Empowering Airflow Users: A framework for performance testing and transparent resource optimization

Bartosz Jankiewicz
bjankiewicz@google.com

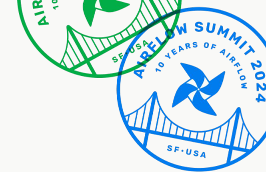
About me

Bartosz Jankiewicz

Engineering Manager
Ex Cloud Composer

Working with Airflow for ~3 years

Motivation



AIP-59

AIP-59 aims to define a testing framework for Apache Airflow.

tldr; Identify performance regressions by introducing regular performance metrics collection mechanism into the Apache Airflow release and deployment process.

https://cwiki.apache.org/confluence/display/AIRFLOW/AIP-59+Performance+tests+framework

Goals

Measure performance changes between Airflow versions.

Identify and communicate changes that affect performance, CPU, memory, disk usage or other key performance characteristics.

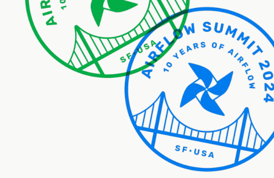
Empower users to measure performance of their own deployments.

Some real stories

Airflow worker memory requirements in Airflow 2.3 are **30% higher** compared to workers in Airflow 2.2 or Airflow 2.1

Airflow memory requirements in Python 3.11 are **10% higher** compared to Python 3.8

Design principles



Focus

Measure Airflow components, not third-party code

Extensibility

Support various Airflow setups (Docker Compose, Cloud Composer, Kubernetes, etc.)

Configurability

Customize scenarios, instances, performance DAGs

How?

Component	Description
Performance DAG	Defines the test scenario (number of DAGs, tasks, task type, etc.)
Instance	Defines the Airflow setup (number of schedulers, worker resources, etc.) and metrics collection mechanism
Test suite	Combines instance and performance DAG, sets placeholder values

Performance DAG

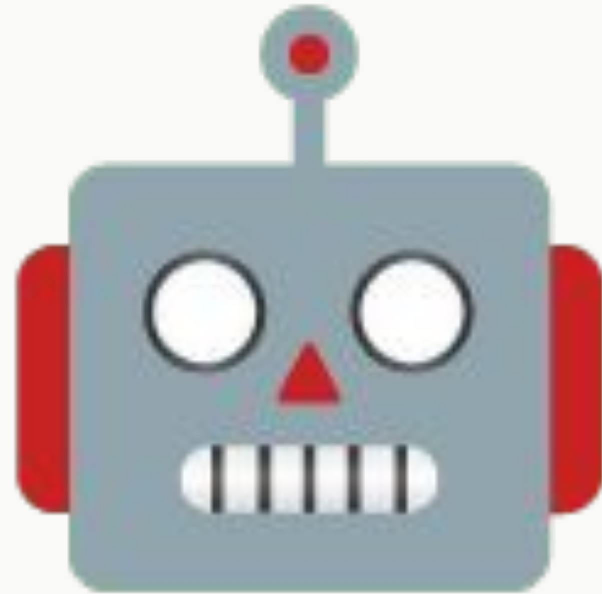
Dynamically creates number of DAGs and tasks

Can be controller by environment variables. Some of them include:

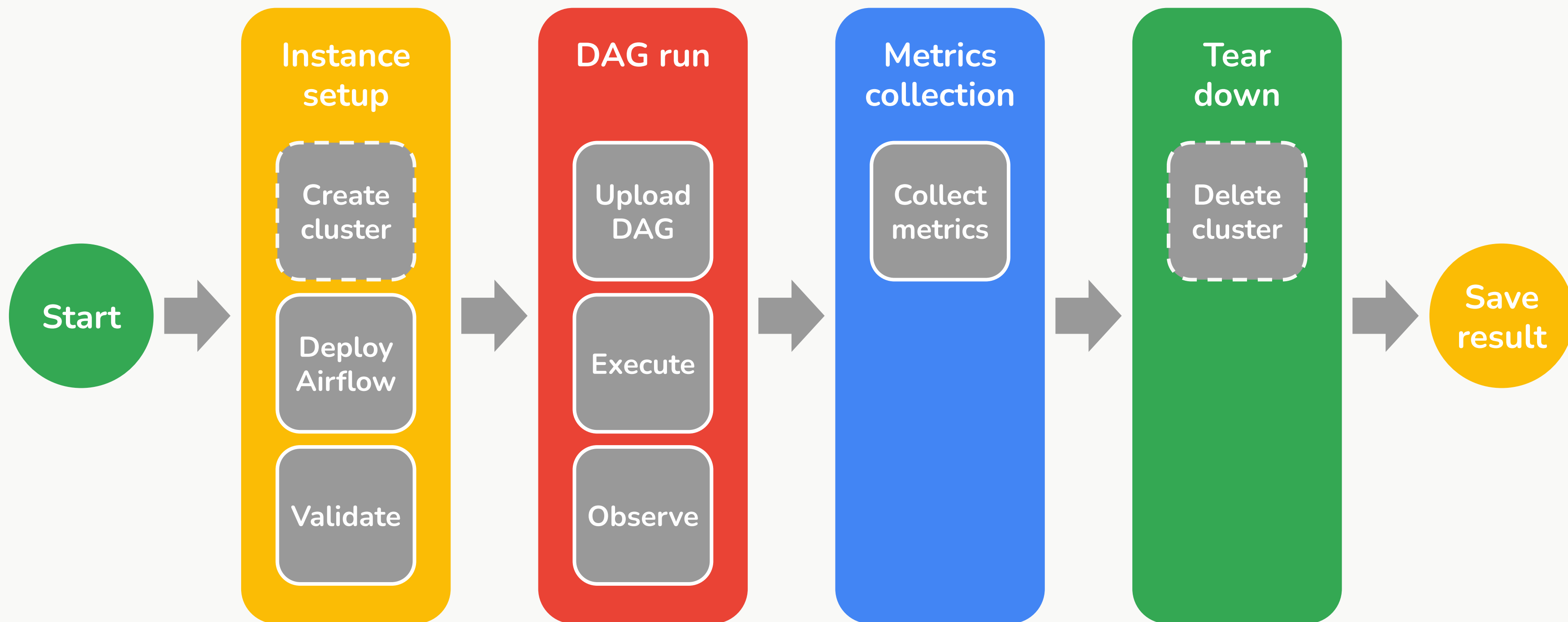
- `PERF_DAGS_COUNT` - number of DAGs to generate
- `PERF_TASKS_COUNT` - tasks count in each DAG
- `PERF_SHAPE` - no structure, linear, grid, star, binary tree
- `PERF_SLEEP_TIME` - time of sleep occurring when each task is executed
- `PERF_OPERATOR_TYPE` - type of operator from predefined set

Instance is a state machine

- Test defines state machine inputs - most importantly the instance type.
- Each instance implements `states_map` method that defines state machine transitions.
- Each state is associated with:
  - Transition to next state **method**
  - **Retryable** property
  - **Sleep time**
- State transition method returns value of the next state.



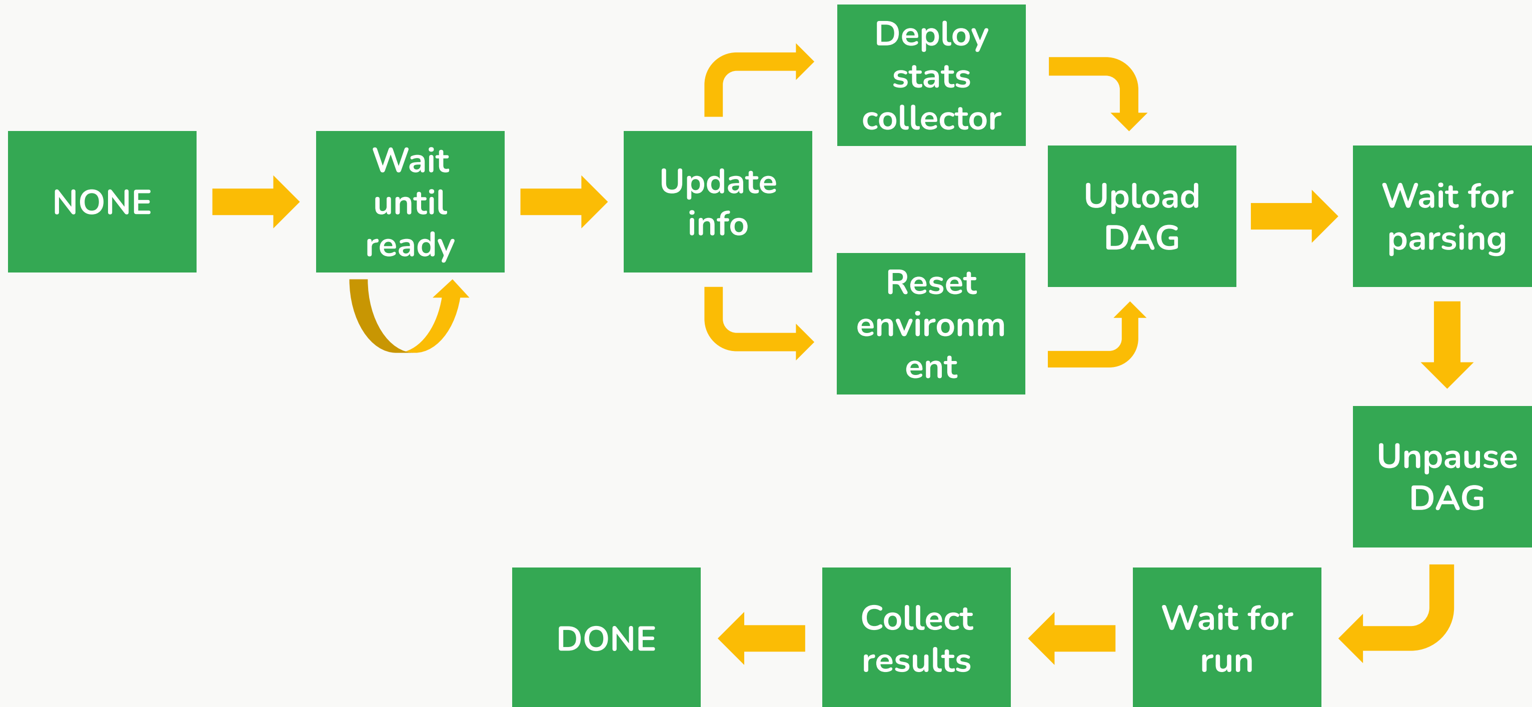
Test life cycle example (simple version)



Example

```
@property
def states_map(self) -> dict[State, Action]:
    """
    Returns a map specifying a method that should be executed for every applicable state
    to move the performance test forward.
    """
    return {
        State.NONE: Action(
            self.prepare_gke_cluster, sleep_time=None, retryable=True
        ),
        State.WAIT_UNTIL_READY: Action(
            self.is_gke_cluster_ready, sleep_time=30.0, retryable=True
        ),
        State.WAIT_UNTIL_CAN_BE_DELETED: Action(
            self.is_gke_cluster_ready, sleep_time=30.0, retryable=True
        ),
        State.DELETING_ENV: Action(
            self._wait_for_deletion, sleep_time=20.0, retryable=True
        ),
        State.UPDATE_ENV_INFO: Action(
            self._update_environment_info, sleep_time=10.0, retryable=True
        ),
        State.WAIT_FOR_DAG: Action(
            self.check_if_dags_have_loaded, sleep_time=30.0, retryable=True
        ),
        State.UNPAUSE_DAG: Action(
            self.unpause_dags, sleep_time=20.0, retryable=True
        ),
        State.WAIT_FOR_DAG_RUN_EXEC: Action(
            self.check_dag_run_execution_status, sleep_time=60.0, retryable=True
        ),
        State.COLLECT_RESULTS: Action(
            self.collect_results, sleep_time=10.0, retryable=True
        ),
    }
```

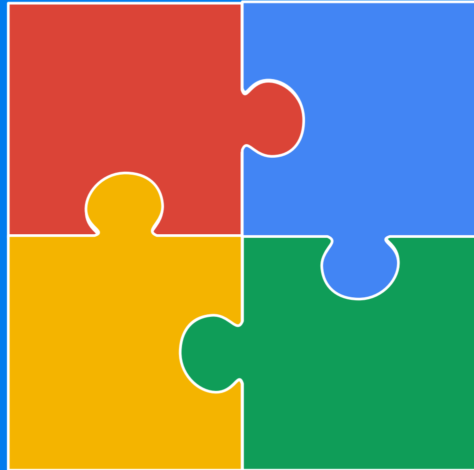
Happy path in (for Cloud Composer)



Metrics

- Set of metrics depends on environment type.
- Typical sources of metrics:
  - Cluster configuration including environment (scheduler count, workers count)
  - Airflow configuration (worker concurrency, parallelism, dag concurrency)
  - DAG run statistics (test duration, run count, min duration, max duration, task durations)
  - Cluster metrics (total cores, cores utilization, memory utilization, restart count)
- Exported to a CSV file

Integration



How to use them?

CI/CD
*for some PRs

upload

compare

Performance metrics storage

Nightly builds

Performance reports

Release process

Integration

- Run tests as part of the build process, combine results with PR
- Export metrics to a dedicated tabular storage
- Review metrics during release process
- Include results in the release documentation

Roadmap

Performance DAG

PR #41961 merged - includes performance DAG code

PR for instance framework well advanced

Instance framework

State machine implementation

Framework for collecting instance metrics from Google Cloud Logging

E2e solution

Framework for collecting metrics from vanilla k8s

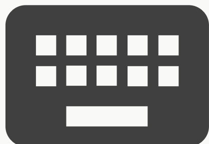
Ready for testing

Documented

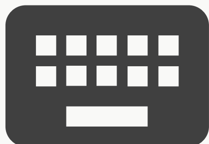
Integration

Start integrating the solution into daily builds

How can I contribute?



Collection of metrics from k8s or other solutions/clouds



Implementation of other than Composer/K8s instance machines

Reviewing the code

Questions?