Writing Dry Code in Airflow

Sarah Krasnik

Airflow Summit

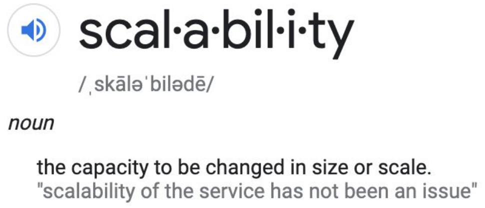
July 14th, 2021

Scalability: Everyone Wants It

Whether it's scaling a team from 5 to 20 or scaling infrastructure workloads, organizational demands grow and other parts of the company must grow in parallel.

By definition, scalability is the ability to grow in size.

However, the less naive definition is tied to growing in size *efficiently*.


scal·a·bil·i·ty

/ˌskāləˈbilədē/

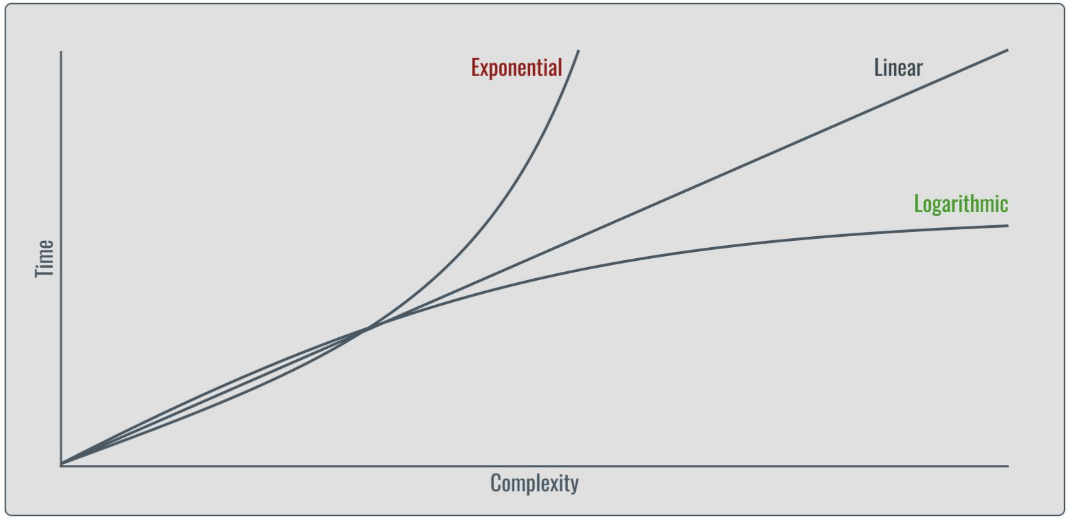
noun

the capacity to be changed in size or scale.
"scalability of the service has not been an issue"

Scalability: Everyone Wants It

Scale can be: **exponential**, **linear** or **logarithmic**, in order of efficiency.

Scalability is the ability to scale most efficiently by having growth in complexity amount to small growth in time to implementation.



What Does Scaling Airflow Mean?

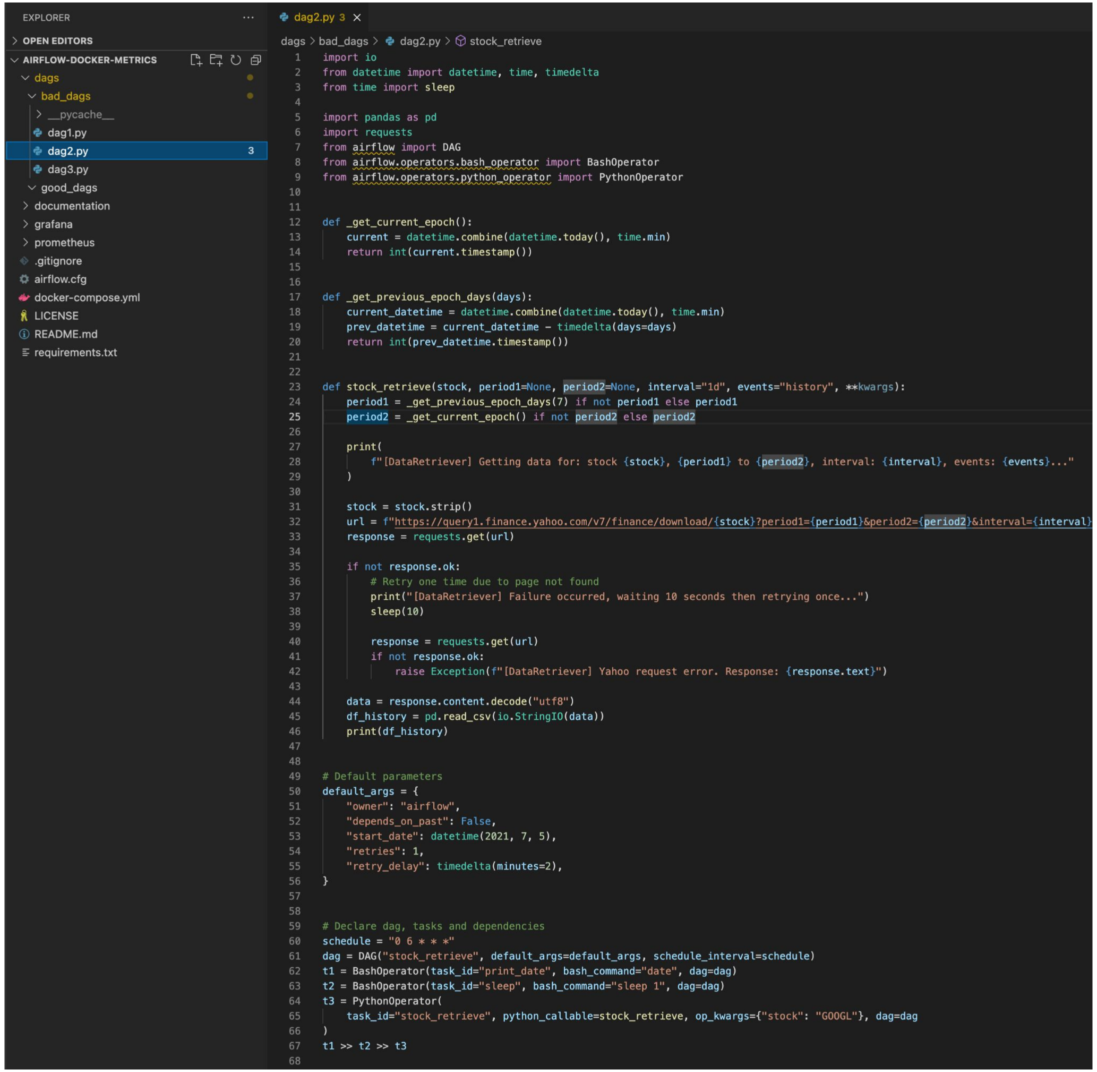
Growing usage of Airflow largely means increasing resources, infrastructure, and a growing number of DAGs (oftentimes, all in parallel).

Focusing on the code: A growing number of DAGs can be a recipe for disaster, or motivation for good organization. Without leveraging base functions or classes, growing a code base from 5 DAGs to 100 could be a nightmare. It doesn’t have to be!

DAG Architecture: What Not To Do

Could a new team member view this file and know what to expect in the Airflow UI?

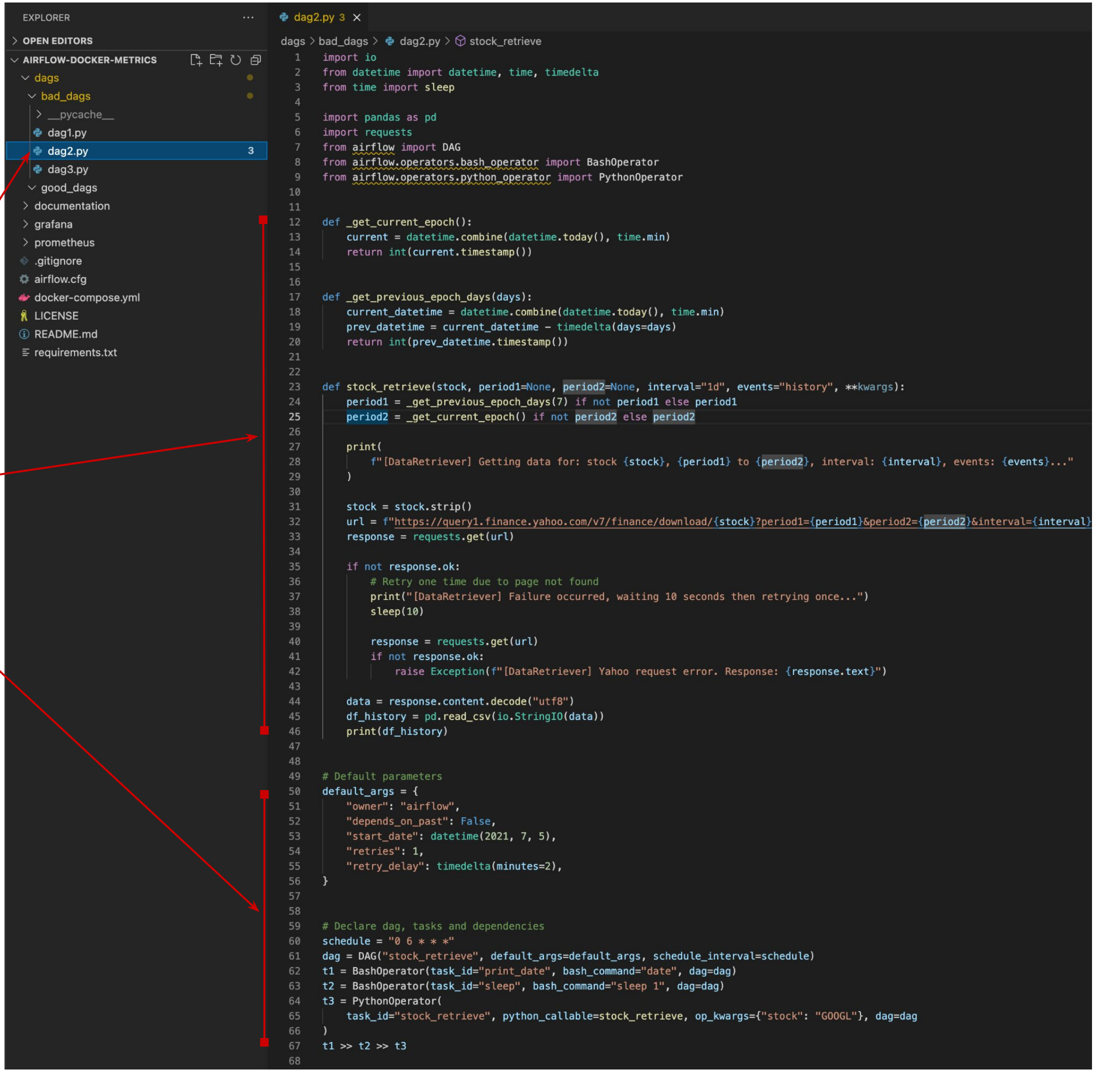
I don’t think so.



```python
import io
from datetime import datetime, time, timedelta
from time import sleep

import pandas as pd
import requests
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator


def _get_current_epoch():
    current = datetime.combine(datetime.today(), time.min)
    return int(current.timestamp())


def _get_previous_epoch_days(days):
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(days=days)
    return int(prev_datetime.timestamp())


def stock_retrieve(stock, period1=None, period2=None, interval="1d", events="history", **kwargs):
    period1 = _get_previous_epoch_days(7) if not period1 else period1
    period2 = _get_current_epoch() if not period2 else period2

    print(
        f"[DataRetriever] Getting data for: stock {stock}, {period1} to {period2}, interval: {interval}, events: {events}..."
    )

    stock = stock.strip()
    url = f"https://query1.finance.yahoo.com/v7/finance/download/{stock}?period1={period1}&period2={period2}&interval={interval}
    response = requests.get(url)

    if not response.ok:
        # Retry one time due to page not found
        print("[DataRetriever] Failure occurred, waiting 10 seconds then retrying once...")
        sleep(10)

        response = requests.get(url)
        if not response.ok:
            raise Exception(f"[DataRetriever] Yahoo request error. Response: {response.text}")

    data = response.content.decode("utf8")
    df_history = pd.read_csv(io.StringIO(data))
    print(df_history)


# Default parameters
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2021, 7, 5),
    "retries": 1,
    "retry_delay": timedelta(minutes=2),
}


# Declare dag, tasks and dependencies
schedule = "0 6 * * *"
dag = DAG("stock_retrieve", default_args=default_args, schedule_interval=schedule)
t1 = BashOperator(task_id="print_date", bash_command="date", dag=dag)
t2 = BashOperator(task_id="sleep", bash_command="sleep 1", dag=dag)
t3 = PythonOperator(
    task_id="stock_retrieve", python_callable=stock_retrieve, op_kwargs={"stock": "GOOGL"}, dag=dag
)
t1 >> t2 >> t3
```

DAG Architecture: What Not To Do

1. File naming: this shouldn't be random
2. Mix business logic with DAG code
3. Write tool-specific DAG and task code over, over, and over again

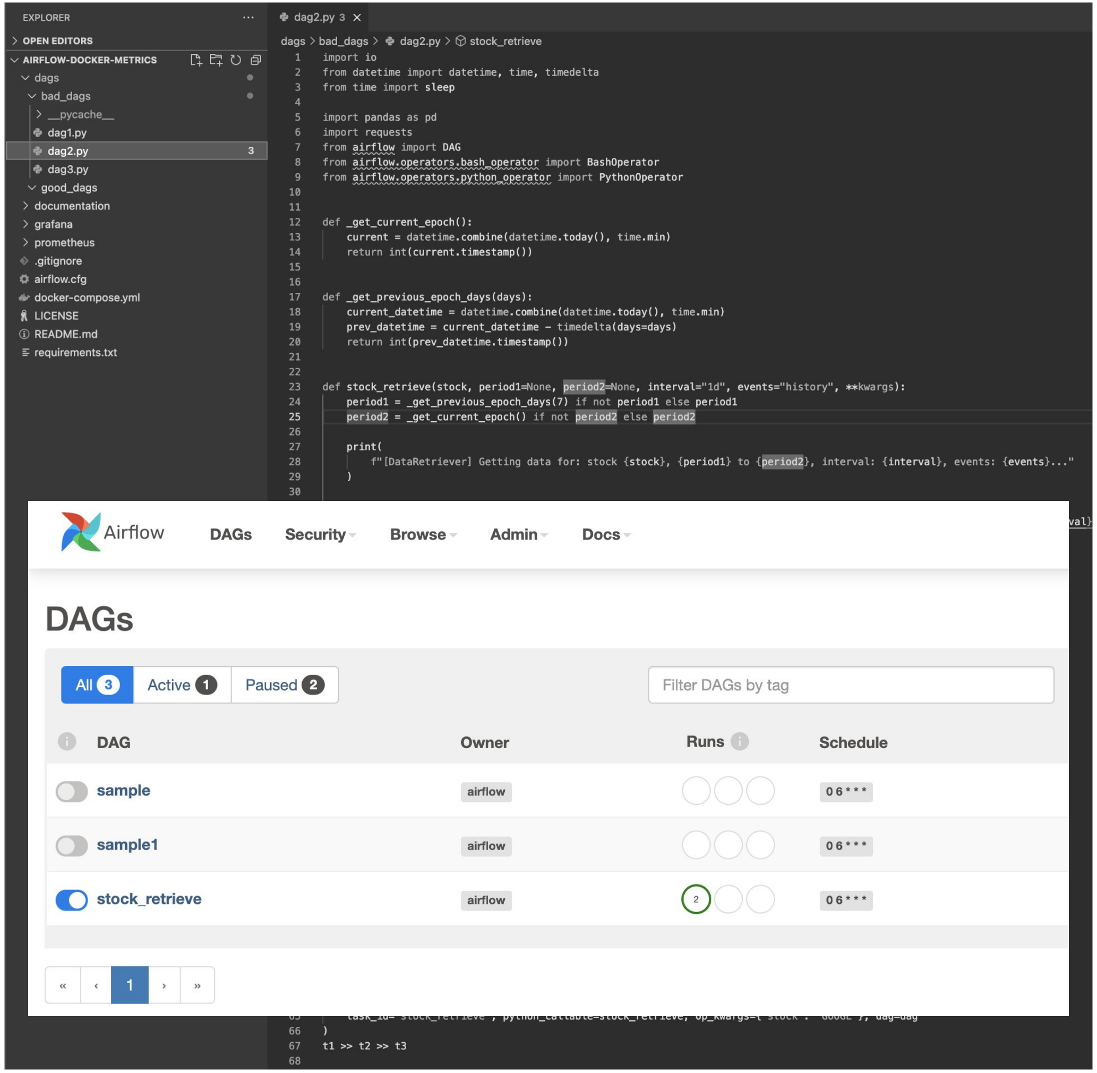


```python
import io
from datetime import datetime, time, timedelta
from time import sleep

import pandas as pd
import requests
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator


def _get_current_epoch():
    current = datetime.combine(datetime.today(), time.min)
    return int(current.timestamp())


def _get_previous_epoch_days(days):
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(days=days)
    return int(prev_datetime.timestamp())


def stock_retrieve(stock, period1=None, period2=None, interval="1d", events="history", **kwargs):
    period1 = _get_previous_epoch_days(7) if not period1 else period1
    period2 = _get_current_epoch() if not period2 else period2

    print(
        f"[DataRetriever] Getting data for: stock {stock}, {period1} to {period2}, interval: {interval}, events: {events}..."
    )

    stock = stock.strip()
    url = f"https://query1.finance.yahoo.com/v7/finance/download/{stock}?period1={period1}&period2={period2}&interval={interval}
    response = requests.get(url)

    if not response.ok:
        # Retry one time due to page not found
        print("[DataRetriever] Failure occurred, waiting 10 seconds then retrying once...")
        sleep(10)

        response = requests.get(url)
        if not response.ok:
            raise Exception(f"[DataRetriever] Yahoo request error. Response: {response.text}")

    data = response.content.decode("utf8")
    df_history = pd.read_csv(io.StringIO(data))
    print(df_history)


# Default parameters
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2021, 7, 5),
    "retries": 1,
    "retry_delay": timedelta(minutes=2),
}


# Declare dag, tasks and dependencies
schedule = "0 6 * * *"
dag = DAG("stock_retrieve", default_args=default_args, schedule_interval=schedule)
t1 = BashOperator(task_id="print_date", bash_command="date", dag=dag)
t2 = BashOperator(task_id="sleep", bash_command="sleep 1", dag=dag)
t3 = PythonOperator(
    task_id="stock_retrieve", python_callable=stock_retrieve, op_kwargs={"stock": "GOOGL"}, dag=dag
)
t1 >> t2 >> t3
```

DAG Architecture: What Not To Do

The difficulties this poses:

1. Someone reviewing this code must understand all of the business logic and Airflow components.
2. Understanding DAGs in the Airflow Webserver UI is not straightforward.
3. Adding to the code is messy.



```python
import io
from datetime import datetime, time, timedelta
from time import sleep

import pandas as pd
import requests
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator


def _get_current_epoch():
    current = datetime.combine(datetime.today(), time.min)
    return int(current.timestamp())


def _get_previous_epoch_days(days):
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(days=days)
    return int(prev_datetime.timestamp())


def stock_retrieve(stock, period1=None, period2=None, interval="1d", events="history", **kwargs):
    period1 = _get_previous_epoch_days(7) if not period1 else period1
    period2 = _get_current_epoch() if not period2 else period2

    print(
        f"[DataRetriever] Getting data for: stock {stock}, {period1} to {period2}, interval: {interval}, events: {events}..."
    )

        task_id='stock_retrieve', python_callable=stock_retrieve, op_kwargs={'stock': 'GOOGL'}, dag=dag
)
t1 >> t2 >> t3
```

DAGs

All 3 | Active 1 | Paused 2

Filter DAGs by tag

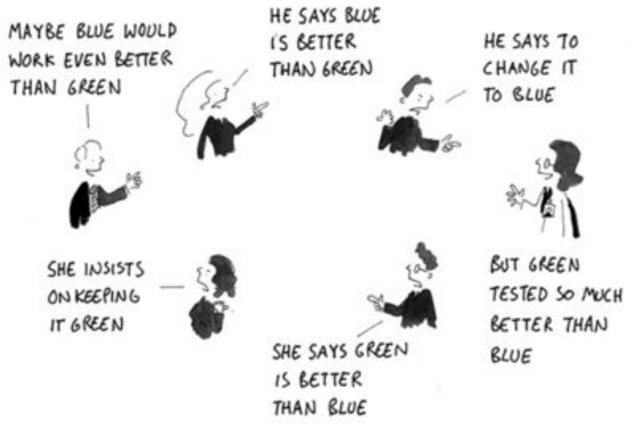
DAG	Owner	Runs	Schedule
sample	airflow		0 6 * * *
sample1	airflow		0 6 * * *
stock_retrieve	airflow	2	0 6 * * *

DRY: Don’t Repeat Yourself

It’s like the game of broken telephone: you repeat what you heard, but then something gets lost in translation and everyone is confused.

Trying to understand a code base that isn’t objectively intuitive is incredibly time consuming for someone who didn’t write it.

Except with code, it’s not funny when a misunderstanding happens while debugging an issue.

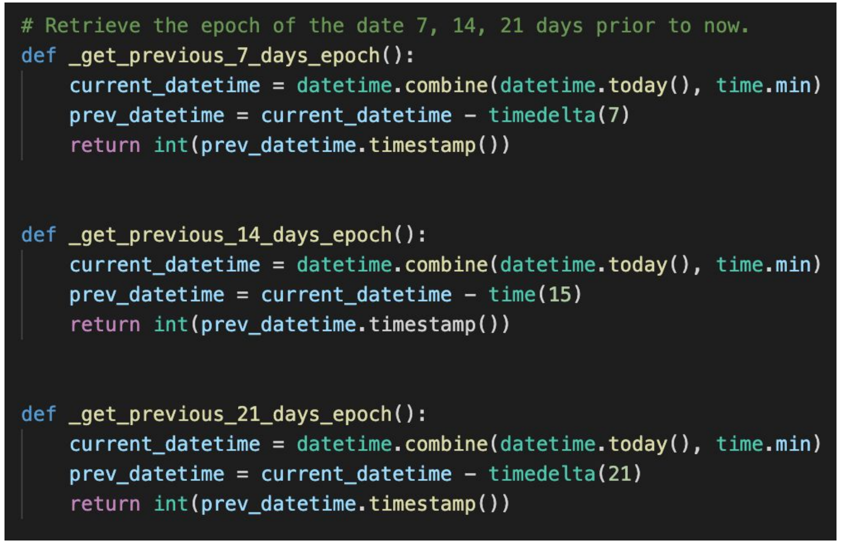

MAYBE BLUE WOULD WORK EVEN BETTER THAN GREEN
HE SAYS BLUE IS BETTER THAN GREEN
HE SAYS TO CHANGE IT TO BLUE
BUT GREEN TESTED SO MUCH BETTER THAN BLUE
SHE SAYS GREEN IS BETTER THAN BLUE
SHE INSISTS ON KEEPING IT GREEN

Source: https://www.insivia.com/wp-content/uploads/2014/04/telephone_game.png

DRY: Don’t Repeat Yourself

Even a simple function of calculating the unix time of 7, 14, and 21 days ago is prone to error.

Do you think you’d be able to find the TWO bugs when debugging why your output isn’t what you’d expect?

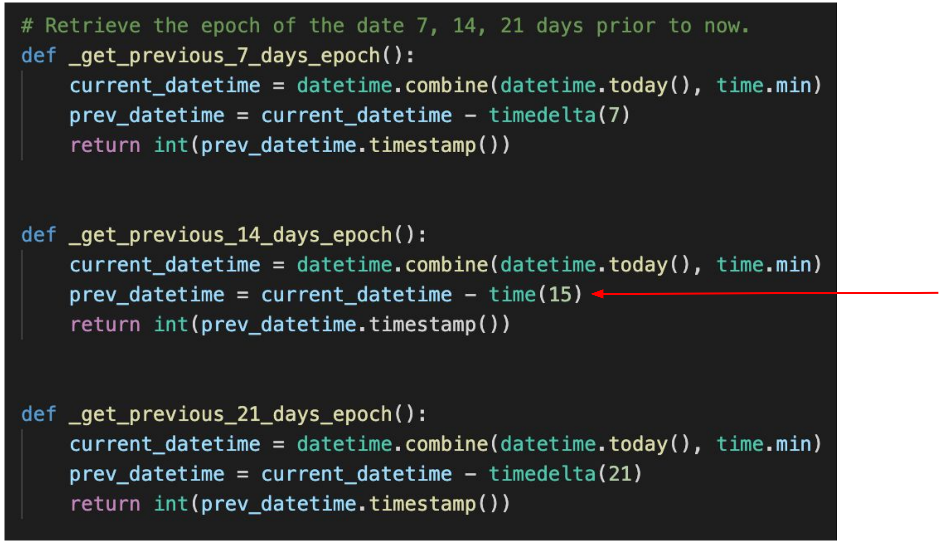


```
# Retrieve the epoch of the date 7, 14, 21 days prior to now.
def _get_previous_7_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(7)
    return int(prev_datetime.timestamp())


def _get_previous_14_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - time(15)
    return int(prev_datetime.timestamp())


def _get_previous_21_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(21)
    return int(prev_datetime.timestamp())
```

DRY: Don’t Repeat Yourself

Even a simple function of calculating the unix time of 7, 14, and 21 days ago is prone to error.

Do you think you’d be able to find the TWO bugs when debugging why your output isn’t what you’d expect?

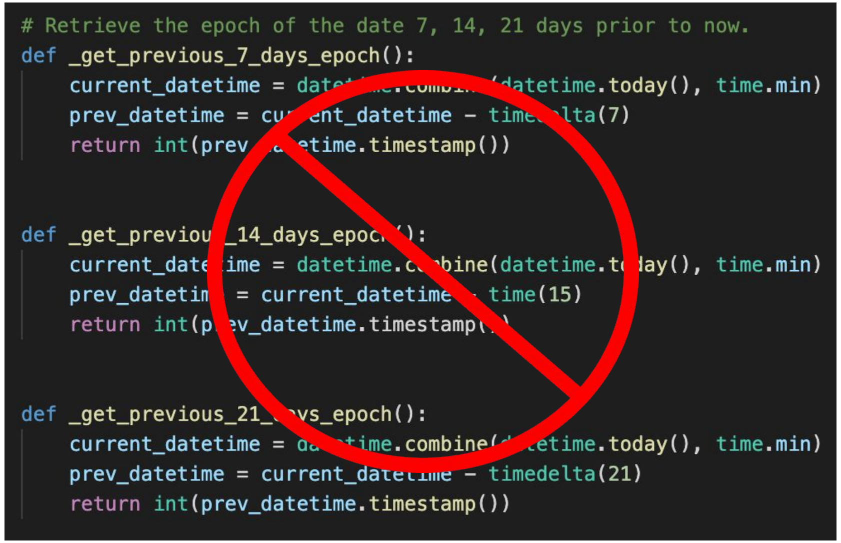


```python
# Retrieve the epoch of the date 7, 14, 21 days prior to now.
def _get_previous_7_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(7)
    return int(prev_datetime.timestamp())


def _get_previous_14_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - time(15)
    return int(prev_datetime.timestamp())


def _get_previous_21_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(21)
    return int(prev_datetime.timestamp())
```

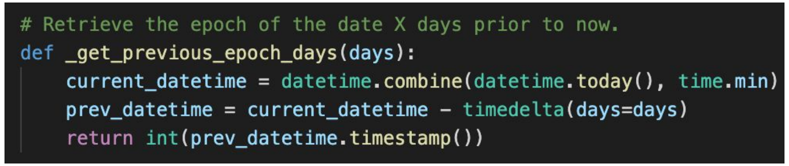
DRY: Don’t Repeat Yourself

Not repeating yourself applies to DAG code too!

Now what does that look like when writing DAGs….



```python
# Retrieve the epoch of the date 7, 14, 21 days prior to now.
def _get_previous_7_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(7)
    return int(prev_datetime.timestamp())


def _get_previous_14_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - time(15)
    return int(prev_datetime.timestamp())


def _get_previous_21_days_epoch():
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(21)
    return int(prev_datetime.timestamp())
```



```python
# Retrieve the epoch of the date X days prior to now.
def _get_previous_epoch_days(days):
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(days=days)
    return int(prev_datetime.timestamp())
```

Dry Airflow DAG Architecture

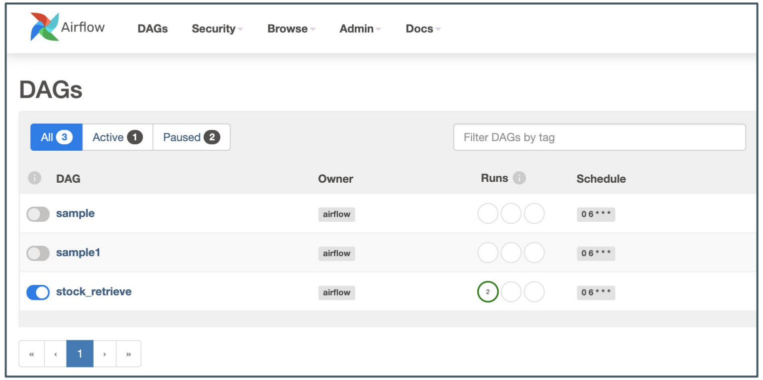
The proposed DAG architecture:

- **Folder structure**. The code implementing a DAG should be within one folder, named the same as the DAG itself. Within the folder, the DAG declaration file should not contain any business logic.
- **Code consolidation.** If any code is shared, create a separate shared utility folder that's referenced across DAGs.
- **Tool specific code**. Any code specific to Airflow or any other tool (Spark, Great Expectations, etc) should be localized to one place, with as little redundancy as possible. With Airflow, the factory pattern works well to achieve generalization.

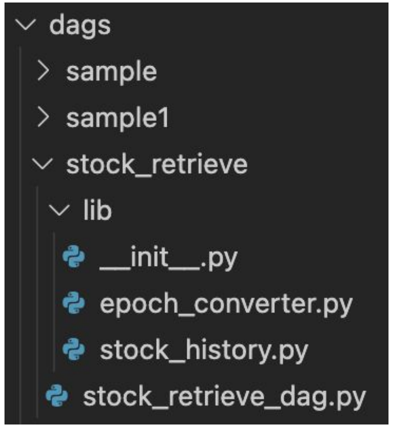
Dry Airflow DAG Architecture: Folder Structure

When an Airflow DAG fails, it should be extremely obvious where in the codebase debugging begins.

Additionally, it should be equally obvious where the DAG is declared versus where business logic is implemented.



DAG	Owner	Runs	Schedule
sample	airflow		0 6 * * *
sample1	airflow		0 6 * * *
stock_retrieve	airflow	2	0 6 * * *



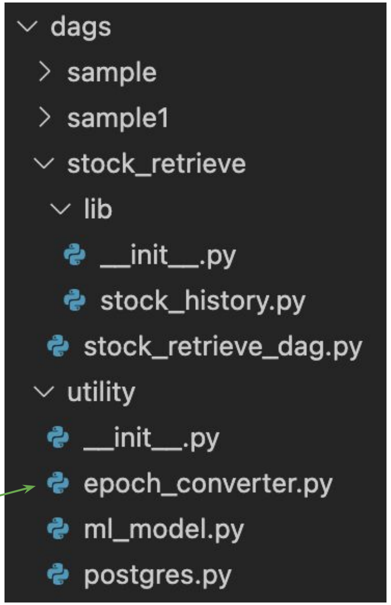
```
dags
  sample
  sample1
  stock_retrieve
    lib
      __init__.py
      epoch_converter.py
      stock_history.py
    stock_retrieve_dag.py
```

Dry Airflow DAG Architecture: Code Consolidation

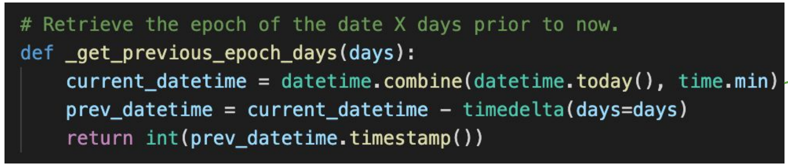
If there are several DAGs handling timestamps and would use the epoch functions mentioned earlier, it can be put into a separate file in a “utility” folder.

DAG code can then use classes from the single folder, instead of copying logic.

Remember: we’re trying to avoid a game of broken telephone.



```
dags
  sample
  sample1
  stock_retrieve
    lib
      __init__.py
      stock_history.py
    stock_retrieve_dag.py
  utility
    __init__.py
    epoch_converter.py
    ml_model.py
    postgres.py
```

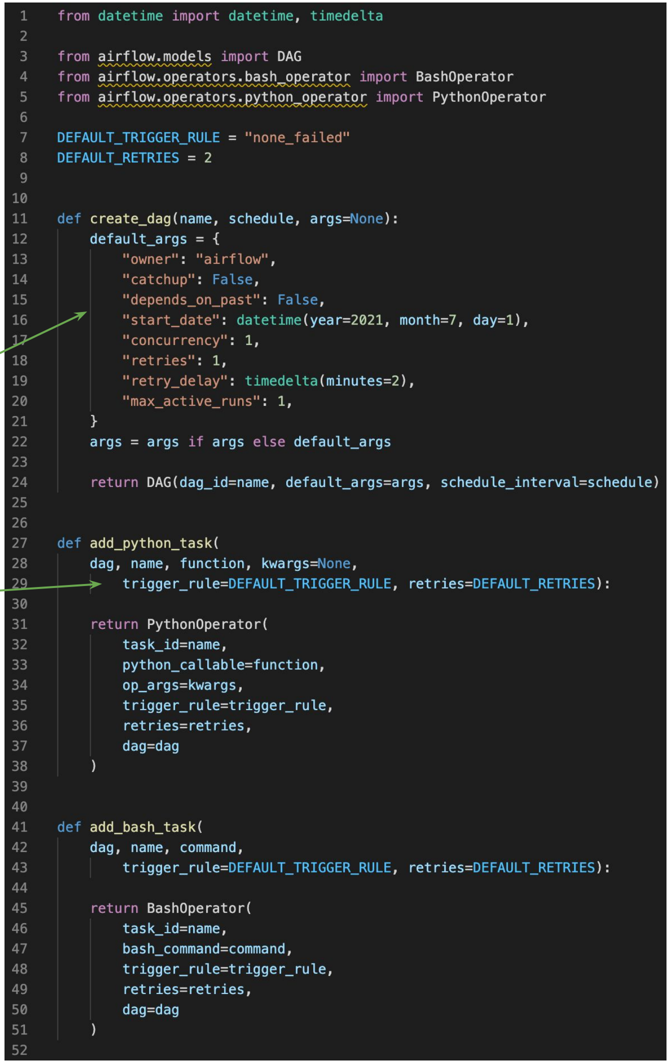


```
# Retrieve the epoch of the date X days prior to now.
def _get_previous_epoch_days(days):
    current_datetime = datetime.combine(datetime.today(), time.min)
    prev_datetime = current_datetime - timedelta(days=days)
    return int(prev_datetime.timestamp())
```

Dry Airflow DAG Architecture: Tool Specific Code

Abstraction let's anyone contribute to the codebase easily, even if they only understand basic Airflow concepts.

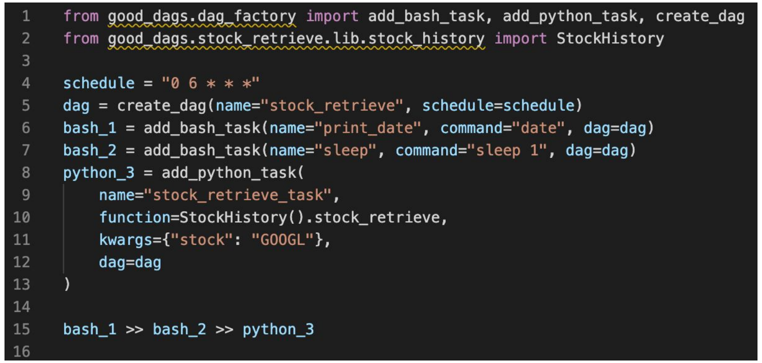
- Supplying recommended defaults
- Generic task argument names



```python
from datetime import datetime, timedelta

from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator

DEFAULT_TRIGGER_RULE = "none_failed"
DEFAULT_RETRIES = 2


def create_dag(name, schedule, args=None):
    default_args = {
        "owner": "airflow",
        "catchup": False,
        "depends_on_past": False,
        "start_date": datetime(year=2021, month=7, day=1),
        "concurrency": 1,
        "retries": 1,
        "retry_delay": timedelta(minutes=2),
        "max_active_runs": 1,
    }
    args = args if args else default_args

    return DAG(dag_id=name, default_args=args, schedule_interval=schedule)


def add_python_task(
    dag, name, function, kwargs=None,
        trigger_rule=DEFAULT_TRIGGER_RULE, retries=DEFAULT_RETRIES):

    return PythonOperator(
        task_id=name,
        python_callable=function,
        op_args=kwargs,
        trigger_rule=trigger_rule,
        retries=retries,
        dag=dag
    )


def add_bash_task(
    dag, name, command,
        trigger_rule=DEFAULT_TRIGGER_RULE, retries=DEFAULT_RETRIES):

    return BashOperator(
        task_id=name,
        bash_command=command,
        trigger_rule=trigger_rule,
        retries=retries,
        dag=dag
    )

```

Dry Airflow DAG Architecture: Tool Specific Code

Then, your DAG declaration files are unbelievably short, abstracted away from specific operators.

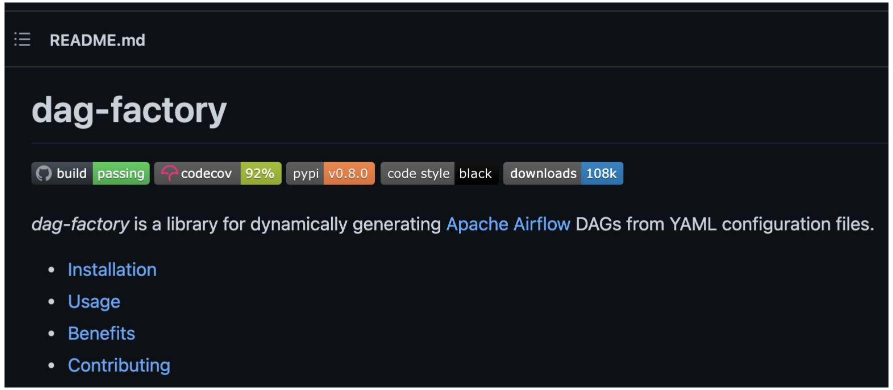


```python
from good_dags.dag_factory import add_bash_task, add_python_task, create_dag
from good_dags.stock_retrieve.lib.stock_history import StockHistory

schedule = "0 6 * * *"
dag = create_dag(name="stock_retrieve", schedule=schedule)
bash_1 = add_bash_task(name="print_date", command="date", dag=dag)
bash_2 = add_bash_task(name="sleep", command="sleep 1", dag=dag)
python_3 = add_python_task(
    name="stock_retrieve_task",
    function=StockHistory().stock_retrieve,
    kwargs={"stock": "GOOGL"},
    dag=dag
)

bash_1 >> bash_2 >> python_3
```

Dry Airflow DAG Architecture: Tool Specific Code

Alternative solutions to abstraction:

- Wrapper functions (example shown)
- DagFactory class
- The dag-factory pip package
  https://github.com/ajbosco/dag-factory


README.md

dag-factory

build passing | codecov 92% | pypi v0.8.0 | code style black | downloads 108k

dag-factory is a library for dynamically generating Apache Airflow DAGs from YAML configuration files.

- Installation
- Usage
- Benefits
- Contributing

Dry Airflow DAG Architecture: Tool Specific Code

This doesn't only apply to Airflow code.

Consider modifications to SQL queries, resulting in changes to data tests.

You may want one person to review the query itself, another to review the data tests, and a third person to review the Python elements of a DAG.

All three people don't necessarily have to keep a deep knowledge of all tools (not that they shouldn't!).

dbt

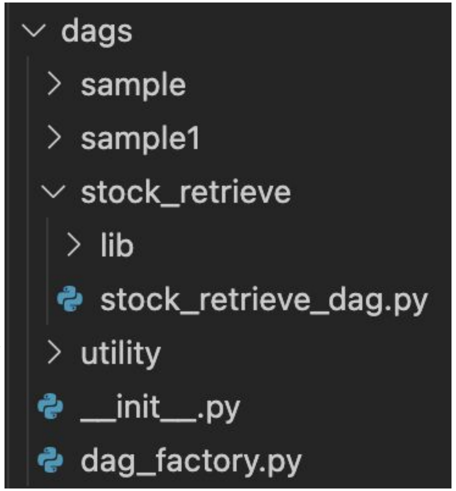
great_expectations

python™

Dry Airflow DAG Architecture: Putting It All Together

So where does this leave us?

- **Folder structure**. Making the jump from the Airflow UI to the codebase doesn't involve a scavenger hunt.
- **Code consolidation.** Centralizing a repository of commonly used functions and classes means no game of developer broken telephone.
- **Tool specific code**. Abstracting over Airflow specific code diversifies who can contribute to and review code.



```
dags
  sample
  sample1
  stock_retrieve
    lib
    stock_retrieve_dag.py
  utility
  __init__.py
  dag_factory.py
```

Final Thoughts

Depending on your use case, you might need more or less complex solutions.

Abstraction and organization are helpful even at a small scale.

Thank you!

Code can be found here:
https://github.com/sarahmk125/airflow-docker-metrics