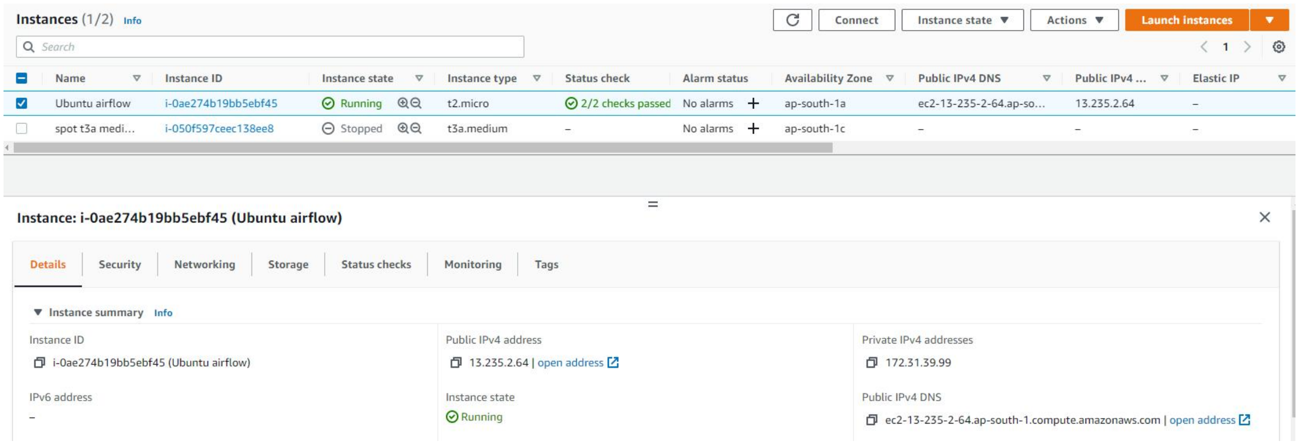
Deployment of airflow on EC2 by Manan Sankhla

Initially airflow is deployed on amazon EC2 instance running with mere t2.micro instances.



	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP
✓	Ubuntu airflow	i-0ae274b19bb5ebf45	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1a	ec2-13-235-2-64.ap-so...	13.235.2.64	–
	spot t3a medi...	i-050f597ceec138ee8	Stopped	t3a.medium	–	No alarms	ap-south-1c	–	–	–

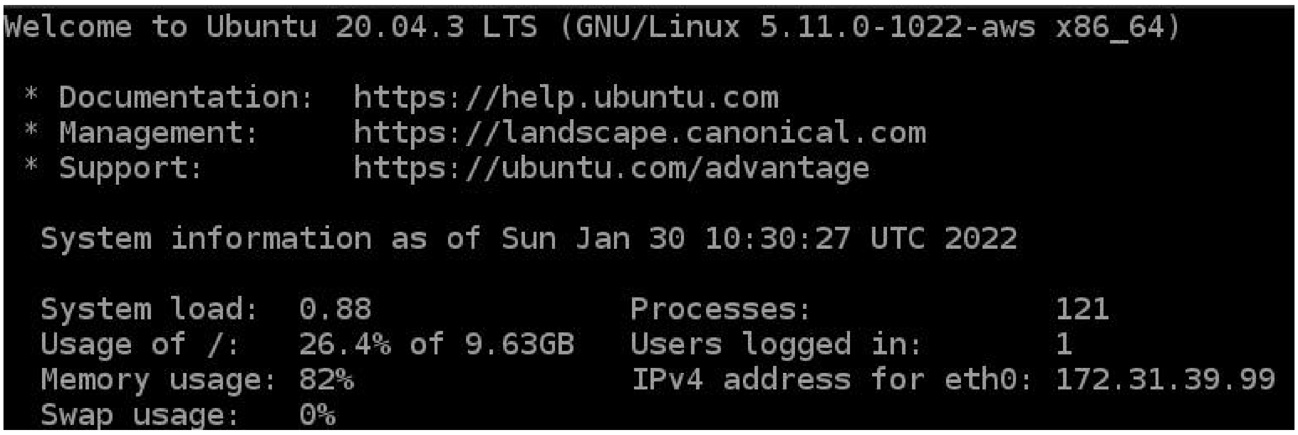
Instance: i-0ae274b19bb5ebf45 (Ubuntu airflow)

Instance summary Info

Instance ID	Public IPv4 address	Private IPv4 addresses
i-0ae274b19bb5ebf45 (Ubuntu airflow)	13.235.2.64 | open address	172.31.39.99
IPv6 address	**Instance state**	**Public IPv4 DNS**
–	Running	ec2-13-235-2-64.ap-south-1.compute.amazonaws.com | open address

Public Url: 13.235.2.64:8080

Ami: ami-0851b76e8b1bce90b



```
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1022-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sun Jan 30 10:30:27 UTC 2022

  System load:  0.88               Processes:             121
  Usage of /:   26.4% of 9.63GB    Users logged in:       1
  Memory usage: 82%                IPv4 address for eth0: 172.31.39.99
  Swap usage:   0%
```

Deployment steps:

1) This amazon ami comes pre equipped with git, python3 & pip.

2) Next we run

    $ sudo -i apt update

    $ python3 -m venv sandbox (this creates a sandbox folder in home directory)

3) After successfull installation of "optional" dependencies, we proceed with final installation.

    $ source ~/sandbox/bin/activate

```
$ pip install "apache-airflow[celery]==2.2.3" --constraint
"https://raw.githubusercontent.com/apache/airflow/constraints-2.2.3/const
 raints-3.8.txt"
```

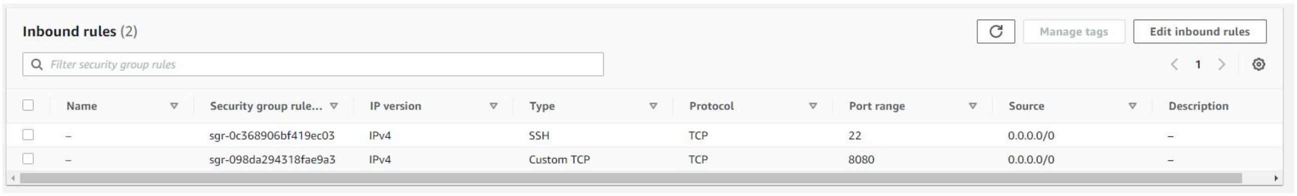
4) This will install the airflow within the virtual sandbox environment. Next we start airflow server, db and scheduler services.

```
$ airflow db init
```

```
$airflow webserver -p 8080 -D --pid
/home/ubuntu/airflow/airflow-webserver.pid --stdout
/home/ubuntu/airflow-webserver.out --stderr
/home/ubuntu/airflow-webserver.err
```

```
$airflow scheduler -D --pid /home/ubuntu/airflow/airflow-scheduler.pid
--stdout /home/ubuntu/airflow-scheduler.out --stderr
/home/ubuntu/airflow-scheduler.err
```

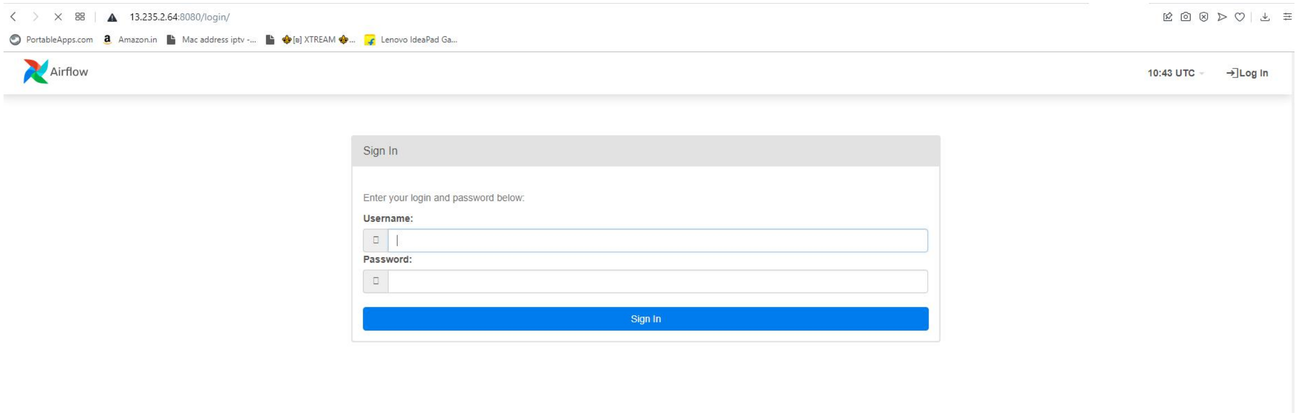
5) Airflow webserver is responsible for the webserver UI which when deployed get enabled and visible on port 8080 of machine. For this we have to allow inbound/outbound rules in order to access our running webserver in security groups of aws cloud attached to the ec2 instance.



Inbound rules (2)

Name	Security group rule...	IP version	Type	Protocol	Port range	Source	Description
–	sgr-0c368906bf419ec03	IPv4	SSH	TCP	22	0.0.0.0/0	–
–	sgr-098da294318fae9a3	IPv4	Custom TCP	TCP	8080	0.0.0.0/0	–

6)Now we visit our public ip on port 8080 to access web ui of airflow.



13.235.2.64:8080/login/

Sign In

Enter your login and password below:

Username:

Password:

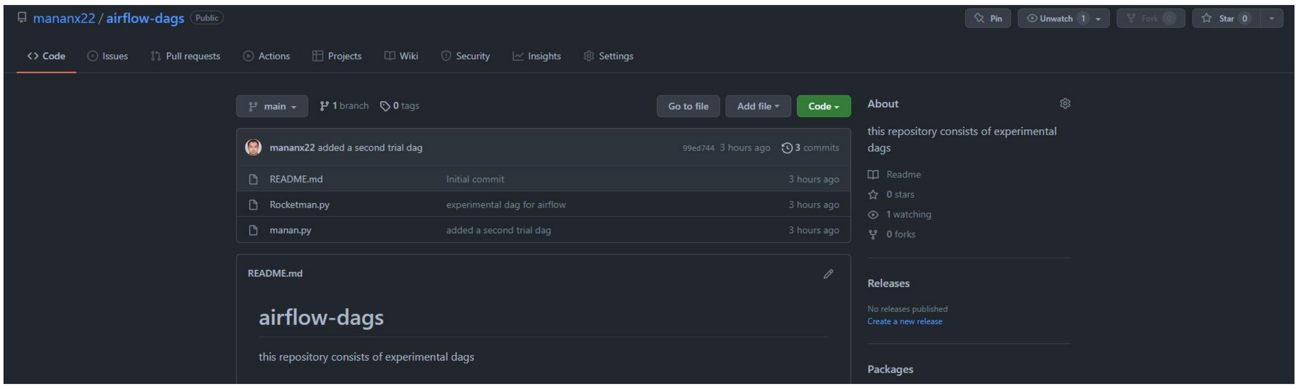
7) since there is no option to create a new user and we have no default user we need to create our own user using airflow cli on terminal

```
$ airflow users create \
 --username admin \
--firstname manan \
--lastname sankhla \
--role Admin \
--email manan@rocketml.com
```

8) This creates a default user and now we can log in to airflow user dashboard.Here we are introduced with multiple example DAGs. We can create custom DAGs as a simple python file and store them in the ~/airflow/dags.

Login info ~ username = admin, password = manan

9) We can link ~/airflow/dags to a github repository that contains DAGs if we want to sync dags to a remote repository as such.



mananx22 / airflow-dags Public

Pin · Unwatch 1 · Fork 0 · Star 0

Code · Issues · Pull requests · Actions · Projects · Wiki · Security · Insights · Settings

main · 1 branch · 0 tags · Go to file · Add file · Code

mananx22 added a second trial dag · 99ed744 3 hours ago · 3 commits

README.md	Initial commit	3 hours ago
Rocketman.py	experimental dag for airflow	3 hours ago
manan.py	added a second trial dag	3 hours ago

README.md

airflow-dags

this repository consists of experimental dags

About

this repository consists of experimental dags

Readme
0 stars
1 watching
0 forks

Releases

No releases published
Create a new release

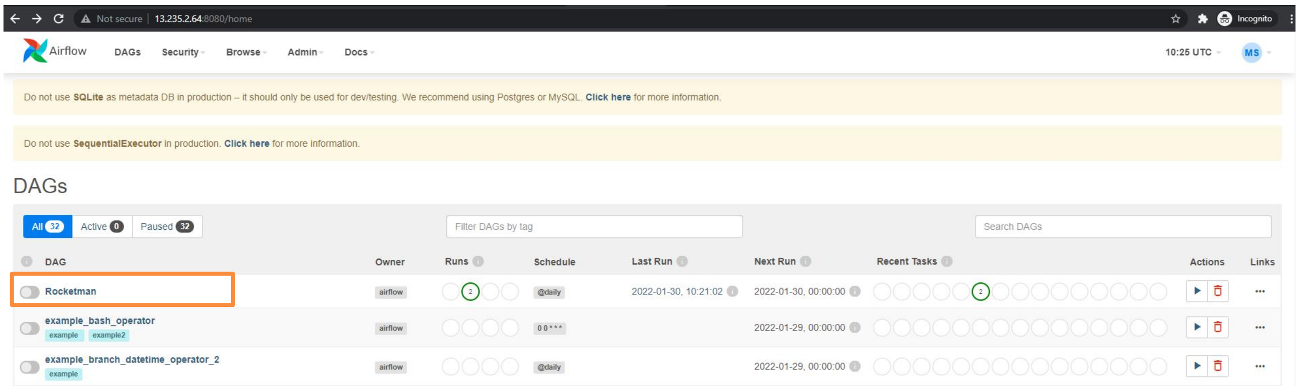
Packages

10) Next in order to sync the DAG repository we use the preinstalled git utility to clone our repository in ~/airflow/dags

```
$ git clone git@github.com:mananx22/airflow-dags.git .
```

11) Next we set up a cronjob if we wish to sync to our repository using

```
$ crontab -e
$ */5 * * * * cd ~/airflow/dags/  && git pull
$ crontab -l
```



Not secure | 13.235.2.64:8080/home

Airflow · DAGs · Security · Browse · Admin · Docs · 10:25 UTC · MS

Do not use **SQLite** as metadata DB in production – it should only be used for dev/testing. We recommend using Postgres or MySQL. **Click here** for more information.

Do not use **SequentialExecutor** in production. **Click here** for more information.

DAGs

All 32 · Active 0 · Paused 32 · Filter DAGs by tag · Search DAGs

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks	Actions	Links
Rocketman	airflow	2	@daily	2022-01-30, 10:21:02	2022-01-30, 00:00:00	2	▶ 🗑	...
example_bash_operator example example2	airflow		0 0 * * *		2022-01-29, 00:00:00		▶ 🗑	...
example_branch_datetime_operator_2 example	airflow		@daily		2022-01-29, 00:00:00		▶ 🗑	...

Here my two custom DAGs rockerman.py and manan.py from git repository are being picked up by scheduler and available in airflow webUI to process and terminate successfully.

Given the time I had and my learning concluded me three ways in which we can expand the deployment on cloud with airflow.

Using Ami - (Least Efficient)

In Aws we can push our own custom ami images. Just like the process I have followed above, we can save our configuration in a custom ami image.

Using terraform we can deploy resources such as EC2 and ELB, where the purpose of elb would be to multiple ec2 resources in case of high load or reduce ec2 resources in case of low nodes.

Tools: Terraform

Cons:

- Generating Ami is time consuming.
- Custom ami usage is Cost incurring.
- Upadtion of airflow from v 2.x to v 3.x would be more time consuming.

Using Ansible - (moderate)

While we can use Terraform to deploy aws resources, terraform can also return the public ip of all ec2 instances it will generate.

This list of all generated EC2 resources will then be passed to a ansible playbook to deploy/install all the necessary tools within the linux using bash operator and scripting.

Tools: Terraform + ansible

Cons:

- Generation and deployment of all scriping and configuration to EC2.
- It brings more errors and troubleshooting, as it introduces more points of failure and vulnerability to your airflow deployment.

Using Kubernetes Helm - (Most Efficient)

Airflow helm chart is officially availble to download and use by airflow community. Although we can deploy a custom airflow docker image but that will take time to deploy each and every componet of airflow.

Kubernetes (EKS on aws & AKS on azure) hold a good position to easily load a airflow helm chart and just deploy the helm chart with the required parameters.

The airflow uses kubernetes executor to execute DAGs as separate pods and will use kubernetes services to contact api within the kubernetes cluster or the cloud service provider.

This also allows us to connect cloud database with airflow to efficient and rapid data storage without need to transport data outside the aws environment.

Tools: AKS/EKS