

## Relatório 18 - Prática: Pipelines de Dados II - Airflow (II)

Lucas Augusto Nunes de Barros

### Descrição das atividades

#### 1. Distributing Apache Airflow

São responsáveis por determinar como as tasks serão executadas, atualmente o airflow possui quatro principais executores o Sequential, Local, Celery e o Kubernetes. Cada um deles com suas características, escolhidos de acordo com a necessidade.

SequentialExecutor é um executor de tarefas sequenciais, em um único processo. Essa característica faz dele um candidato para ambientes de teste e desenvolvimento, pois não há necessidade de paralelismo, como em um ambiente de produção.

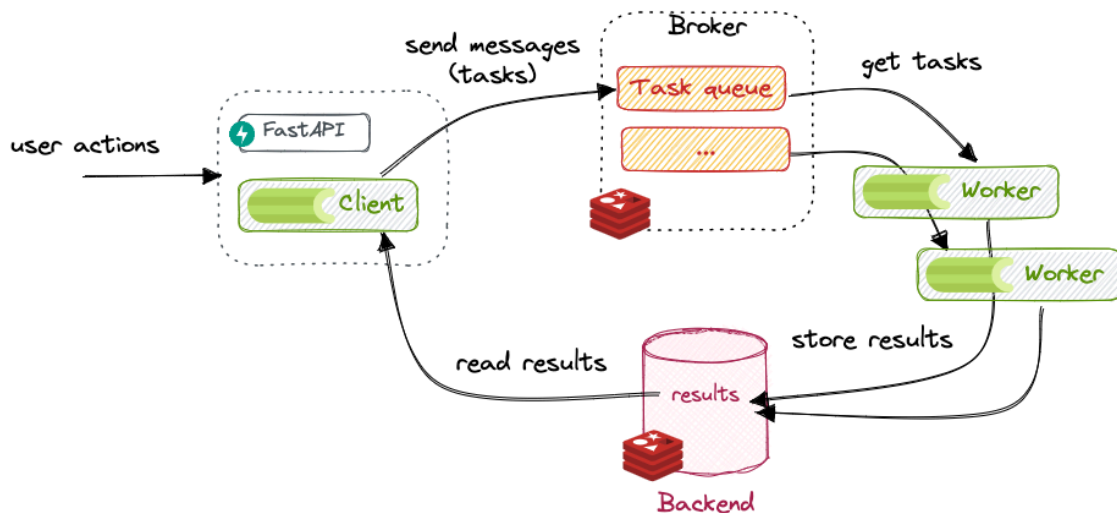
No arquivo airflow.cfg

```
executor=SequentialExecutor
```

```
sqlalchemy_conn=sqlite:///home/airflow/airflow.db
```

Já o LocalExecutor é utilizado em pequenos projetos, onde existe uma necessidade de paralelismo entre processos mas ainda não é necessário distribuir os processos em múltiplas máquinas.

O CeleryExecutor utiliza um software externo, o Celery, que opera gerenciando a fila de tasks assincronamente, ou seja, processos em paralelo são distribuídos para diferentes dispositivos, os workers. Além de suportar paralelismo e escalabilidade horizontal, ele ainda possui dois componentes complementares, que o permitem realizar a distribuição dos processos, o Broker e os Workers. Como já foi explicado, workers são dispositivos que executam tarefas distribuídas pelo Celery, enquanto o broker é um servidor de mensagens que faz a comunicação entre os workers e clientes Celery.



```

- => writing image sha256:9387a8bf19f50d1b012f54f01912c9591408750d38a1197f7ee7b9361b1e0c7 0.0s
- => naming to docker.io/library/airflow-section-5-worker 0.25s
- => [worker] resolving provenance for metadata file 0.0s
[+] Running 8/9
  ✓ flower Built 0.0s
  ✓ scheduler Built 0.0s
  ✓ worker Built 0.0s
  ✓ Container airflow-section-5-postgres-1 Running 0.0s
  ✓ Container airflow-section-5-redis-1 Started 5.8s
  ✓ Container airflow-section-5-flower-1 Started 6.4s
  ✓ Container airflow-section-5-webserver-1 Starting 6.4s
  ✓ Container airflow-section-5-scheduler-1 Created 1.1s
  ✓ Container airflow-section-5-worker-1 Created 1.0s
Error response from daemon: failed to create task for container: failed to create shim task: OCI runtime create failed: runc create failed: unable to
start container process: error during container init: exec: "/entrypoint.sh": permission denied: unknown
(venv) lucas@fdn:~/LAMIA/airflow-materials/airflow-section-5$

```

Erro ao iniciar o webserver airflow, permissão de *entrypoint.sh* negada.

```
COPY --chmod=755 script/entrypoint.sh /entrypoint.sh
COPY config/airflow.cfg ${AIRFLOW_USER_HOME}/airflow.c
```

```
(venv) lucas@fdp:~/LAMIA/airflow-materials/airflow-section-5$ docker compose -f docker-compose-CeleryExecutor.yml up -d
[*] Running 7/7
✓ Network airflow-section-5_default Created
✓ Container airflow-section-5-redis-1 Started
✓ Container airflow-section-5-postgres-1 Started
✓ Container airflow-section-5-flower-1 Started
✓ Container airflow-section-5-webserver-1 Started
✓ Container airflow-section-5-scheduler-1 Started
✓ Container airflow-section-5-worker-1 Started
```

Feitas as alterações e ao tentar iniciar os containers corretamente, houve outro problema ao tentar logar no *Airflow*, mesmo criando novos usuários, com as permissões corretas e reiniciando o banco de dados para apagar possíveis dados errados e corrompidos. Nenhuma dessas soluções funcionou, sendo necessário desativar a autenticação do *Airflow* para então poder prosseguir com o card.

Primeiro foi necessário copiar o arquivos do container para o *host*:

```
docker cp airflow-section-5-webserver-1:/usr/local/airflow/webserver config.py ./temp.py
```

Onde:

*airflow-section-5-webserver-1* : nome do container (origem do arquivo)

`./usr/local/airflow/webserver_config.py` : caminho do arquivo, dentro do container (origem do arquivo)

`./temp.py` : caminho para o arquivo copiado no host (destino do arquivo)

No arquivo `temp.py`, descomentar a linha `AUTH_ROLE_PUBLIC` e modificar seu valor para `'Admin'`

No arquivo `temp.py`:

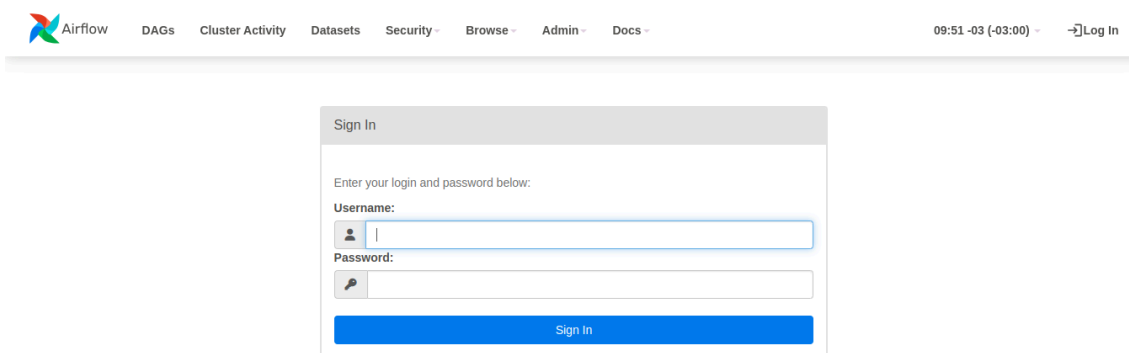
```
AUTH_ROLE_PUBLIC = 'Admin'
```

Feito isso, deve-se copiar novamente o arquivo modificado para dentro do container, executando o comando:

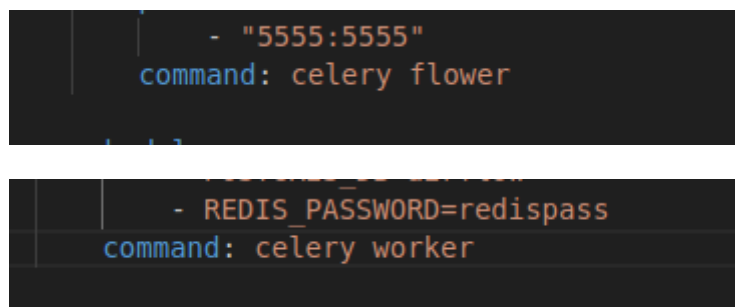
```
docker cp /temp.py airflow-section-5-webserver-1:/usr/local/airflow/webserver_config.py
```

E então reiniciá-lo:

```
docker restart airflow-section-5-webserver-1
```



Devido a atualização de versão, alguns comandos foram alterados, como por exemplo os comandos para iniciar os serviços do flower e do worker.



E finalmente todos os container iniciaram de forma perfeita.

```
lucas@fdp:~/LAMIA/airflow-materials/airflow-section-5$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS
2118c672d40e   airflow-section-5-worker            "/entrypoint.sh cele..." 27 minutes ago Up 27 minutes
274faac141cb   airflow-section-5-scheduler         "/entrypoint.sh sche..." 27 minutes ago Up 26 minutes
d9b08690cbf8   airflow-section-5-webserver         "/entrypoint.sh webs..." 27 minutes ago Up 27 minutes (healthy)
080/tcp        airflow-section-5-webserver-1       "/entrypoint.sh cele..." 27 minutes ago Up 27 minutes
af5f17679e31   airflow-section-5-flower            "/entrypoint.sh cele..." 27 minutes ago Up 27 minutes
793/tcp        airflow-section-5-flower-1          "/entrypoint.sh cele..." 27 minutes ago Up 27 minutes
9f1641564c3d   postgres:9.6                        "docker-entrypoint.s..." 27 minutes ago Up 27 minutes
a1996c490516   airflow-section-5-postgres-1        "docker-entrypoint.s..." 27 minutes ago Up 27 minutes
airflow-section-5-redis-1
```

O CeleryExecutor é um executor do *Airflow* que permite a distribuição e o processamento assíncrono de tarefas. O Celery é um sistema de filas distribuído baseado em mensagens, distribuindo-as entre múltiplos *workers*.

Flower
Dashboard
Tasks
Broker
Monitor
Logout
Docs
Code

Worker:
**celery@b4b3e6f1a39a**
Refresh

Pool
Broker
Queues
Tasks
Limits
Config
System

Worker pool options

Max concurrency	12
Processes	42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53
Max tasks per child	N/A
Put guarded by semaphore	False
Timeouts	0, 0
Writes	{'total': 0, 'avg': '0.00%', 'all': '', 'raw': '', 'strategy': 'fair', 'inqueues': {'total': 12, 'active': 0}}
Worker PID	1
Prefetch Count	12

Pool size control

Pool size
1
Grow
Shrink

Min/Max autoscale

Apply

O CeleryExecutor utiliza um *broker* de mensagens (como Redis ou RabbitMQ) para coordenar a fila de tarefas, garantindo alta disponibilidade e balanceamento de carga. O Flower pode ser integrado para monitorar o desempenho dos *workers* e das tarefas em tempo real, proporcionando controle mais fino sobre a execução dos pipelines.

Ainda na seção 5, é lembrado o orquestrador de containers, Kubernetes, uma plataforma *open source*, originalmente desenvolvida pelo Google, ela automatiza e simplifica operações de infraestrutura. Ele facilita práticas modernas de CI/CD, permitindo *deploy* contínuo de atualizações e estratégias avançadas, bem como oferece recursos para auto-escalonamento horizontal, alta disponibilidade, tolerância a

falhas e resiliência de aplicações. Sua arquitetura distribuída e conjunto abrangente de funcionalidades o tornam uma solução robusta.

O KubernetesExecutor é um executor do Apache *Airflow* que proporciona escalabilidade automática e isolamento entre tarefas. Ao contrário de outros executores, esse cria um ambiente separado para cada tarefa. Essa abordagem é particularmente eficaz em ambientes de nuvem, onde a carga de trabalho pode variar significativamente. É uma solução robusta para *workflows* de dados que exigem alta disponibilidade, isolamento e escalabilidade.

O Vagrant provisiona máquinas virtuais e o Rancher gerencia múltiplos clusters Kubernetes, o foco do Kubernetes, utilizado com Rancher, é a orquestração de containers

## 2. Improving your DAGs with advanced concepts

Uma forma eficiente de minimizar padrões repetitivos é agrupando tarefas recorrentes em um único componente reutilizável, geralmente chamado de subDAG. Funcionam como DAG's encapsuladas e que podem ser chamadas em múltiplas outras DAG's. Embora úteis para organizar lógicas repetitivas e evitar repetição, é importante ter cautela, pois podem influenciar a performance e a legibilidade do pipeline. Alternativas como TaskGroups (a partir do *Airflow* 2.0) oferecem uma abordagem mais leve para agrupamento de tarefas.

Durante a seção 6 alguns erros de importação foram encontrados, sendo necessário alterar alguns códigos, devido a alteração nas versões dos *softwares*.

```
subdag_4 = SubDagOperator(  
    task_id='subdag-4',  
    subdag=factory_subdag(DAG_NAME, 'subdag-4', default_args),  
    executor=CeleryExecutor()  
)
```

Na declaração da SubDagOperator não deve ser definido um executor, pois a SubDag herda o executor da Dag-Pai.

Todos os operadores têm um argumento *trigger\_rule* que define a regra pela qual a tarefa gerada é acionada. O valor predefinido para *trigger\_rule* é *all\_success* e pode ser definido como “acionar esta tarefa quando todas as tarefas diretamente a montante tiverem sido bem sucedidas”.

- *all\_success*: (default) todos processos anteriores bem sucedidos.
- *all\_failed*: todos processos anteriores falham.
- *all\_done*: todos processos anteriores foram finalizados.
- *one\_failed*: dispara assim que o primeiro processo falha, sem aguardar a execução dos outros processos.
- *one\_success*: dispara assim que o primeiro processo é bem sucedido, sem aguardar a execução dos outros processos.
- *none\_failed*: nenhum dos processos anteriores falha.
- *none\_skipped*: nenhum dos processos anteriores foi ignorado.

O *Airflow* utiliza *templates* Jinja em conjunto com *placeholders* para permitir a renderização dinâmica de valores durante a execução das tarefas. Essa funcionalidade é especialmente útil para acessar variáveis de contexto, parâmetros de execução ou informações de tarefas anteriores. Os operadores do *Airflow* suportam nativamente *templates* em seus parâmetros, essa integração permite a construção simplificada de fluxos de trabalho.

XComs (*Cross-Communication*) no *Airflow* funcionam como mecanismos de compartilhamento de dados entre tarefas dentro de um mesmo DAG. Funcionam como um sistema de mensagens que permite que tarefas troquem algumas informações. Funcionam com base em pares *key-value* armazenados temporariamente no banco de dados. Eles permitem que tarefas troquem informações como IDs, status ou parâmetros usando os métodos *xcom\_push* (envio de dados) e *xcom\_pull* (recuperação de dados).

São particularmente úteis na coordenação de fluxos complexos ou tomada de decisões. No entanto, têm algumas limitações:

- Não são adequados para grandes volumes de dados, apenas informações leves.
- O uso excessivo pode impactar o desempenho devido ao acesso recorrente ao banco de dados.

São ideais para comunicação pontual, mas exigem soluções externas caso seja necessário comunicar dados mais pesados.

O *TriggerDagRunOperator* é um operador que permite o acionamento de um DAG a partir de outro DAG, aumentando a gama de possibilidades para configurar um fluxo de trabalho, se mostra uma ferramenta muito útil em combinação com outros operadores estudados e em cenários que demandam execução condicional.

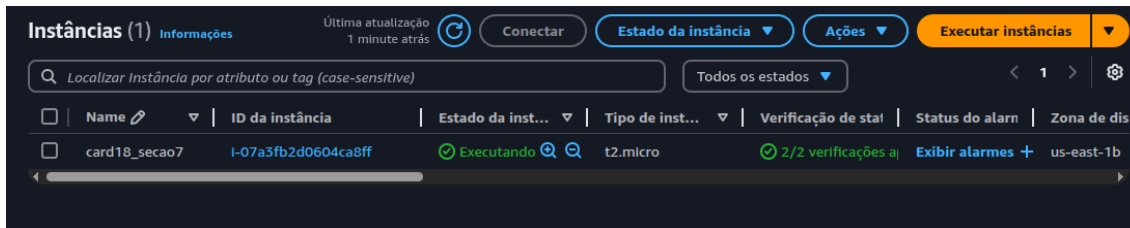
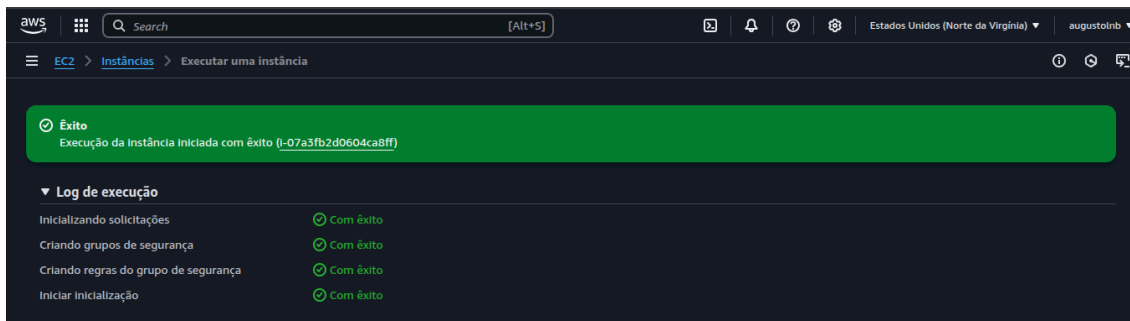
O *ExternalTaskSensor* é um sensor que monitora a conclusão de uma determinada tarefa de outro DAG, aguardando para iniciar a execução apenas após a *task* monitorada ser finalizada. É útil na sincronização de fluxos de trabalho independentes.

### 3. Deploying Airflow on AWS EKS with Kubernetes Executors and Rancher

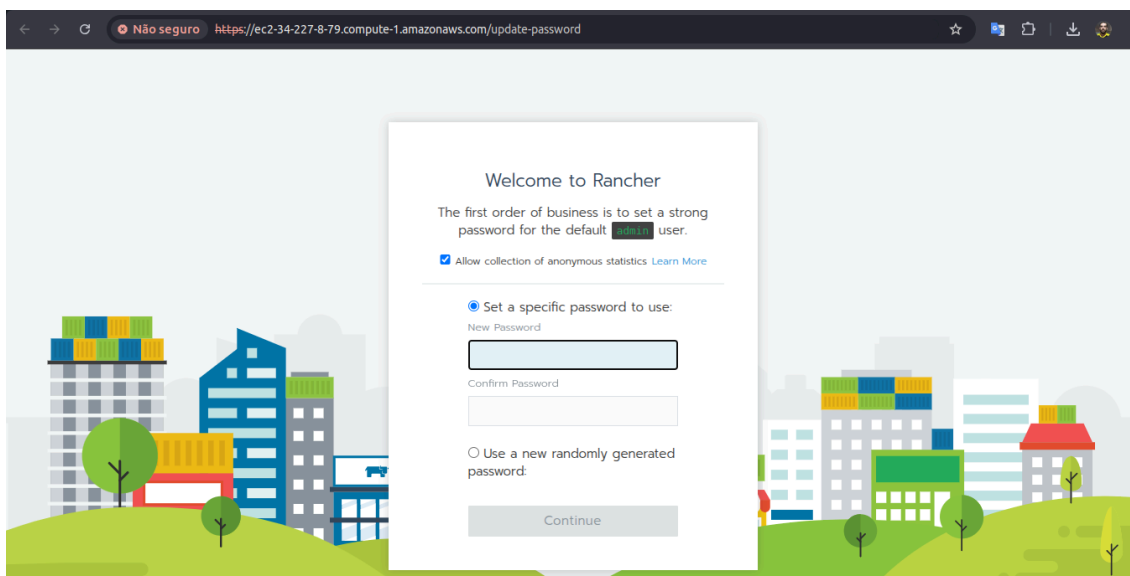
O *Amazon Elastic Kubernetes Service* - EKS, é um serviço disponibilizado pela AWS para execução e gerência do Kubernetes na nuvem, sem a necessidade de manter manualmente um cluster. Ele permite implementar, gerenciar e escalar aplicações em contêineres usando a infraestrutura da AWS, bem como permite a integração com outros serviços do ecossistema AWS.

Durante esta etapa serão utilizados serviços da AWS, sendo assim necessário criar uma conta para ter acesso ao ecossistema.

As instâncias EC2 - *Elastic Compute Cloud*, fornecidas pela AWS são servidores virtuais implementados em nuvem. Oferecem diferentes tipos de instâncias para diferentes cargas de trabalho, além de já serem integradas a outros serviços da AWS.



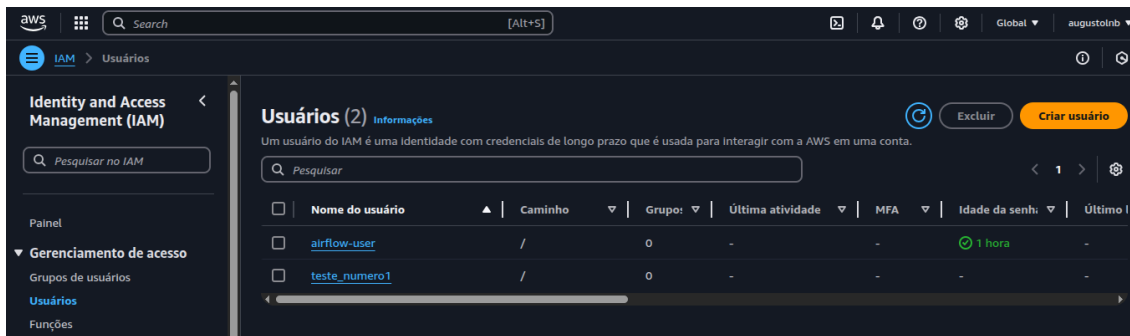
Primeira instância AWS criada com sucesso.



O Rancher é executado diretamente da instância EC2.

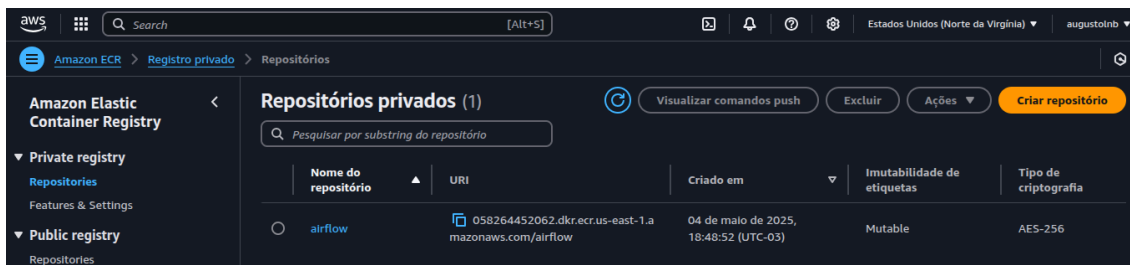
O Rancher é uma plataforma de gerenciamento para o Kubernetes, projetada para simplificar a manutenção de clusters em ambientes de produção.

Usuários IAM (*Identity and Access Management*) são entidades da AWS que permitem o gerenciamento de permissões para serviços e recursos. Suporta autenticação multifator e permite a criação de funções para delegação temporária de acesso entre serviços AWS.



Usuários IAM criados.

O Amazon ECR - *Elastic Container Registry*, é um serviço gerenciado de repositório privado, similar ao popular GitHub, usado para armazenar e distribuir imagens de containers. Integra-se nativamente com outros serviços do ecossistema com intuito de simplificar os processos envolvidos. O ECR utiliza usuários IAM para controle de acesso.

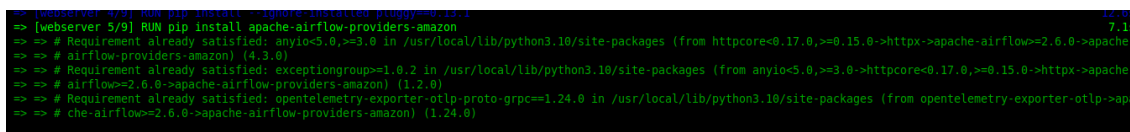


Repositório criado no ECR.

## 4. Monitoring Apache Airflow

Os logs do Airflow são o histórico de suas atividades. Eles auxiliam no monitoramento e na execução das suas tarefas, identificando problemas de forma mais ágil e permitindo criar uma linha do tempo para compreender o funcionamento do fluxo de trabalho ao longo de determinado período. É uma ferramenta essencial para manter pipelines de dados operando bem no longo prazo.

O *Amazon Simple Storage Service* - Amazon S3, é um serviço de armazenamento altamente escalável, disponível em nuvem e com alta performance, projetado para o armazenamento de dados através da web. Sua arquitetura fundamenta-se em *buckets*, que atuam como containers para dados armazenados.



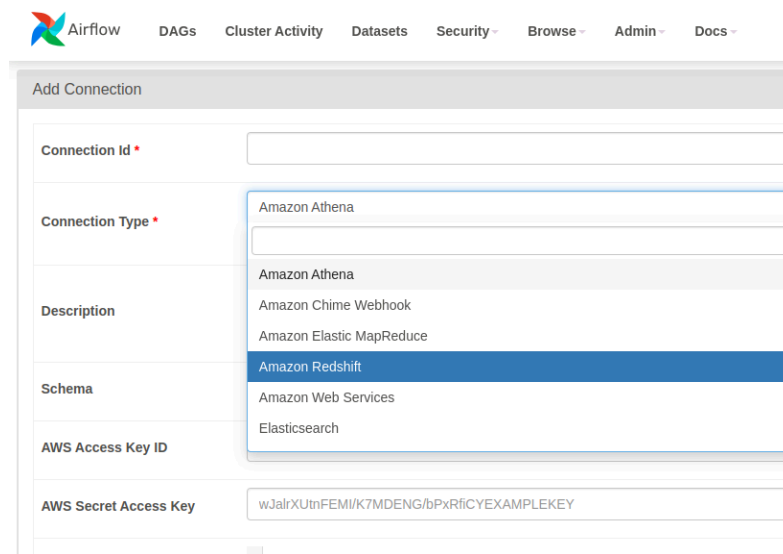
Instalação do pacote *apache-airflow-providers-amazon*



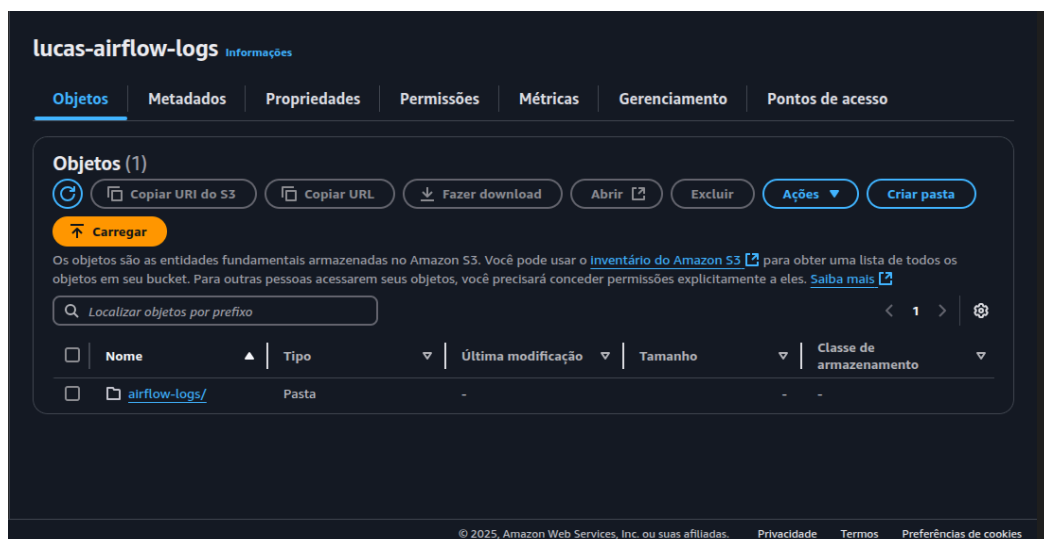
Após a criação do *bucket* e do novo usuário IAM para manipular os arquivos de log, ao tentar criar a conexão entre *Airflow* e o serviço AWS S3 foi notada a falta do conector necessário, para prosseguir com o card foi necessário atualizar alguns comandos e adicionar alguns pacotes ao *Dockerfile*, para então recriar uma imagem atualizada com todos os pacotes necessários.

```
RUN pip install apache-airflow-providers-amazon
```

Após a atualização os conectores já estavam disponíveis.



Com a conexão estabelecida foi possível ao *Airflow* criar sua pasta de logs utilizando da conexão com o serviço AWS S3.



**lucas-airflow-logs** Informações

Objetos | Metadados | Propriedades | Permissões | Métricas | Gerenciamento | Pontos de acesso

**Objetos (1)**

Carregar

Os objetos são as entidades fundamentais armazenadas no Amazon S3. Você pode usar o [Inventário do Amazon S3](#) para obter uma lista de todos os objetos em seu bucket. Para outras pessoas acessarem seus objetos, você precisará conceder permissões explicitamente a eles. [Saiba mais](#)

Localizar objetos por prefixo

	Nome	Tipo	Última modificação	Tamanho	Classe de armazenamento
<input type="checkbox"/>	airflow-logs/	Pasta	-	-	-

© 2025, Amazon Web Services, Inc. ou suas afiliadas. Privacidade | Termos | Preferências de cookies

Prosseguindo é possível atualizar o serviço AWS S3 com os logs do *Airflow*, agora disponíveis via web. Para completar o escopo da aplicação, ainda são usados o *Elasticsearch* e o *Kibana*.

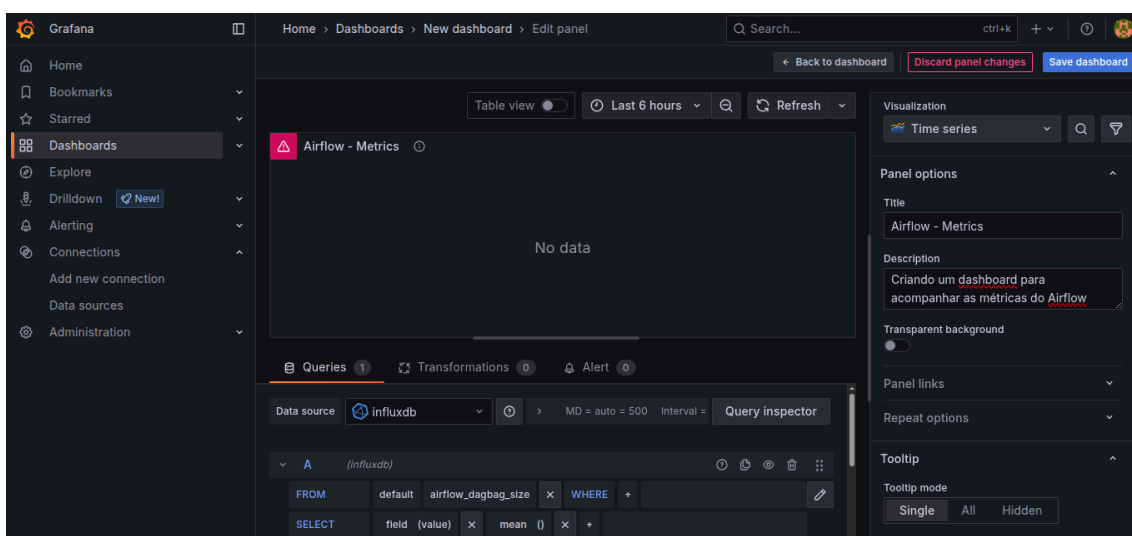
O *Elasticsearch* é um motor de busca distribuído, permite indexar e pesquisar grandes volumes de dados em tempo real. A capacidade de realizar buscas complexas, agregações e análises espaciais em dados não estruturados e semiestruturados o diferencia como uma solução robusta para explorar informações de maneira eficiente.

Complementando o *Elasticsearch*, o *Kibana* oferece uma interface de visualização e exploração de dados. Com ele os usuários podem criar *dashboards*, gráficos, tabelas e outros tipos de visualizações a partir dos dados indexados no *Elasticsearch*.

O InfluxDB também é utilizado durante o desenvolvimento do card, ele consiste em um sistema de gerenciamento de banco de dados para séries temporais de código aberto, projetado especificamente para lidar com dados marcados por tempo. Sua arquitetura otimizada permite a entrada e o armazenamento eficientes de grandes volumes de dados com alta granularidade temporal, tornando-o ideal para aplicações como monitoramento de infraestrutura, telemetria e Internet das Coisas.

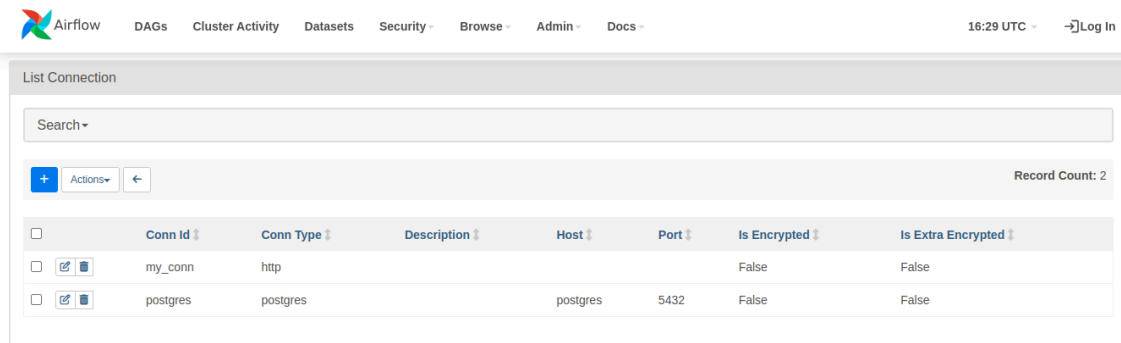
O Telegraf, por sua vez, atua como um agente de coleta de dados, *open source*. Sua principal função é coletar dados de diversas fontes (sistemas operacionais, aplicações, serviços de nuvem, sensores IoT, etc.) e então enviá-los para um ou mais destinos, sendo o InfluxDB uma das opções mais comuns.

O Grafana, por sua vez, se destaca como uma plataforma de visualização de dados, também de código aberto. Embora possa ser integrado ao *Elasticsearch* para visualizar dados indexados, o Grafana possui capacidade de agregar informações vindas de diversas fontes e ferramentas de análise. Seus dashboards e recursos de alerta o tornam essencial para monitorar a saúde e o desempenho de sistemas complexos.



## 5. Security in Apache Airflow

Por fim, o card traz à tona o Fernet, que no contexto do *Airflow*, opera como uma camada de criptografia, para proteger informações como senhas, variáveis e outros dados confidenciais. Seu funcionamento é baseado no algoritmo de criptografia simétrica AES - *Advanced Encryption Standard*, utilizando SHA256, um algoritmo hash criptográfico que produz um valor de hash com 256 bits. Essa combinação garante tanto a confidencialidade quanto a integridade dos dados criptografados.



Conn Id	Conn Type	Description	Host	Port	Is Encrypted	Is Extra Encrypted
my_conn	http				False	False
postgres	postgres		postgres	5432	False	False

É explicado que as duas colunas mais à direita das conexões indica o nível de criptografia utilizada, sendo a coluna *Is Encrypted* verdadeira quando apenas as senhas são criptografadas, já a coluna *Is Extra Encrypted* será verdadeira quando todo o JSON comunicado através da conexão for criptografado.

```
login | password | extra
-----|-----|-----
my_login | pitoco32 | {"access_key": "my_key", "secret_key": "my_secret"}
(1 row)

airflow# SELECT login, password, extra FROM connection WHERE conn_id='my_conn';
login | password | extra
-----|-----|-----
my_login | 2a233391e9694395ca68d6568af792223003eb6bab4fa93ebce36466b7b64f47 | {"access_key": "my_key", "secret_key": "my_secret"}
(1 row)

airflow#
```

Informações da conexão HTTP do *Airflow*, inicialmente a senha era legível, antes da aplicação do Fernet, após a criptografia a senha da conexão no banco de dados está protegida.

Além do Fernet, são apresentadas algumas práticas de segurança como a alternância entre as chaves de criptografia do próprio Fernet, ocultar variáveis do sistema e a filtragem de Dag's por dono.

Uma outra prática de segurança abordada é a apresentação do *RBAC UI Security - Role-Based Access Control User Interface Security*, que é um sistema de controle de acesso baseado em funções aplicado à interface web do *Airflow*. Ele permite que administradores definam permissões e atribuam essas permissões a funções (*roles*). Em seguida, os usuários são associados a uma ou mais funções, herdando assim as permissões concedidas a essas funções.

## Conclusão

O Apache Airflow até agora foi uma das aplicações que mais me deu problemas mas apesar disso gostei muito de aprender a lidar com essa ferramenta, todos os erros que obtive durante o desenvolvimento do card se mostraram úteis no aprendizado de algo novo, além de ser muito satisfatório aprender a contornar os erros e entender qual era o problema, eles me forneceram uma visão mais abrangente do ambiente de desenvolvimento em que são necessários diversos containers e aplicações executando simultaneamente e trocando informações entre si.

Esse card em específico me foi muito querido pois além de apresentar conceitos mais avançados do *Airflow* abordou também outras aplicações que há muito tempo eu conhecia mas não havia tido a oportunidade de manejar, muito menos a oportunidade de ser ensinado a como manejar, exemplo disso foram os serviços da AWS, bem como o Kubernetes e o Grafana, dos quais já tinha ouvido falar bastante durante meus estudo de IOT.

## Referências

[1] YOUTUBE.COM. How to create S3 connection for AWS and MinIO in latest airflow version | Airflow Tutorial Tips 3 Disponível em: <<https://www.youtube.com/watch?v=sVNvAtlZWdQ>>

, Acesso em 05 de maio de 2025

[2] PYPI.ORG. apache-airflow-providers-amazon 9.6.1 Disponível em:  
<<https://pypi.org/project/apache-airflow-providers-amazon/>> , Acesso em 05 de maio de 2025

[3] HEVODATA.COM. Apache Airflow S3 Connection. Disponível em:  
<<https://hevodata.com/learn/airflow-s3-connection/#s3>> Acesso em 02 de maio de 2025

[4] [WWW.BOTREETECHNOLOGIES.COM](http://WWW.BOTREETECHNOLOGIES.COM) Implementing Celery using Django for Background Task Processing . Disponível em:  
<<https://www.botreetechnologies.com/blog/implementing-celery-using-django-for-background-task-processing/>> , Acesso em 06 de abril de 2025

[5] [WWW.GITHUB.COM](http://WWW.GITHUB.COM) No SecretsMasker found! #17681. Disponível em:  
<<https://github.com/apache/airflow/issues/17681>> , Acesso em 27 de abril de 2025

[6] [WWW.GITHUB.COM](http://WWW.GITHUB.COM) airflow\_local\_settings.py . Disponível em:  
<[https://github.com/AllenInstitute/incubator-airflow/blob/master/airflow/config\\_templates/airflow\\_local\\_settings.py](https://github.com/AllenInstitute/incubator-airflow/blob/master/airflow/config_templates/airflow_local_settings.py)> , Acesso em 23 de abril de 2025