Airflow at Shopify

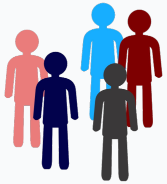
Keeping Users Happy While Running Airflow at Scale

Sam Wheating

Megan Parker

Developers on the Data Foundations team

Why are we here?



Operations Team
(This is us)

Airflow

There's only a few of us, we have a lot of users
Users have high expectations and diverse requirements

How can we keep our users happy?

1) Make Airflow reliable
2) Make Airflow easy to use

Contents

How we use Airflow

10,000+

Dags in a single environment

7,000,000

Tasks executed every month

~3 second

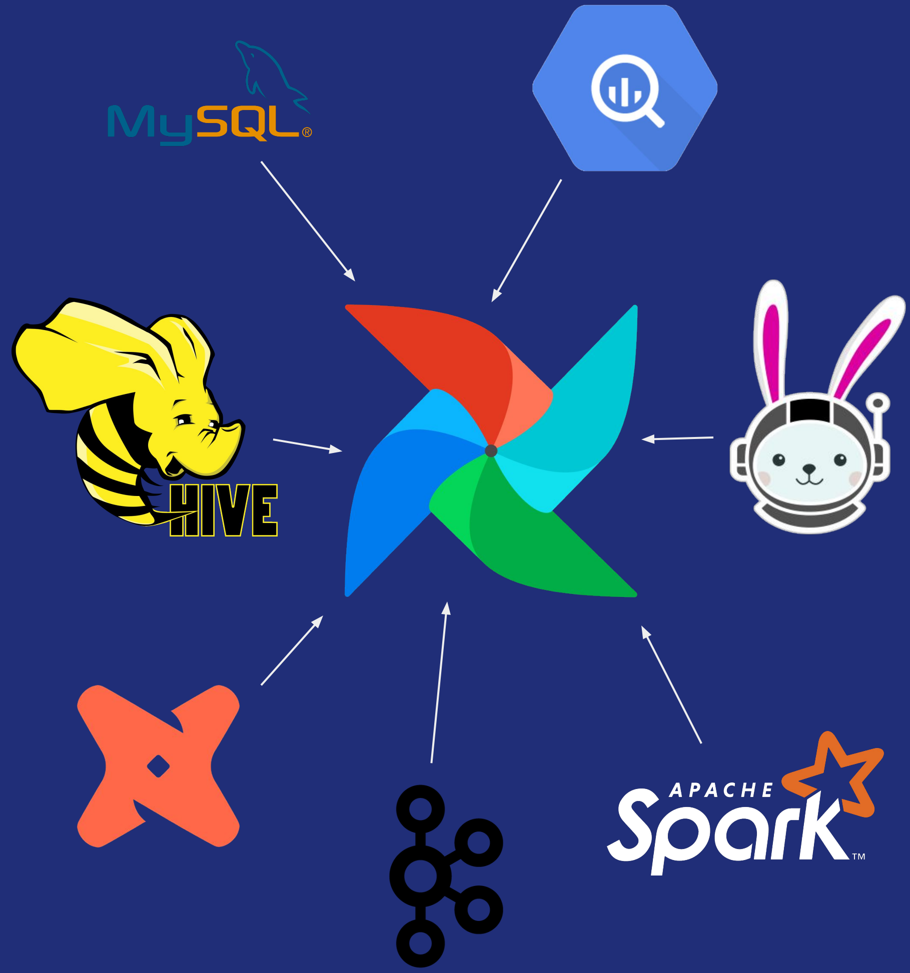
First task start latency

~3

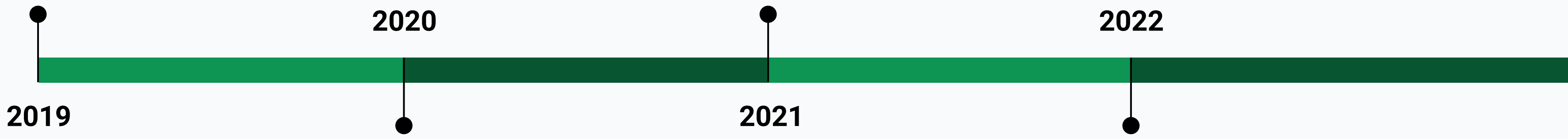
Infrastructure engineers managing Airflow

What runs on Airflow?

- Data ingestion (Streams, DBs and APIs)
- Trino and BigQuery-powered DBT models
- ML Model Training
- Machine Learning Offline Inference
- Apache Iceberg Table Maintenance
- Data Expiration and Deletion
- Data Aggregations and Loads



History of Airflow at Shopify



Moving towards a unified scheduler

Running Oozie, Azkaban and some single-team Airflow instances
Decided to commit to Airflow as a unified scheduling solution

Centralized Airflow environment

Started to move decentralized jobs to a centralized Airflow environment

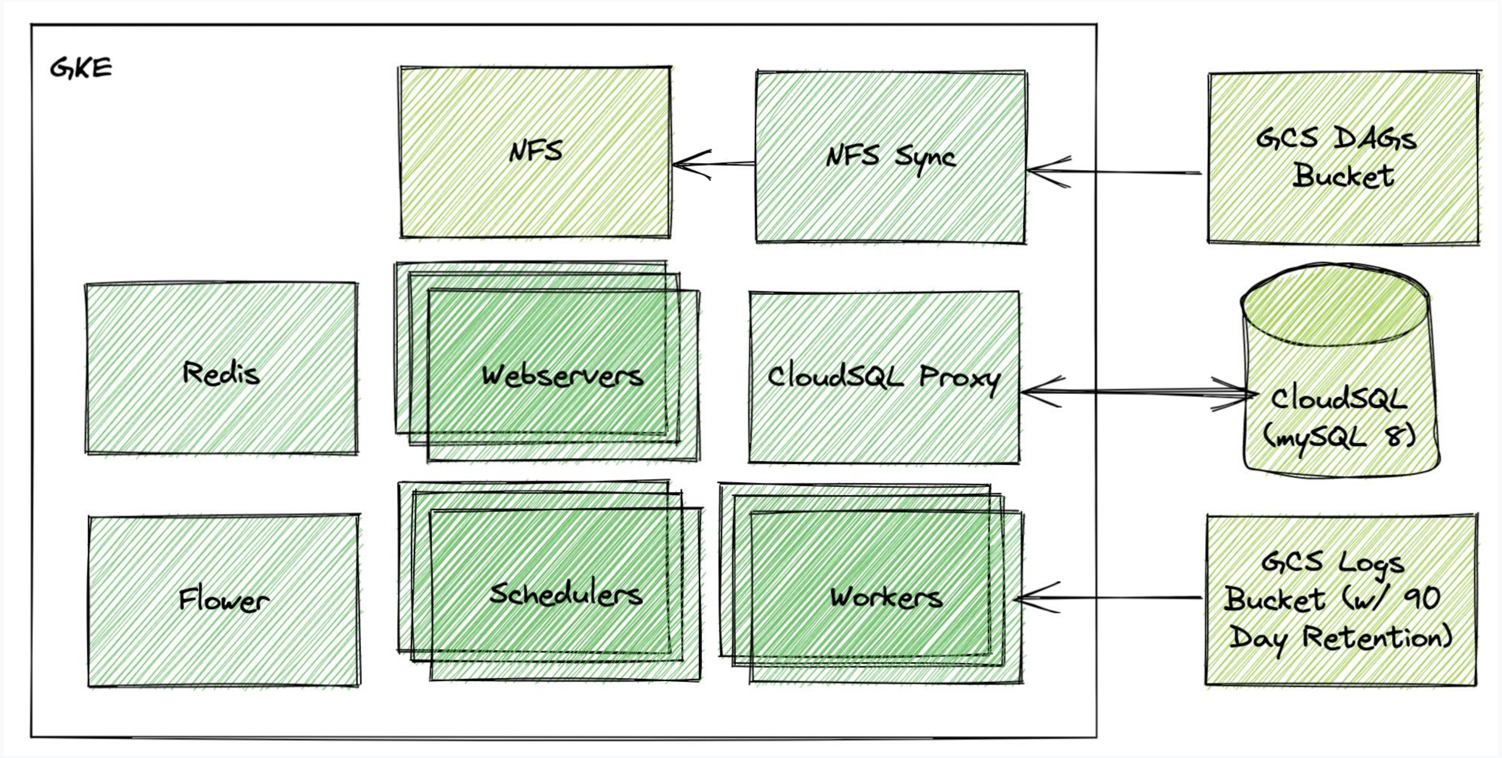
Airflow adoption

Upgraded Airflow to 2.0. Completion of Azkaban migration

Airflow is status quo

Completion of Oozie migration (Exact date TBD)

Airflow Architecture



Airflow Reliability
(Making our Job Easier)

Case Study 1: Testing

“How do I know that this Airflow upgrade won’t break things?”

Testing, Testing, Testing

Thorough test coverage can be used to minimize disruptions and prevent errors.

We test our Airflow environment in multiple ways to ensure smooth operation for our users.

	Unit Tests	**Smoke Tests**	**Load Tests**
What are we Testing?	DAGs and Operators	Operators	Airflow Infrastructure
Written by?	DAG + Operator Authors	Us	Us
When are they run?	Constantly	During Upgrades	During Upgrades

Unit Tests

Test the basic functionality of all DAGs and Operators.

It can be pretty hard to test the functionality of a DAG end-to-end, but we can at least make assertions about the structure and composition of the job.

Other good things to test:

- That there’s no duplicated DAG IDs
- That all DAGs are importable
- That any DAG Factory or DAG templating code is generating valid DAGs.

This is also a good time to pick up on transitive dependency issues[1]

[1] You should pre-compile the dependencies in your production containers!

Smoke Tests

Airflow upgrades can introduce changes to operator interfaces as well as underlying libraries.

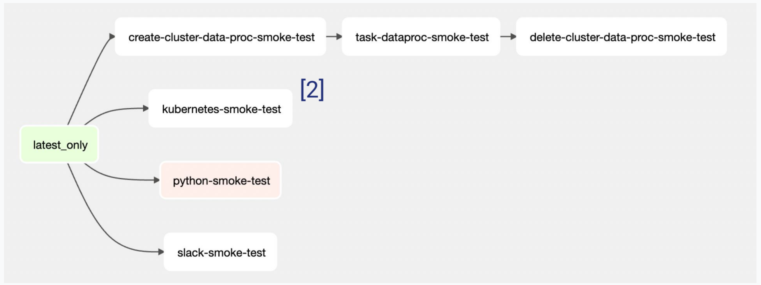
For example, sometimes operators which subclass an upstream operator can be sensitive to changes in the parent class.

How can we ensure that a new upgrade is compatible with ~~all~~ most[1] of our DAGs?

1) Decide on a set of “Supported” operators and components

2) Create a DAG which uses all of these operators in a production-like way

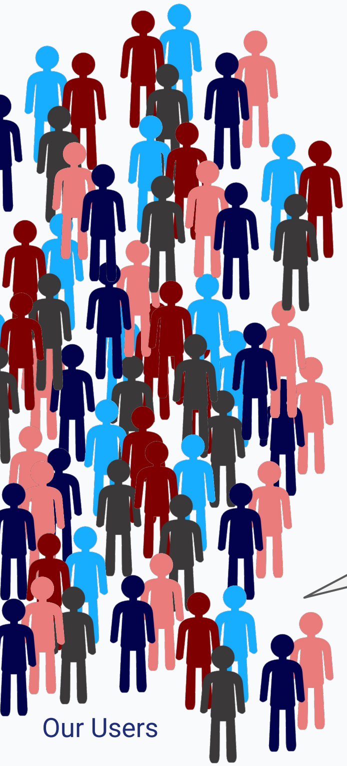
3) Just run this DAG every time something changes.



[1] Without restricting Airflow usage to a set of managed abstractions, it’s not realistic to guarantee no issues after an upgrade

[2] We use the KubernetesPodOperator for the vast majority of our tasks - shout out to Bluecore for that awesome blog post

Load Tests

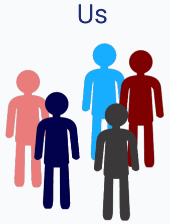


"Will Airflow withstand the increased volumes of data around Black Friday?"

"How does the frequency of jobs correlate to the first-task start latency latency?"

"We're planning on adding 15,000 DAGs this year, is that gonna break anything?"

"Are there any performance regressions in this new version of Airflow, with our scale and infrastructure?"



We need a way to answer these questions in a production-like environment, without the fear of disrupting production workloads.

Load Tests

- Create a randomized, parameterizable ensemble of jobs from a single python file.
- Try to emulate production workloads as closely as possible.
- Scale up the non-production environment to the same scale as the production one.

```
NUM_DAGS = Variable.get('LOAD_TEST.COUNT')
MAX_DAG_DEPTH = Variable.get('LOAD_TEST.MAX_DEPTH')
random.seed(1)

for i in range(NUM_DAGS):
   dag = DAG(f'load-test.dag-{i}', schedule_interval=get_interval())
   for i in range(randint(MAX_DAG_DEPTH))
       task = KubernetesPodOperator(
           dag=dag,
           task_id='task_{i}'
   ...
   globals()[f'dag-{i}'] == dag
```

** abridged to fit on a slide*

Code Sample 1: Creating a parameterized ensemble of DAGs in one file

Gotchas:

- This does not test the capacity of downstream systems, such as your processing infrastructure
- Remember to vary the shape of your DAGs as well, to better emulate a real workload
- Remember to seed your RNG, so that the file processor sees the same DAGs every time.

Case Study 2: Policies and Guardrails

“How do we maintain control over all of these DAGs?”

Multi-tenancy via Airflow Manifests

$AIRFLOW_HOME/airflow_projects.yaml

```
projects:
 data_extracts:
   owner_email: 'etl-team'
   source_repository: 'https://github.com/my_org/extracts'
   constraints:
     namespaces:
       - 'etl-jobs'
     pools:
       - 'extracts'
 batch_processing:
   owner_email: 'spark-team'
   source_repository: 'https://github.com/my_org/batch_jobs'
   constraints:
     namespaces:
       - 'batch'
     pools:
       - 'batch'
```

All workloads have to be registered (it's easy!)

This file is used to determine which files to load to the Airflow environment.

It also allows us to create specifications which each workload must conform to.

A helper script allows users to easily register a new “namespace” in Airflow, define its constraints and specify which environments it is deployed to.

Code Sample 2:
An example of an airflow environment's manifest file with two namespaces.

dag_policy

Implementing a *dag_policy* allows you to conditionally reject DAGs at time of loading.

We read the manifest file within a *dag_policy* function and ensure that DAGs conform to their specification.

$AIRFLOW_HOME/airflow_local_settings.yaml

```
def dag_policy(dag: DAG) -> None:

    airflow_home = os.environ.get('AIRFLOW_HOME', '~/airflow')
    manifest_path = f"{airflow_home}/airflow_manifest.yaml"
    with open(manifest_path, "r", encoding="UTF-8") as manifest_file:
        manifest = yaml.safe_load(manifest_file)

    dag_namespace = dag.dag_id.split(".")[0]
    if dag_namespace not in manifest["projects"]:
        raise AirflowClusterPolicyViolation(
            f"Namespace {dag_namespace} is not registered."
        )

    constraints = manifest["projects"][dag_namespace]["constraints"]

    validate_pools(dag, constraints["pools"])
```

Code Sample 3:
This *dag_policy* will ensure that a DAG is registered in the manifest file and only launches tasks in the permitted pools

Case Study 3: Platform Considerations

“How can we minimize the amount of downtime?”

Separation of Environments

	Production	**Staging**	**Development**
Scale	Large	Small	Small
Who Can Upload?	Continuous Deployment	Everyone	Operators Only
Sends Alerts	Yes	Yes	No
Sends Pages	Yes	No	No

Roll out updates in reverse-priority: Development -> Staging -> Production

Gotchas:

- Version / Provider / Plugin drift between Production and Staging
- Different behaviour performance across environments due to scale

Why use a remote staging environment?

Pros:

- Full auditability of all user activity
- No need to maintain credentials and connections on users' machines
- Provides an environment which is consistent with production
- Faster to get up and running with DAG Development

Cons:

- Creates an additional environment to support
- A single point of failure if (when) someone pushes a troublesome job
- Syntax highlighting and some IDE features don't work well locally

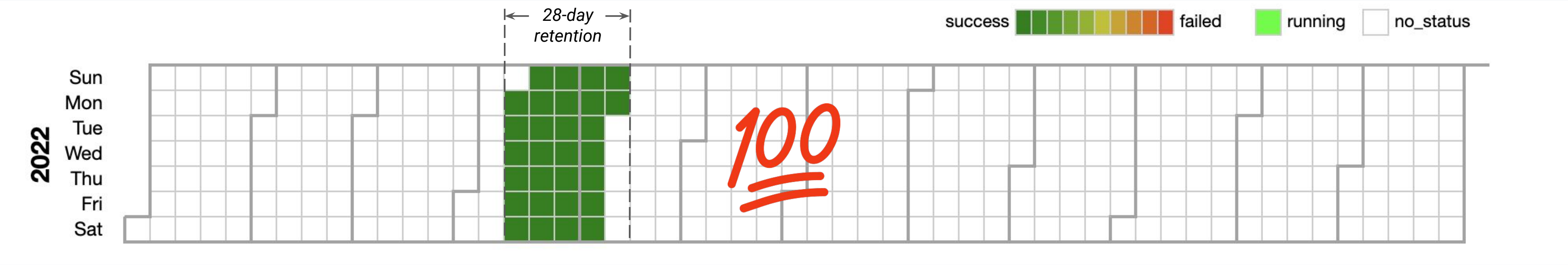
*** Interested in these tradeoffs?**
Check out Lyft's talk later this week to learn about how they create single-user remote development environments.

Monitoring

Use a monitoring DAG!

This DAG runs every minute and emits some basic metrics. If those metrics go missing then we can assume airflow isn't happy.

The calendar view also gives a nice overview of uptime:



Gotchas:

- Ensure that this DAG is running with a high *priority_weight*
- If you're using separate queues, you may need one task per queue.

Metadata Truncation

Large volumes of metadata can **seriously** impede in-place upgrades of Airflow, especially when larger migrations are involved

Just use a DAG to delete old data (AIP-44 will break this!)

Fun Fact: We took down our production Airflow environment for ~3 hours while upgrading from 2.1.2 to 2.2.2

** this is a simplification*

```
with DAG(
    'truncate_metadata',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False
) as dag:

    cleanup_task = PythonOperator(
        task_id='truncate_metadata'
        python_callable=delete_metadata
    )
```

Code Sample 4: Obligatory DAG Code

Gotchas:

- Be careful not to delete dagrun history which is relied on by backfills
- Be careful not to delete “active” data (currently running jobs, DAGRuns, etc)

User Experience
(Making our User’s Job Easier)

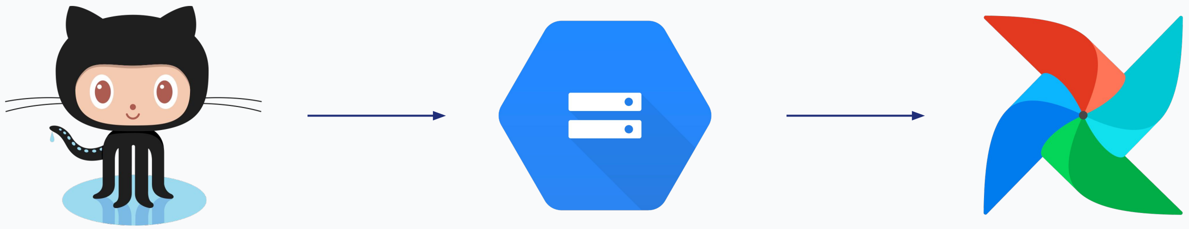
Case Study 4: GitOps

“I need to add a Pool, but I’m not an Admin”

Managing Connections via Github

Airflow connections in Github? It’s easier than you think!

1) Use Shopify’s open source *ejson*[1] to store encrypted sensitive values alongside DAGs.
2) Deploy these changes to your Airflow environment alongside your DAGs.
3) A project loader script running on the scheduler decrypts these files and load connections into the DB



Gotchas:

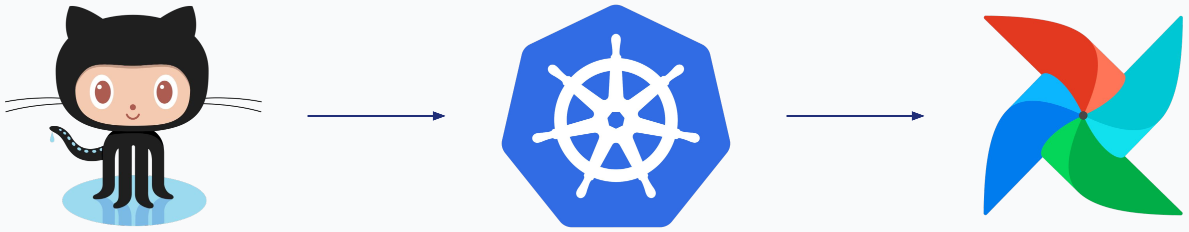
- Use a pre-commit hook to ensure nobody commits unencrypted secrets
- Prefix connections with workspace name to maintain a clear path of ownership

[1] https://github.com/Shopify/ejson

Managing Pools and Users from Github

Similar to before, but with less secrecy. Also these aren't namespace-specific.

1) Specify the connections and admin users in a *.yaml* file which is mounted as a Kubernetes ConfigMap
2) In an Airflow DAG, load up this file and create / update pools and users as required



Gotchas:

- Use CODEOWNERS to prevent users from escalating their own privilege.

Case Study 5: Surf

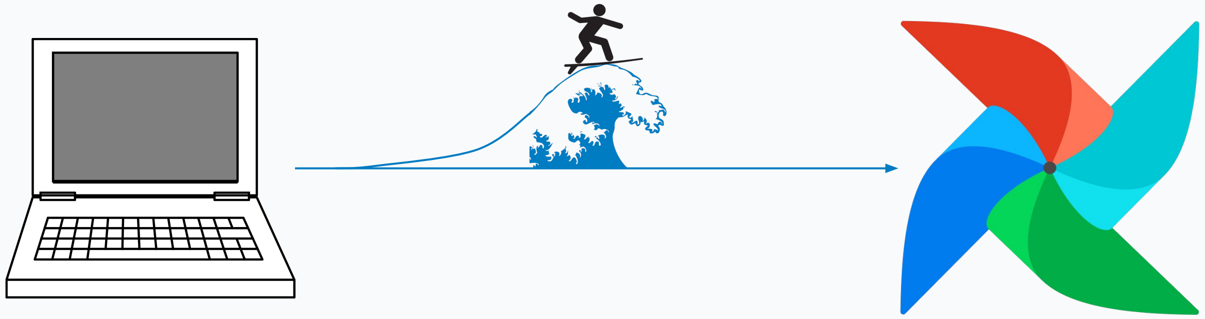
“How can I interact with the remote Environment?”

What is *surf*?

- Opinionated CLI for interacting with remote Airflow environments
- Integrated with a browser-based auth flow for secure API access

What can *surf* do?

- Upload DAGs directly to staging, with immediate user feedback.
- Trigger, List and Monitor DAGs
- Upload Connections to staging
- Pause, Resume or Fail DAGs in bulk



(Partial) Syntax Reference

```
surf dags list <prefix>
```

Lists all DAGs matching the provided prefix

```
surf dags <pause|resume> <prefix>
```

Pause or resume multiple DAGs based on the provided pattern

```
surf dags trigger <dag_id> -d <key>=<value>
```

Trigger the specified DAG, with the option of adding dagrun config

```
surf dags fail <pattern>
```

Mark the most recent in-progress run of a given DAG as failed.

```
surf staging dags push <filepath>
```

Upload the specified file to the staging environment and trigger its import.

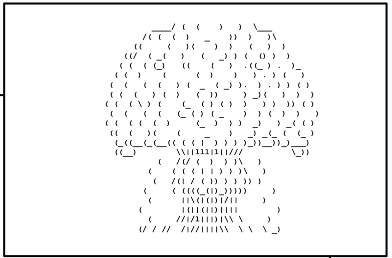
```python
from airflow.models import DAG
from airflow.operators.bash import BashOperator

MESSAGE = """
 _       __     __                             __
| |     / /__  / /________  ____ ___  ___     / /_____
| | /| / / _ \/ / ___/ __ \/ __  __ \/ _ \   / __/ __ \
| |/ |/ /  __/ / /__/ /_/ / / / / / /  __/  / /_/ /_/ /
|__/|__/\___/_/\___/\____/_/ /_/ /_/\___/___\__/\____/                        _ __
   /   |  (_)_____/ __/ /___ _      __   / ___/__  ______ ___  ____ ___  (_) /_
  / /| | / / ___/ /_/ / __ \ | /| / /   \__ \/ / / / __  __ \/ __  __ \/ / __/
 / ___ |/ / /  / __/ / /_/ / |/ |/ /   ___/ / /_/ / / / / / / / / / / / / /_
/_/  |_/_/_/  /_/ /_/\____/|__/|__/   /____/\__,_/_/ /_/ /_/_/ /_/ /_/_/\__/
"""

dag = DAG(
   dag_id='data-infrastructure-examples.my-dag',
   schedule_interval=None,
   catchup=False
)

task = BashOperator(
   dag = dag,
   task_id='say_hello',
   bash_command=f'echo "{MESSAGE}"'
)
```

Code Sample 4: *my_dag.py*

I've been working on this DAG in preparation for Airflow Summit.

I wanna make sure it works, so I'd like to run it in staging ASAP!

Examples:



```
> time surf staging dags push dags/data_infrastructure_examples/my_dag.py

Checking dags/data_infrastructure_examples/my_dag.py for syntax errors
Syntax error check successful!
Starting upload to GCS…
Successfully uploaded DAG to GCS

'Traceback (most recent call last):\n  File
"/usr/local/lib/python3.9/site-packages/airflow/models/baseoperator.py", line 840, in dag\n
dag.add_task(self)\n  File "/usr/local/lib/python3.9/site-packages/airflow/models/dag.py",
line 2139, in add_task\n    raise AirflowException("Task is missing the start_date
parameter")\nairflow.exceptions.AirflowException: Task is missing the start_date parameter\n'

     Your DAG has uploaded with errors, please review the logs and resolve any errors before
uploading again

     2.34s user 0.26s system 62% cpu 4.189 total
```

```python
import pendulum  # needed this
from airflow.models import DAG
from airflow.operators.bash import BashOperator

MESSAGE = """

_       __     __                             __
| |     / /__  / /________  ____ ___  ___     / /_____
| | /| / / _ \/ / ___/ __ \/ __  __ \/ _ \   / __/ __ \
| |/ |/ /  __/ / /__/ /_/ / / / / / /  __/  / /_/ /_/ /
|__/|__/\___/_/\___/\____/_/ /_/ /_/\___/   \__/\____/                    _ __
   /   |  (_)____/ __/ /___ _      __   / ___/__  ______ ___  ____ ___  (_) /_
  / /| | / / ___/ /_/ / __ \ | /| / /   \__ \/ / / / __  __ \/ __  __ \/ / __/
 / ___ |/ / /  / __/ / /_/ / |/ |/ /   ___/ / /_/ / / / / / / / / / / / / /_
/_/  |_/_/_/  /_/ /_/\____/|__/|__/   /____/\__,_/_/ /_/ /_/_/ /_/ /_/_/\__/
"""

dag = DAG(
    dag_id='data-infrastructure-examples.my-dag',
    schedule_interval=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"), # also this
    catchup=False
)

task = BashOperator(
    dag = dag,
    task_id='say_hello',
    bash_command=f'echo "{MESSAGE}"'
)
```

Code Sample 5:

git commit . -m “oops”

Examples:

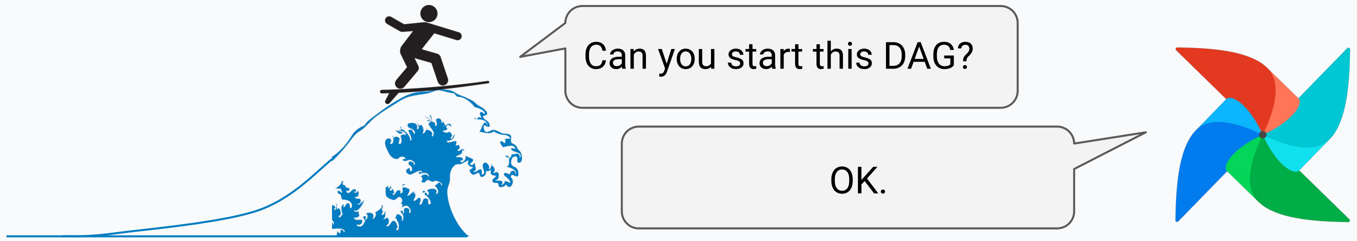
```
> time surf staging dags push dags/data_infrastructure_examples/my_dag.py

Checking dags/data_infrastructure_examples/my_dag.py for syntax errors
Syntax error check successful!
Starting upload to GCS…
Successfully uploaded DAG to GCS
Parsed 1 DAG(s)
Successfully uploaded DAG to Airflow Scheduler

     2.40s user 0.22s system 69% cpu 3.765 total
```

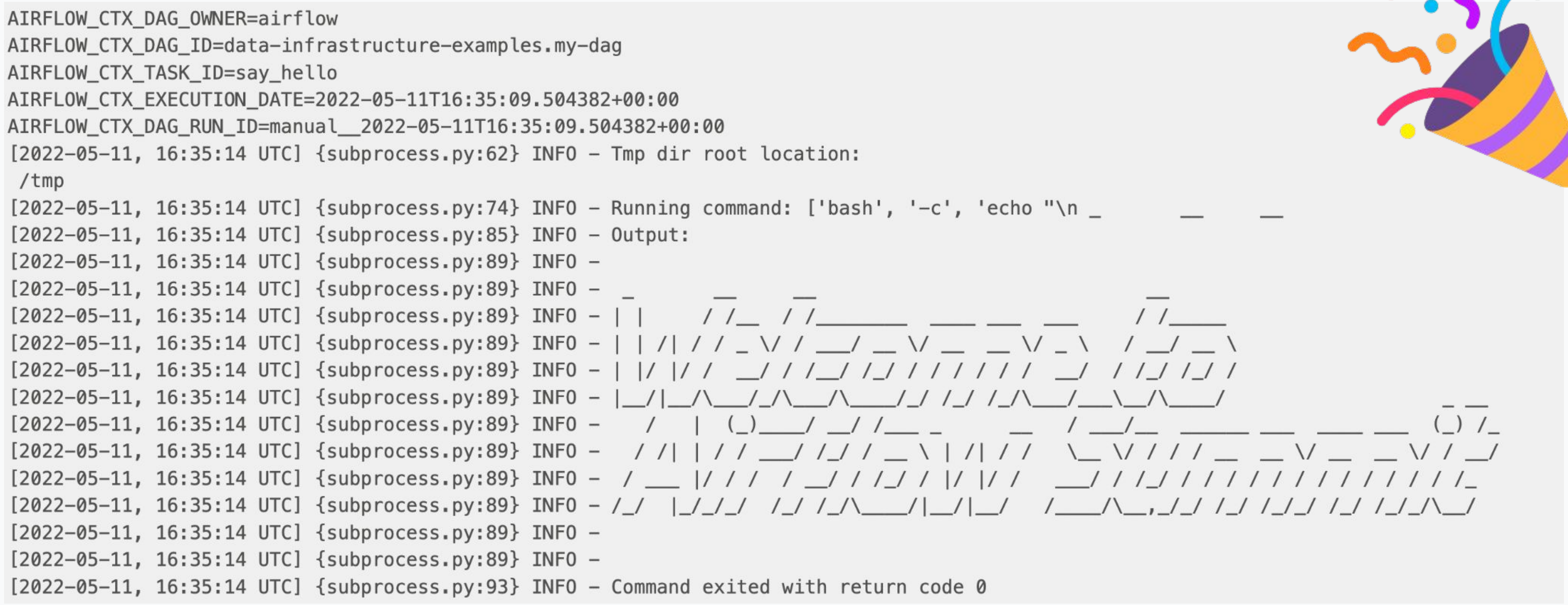
"With a remote development experience like this, who needs a local environment?"

- I said this to our users

Examples:

```
> surf dags trigger data-infrastructure-examples.my-dag

Using environment: staging
Found 1 DAG:
 - data-infrastructure-examples.my-dag

Trigger 1 DAG? [y/N]: y
[ OK     ] Trigger data-infrastructure-examples.my-dag

https://airflow.staging.webserver/graph?dag_id=data-infrastructure-examples.my-dag&dag_run_id
=manual__2022-05-11T16%3A35%3A09.504382%2B00%3A00&exec_date=2022-05-11T16%3A35%3A09.504382%2B
00%3A00
```





```
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=data-infrastructure-examples.my-dag
AIRFLOW_CTX_TASK_ID=say_hello
AIRFLOW_CTX_EXECUTION_DATE=2022-05-11T16:35:09.504382+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2022-05-11T16:35:09.504382+00:00
[2022-05-11, 16:35:14 UTC] {subprocess.py:62} INFO - Tmp dir root location:
 /tmp
[2022-05-11, 16:35:14 UTC] {subprocess.py:74} INFO - Running command: ['bash', '-c', 'echo "\n _       __     __
[2022-05-11, 16:35:14 UTC] {subprocess.py:85} INFO - Output:
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -  _       __     __                             __
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO - | |     / /__  / /________  ____ ___  ___     / /_____
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO - | | /| / / _ \/ / ___/ __ \/ __  __ \/ _ \   / __/ __ \
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO - | |/ |/ /  __/ / /__/ /_/ / / / / / /  __/  / /_/ /_/ /
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO - |__/|__/\___/_/\___/\____/_/ /_/ /_/\___/___\__/\____/                   _ __
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -    /   |  (_)____/ __/ /___ _      __   / ___/__  ______ ___  ____ ___  (_) /_
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -   / /| | / / ___/ /_/ / __ \ | /| / /   \__ \/ / / / __  __ \/ __  __ \/ / __/
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -  / ___ |/ / /  / __/ / /_/ / |/ |/ /   ___/ / /_/ / / / / / / / / / / / / /_
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO - /_/  |_/_/_/  /_/ /_/\____/|__/|__/   /____/\__,_/_/ /_/ /_/_/ /_/ /_/_/\__/
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -
[2022-05-11, 16:35:14 UTC] {subprocess.py:89} INFO -
[2022-05-11, 16:35:14 UTC] {subprocess.py:93} INFO - Command exited with return code 0
```

What's Next?

Multiple Prod Environments

Kubernetes Executor ☸️

More Contributions 📈

30,000 DAGs 🚀

Airflow 2.3 🎉

Thanks for Watching

Thanks again to:

The rest of the Data Foundations team at Shopify

The Airflow development team

The Airflow Summit organizers

Wanna chat about our presentation? You can find either of us on the official Airflow Slack.