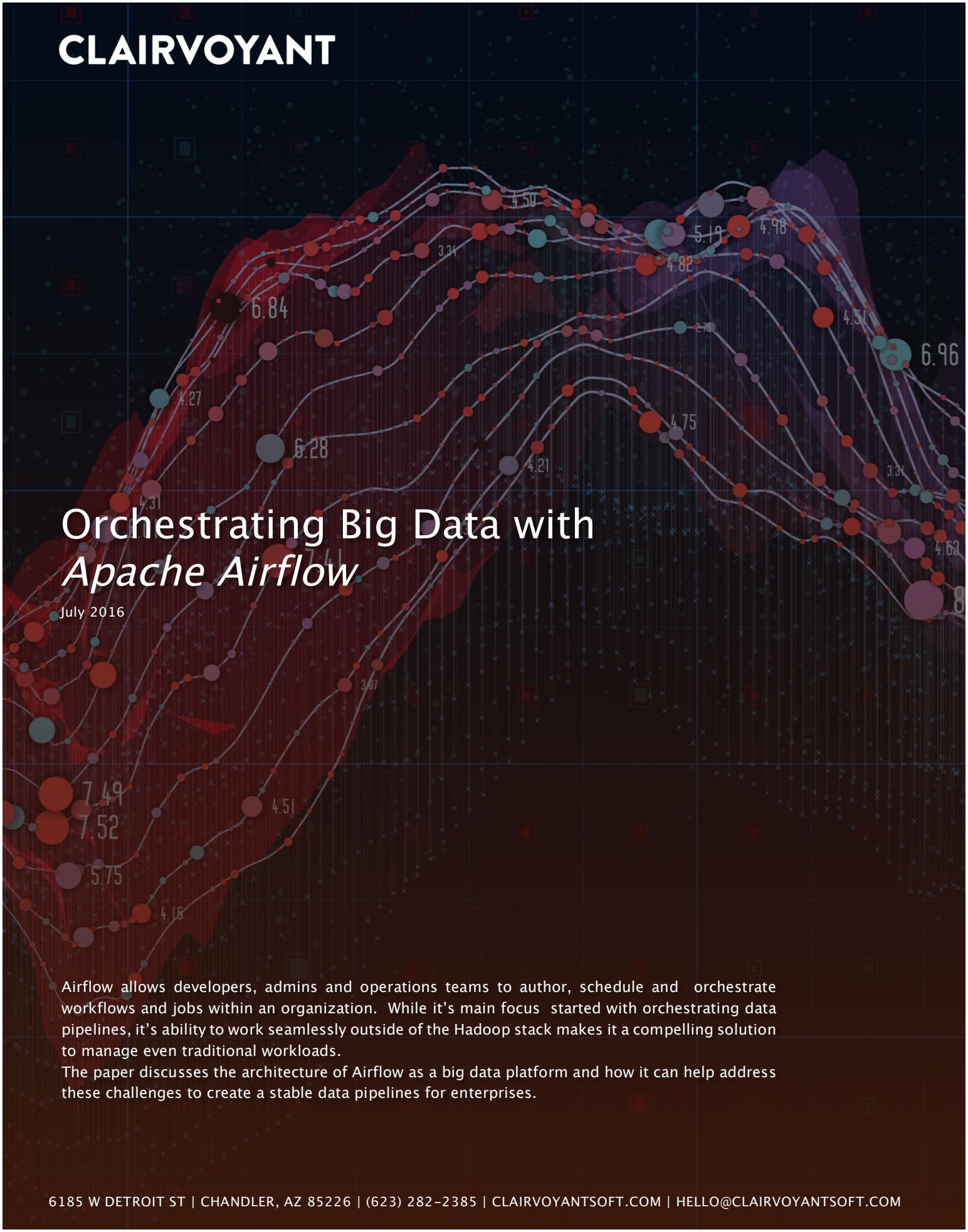
CLAIRVOYANT



Orchestrating Big Data with *Apache Airflow*

July 2016

Airflow allows developers, admins and operations teams to author, schedule and orchestrate workflows and jobs within an organization. While it's main focus started with orchestrating data pipelines, it's ability to work seamlessly outside of the Hadoop stack makes it a compelling solution to manage even traditional workloads.

The paper discusses the architecture of Airflow as a big data platform and how it can help address these challenges to create a stable data pipelines for enterprises.

6185 W DETROIT ST | CHANDLER, AZ 85226 | (623) 282-2385 | CLAIRVOYANTSOFT.COM | HELLO@CLAIRVOYANTSOFT.COM

OVERVIEW

Data Analytics is playing a key role in the decision-making process at various stages of business in many industries. Data is being generated at a very fast pace through various sources across the business. Applications that automate the business processes are literally fountains of data today. Implementing solutions for use cases like "real time data ingestion from various sources", "processing the data at different levels of the data ingestion" and preparing the final data for analysis is a serious challenge given the dynamic nature of the data that is being generated. Proper orchestrating, scheduling, managing and monitoring the data pipelines is a critical task for any data platform to be stable and reliable. The dynamic nature of the data sources, data inflow rates, data schema, processing needs, etc., the work flow management (pipeline generation / maintenance/monitoring) creates these challenges for any data platform.

This whitepaper provides a view on some of the open source tools available today. The paper also discusses the unique architecture of Airflow as a big data platform and how it can help address these challenges to create a stable data platform for enterprises. In addition, ingestion won't be halted at the first sign of trouble.

INTRODUCTION TO AIRFLOW

Airflow is a platform to programmatically author, schedule and monitor data pipelines that meets the need of almost all the stages of the lifecycle of Workflow Management. The system has been built by Airbnb on the below four principles:

- **Dynamic:** Airflow pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.
- **Extensible:** Easily define your own operators, executors and extend the library so that it fits the level of abstraction that suits your environment.
- **Elegant:** Airflow pipelines are lean and explicit. Parameterizing your scripts is built into the core of Airflow using the powerful Jinja templating engine.
- **Scalable:** Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale to infinity.

Basic concepts of Airflow

- **DAGs: Directed Acyclic Graph** – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.
    - DAGs are defined as python scripts and are placed in the DAGs folder (could be any location, but needs to be configured in the airflow config file).
    - Once a new DAG is placed into the DAGS folder, the DAGS are picked up by Airflow automatically within a minute's time.
- **Operators:** An operator describes a single task in a workflow. While DAGs describe *how* to run a workflow, Operators determine what gets done.
    - Task: Once an operator is instantiated using some parameters, it is referred to as a "task"
    - Task Instance: A task executed at a time is called Task Instance.
- **Scheduling the DAGs/Tasks:** The DAGs and Tasks can be scheduled to be run at certain frequency using the below parameters.
    - **Schedule interval:** Determines when the DAG should be triggered. This can be a cron expression or a datetime object of python.
- **Executors**: Once the DAGs, Tasks and the scheduling definitions are in place, someone need to execute the jobs/tasks. Here is where Executors come into picture.
    - There are three types of executors provided by Airflow out of the box.
        - **Sequential**: A Sequential executor is for test drive that can execute the tasks one by one (sequentially). Tasks cannot be parallelized.
- **Local:** A local executor is like Sequential executor. But it can parallelize task instances locally.
- **Celery:** Celery executor is a open source Distributed Tasks Execution Engine that based on message queues making it more scalable and fault tolerant. Message queues like RabbitMQ or Redis can be used along with Celery.
- This is typically used for production purposes.

Airflow has an edge over other tools in the space

Below are some key features where Airflow has an upper hand over other tools like Luigi and Oozie:

- Pipelines are configured via **code** making the pipelines dynamic
- A graphical representation of the DAG instances and Task Instances along with the metrics.
- Scalability: Distribution of Workers and Queues for Task execution
- Hot Deployment of DAGS/Tasks
- Support for Celery, SLAs, great UI for monitoring matrices
- Has support for Calendar schedule and Crontab scheduling
- Backfill: Ability to rerun a DAG instance in case of a failure.

- Variables for making the changes to the DAGS/Tasks quick and easy

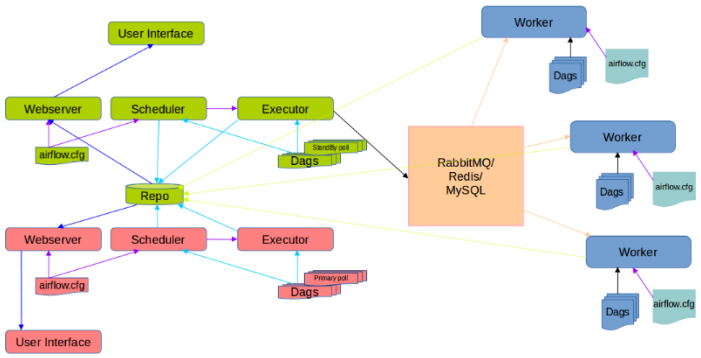
ARCHITECTURE OF AIRFLOW

Airflow typically constitutes of the below components.

- **Configuration file:** All the configuration points like "which port to run the web server on", "which executor to use", "config related to RabbitMQ/Redis", workers, DAGS location, repository etc. are configured.
- **Metadata database (MySQL or postgres):** The database where all the metadata related to the DAGS, DAG runs, tasks, variables are stored.
- **DAGs (Directed Acyclic Graphs):** These are the Workflow definitions (logical units) that contains the task definitions along with the dependencies info. These are the actual jobs that the user would be like to execute.
- **Scheduler:** A component that is responsible for triggering the DAG instances and job instances for each DAG. The scheduler is also responsible for invoking the Executor (be it Local or Celery or Sequential)
- **Broker** (Redis or RabbitMQ): In case of a Celery executor, the broker is required to hold the messages and act as a communicator between the executor and the workers.
- **Worker nodes:** The actual workers that execute the tasks and return the result of the task.
- **Web server:** A web server that renders the UI for Airflow through which one can view the DAGs, its status, rerun, create variables, connections etc.

HOW IT WORKS

- Initially the primary (instance 1) and standby (instance 2) schedulers would be up and running. The instance 1 would be declared as primary Airflow server in the MySQL table.
- The DAGs folder for primary instance (instance 1) would contain the actual DAGs and the DAGs folder and the standby instance (instance 2) would contain Counterpart Poller (PrimaryServerPoller).
    - Primary server would be scheduling the actual DAGs as required.
    - Standby server would be running the PrimaryServerPoller which would continuously poll the Primary Airflow scheduler.
- Let's assume, the Primary server has gone down. In that case, the PrimaryServerPoller would detect the same and
    - Declare itself as the primary Airflow server in the MySQL table.
    - Move the actual DAGs out of DAGs folder and moves the PrimaryServerPoller DAG into the DAGs folder on the older primary server (instance 1)
    - Move the actual DAGs into DAGs folder and move the PrimaryServerPoller DAG out of DAGs folder on the older standby (instance 2).
- So here, the Primary and Standby servers have swapped their positions.
- Even if the airflow scheduler on the current standby server (instance 1) comes back, since there would be only the CounterServerPoller DAG running on it, there would be no harm. And this server (instance 1) would remain to be standby till the current Primary server (instance 2) goes down.
- In case the current primary server goes down, the same process would repeat and the airflow running on instance 1 would become the Primary server.



DEPLOYMENT VIEWS

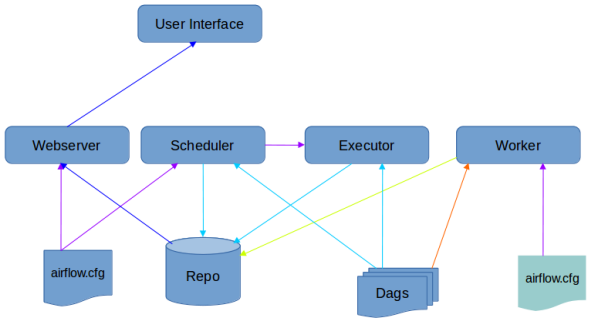
Based on the needs, one may have to go with a simple setup or a complex setup of Airflow. There are different ways Airflow can be deployed (especially from an Executor point of view). Below are the deployment options along with the description for each.

Standalone mode of deployment

Description: As mentioned in the above section, the typical installation of Airflow will start as follows.

- **Configuration file (airflow.cfg)**: which contains the details of where to pick the DAGs from, what Executor to run, how frequently the scheduler should poll the DAGs folder for new definitions, which port to start the webserver on etc.
- **Metadata Repository**: Typically, MySQL or postgres database is used for this purpose. All the metadata related to the DAGs, their metrics, Tasks and their statuses, SLAs, Variables, etc. are stored here.

- **Web Server**: This renders the beautiful UI that shows all the DAGs, their current states along with the metrics (which are pulled from the Repo).
- **Scheduler**: This reads the DAGs, put the details about the DAGs into Repo. It initiates the Executor.
- **Executor**: This is responsible for reading the schedule interval info and creates the instances for the DAGs and Tasks into Repo.
- **Worker:** The worker reads the tasks instances and perform the tasks and writes the status back to the Repo.



Distributed mode of deployment

Description: The description for most of the components mentioned in the Standalone section remain the same except for the Executor and the workers.

- **RabbitMQ**: RabbitMQ is the distributed messaging service that is leveraged by Celery Executor to put the task instances into. This is where the workers would typically read the tasks for execution. Basically, there is a broker URL that is exposed by RabbitMQ for the Celery Executor and Workers to talk to.
- **Executor:** Here the executor would be Celery executor (configured in airflow.cfg). The Celery executor is configured to point to the RabbitMQ Broker.
- **Workers:** The workers are installed on different nodes (based on the requirement) and they are configured to read the tasks info from the RabbitMQ brokers. The workers are also configured with a Worker_result_backend which typically can be configured to be the MySQL repo itself.

The important point to be noted here is:

The Worker nodes is nothing but the airflow installation. The DAG definitions should be in sync on all the nodes (both the primary airflow installation and the Worker nodes)

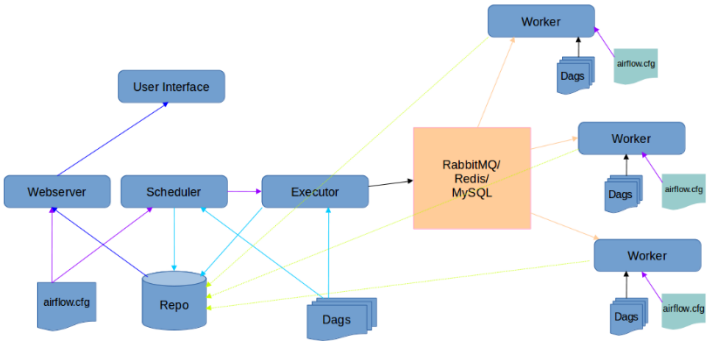
Distributed mode of deployment with High Availability set up

Description: As part of the setup for high availability of Airflow installation, we are assuming if MySQL repository is configured to be highly available and RabbitMQ would be highly available. The focus is on how to make the airflow components like the Web Server and the Scheduler highly available.

The description for most of the components would remain the same as above. Below are what changes:

- New airflow instance (standby): There would be another instance of airflow setup as a standby.
  - The ones shown in Green is the primary airflow instance.
  - The one in red is the stand by one.
- A new DAG must be put in place. Something called "CounterPart Poller". The purpose of this DAG would be two-fold
  - To continuously poll the counterpart scheduler to see if it is up and running.
  - If the counterpart instance is not reachable (which means the instance is down),
- Declare the current airflow instance as the Primary
- Move the DAGs of the (previous) primary instance out of DAGs folder and move the Counterpart Poller DAG into the DAGs folder.
- Move the actual DAGs into the DAGs folder and move the Counterpart Poller out of DAGs folder on the standby server (the one which declares itself as primary now).

Note that the declaration as primary server by the instances can be done as a flag in some MySQL table.



TYPICAL STAGES OF WORKFLOW MANAGEMENT

The typical stages of the life cycle for Workflow Management of Big Data are as follows:

- Create Jobs to interact with systems that operate on Data
  - Use of tools/products like: Hive / Presto / HDFS/Postgres/S3 etc.
- (Dynamic) Workflow creation

    - Based on the number of sources, size of data, business logic, variety of data, changes in the schema, and the list goes on.
- Manage Dependencies between Operations
    - Upstream, Downstream, Cross Pipeline dependencies, Previous Job state, etc.
- Schedule the Jobs/Operations
    - Calendar schedule, Event Driven, Cron Expression etc.
- Keep track of the Operations and the metrics of the workflow
    - Monitor the current/historic state of the jobs, the results of the jobs etc.
- Ensuring Fault tolerance of the pipelines and capability to back fill any missing data, etc.

This list grows as the complexity increases.

TOOLS THAT SOLVE WORKFLOW MANAGEMENT

There are a many Workflow Management Tools in the market. Some have support for Big Data operations out of the box, and some that need extensive customization/extensions to support Big Data pipelines.

- **Oozie**: Oozie is a workflow scheduler system to manage Apache Hadoop jobs
- **BigDataScript**: BigDataScript is intended as a scripting language for big data pipeline
- **Makeflow**: Makeflow is a workflow engine for executing large complex workflows on clusters, and grids
- **Luigi**: Luigi is a Python module that helps you build complex pipelines of batch jobs. (This is a strong contender for Airflow)
- **Airflow**: Airflow is a platform to programmatically author, schedule and monitor workflows
- **Azkaban**: Azkaban is a batch workflow job scheduler created at LinkedIn to run Hadoop jobs
- **Pinball**: Pinball is a scalable workflow manager developed at Pinterest

Most of the mentioned tools meets the basic need of the workflow management. When it comes to dealing with the complex workflows, only few of the above shine. Luigi, Airflow, Oozie and Pinball are the tools preferred (and are being used in Production) by most teams across the industry.

None of the existing resources (on the web) talk about architecture and about the setup of Airflow in production with CeleryExecutor and more importantly on how Airflow needs to be configured to be highly available. Hence here is an attempt to share that missing information.

INSTALLATION STEPS FOR AIRFLOW AND HOW IT WORKS.

Install pip

Installation steps

```
sudo yum install epel-release

sudo yum install python-pip python-wheel
```

Install Erlang

Installation steps

```
sudo yum install wxGTK

sudo yum install erlang
```

RabbitMQ

Installation steps

```
wget https://www.rabbitmq.com/releases/rabbitmq-server/v3.6.2/rabbitmq-server-3.6.2-1.noarch.rpm

sudo yum install rabbitmq-server-3.6.2-1.noarch.rpm
```

Celery

Installation steps

```
pip install celery
```

Airflow: Pre-requisites

Installation steps

```
sudo yum install gcc-gfortran libgfortran numpy redhat-rpm-config python-devel gcc-c++
```

Airflow

Installation steps

```
# create a home directory for airflow
mkdir ~/airflow
# export the location to AIRFLOW_HOME variable
export AIRFLOW_HOME=~/airflow
pip install airflow
```

Initialize the Airflow Database

Installation Steps

```
airflow initdb
```

By default, Airflow installs with SQLite DB. Above step would create a airflow.cfg file within "$AIRFLOW_HOME"/ directory. Once this is done, you may want to change the Repository database to some well-known (Highly Available) relations database like "MySQL", Postgres etc. Then reinitialize the database (using airflow initdb command). That would create all the required tables for airflow in the relational database.

Start the Airflow components

Installation steps

```
# Start the Scheduler
airflow scheduler
# Start the Webserver
airflow webserver
# Start the Worker
airflow worker
```

Here are few important Configuration points in airflow.cfg file

- Dags_folder = /root/airflow/DAGS
    - The folder where your airflow pipelines live
- executor = LocalExecutor
    - The executor class that airflow should use.
- sql_alchemy_conn = mysql://root:root@localhost/airflow
    - The SqlAlchemy connection string to the metadata database.
- base_url = http://localhost:8080
    - The hostname and port at which the Airflow webserver runs
- broker_url = sqla+mysql://root:root@localhost:3306/airflow
    - The Celery broker URL. Celery supports RabbitMQ, Redis and experimentally a sqlalchemy database
- celery_result_backend = db+mysql://root:root@localhost:3306/airflow
    - A key Celery setting that determines the location of where the workers write the results.

This should give you a running airflow instance and set you on the path to run it in production.

If you have any questions, feedback – we would love to hear from you, do drop in a line to airflow@clairvoyantsoft.com. We have used Airflow extensively in production and would love to hear more about your needs and thoughts.

ABOUT CLAIRVOYANT

Clairvoyant is a global technology consulting and services company. We help organizations build innovative products and solutions using big data, analytics, and the cloud. We provide best-in-class solutions and services that leverage big data and continually exceed client expectations. Our deep vertical knowledge combined with expertise on multiple, enterprise-grade big data platforms helps support purpose-built solutions to meet our client's business needs.