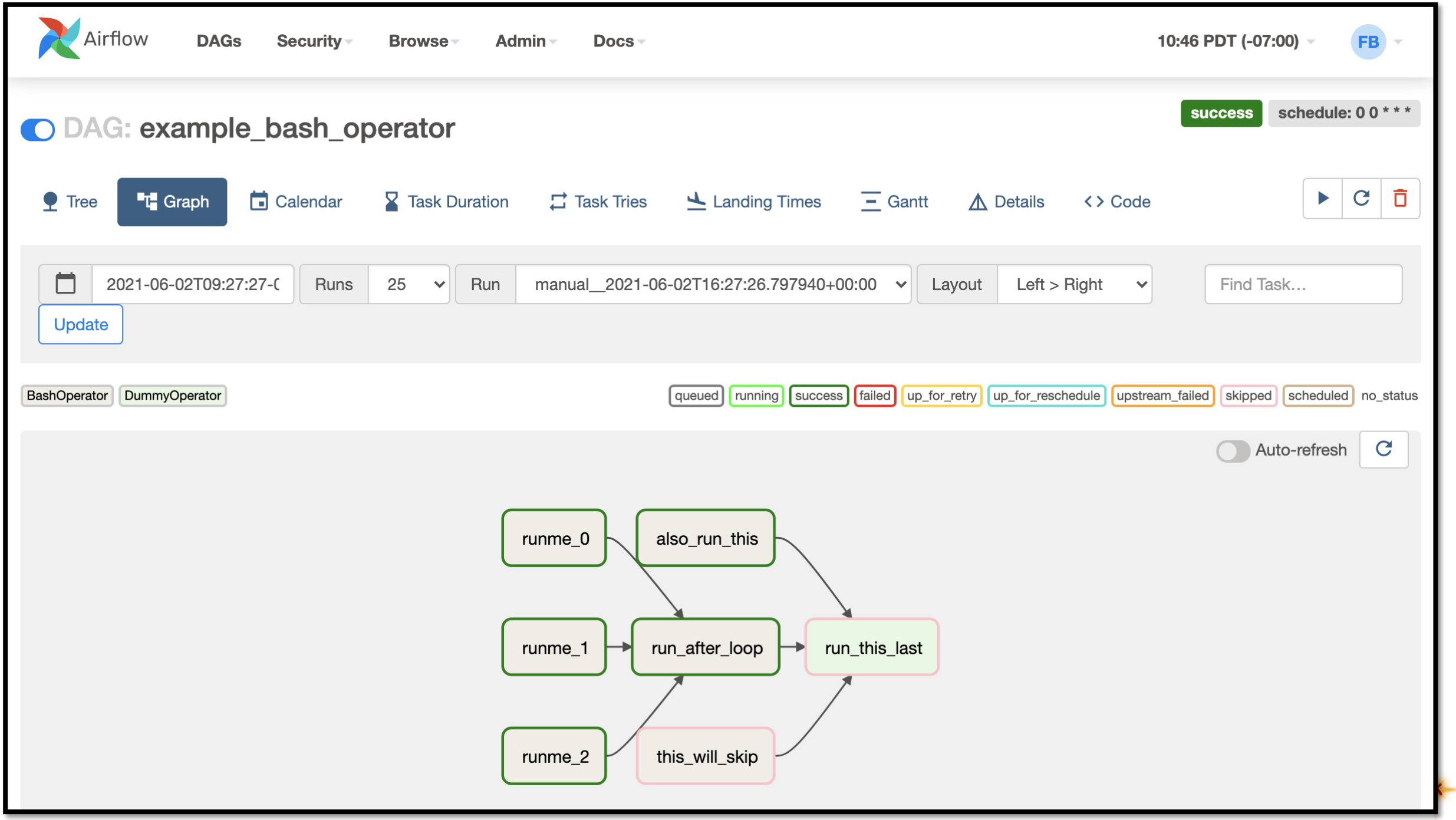
Airflow Components

- A scheduler, which handles both triggering scheduled workflows, and submitting Tasks to the executor to run.
- An executor, which handles running tasks.
- A *webserver*, for Airflow UI to inspect, trigger and debug the behaviour of DAGs and tasks.
- A folder of *DAG files*, read by the scheduler and executor (and any workers the executor has)
- A *metadata database*, used by the scheduler, executor and webserver to store state.



Airflow UI


Airflow
DAGs
Security
Browse
Admin
Docs
21:11 UTC
RH
DAGs
All 26
Active 10
Paused 16
Filter DAGs by tag
Search DAGs
DAG
Owner
Runs
Schedule
Last Run
Recent Tasks
Actions
Links
example_bash_operator
example
example2
airflow
0 0 * * *
2020-10-26, 21:08:11
example_branch_dop_operator_v3
example
airflow
*/1 * * * *
example_branch_operator
example
example2
airflow
@daily
2020-10-23, 14:09:17
example_complex
example
example2
example3
airflow
None
2020-10-26, 21:08:04
example_external_task_marker_child
airflow
None
2020-10-26, 21:07:33
example_external_task_marker_parent
airflow
None
2020-10-26, 21:08:34
example_kubernetes_executor
example
example2
airflow
None
example_kubernetes_executor_config
example3
airflow
None
2020-10-26, 21:07:40
example_nested_branch_dag
example
airflow
@daily
2020-10-26, 21:07:37
example_passing_params_via_test_command
example
airflow
*/1 * * * *


Airflow
DAGs
Security
Browse
Admin
Docs
10:46 PDT (-07:00)
FB
DAG: example_bash_operator
success
schedule: 0 0 * * *
Tree
Graph
Calendar
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
2021-06-02T09:27:27-(
Runs
25
Run
manual__2021-06-02T16:27:26.797940+00:00
Layout
Left > Right
Find Task...
Update
BashOperator
DummyOperator
queued
running
success
failed
up_for_retry
up_for_reschedule
upstream_failed
skipped
scheduled
no_status
Auto-refresh
runme_0
also_run_this
runme_1
run_after_loop
run_this_last
runme_2
this_will_skip

Airflow DAGs Security Browse Admin Docs 10:46 PDT (-07:00) FB

DAG: example_bash_operator

schedule: 0 0 * * *

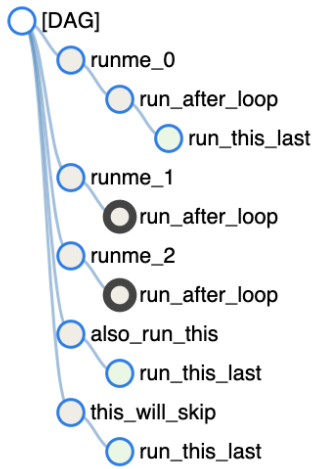
Tree | Graph | Calendar | Task Duration | Task Tries | Landing Times | Gantt | Details | Code

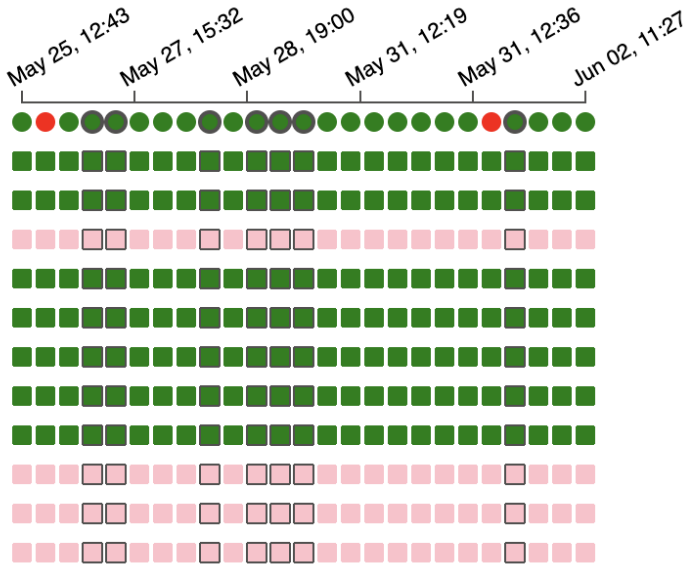
2021-06-02T09:27:26-(Runs 25 Update

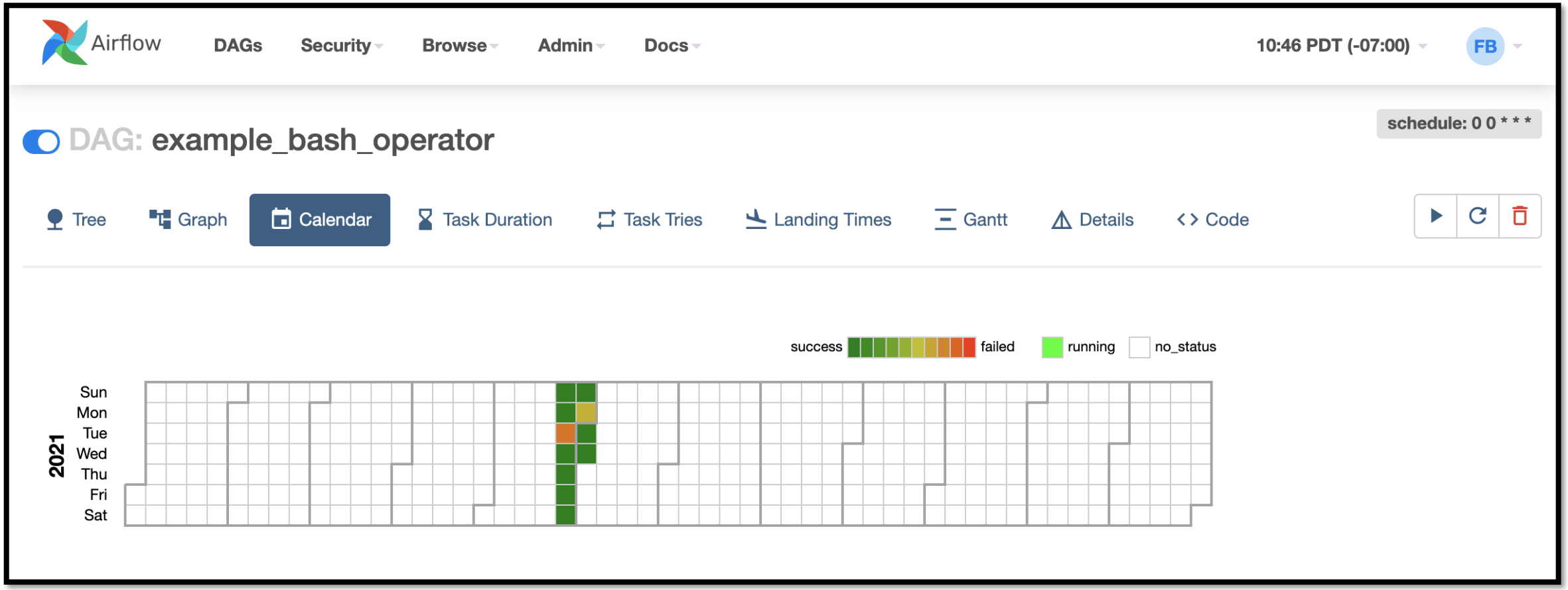
BashOperator DummyOperator

queued running success failed up_for_retry up_for_reschedule upstream_failed skipped scheduled no_status

Auto-refresh






Airflow
DAGs
Security
Browse
Admin
Docs
10:46 PDT (-07:00)
FB
DAG: example_bash_operator
schedule: 0 0 * * *
Tree
Graph
Calendar
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
success
failed
running
no_status
2021
Sun
Mon
Tue
Wed
Thu
Fri
Sat

Building DAGs

```
dag = DAG(
            dag_id = "download_rocker_launches",
            start_date = airflow.utils.dates.days_ago(30),
            schedule_interval = "@daily", )
```

```
dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
)
```

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}
```

DAG Runs (schedule_intervals)

- Each DAG may or may not have a schedule (**why not?**), which informs how DAG Runs are created.
  - **`schedule_interval`** is defined as a DAG argument, and receives preferably a cron expression as a **`str`**, or a **`datetime.timedelta`** object.

preset	meaning	cron
`None`	Don't schedule, use for exclusively "externally triggered" DAGs	
`@once`	Schedule once and only once	
`@hourly`	Run once an hour at the beginning of the hour	`0 * * * *`
`@daily`	Run once a day at midnight	`0 0 * * *`
`@weekly`	Run once a week at midnight on Sunday morning	`0 0 * * 0`
`@monthly`	Run once a month at midnight of the first day of the month	`0 0 1 * *`
`@quarterly`	Run once a quarter at midnight on the first day	`0 0 1 */3 *`
`@yearly`	Run once a year at midnight of January 1	`0 0 1 1 *`

Crontabguru (https://crontab.guru/)

*	any value
,	value list separator
-	range of values
/	step values
@yearly	(non-standard)
@annually	(non-standard)
@monthly	(non-standard)
@weekly	(non-standard)
@daily	(non-standard)
@hourly	(non-standard)
@reboot	(non-standard)

SPARK

Operators

- While DAGs describe how to run a workflow, **Operators** determine what actually gets done by a task.
- The DAG will make sure that operators run in the correct order
  - Once the dependencies are met, operators generally run independently.
  - In fact, they may run on two completely different machines.
  - If two operators need to share information, (e.g., small amount data, variable name, etc), consider combining them into a single operator.
  - If it absolutely can't be avoided, Airflow does have a feature for operator cross-communication called XCom

Operators (cont…)

Airflow provides operators for many common tasks, including:

- **`BashOperator`** - executes a bash command
- **`PythonOperator`** - calls an arbitrary Python function
- **`EmailOperator`** - sends an email
- **`SimpleHttpOperator`** - sends an HTTP request
- **`MySqlOperator`**, **`SqliteOperator`**, **`PostgresOperator`**, **`MsSqlOperator`**, **`OracleOperator`**, **`JdbcOperator`**, etc. - executes a SQL command
- `Sensor` - an Operator that waits (polls) for a certain time, file, database row, S3 key, etc…

In addition to these basic building blocks, there are many more specific operators:
`DockerOperator`, **`HiveOperator`**, **`S3FileTransformOperator`**, **`PrestoToMySqlTransfer`**, **`SlackAPIOperator`** … you get the idea!

Airflow Tasks

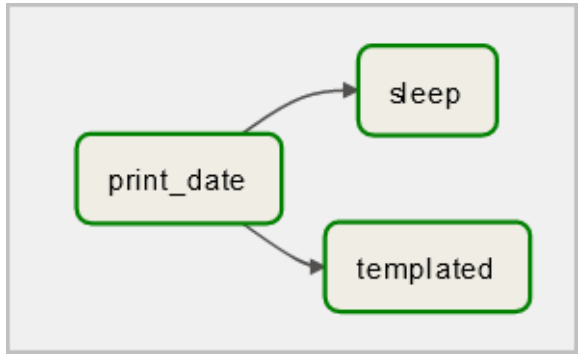
- A Task is the basic unit of execution in Airflow
  - Tasks instantiate Airflow Operators or Sensors (Triggers) to execute predefined functionality
  - Tasks are arranged (i.e., added/inserted) into DAGs

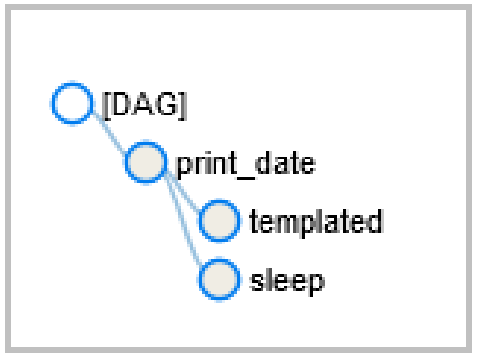
```
task_print_date = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag = dag,
)
```

```
templated_task = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag = dag,
)
```

  - Upstream and downstream dependencies are set between the tasks into order to express the order they should run in.
    - First declare the tasks, then declare its dependencies

```
task_print_date >> [task_sleep, templated_task]
```





Sensors

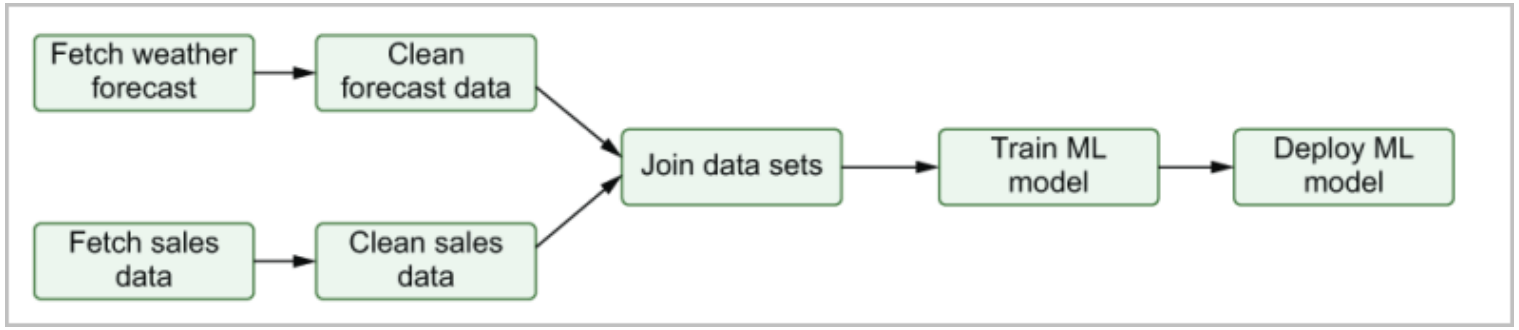
- Sensors are a special type of Operator that are designed to do exactly one thing - wait for something to occur.
- It can be **time-based**, or **waiting for a file**, or an **external event**, but all they do is wait until something happens, and then succeed so their downstream tasks can run.

https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/sensors/index.html

- airflow.sensors.base
- airflow.sensors.base_sensor_operator
- airflow.sensors.bash
- airflow.sensors.date_time
- airflow.sensors.date_time_sensor
- airflow.sensors.external_task
- airflow.sensors.external_task_sensor
- airflow.sensors.filesystem
- airflow.sensors.hdfs_sensor
- airflow.sensors.hive_partition_sensor
- airflow.sensors.http_sensor
- airflow.sensors.metastore_partition_sensor
- airflow.sensors.named_hive_partition_sensor

- airflow.sensors.python
- airflow.sensors.s3_key_sensor
- airflow.sensors.s3_prefix_sensor
- airflow.sensors.smart_sensor
- airflow.sensors.sql
- airflow.sensors.sql_sensor
- airflow.sensors.time_delta
- airflow.sensors.time_delta_sensor
- airflow.sensors.time_sensor
- airflow.sensors.web_hdfs_sensor
- airflow.sensors.weekday

DAG Workflows



- Linear Dependency: 2 Workflow Path executing in parallel:

```
fetch_weather_forecast >> clean_weather_forecast
fetch_sales_data >> clean_sales_data
```

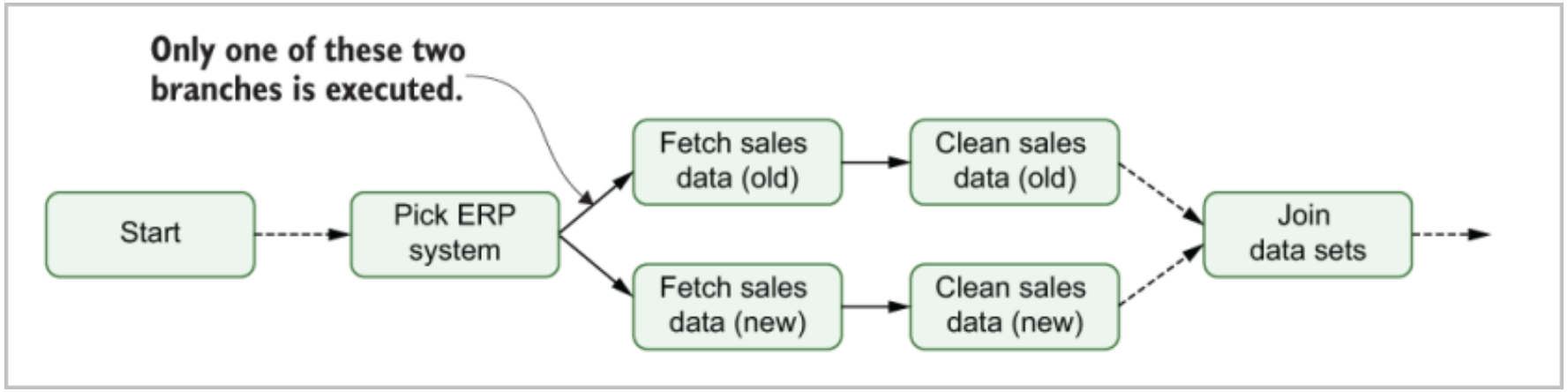
- Fan-out: One-2-Many Dependency (implicit in our case)

```
start = DummyOperator(task_id="start")
start >> [fetch_weather_forecast, fetch_sales_data]
```

- Fan-in: Multiple-2-One Dependency (implicit in our case)

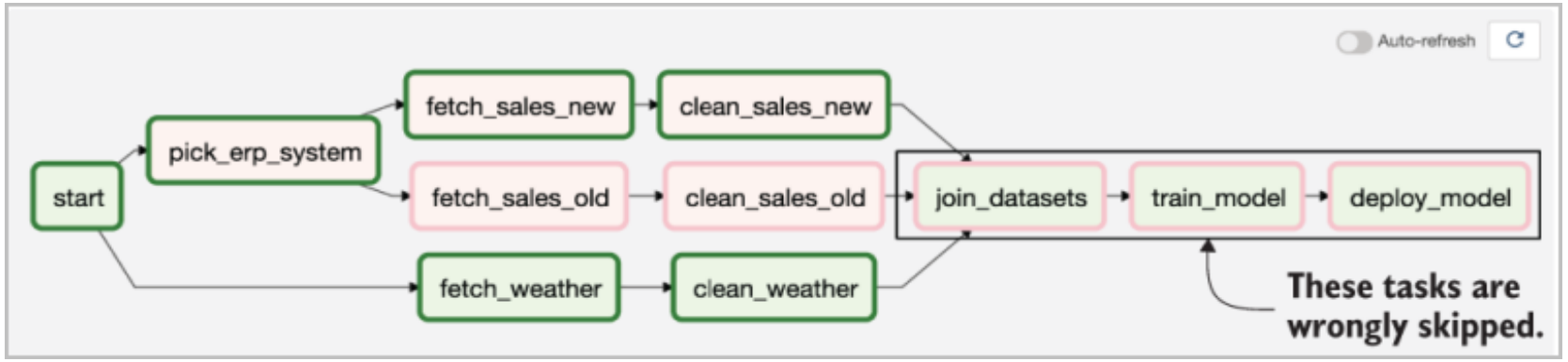
```
[clean_weather_forecast, clean_sales_data] >> join_data_sets
```

Branching



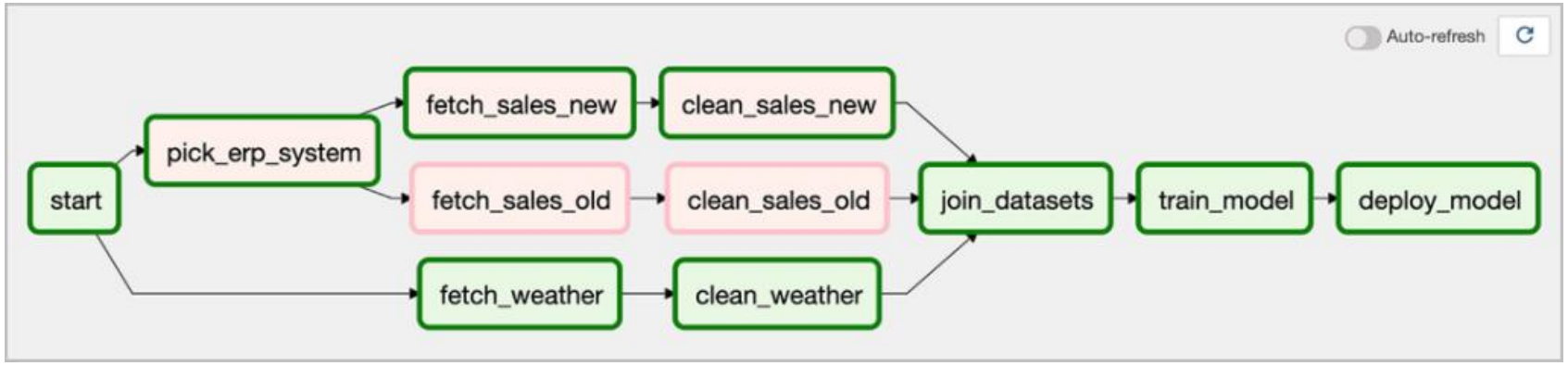

```
def _pick_erp_system(**context):
    if context["execution_date"] < ERP_SWITCH_DATE:
        return "fetch_sales_old"
    else:
        return "fetch_sales_new"

pick_erp_system = BranchPythonOperator(
    task_id="pick_erp_system",
    python_callable=_pick_erp_system,
)

pick_erp_system >> [fetch_sales_old, fetch_sales_new]
```

Trigger Rules



`trigger_rule = "all_success"`

```
join_datasets = PythonOperator(
    ...,
    trigger_rule="none_failed",
)
```



Trigger Rules (cont...)

- `all_success` (default): All upstream tasks have succeeded
- `all_failed` : All upstream tasks are in a `failed` or `upstream_failed` state
- `all_done` : All upstream tasks are done with their execution
- `one_failed` : At least one upstream task has failed (does not wait for all upstream tasks to be done)
- `one_success` : At least one upstream task has succeeded (does not wait for all upstream tasks to be done)
- `none_failed` : All upstream tasks have not `failed` or `upstream_failed` - that is, all upstream tasks have succeeded or been skipped
- `none_failed_min_one_success` : All upstream tasks have not `failed` or `upstream_failed` , and at least one upstream task has succeeded.
- `none_skipped` : No upstream task is in a `skipped` state - that is, all upstream tasks are in a `success` , `failed` , or `upstream_failed` state
- `always` : No dependencies at all, run this task at any time

Task Cross-Communication (xcom)

- Airflow xcom is used to “pass” data between tasks
  - Tasks are completely isolated otherwise
  - Guidance is that it should be used for small amounts of data.
  - Source Code implies that max xcom size is 48kb but is never used
  - https://airflow.apache.org/docs/apache-airflow/stable/_modules/airflow/models/xcom.html
    - Seems to be dependent on backend database blob size:
    - Sqlite: default max of 1 GB (absolute maximum of 2 GB)
    - Mysql: 64 kb
- **`xcom_push`**: pushes data into xcom
- **`xcom_pull`**: pulls data from xcom

SLA

- Service Level Agreements (SLAs), or the **time** by which a task or DAG should have succeeded, can be set at a task level as a timedelta.
- If one or many instances have not succeeded by that time, an alert email is sent with the list of tasks that missed their SLA.
  - The SLA miss event is also recorded in the database and made available in the web UI under Browse->SLA Misses where events can be analyzed and documented.
- SLAs can be configured for scheduled tasks by using the **`sla`** parameter.
  - In addition to sending alerts, the **sla_miss_callback** specifies an additional method to be invoked when the SLA is not met to invoke any custom functionality

Providers Packages

- Airbyte
- Alibaba
- Amazon
- Apache Beam
- Apache Cassandra
- Apache Drill
- Apache Druid
- Apache HDFS
- Apache Hive
- Apache Kylin
- Apache Livy
- Apache Pig
- Apache Pinot
- Apache Spark
- Apache Sqoop
- Asana
- Celery
- IBM Cloudant
- Kubernetes
- Databricks
- Datadog
- Dingding
- Discord
- Docker
- Elasticsearch
- Exasol
- Facebook
- File Transfer Protocol (FTP)
- Google
- gRPC
- Hashicorp
- Hypertext Transfer Protocol (HTTP)
- Influx DB
- Internet Message Access Protocol (IMAP)
- Java Database Connectivity (JDBC)
- Jenkins
- Jira
- Microsoft Azure
- Microsoft PowerShell Remoting Protocol (PSRP)
- Microsoft SQL Server (MSSQL)
- Windows Remote Management (WinRM)
- MongoDB
- MySQL
- Neo4J
- ODBC
- OpenFaaS
- Opsgenie
- Oracle
- Pagerduty
- Papermill
- Plexus
- PostgreSQL
- Presto
- Qubole
- Redis
- Salesforce
- Samba
- Segment
- Sendgrid
- SFTP
- Singularity
- Slack
- Snowflake
- SQLite
- SSH
- Tableau
- Telegram
- Trino
- Vertica
- Yandex
- Zendesk