An Introduction to Airflow Cluster Policies

Philippe Gagnon
Airflow Summit 2023
Toronto, Canada

Agenda

- Your Speaker
- What are Cluster Policies?
- Available Policy Functions
- Use Cases
- Defining your policy functions
- Using the pluggy mechanism

Philippe Gagnon

- Senior Solutions Architect @ Astronomer, inc.
- Fancycron Enthusiast since 2017.

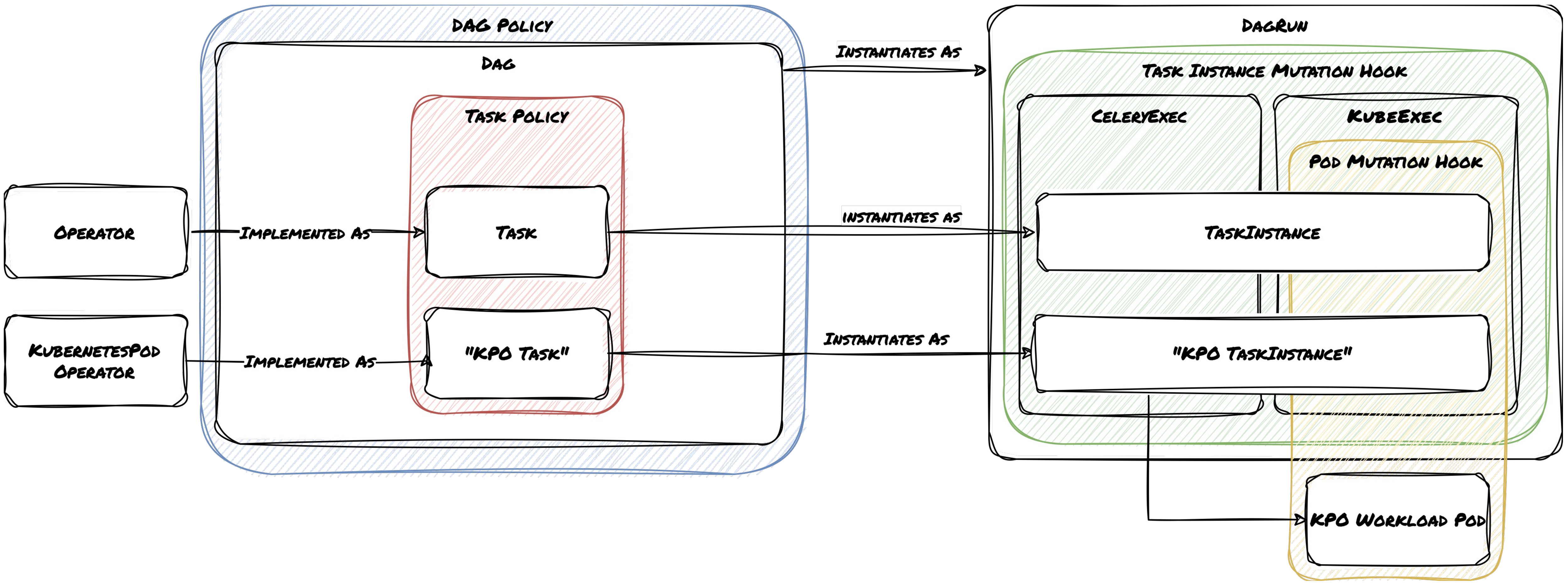
What are Cluster Policies?

- *Cluster Policies* 📜 are a set of functions Airflow administrators 👮 can define in their `airflow_local_settings*` module to perform custom logic on a few important Airflow objects.
- They can either
  - *Mutate* 🧟 the object they are applied on;
  - or (for DAG or task policies), *skip* ⏭️ ;
  - or *deny* 🚫 a DAG from being added to the DagBag.

Available Policy Functions in Airflow

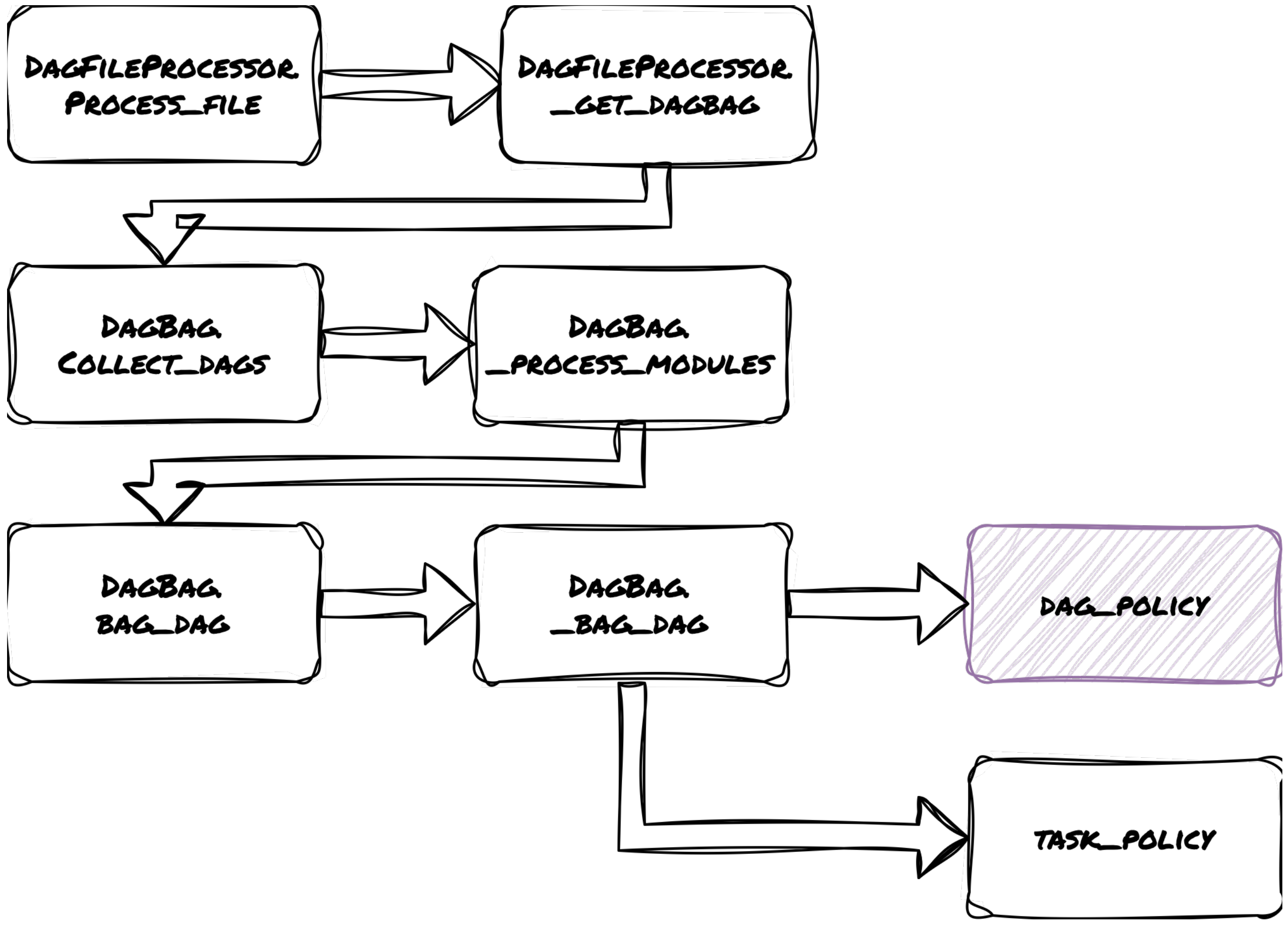
- `dag_policy`
- `task_policy`
- `task_instance_mutation_hook`
- `pod_mutation_hook`
- `get_airflow_context_vars`

High-level Overview



dag_policy

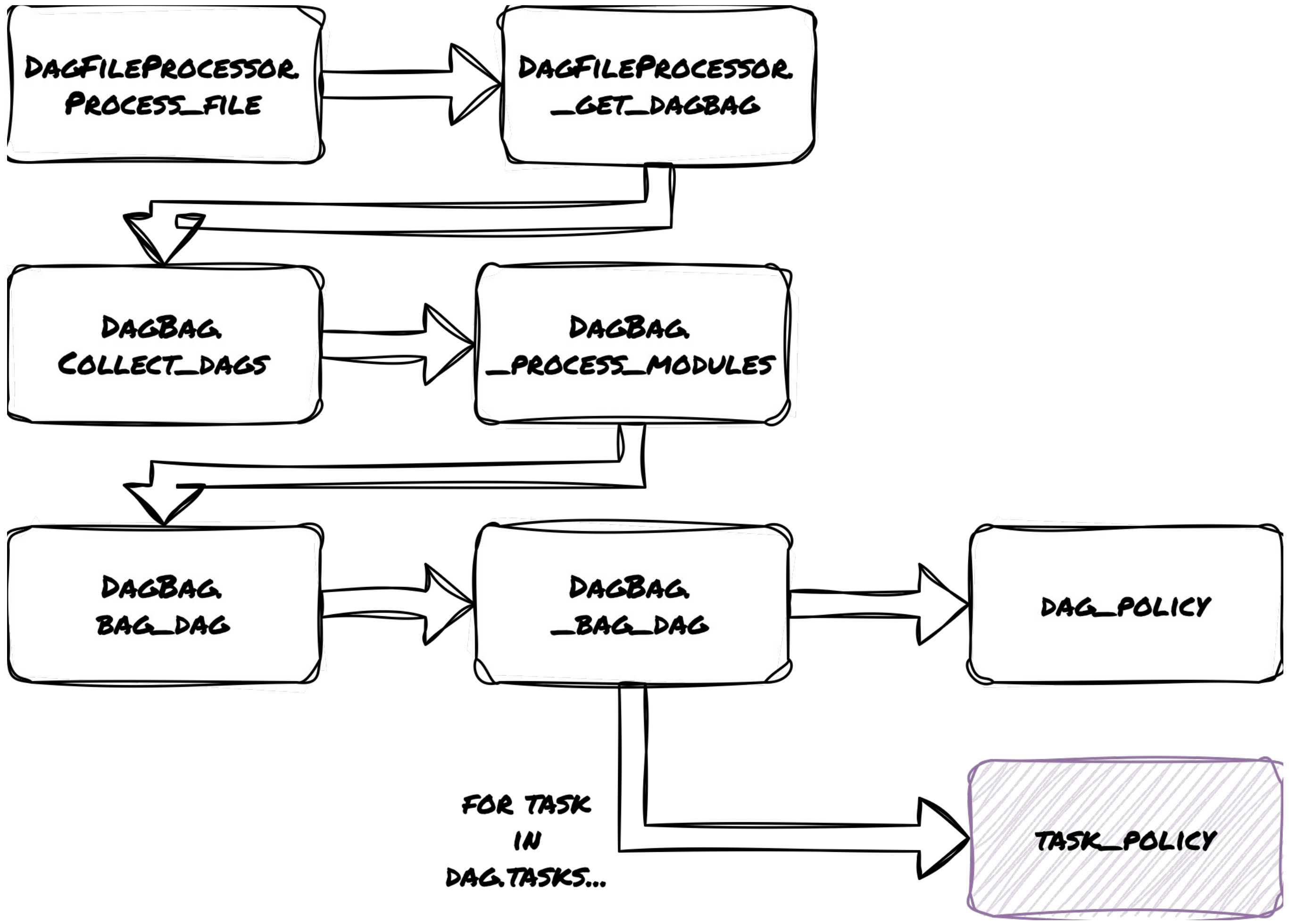
- Mutates DAG objects after they are loaded in the DagBag.
- Runs *after your DAG has been fully generated*.
- However, dag_policy is still applied before task_policy.
- It also means that the DAG processor parses all DAG files even if skipped or denied.



`task_policy`

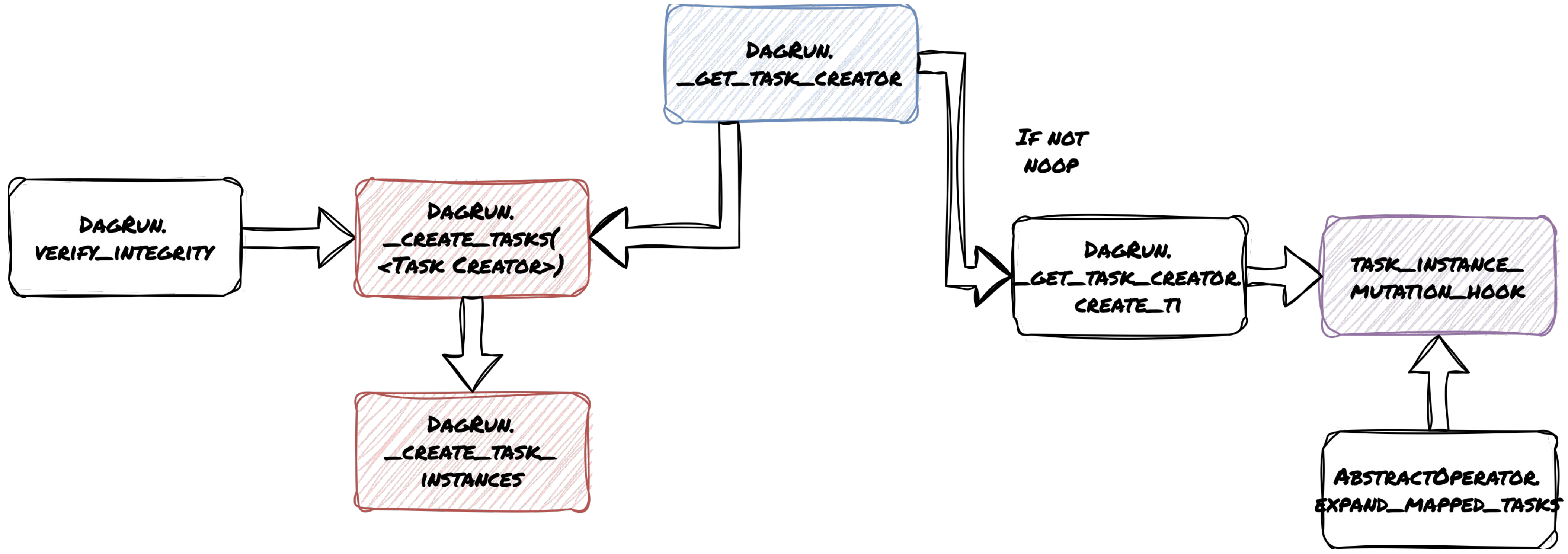
- Mutates tasks <u>after</u> they have been added to a DAG.
- It receives a “BaseOperator” as an argument* and can issue skip/deny exceptions.

* This is actually a bug, and we’ll see why later... 🤫



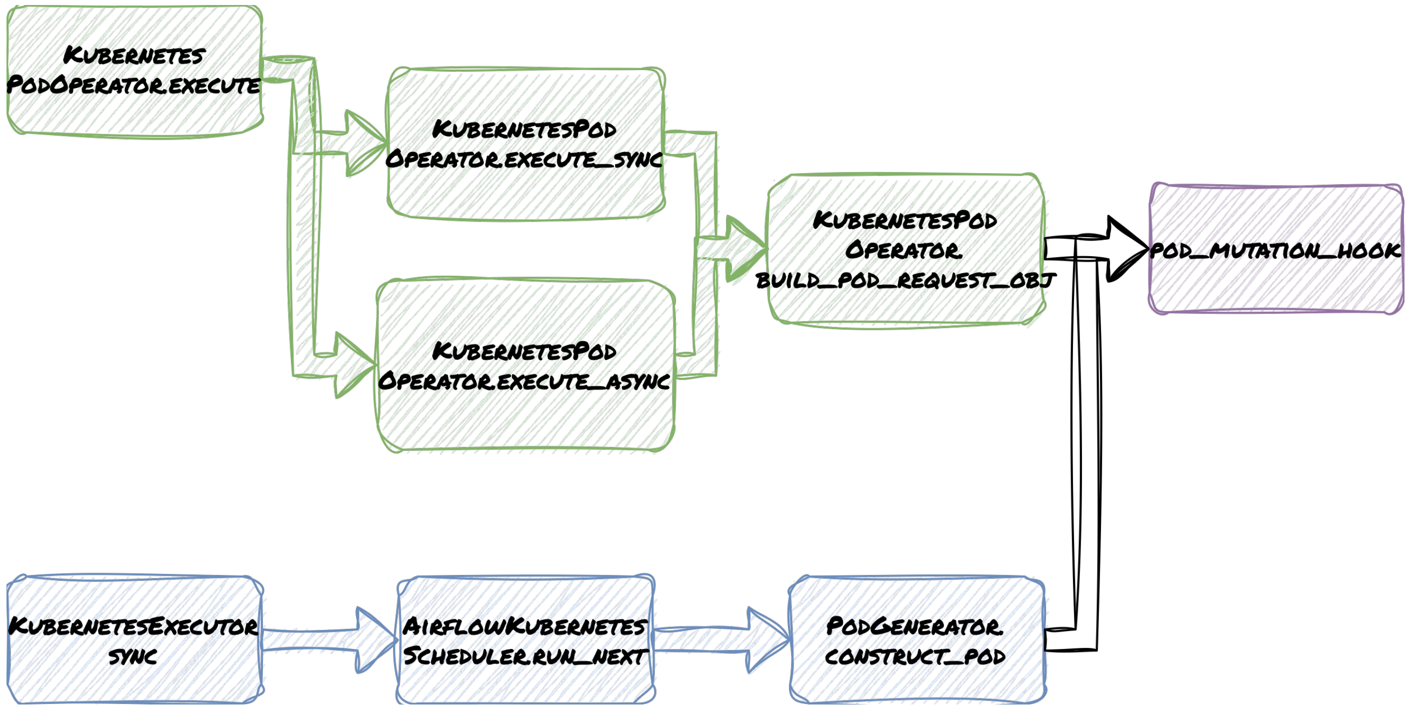
`task_instance_mutation_hook`

- Similar to task policies, but applies to TaskInstance objects.
- The main difference between these two functions is that, while task policies mutate and inspect tasks “as defined”, task instance policies mutate and inspect task instances before they are executed.



pod_mutation_hook

- This is the original policy function.
- It takes a Pod object as an argument and can mutate it before it is scheduled on a Kubernetes cluster by Airflow.
- It is applied to Pod objects generated by both KubernetesPodOperator and KubernetesExecutor. ✌️


KubernetesPodOperator.execute
KubernetesPodOperator.execute_sync
KubernetesPodOperator.execute_async
KubernetesPodOperator.build_pod_request_obj
pod_mutation_hook
KubernetesExecutor.sync
AirflowKubernetesScheduler.run_next
PodGenerator.construct_pod

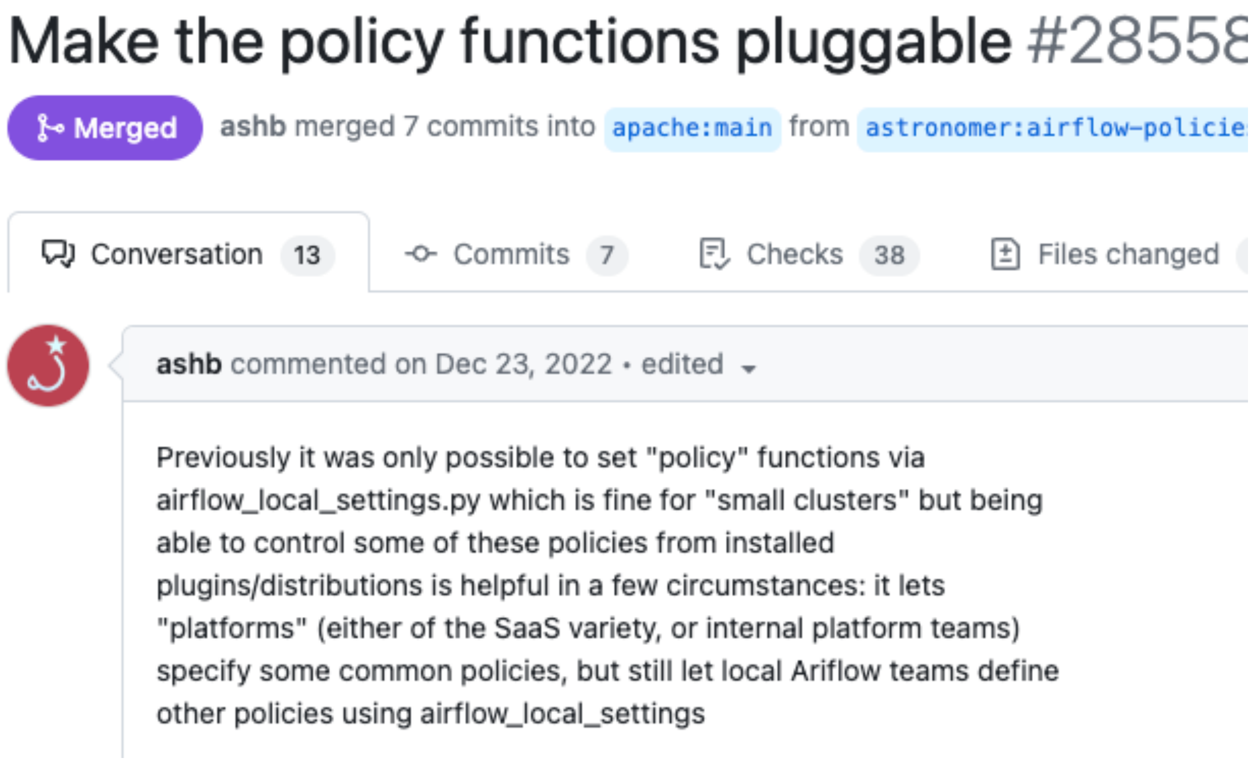
Defining your policy function

- Two methods: airflow_local_settings or via pluggy.

Using `airflow_local_settings`

- Create a module named `airflow_local_settings` and ensure it is added on your `sys.path`.
- The module should contain functions that match one or more of the policy functions defined in Airflow.

Using the pluggy interface



Make the policy functions pluggable #2855

Merged ashb merged 7 commits into `apache:main` from `astronomer:airflow-policie`

ashb commented on Dec 23, 2022 • edited

Previously it was only possible to set "policy" functions via airflow_local_settings.py which is fine for "small clusters" but being able to control some of these policies from installed plugins/distributions is helpful in a few circumstances: it lets "platforms" (either of the SaaS variety, or internal platform teams) specify some common policies, but still let local Ariflow teams define other policies using airflow_local_settings

Since Airflow 2.6, a new policy function configuration mechanism exists.

Using the pluggy interface

```
from airflow.policies import
hookimpl
@hookimpl
def task_policy(task) -> None:
# Mutate task in place # ...
print(f"Hello from {__file__}")
```

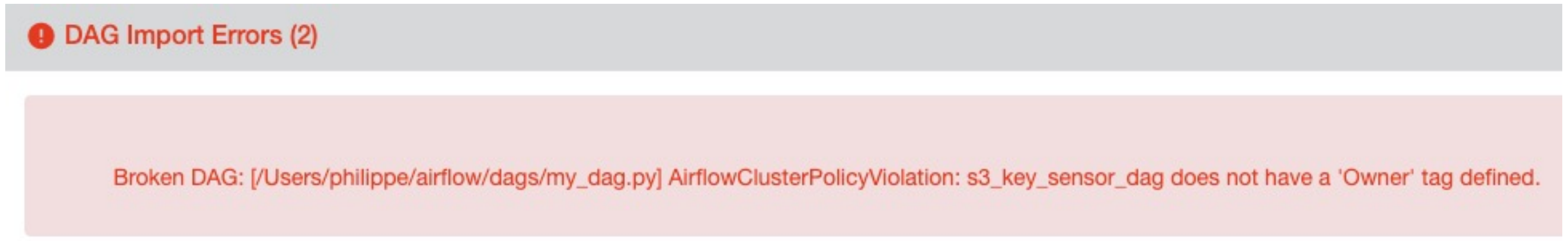
```
[build-system]
requires = ["setuptools",
"wheel"]
build-backend =
"setuptools.build_meta"
[project]
name = "my-airflow-plugin”
 version = "0.0.1" # ...
 dependencies = ["apache-
airflow>=2.6"] [project.entry-
points.'airflow.policy'] _ =
'my_airflow_plugin.policies'
```

Example Use Cases

- Ensuring DAGs are tagged
- Ensuring DAGs in development do not run in production
- Enforcing a task timeout
- Setting resource requests and limits
- Replacing an operator with its deferrable counterpart
- Using a different environment for different operators

Ensuring DAGs are tagged

```python
def ensure_dags_are_tagged(dag: "DAG") -> None:
    tag_labels = [tag.split(":")[0] for tag in dag.tags]
    if not "Owner" in tag_labels:
        raise AirflowClusterPolicyViolation(
            f"{dag.dag_id} does not have a 'Owner' tag defined."
        )
def dag_policy(dag: "DAG"):
    ensure_dags_are_tagged(dag)
```


DAG Import Errors (2)

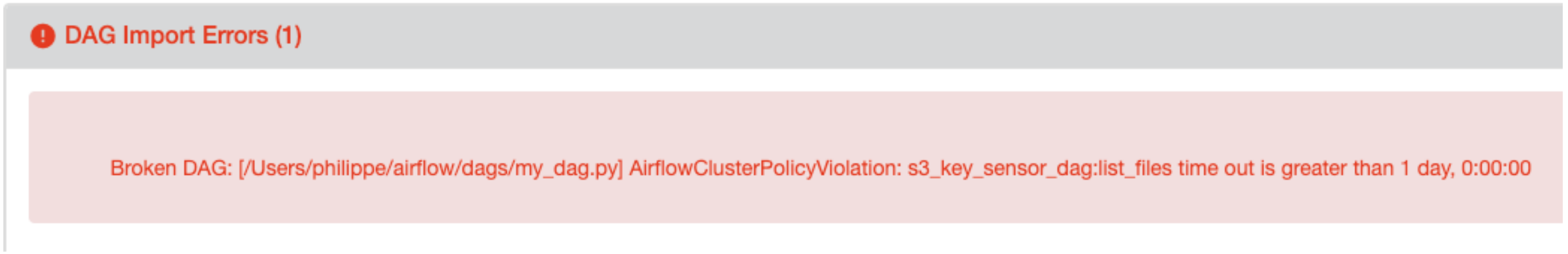
```
Broken DAG: [/Users/philippe/airflow/dags/my_dag.py] AirflowClusterPolicyViolation: s3_key_sensor_dag does not have a 'Owner' tag defined.
```

Ensuring DAGs in development do not run in production

```
def ensure_no_dev_dags_in_production(dag: "DAG") -> None:
    if not ”Maturity:Production" in dag.tags:
        raise AirflowClusterPolicySkipDag(
            f"Skipping DAG '{dag.dag_id}’ (missing 
Maturity:Production tag)"
        )
```

Enforcing a task timeout

```
def task_policy(task: "BaseOperator") -> None:
    min_timeout = datetime.timedelta(hours=24)
    if not task.execution_timeout or task.execution_timeout
> min_timeout:
        raise AirflowClusterPolicyViolation(
            f"{task.dag.dag_id}:{task.task_id} time out is
greater than {min_timeout}"
        )
```


DAG Import Errors (1)

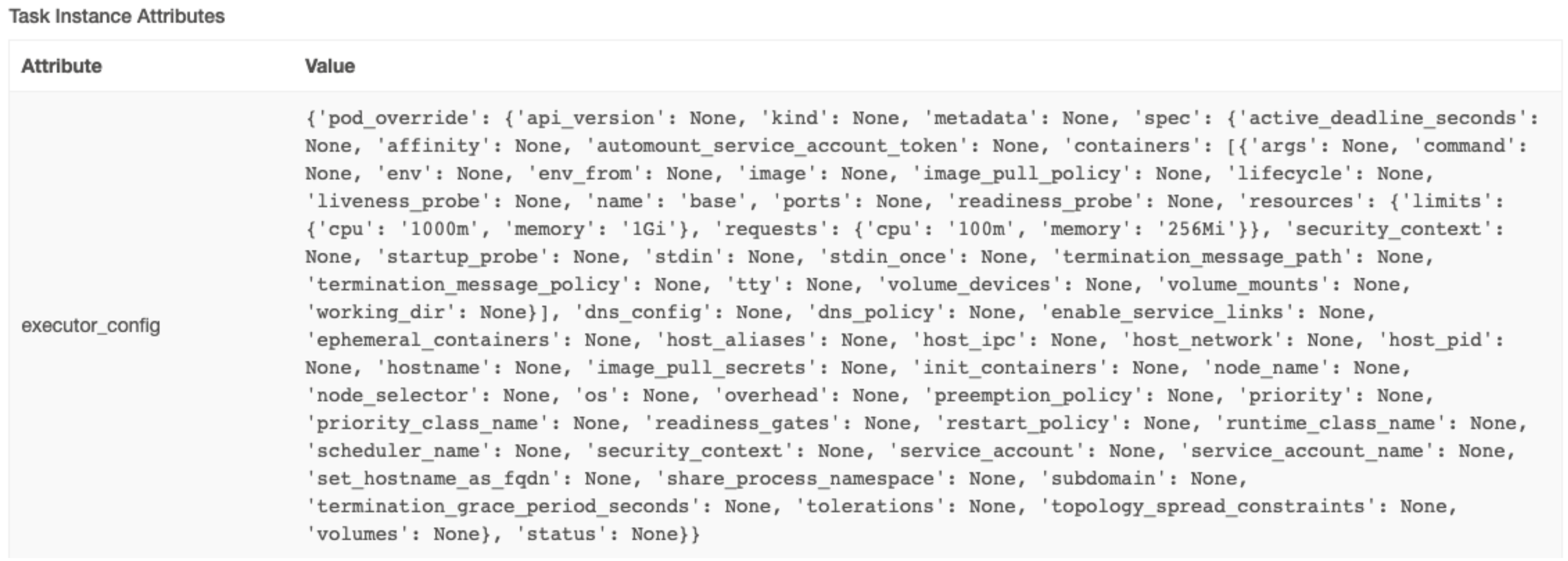
Broken DAG: [/Users/philippe/airflow/dags/my_dag.py] AirflowClusterPolicyViolation: s3_key_sensor_dag:list_files time out is greater than 1 day, 0:00:00

Setting resource requests and limits

```
def task_policy(task: "BaseOperator") -> None:
    executor_config = {
            "pod_override": k8s.V1Pod(
                spec=k8s.V1PodSpec(
                    containers=[
                        k8s.V1Container(
                            name="base",
                            resources=k8s.V1ResourceRequirements(
                                requests={
                                    "cpu": "100m",
                                    "memory": "256Mi",
                                },
                                limits={
                                    "cpu": "1000m",
                                    "memory": "1Gi",
                                },
                            ),
                        )
                    ]
                )
            )
        }

    task.executor_config = executor_config
```

Setting resource requests and limits (2)



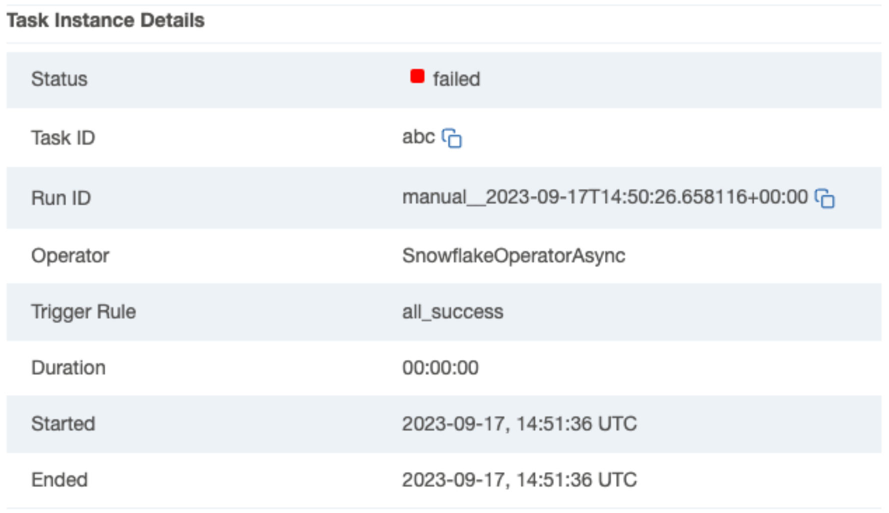
Task Instance Attributes

Attribute	Value
executor_config	{'pod_override': {'api_version': None, 'kind': None, 'metadata': None, 'spec': {'active_deadline_seconds': None, 'affinity': None, 'automount_service_account_token': None, 'containers': [{'args': None, 'command': None, 'env': None, 'env_from': None, 'image': None, 'image_pull_policy': None, 'lifecycle': None, 'liveness_probe': None, 'name': 'base', 'ports': None, 'readiness_probe': None, 'resources': {'limits': {'cpu': '1000m', 'memory': '1Gi'}, 'requests': {'cpu': '100m', 'memory': '256Mi'}}, 'security_context': None, 'startup_probe': None, 'stdin': None, 'stdin_once': None, 'termination_message_path': None, 'termination_message_policy': None, 'tty': None, 'volume_devices': None, 'volume_mounts': None, 'working_dir': None}], 'dns_config': None, 'dns_policy': None, 'enable_service_links': None, 'ephemeral_containers': None, 'host_aliases': None, 'host_ipc': None, 'host_network': None, 'host_pid': None, 'hostname': None, 'image_pull_secrets': None, 'init_containers': None, 'node_name': None, 'node_selector': None, 'os': None, 'overhead': None, 'preemption_policy': None, 'priority': None, 'priority_class_name': None, 'readiness_gates': None, 'restart_policy': None, 'runtime_class_name': None, 'scheduler_name': None, 'security_context': None, 'service_account': None, 'service_account_name': None, 'set_hostname_as_fqdn': None, 'share_process_namespace': None, 'subdomain': None, 'termination_grace_period_seconds': None, 'tolerations': None, 'topology_spread_constraints': None, 'volumes': None}, 'status': None}}

Replacing an operator with its deferrable counterpart

```
def make_snowflake_operators_async(dag: "DAG") -> None:
    from airflow.providers.snowflake.operators.snowflake import SnowflakeOperator
    from astronomer.providers.snowflake.operators.snowflake import
SnowflakeOperatorAsync

    for task_id, task in dag.task_dict.copy().items():
        if isinstance(task, SnowflakeOperator):
            task = SnowflakeOperatorAsync(
                task_id=task.task_id,
                sql=task.sql,
                snowflake_conn_id=task.conn_id,
                database=task.database,
                return_last=task.return_last,
            )
            dag.task_dict["task_id"] = task
```


Task Instance Details

Status	🟥 failed
Task ID	abc
Run ID	manual__2023-09-17T14:50:26.658116+00:00
Operator	SnowflakeOperatorAsync
Trigger Rule	all_success
Duration	00:00:00
Started	2023-09-17, 14:51:36 UTC
Ended	2023-09-17, 14:51:36 UTC

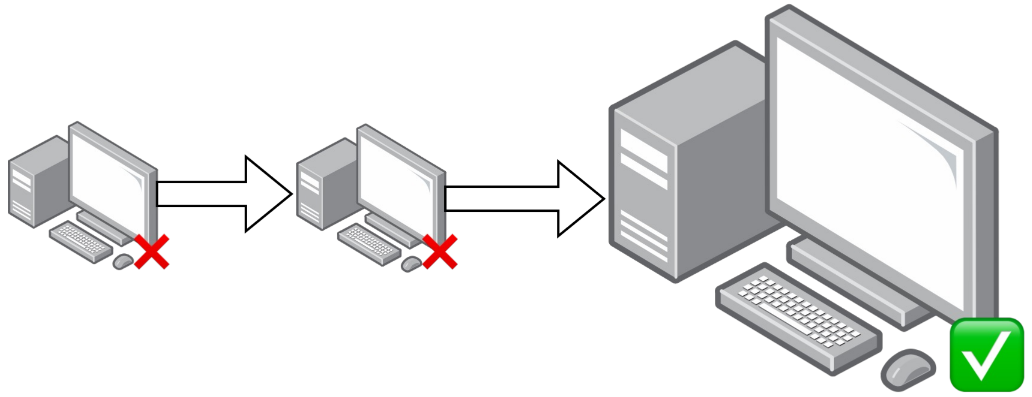
Using a different image depending on operator

```python
def task_policy(task: "BaseOperator") -> None:
    from airflow.providers.apache.spark.operators.spark_submit import (
        SparkSubmitOperator,
    )

    if isinstance(task, SparkSubmitOperator):
        executor_config = {
            "pod_override": k8s.V1Pod(
                spec=k8s.V1PodSpec(
                    containers=[
                        k8s.V1Container(name="base", image="airflow-with-spark"),
                    ]
                )
            )
        }
        task.executor_config = executor_config
        task.doc = "⚠️ Warning! This task has been mutated by your friendly Airflow admin!"
```

Retrying a task on a different queue

```
def task_instance_mutation_hook(task_instance:
TaskInstance):
    if task_instance.try_number >= 3:
        task_instance.queue = "big-machine"
```



Special Case: Mapped Operators

- may run into a problem because most properties of `MappedOperator` are not mutable.
- This isn't generally a problem for deny/skip policies, but it is for mutations.
- Fortunately, there is a workaround.
- You can get past this with the `partial_kwargs`, which is mutable.

```
def task_policy(task:
"BaseOperator") -> None:
    doc_str = "⚠️ Warning! This
task has been mutated by your
friendly Airflow admin!"

    if isinstance(task,
MappedOperator):
        task.partial_kwargs["doc"]
= doc_str
    else:
        task.doc = doc_str
```

Takeaways

- Airflow policy functions are a powerful ⚡ yet relatively unknown 🤫 feature available to Airflow cluster administrators.
- They are essential to a cluster administrator’s toolbox 🧰 to ensure that your Airflow instances are governed properly.
- You should use them. 😉
- But try not to surprise your users! ⚠️

Thank you

ASTRONOMER

(After) Party Under the Stars

Wednesday, September 20th
6:30pm-10:00pm

The Sheraton Centre
123 Queen St W
(7 min walk)

RSVP Now

Airflow Summit
September 19 - 21, 2023
Toronto, Canada
airflowsummit.org

Let's flow together

Workshop

Get Airflow Certified

Thursday, September 21st
12:00 pm in Trinity 4

Marc Lamberti
Head of Customer Education
at Astronomer