

# Performance Optimization

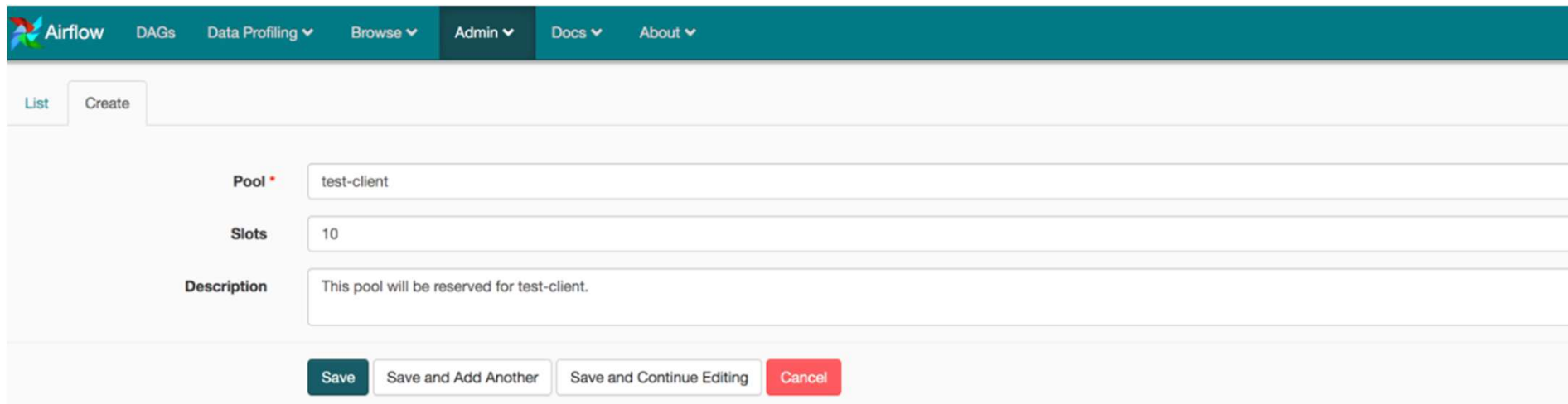
# Managing Task Pools for Resources

# Introduction

- Pools enable to allocate resources, prioritize tasks, and prevent over-utilization, resulting in optimized workflow execution and improved overall performance
- Pools allow to limit the number of task instances that can run simultaneously for a specific set of tasks
- Each pool has a defined size or limit, and when the limit is reached, new task instances will be queued and wait for a slot to become available
- Pools are useful for
  - Managing shared resources
  - Preventing resource over-utilization in your Airflow environment

# Use of Pools

- For example, with the default airflow config settings, and a DAG with 50 tasks to pull data from a REST API, when the DAG starts, you would get 16 workers hitting the API at once and you may get some throttling errors back from your API.
- You can create a pool and give it a limit of 5. Then assign all of the tasks to that pool. Even though you have plenty of free workers, only 5 will run at one time.



The screenshot shows the Airflow Admin interface. The top navigation bar is teal with the Airflow logo and links for DAGs, Data Profiling, Browse, Admin, Docs, and About. Below the navigation bar, there are two tabs: 'List' and 'Create'. The 'Create' tab is active, and the form for creating a new pool is displayed. The form has three main sections: 'Pool', 'Slots', and 'Description'. The 'Pool' section has a text input field with the value 'test-client'. The 'Slots' section has a text input field with the value '10'. The 'Description' section has a text area with the text 'This pool will be reserved for test-client.' At the bottom of the form, there are four buttons: 'Save', 'Save and Add Another', 'Save and Continue Editing', and 'Cancel'.

Airflow Admin interface showing the 'Create' form for a new pool.

Fields and values:

- Pool:** test-client
- Slots:** 10
- Description:** This pool will be reserved for test-client.

Buttons at the bottom:

- Save
- Save and Add Another
- Save and Continue Editing
- Cancel

# Implementation Approach

- We wanted an approach for the best utilisation of resources
- If we start enforcing pools, let's say
  - Client A & B with limits 30 & 20 respectively and
  - Default pool size is 128 for non-pooled clients
- Then following should be possible
  - Non-Pooled tasks are 50, Client A has 30 & Client B has 20 tasks to run
    - All tasks will start running
  - Non-Pooled tasks are 150, Client A has 35 & Client B has 25 tasks to run
    - 30 tasks of Client A & 20 tasks of B and 128 non-pooled will start running
  - Non-Pooled tasks are 150 & Client B has 5 tasks to run
    - 5 tasks of B and 128 non-pooled will start running

# Resource Management

- Pools allow to allocate and manage resources effectively
- Define pools based on the available resources in your environment, such as
  - The number of worker nodes
  - CPU cores
  - memory or any other shared resource
- By limiting the concurrency of tasks using pools, you can prevent resource contention and ensure fair allocation of resources.

# Throttling

- Pools can be used to throttle or limit the rate at which certain tasks or workflows are executed
- For example, if you have tasks that interact with external APIs or services that have rate limits, you can define a pool for those tasks and set the pool size to match the allowed rate limit
- This ensures that your tasks do not exceed the rate limit and helps you comply with API usage policies.

# Prioritization

- Pools can be used to prioritize certain tasks or workflows over others
- By assigning different pool sizes to different sets of tasks, you can control the order in which tasks are executed
- For example, you can define a “high-priority” pool with a larger size and a “low-priority” pool with a smaller size
- This allows high-priority tasks to have more concurrency and be processed faster than low-priority tasks.



# Preventing Resource Over-utilization

- Pools help prevent over-utilization of resources, especially in cases where tasks may consume significant resources or have dependencies on external systems.
- By setting a limit on the pool size, you can prevent excessive resource usage and ensure that other tasks or workflows have enough resources to execute without performance degradation.

# Load Balancing

- Pools can be used for load balancing tasks across multiple workers or nodes
- By setting the pool size based on the capacity of your worker nodes, you can distribute the workload evenly and avoid overloading any single node
- This helps improve the overall performance and stability of your Airflow deployment

# Using multiple pool slots

- Each Airflow task will occupy a single pool slot by default
- They can be configured to occupy more with the `pool_slots` argument
- Particularly useful when several tasks that belong to the same pool don't carry the same “computational weight”.

# Using multiple pool slots

- For instance, consider a pool with 2 slots, `Pool(pool='maintenance', slots=2)`, and the these tasks on right.
- `heavy_task` will use all the slots and the other 2 `light_task1` and `light_task2` will be queued
- Here, in terms of resource usage, the heavy task is equivalent to two light tasks running concurrently.

```
BashOperator(  
    task_id="heavy_task",  
    bash_command="bash backup_data.sh",  
    pool_slots=2,  
    pool="maintenance",  
)  
  
BashOperator(  
    task_id="light_task1",  
    bash_command="bash check_files.sh",  
    pool_slots=1,  
    pool="maintenance",  
)  
  
BashOperator(  
    task_id="light_task2",  
    bash_command="bash remove_files.sh",  
    pool_slots=1,  
    pool="maintenance",  
)
```

# Using multiple pool slots

- This implementation can prevent overwhelming system resources, which (in this example) could occur when a heavy and a light task are running concurrently
- On the other hand, both light tasks can run concurrently since they only occupy one pool slot each, while the heavy task would have to wait for two pool slots to become available before getting executed.

```
BashOperator(  
    task_id="heavy_task",  
    bash_command="bash backup_data.sh",  
    pool_slots=2,  
    pool="maintenance",  
)  
  
BashOperator(  
    task_id="light_task1",  
    bash_command="bash check_files.sh",  
    pool_slots=1,  
    pool="maintenance",  
)  
  
BashOperator(  
    task_id="light_task2",  
    bash_command="bash remove_files.sh",  
    pool_slots=1,  
    pool="maintenance",  
)
```

# Provision of Pools through DAG

- `run_bash_job = BashOperator(  
 task_id='test_task_1', bash_command='echo 1 && sleep 1400m', retries=3,  
 retry_delay=timedelta(seconds=300), pool='test-client', dag=dag)`
- `run_spark_job = SparkOperator(  
 json={"name":"HelloWorld","sparkConf":{},"envVars":{},"className":"HelloWorld",  
 "jar":"https://xyz.com/artifactory/hello.jar",  
 "args":["/usr/local/spark/README.md"],"driverMemory":1024,"driverCores":1,"executorMemory":1024,"executorCores":2,"numExecutors":1},  
 retries=1, retry_delay=timedelta(seconds=5), task_id='Spark_0_TrainingJob', pool='test',  
 dag=dag)`
- `run_bash_job.set_upstream(run_compute_job)`

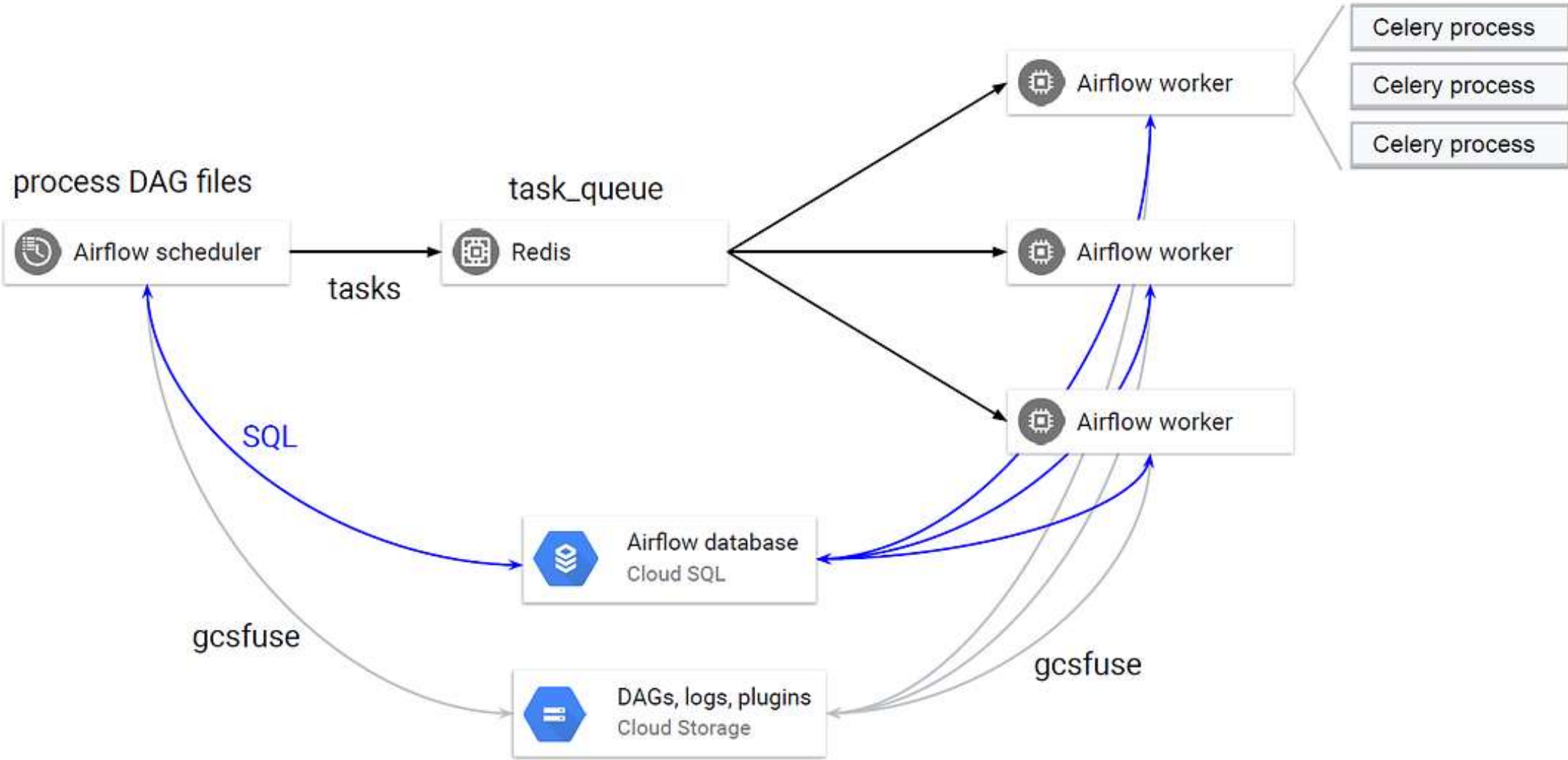
# Provision of Pools through DAG

- The pool parameter can be used in conjunction with `priority_weight` to define priorities in the queue, and which tasks get executed first as slots open up in the pool
- The default `priority_weight` is 1, and can be bumped to any number
- When sorting the queue to evaluate which task should be executed next, we use the `priority_weight`, summed up with all of the `priority_weight` values from tasks downstream from this task
- You can use this to bump a specific important task and the whole path to that task gets prioritised accordingly.
- Tasks will be scheduled as usual while the slots fill up. Once capacity is reached, runnable tasks get queued
- As slots free up, queued tasks start running based on the `priority_weight`

Scaling for 1000+ DAGs

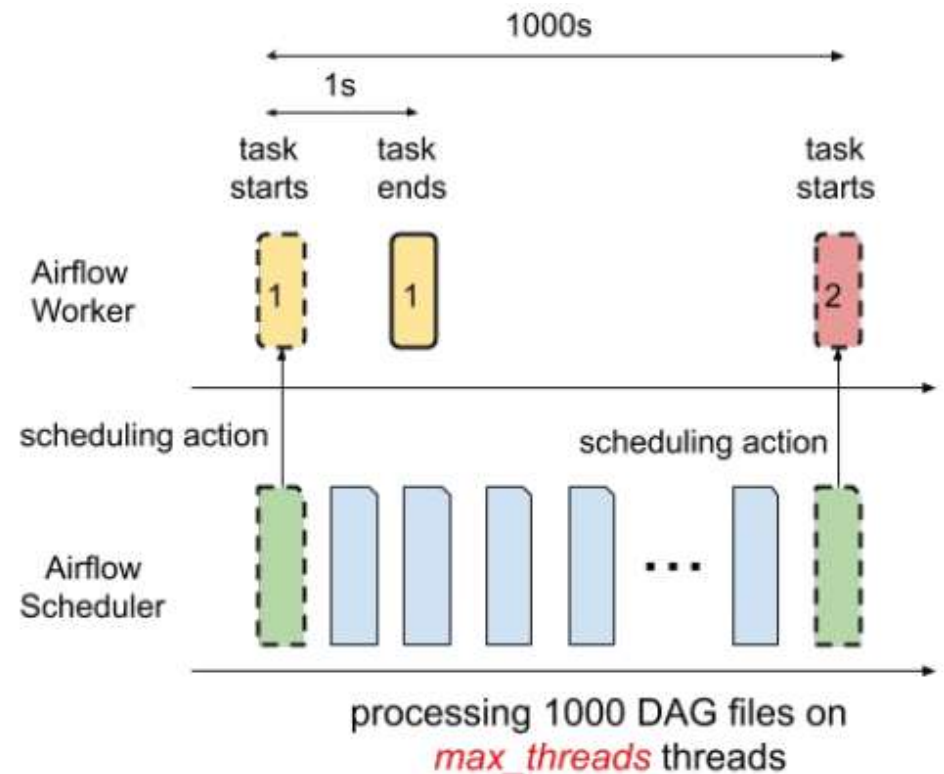
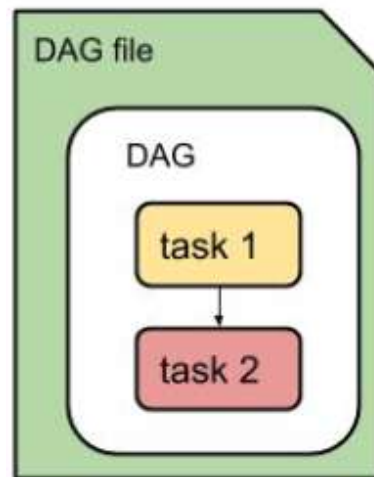


# Airflow Internals



# Airflow Scheduler

- Checks all dags and tasks at a predefined interval to see if all the pre requisites are met by system to trigger next steps
- It monitors and stays in sync with a folder for all DAG objects it may contain, and periodically (every minute or so) inspects active tasks to see whether they can be triggered.



# Airflow Scheduler

- Scheduler takes actions by processing a DAG file:
  - Starts/ends a DAG run
  - Triggers next task
- How fast a scheduler takes actions depends on
  - Total time to process all DAG files
    - $\text{\#files} * \text{time\_each\_file}$
    - Multiple DAGs (~100) in one file is more efficient
  - Other files in DAG folder
  - Use `max_threads` processes

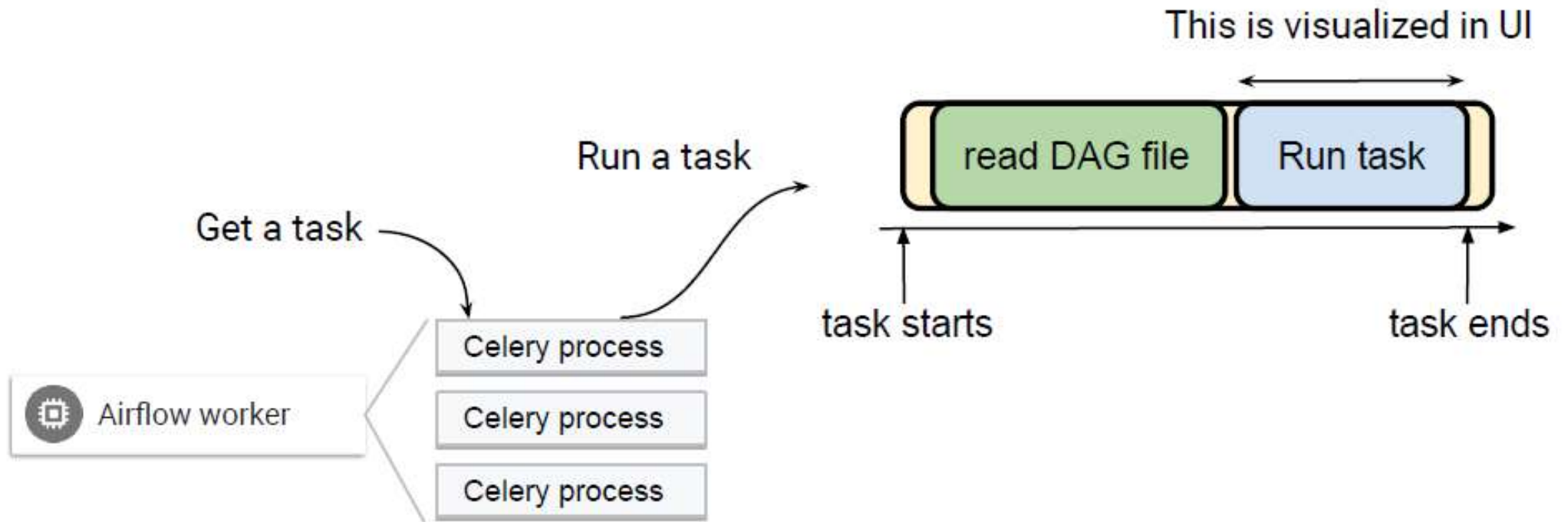
# Task Queue

- All the tasks which were open for execution as per Airflow scheduler is submitted to a task queue
- In case of google composer, tasks queue is built on top of redis

# Executors

- There are multiple strategies to perform this
- Most commonly used strategy is celery executor
- CeleryExecutor is one of the ways you can scale out the number of workers
- With K8s running in the background, this makes more sense and natural fit to data intensive applications

# Workers



# Workers

- For short tasks:
  - The DAG file is read before running task
  - A large DAG file slows down the task
- For long tasks (e.g., blocking, sleeping tasks):
  - A task takes away a celery process
  - Others can not run if no available celery processes


# Airflow Database

- In most cases its a My SQL database or My SQL as PaaS
- The responsibility of his component is to save monitoring, history and audit details of executions of job




# Airflow Scalability Considerations

- Scalability

 Composer

[← Environment details](#) [REFRESH](#) [DELETE](#)

 **stackdriver-test-core8**

This environment is running.

[ENVIRONMENT CONFIGURATION](#) [AIRFLOW CONFIGURATION OVERRIDES](#) [ENVIRONMENT VARIABLES](#) [LABELS](#) [PYPI PACKAGES](#)

[EDIT](#)

The configuration properties are stored in a file called `airflow.cfg` the first time you run Apache Airflow. You can choose to change these properties and override the default values. Sections include: core, cli, api, operators, hive, webserver, email, smtp, celery, celery\_broker\_transport\_options, dask, scheduler, ldap, mesos, kerberos, github\_enterprise, admin.

celery	
worker_concurrency	128
core	
dag_concurrency	1000
parallelism	1000
scheduler	
max_threads	64

# Key Airflow Configurations

# max-threads =8

- Its a scheduler configuration
- Number of processes to process DAG files
- estimate = num\_cpu\_per\_node

`worker_concurrency = 32`

- Number of celery processes per Airflow worker
- $\text{estimate} = \frac{\text{num\_dags} * \text{num\_tasks\_per\_dag} * \text{execution\_duration\_per\_task}}{\text{dag\_scheduling\_period} / \text{num\_airflow\_workers}}$
- $\text{estimate} = \text{num\_cpu\_per\_node} * 6$
- use lesser of the two estimates

# parallelism = 96

- Its a core configuration
- The amount of parallelism as a setting to the executor. This defines the max number of task instances that should run simultaneously
- $\text{estimate} = \text{worker\_concurrency} * \text{num\_airflow\_workers}$

`dag_concurrency = 96`

- The number of task instances allowed to run concurrently by the scheduler
- `estimate = parallelism`

`non_pooled_task_slot_count = 96`

- When not using pools, tasks are run in the “default pool”, whose size is guided by this config element
- `estimate = parallelism`

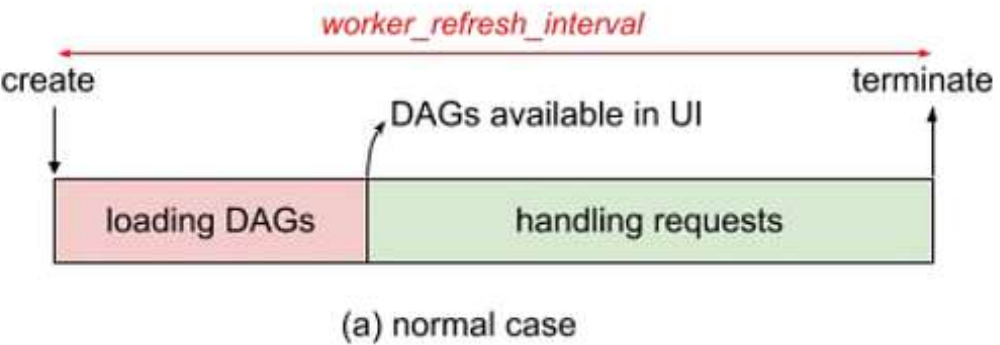
# Scalability : RDBMS Database Configuration

- Scheduler, workers, and webserver communicate with Airflow Database to make progress
- Cloud SQL database hosted in Google managed project with a 2 vCPU, 7.5G memory instance by default
- For most cases, customers don't have to worry about DB as it's sufficient for  $O(1000)$  dags.
- Database may be a bottleneck for a high amount of concurrent DAGs, e.g., > 10K DAGs



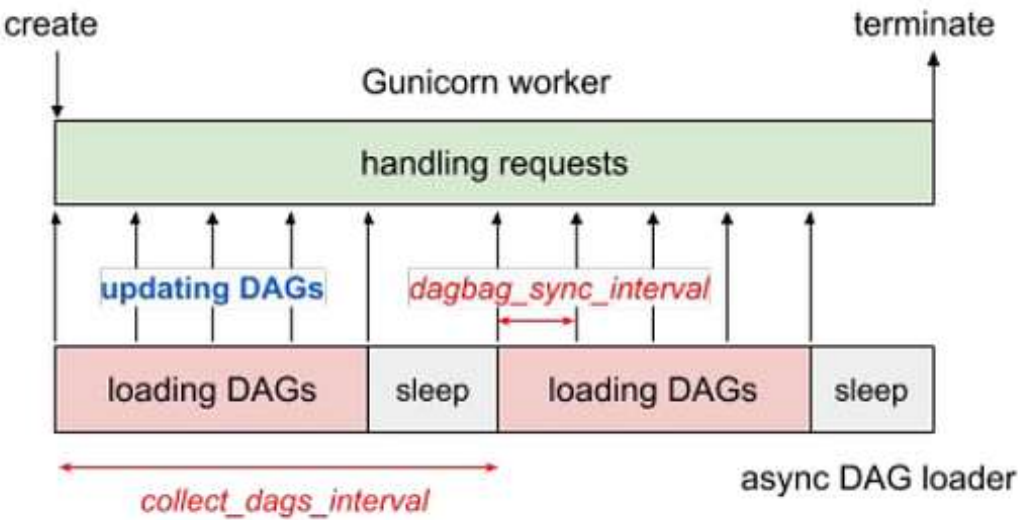
# Scalability : Webserver Configuration

## Original Airflow webserver:



(b) loading DAGs takes too long: webserver down

## 😊 Asynchronous DAGs Loader:



(c) loading DAGs in a separate process (async DAG loader), webserver is never blocked

# Scalability : Webserver Configuration

- `worker_refresh_interval = 3600`
- `async_dagbag_loader = True`
- `workers = 1` (default is 3)

# Steps of Scaling

# Machine Type

- Decide machine type: how many CPUs does the scheduler need?
  - Throughput
    - e.g., 1K DAGs completes in 1 hour
  - Delay
    - e.g., task triggering delay < 1 min
  - Select number of CPUs based on current Composer use cases
  - Assume linear performance gain: 2x CPUs = 2x throughput, ½ Delay
- Number of nodes can be changed on-the-fly
- Set proper Airflow configurations

# Example — 10 k DAGS

- Machine type
  - 16 vCPU
- Machine number
  - 8
- Number of DAGs
  - 10K DAGs with 1 bash task
- SQL instance
  - 2vCPUs (default, 90% SQL CPU usage)
- Completion time
  - ~ 45 min (~20min, if SQL CPU=4, 60% SQL CPU usage)
- Throughput
  - 7 tasks/s (10 tasks/s, if SQL CPU=4)

## Example — 10 k DAGS - Airflow Config

- [celery]
- worker\_concurrency = 96 # Celery process per worker
- [core]
- non\_pooled\_task\_slot\_count = 1000 # tasks sent for running at most
- dag\_concurrency = 1000 # workflows scheduled for running at most
- parallelism = 1000 # total number of concurrent tasks on all workers
- [scheduler]
- max\_threads = 32 # number of processes to process DAG files

# Example — 100K DAGs

- Machine type
  - 64 vCPU
- Machine number
  - 8
- Number of DAGs
  - 100K DAGs with 1 bash task
- SQL instance
  - 8vCPUs (need to contact support to modify it)
- Completion time
  - ~ 2 hour
- Throughput
  - 15 tasks/s

## Example — 100K DAGs - Airflow Config

- [celery]
  - worker\_concurrency = 256 # Celery process per worker
- [core]
  - non\_pooled\_task\_slot\_count = 2000 # tasks sent for running at most
  - dag\_concurrency = 2000 # workflows scheduled for running at most
  - parallelism = 2000 # total number of concurrent tasks on all workers
- [scheduler]
  - max\_threads = 64 # number of processes to process DAG files



Monitoring

# Log types

- Airflow logs

- These logs are associated with single DAG tasks
- You can view the task logs in the Cloud Storage logs folder associated with the Cloud Composer environment
- You can also view the logs in the Airflow web interface.

- Streaming logs

- These logs are a superset of the logs in Airflow
- To access streaming logs, you can go to the logs tab of Environment details page in Google Cloud console, use the Cloud Logging, or use Cloud Monitoring.

# Logs in Cloud Storage

- Cloud Composer creates a Cloud Storage bucket
- Cloud Composer stores logs for single DAG tasks in logs folder in the bucket

# Log folder directory structure

- The logs folder includes folders for each workflow that has run in the environment
- Each workflow folder includes a folder for its DAGs and sub-DAGs
- Each folder contains log files for each task
- The task filename indicates when the task started.
- You must manually delete logs from Cloud Storage.

```
us-central1-my-environment-60839224-bucket
├── dags
│   ├── dag_1
│   ├── dag_2
│   └── ...
├── logs
│   ├── dag_1
│   │   ├── task_1
│   │   │   ├── datefile_1
│   │   │   ├── datefile_2
│   │   │   └── ...
│   │   └── task_2
│   │       ├── datefile_1
│   │       ├── datefile_2
│   │       └── ...
│   └── dag_2
│       └── ...
```

# Viewing streaming logs in the Google Cloud console

- These logs can be viewed on the logs tab of the Environment details page
- Cloud Composer produces the following logs:
  - airflow: The uncategorized logs that Airflow pods generate.
  - airflow-upgrade-db: The logs Airflow database initialization job generates (previously airflow-database-init-job).
  - airflow-scheduler: The logs the Airflow scheduler generates.
  - dag-processor-manager: The logs of the DAG processor manager (the part of the scheduler that processes DAG files).
  - airflow-triggerer: The logs the Airflow triggerer generates.
  - airflow-webserver: The logs the Airflow web interface generates.
  - airflow-worker: The logs generated as part of workflow and DAG execution.

# Viewing streaming logs in the Google Cloud console

- Cloud Composer produces the following logs:
  - `clouddaudit.googleapis.com/activity`: The logs Admin Activity generates.
  - `composer-agent`: The logs generated as part of create and update environment operations.
  - `gcs-syncd`: The logs generated by the file syncing processes.
  - `build-log-worker-scheduler`: The logs from the local build of the Airflow worker image (during upgrades and Python package installation).
  - `build-log-webserver`: The logs from the build of the Airflow webserver image (during upgrades and python package installation).
  - `airflow-monitoring`: The logs that Airflow monitoring generates.

# Steps of scaling up: my DAG is slow?

- Check whether suggested Airflow configurations are applied
- Check resource usage:
  - If the scheduler uses 100% CPU, other nodes are under-utilized:
    - Scheduler-bounded: upgrade machine type to have more CPUs per node
  - If all nodes are under-utilized:
    - Are there many long running tasks? The number of Celery processes may be not enough, try to increase `worker_concurrency`
    - Are there many DAGs? e.g., 10K- Database may be the bottleneck, contact the team to upgrade database
    - Otherwise, if all nodes are fully utilized, check `task_queue_length` in Stackdriver monitoring:
      - If it increases to infinity, it is worker-bounded
        - Increase number of nodes and revise

Thanks