TP 4

⚠️ Ceci est un TD noté. Voici quelques règles :

- Les fichiers créés pendant le TD seront à rendre avant la fin sous forme d'archive (`.zip, .tar, .rar, ...`) sur teams ou par email à qnambot@gmail.com
- C'est un TD classique : vous avez le droit de communiquer, travailler ensemble, utiliser internet, demander de l'aide au professeur, etc.
- Il est cependant **interdit** de *copier-coller* du code (de quelqu'un d'autre, d'une GenAI type ChatGPT/Copilot). Vous pouvez vous partager du code, mais faites l'effort de le comprendre, et de le ré-écrire à votre façon

Pour rappel : chaque occurence de `{UserName}` doit être remplacer par son user name AWS (visible en haut à droite de l'interface AWS)

Setup Environnement

Ce TP s'effectue sur Airflow, vous allez avoir besoin d'avoir un Airflow local, voici plusieurs options pour installer Airflow :

- **Sur UNIX** (mac, linux) : Installation avec pip

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

  Puis lancer `airflow standalone,` si tout se passe bien airflow est accessible sur http://localhost:8080 (documentation complète Airflow).

  Les identifiants ont été générés automatiquement, un message dans le terminal doit afficher leur emplacement (normalement `$HOME/airflow/simple_auth_manager_passwords.json.generated`)

  Airflow va créer un dossier `$HOME/airflow` . Il faut créer un sous-dossier `dags/` ici.

  Astuce : Modifier le fichier `$HOME/airflow/airflow.cfg` pour passer la variable `load_examples = True` à False, puis lancer `airflow db reset` : cela va supprimer les 80 dags d'exemple.

- **Sur Windows avec WSL** (Windows Subsystem for Linux, pour l'installation de WSL voir la documentation microsoft) : Même instruction que ci-dessus (étant donné que que cela revient à avoir Linux)

- **Avec Docker** : Il faut avoir Docker et docker compose d'installé. Puis télécharger ce fichier dans un dossier, ouvrir un terminal dans ce même dossier. (Sur linux uniquement : Lancer `mkdir -p ./dags ./logs ./plugins ./config; echo -e "AIRFLOW_UID=$(id -u)" > .env;`). Sur tous les OS : Lancer `docker compose up airflow-init` pour initilaiser airflow, puis `docker compose up` pour le lancer. Une fois prêt, Airflow devrait être accessible via l'adresse http://localhost:8080, identifiant: admin admin. (documentation complète)

- **Si aucune de ces solutions ne marchent** (typiquement si vous avez un windows sans être admin dessus, et que vous n'avez pas WSL d'installé), il est possible d'utiliser un Airflow managé par Astronomer. Suivre les instructions ci-dessous pour avoir un environnement avec Astronomer :

  1. Aller sur le site d'Astronomer et s'inscrire à la période d'essai de 14 jours
  2. Une fois connecté, dans la section *Deployment*, ajouter un nouveau *deployement* pour créer une instance d'Airflow avec :
     - Un nom au choix
     - Cluster sur AWS

    - Region eu-central-1
    - Dans la section Execution, modifier le *Max Workers* pour mettre 2 La création peut prendre quelques minutes, pendant ce temps, avancer sur la suite :
3. Dans la section Workspace Settins aller dans Access Management et ajouter un user : `qnambot@myges.fr`
4. Installer la CLI d'Astro en suivant les instructions de la documentation
5. Lancer la commande suivante pour configurer la CLI

```
astro login
```

6. Créer un dossier vide pour le TD, se placer dedans et lancer la commande

```
astro dev init
```

Cela devrait créer de nombreux dossiers et fichiers, dont un dossier `dags/`.

7. Il y a deux fichiers d'exemple dans le fichiers `dags/` : il faut les supprimer (ou les déplacer dans dehors du projet pour les garder en exemple)

À la fin du TD, **il faudra rendre un archive avec le dossier `dags/`**

⚠️ Bien penser à supprimer toutes les ressources à la fin du TD.

Exercice 1

Dans cet exercice nous allons utiliser les API publiques 7timer et Open-Meteo pour récupérer des informations meteo périodiquement, les fusionner, et les stocker sur S3.

La première API nous permet d'avoir des informations sur les conditions du ciel (nuageux, etc.), et la deuxième nous donnes informations de température et humidité

1. Commençons par créer et déployer un DAG assez basique.

Pour créer un DAG, il faut créer un fichier python (peu importe son nom) dans le dossier `dags/`. On peut utiliser un code comme ceci :

```python
from datetime import datetime

from airflow.sdk import DAG, task

with DAG(
    dag_id="...",
    start_date=datetime(...),
    schedule=None,
    catchup=False,
) as dag:
    pass
```

- `dag_id` : Le nom du dag. **Donner comme nom `weather`**
- `start_date` : Une date à minimum à partir de laquelle le DAG doit s'exécuter. Si elle est dans le futur, il ne commencera pas avant. On peut mettre n'importe quel jour du passé ici

- `schedule` : La périodicité du DAG. Peut être un CRON (ex : `"0 8 * * * "` pour *tous les jours à 8h*) ou un objet python timedelta. **Définir une périodicité de 15 minutes**
- `catchup` : (Par défaut `True`). Si *true*, au moment de son activation Airflow va rattraper tous les runs ratés entre la start_date et maintenant

On peut maintenant **ajouter une première tâche** EmptyOperator. Cette tâche ne fait rien, elle prend simplement comme argument `task_id` (comme tous les opérateurs). Elle va nous permettre d'avoir une première tâche. Lui donner comme nom `start`

> *Pour utiliser le `EmptyOperator` , il faut l'importer. Pour trouver l'import, il suffit de regarder la documentation de l'opérateur : le titre de la page est le nom du package qui contient l'operateur. Donc : `from {page_title} import EmptyOperator`*

2. Si le DAG a bien été créé dans le dossier `dags/` , il devrait être visible dans la web interface (dans la section dags). Si besoin de rafraichir manuellement, il suffit de lancer la commande `airflow dags list` si Airflow est en local, `docker exec airflow-docker-airflow-scheduler-1 airflow dags list` si Airflow tourne avec docker compose, ou `astro deploy --dag` pour Astronomer. Le DAG peut mettre une à deux minutes pour apparaître sur Airflow. Si le scheduler n'arrive pas à le lire, une popup rouge apparaît en haut d'Airflow avec plus d'informations. Il est également possible d'avoir plus de logs sur l'interface d'astro, dans l'onglet "Logs" du deployment.

Ajouter le DAG, patienter quelques minutes, et vérifier qu'il apparaît.

3. **Aller sur l'interface du DAG**. Sur la gauche on doit voir apparaître le nom de l'operateur. Avec un des logos en haut à gauche, on peut afficher une visino sous forme de graphe du DAG. Pour la lancer manuellement il suffit de cliquer sur le bouton *Trigger* en haut à droite. **Lancer le DAG et vérifier que la tâche passe bien au vert**.

4. Le but du DAG que nous allons créer ici est de récupérer périodiquement des informations météo de plusieurs APIs et de les regrouper. Commençons par Open-Meteo.

L'API d'OpenMeteo a une URL qui ressemble à ça :

```
https://api.open-meteo.com/v1/forecast?
latitude=***&longitude=***&current=temperature_2m,wind_speed_10m
```

Avec en argument :

- latitude/longitude : Les coordonnées du lieu où l'on souhaite récupérer les infors météo
- current : une liste d'éléments à récupérer et qui correspondent à la météo des 15 dernières minutes

Voici un exemple de script pour récupérer les informations meteo de Lyon.

```
import requests

# Lyon coordinates
lon, lat = 4.85, 45.7

# Call openmeteo API
results = requests.get(
    f"https://api.open-meteo.com/v1/forecast?latitude={lat}&longitude=
{lon}&current=temperature_2m,wind_speed_10m"
)
```

```
# Raise an error if API return status != 2xx
results.raise_for_status()

print(results.json())  # Affiche le contenue de la réponse sous forme de texte brut
```

Lancer le script localement et vérifier regarder le résultat.

5. **Créer une première tâche avec le décorateur `@task` qui** :

- Fait un appel à l'API d'OpenMeteo
- Affiche le résultat avec un print
- S'exécute *après* la tâche *start*
- *raise* une exception si le status_code ne vaut pas 200

Déployer le DAG, et vérifier ce qui a été *print* en allant voir les logs de la tâche. (Astuce : dans l'interface du DAG il y a une sectino *Code* qui permet de voir le code source actuelle du DAG et ainsi savoir si Airflow l'a actualisé ou pas encore).

> *Si la tâche est en erreur, il faut re-déployer le DAG. Pour vérifier que le code a bien été mis à jour il y a un onglet "Code" qui permet de voir le code source du DAG.*

> *Une tâche en erreur peut être relancée en cliquant dessus puis Clear qui va relancer la tâche et toutes ses dépendances en aval*

6. Pour qu'Airflow puisse intéragir avec notre compte AWS, il faut créer un *user* IAM qu'Airflow pourra utiliser pour avoir des droits (comme pour Preset lors du premier TD). Dans le service IAM, **créer un user** :

- Nom : `EsgiIabdM2Airflow{UserName}`
- Permissions : Lui associer la permission `EsgiIabdM2Airflow{UserName}` via la section `Attach policies directly`.
- Tag : `school = esgi, promotion = iabd-m2, user = {UserName}`

⚠️ En tant qu'élève vous n'avez que très peu de droits sur IAM. Il y a donc de nombreux messages d'erreurs, mais vous avez suffisament de droit pour créer l'utilisateur.

Ensuite, aller sur la page de ce user :

```
https://us-east-1.console.aws.amazon.com/iamv2/home?region=eu-west-
3#/users/details/EsgiIabdM2Airflow{UserName}
```

Puis dans *Security Credentials*, et dans la section *Access keys*, **créer une paire access/secret key** pour l'utilisateur (attention à bien noter la valeur). Airflow va pouvoir utiliser cette pair de clef pour pouvoir se connecter sur AWS en tant que ce user.

7. Il ne reste plus qu'à déclarer une connexion qui pourra ensuite être utiliser par un operator pour intéragir avec AWS au nom de ce user.

Sur l'interface d'Airflow, aller dans la section *Admin* puis *Connecions*, et **créer une connexion** avec :

- Nom : `aws_default`
- Type : Amazon Web service
- Définir l'access key et secret key avec ceux du rôle précédemment créé
- En extra : `{"region_name": "eu-west-3"}`

On peut valider que la connexion marche grâce au bouton *Test*. Une fois fonctionnel, **appeler le professeur pour valider la question**.

8. On peut désormais améliorer notre première task pour qu'elle créer un fichier S3 :

- Sous le préfixe `s3://esgi-lyon-iabd-m2-cloud/{UserName}/weather/bronze/openmeteo/`
- Le nom du fichier doit contenir le timestamp d'exécution du DAG grâce à un argument templaté
- La fonction doit renvoyer le chemin de l'objet créé

💡 Pour accéder à un argument templaté dans une tâche, il faut utiliser les kwargs. Exemple :

```
@task
def my_task(**kwargs):
    print(kwargs["some_templated_argument"])
```

💡 Pour accéder à un argument templaté dans un opérator, il faut utiliser les doubles accolades :

```
run_lambda = LambdaInvokeOperator(some_arg="{{ some_templated_arg }}")
```

Airfow fourni des objets appelé *Hook* qui permettent d'encapsuler les clients vers des providers extérieurs et de gérer la connexion. Il existe par exemple un S3Hook, qui gère la connexion AWS et le client boto3.

Le S3Hook possède une méthode load_string permettant de créer un objet sur S3 à partir d'une string.

```
hook = S3Hook(aws_conn_id="my_aws_connection") # Note that aws_conn_id default value is
"aws_default"
hook.conn # A boto3 client instance
hook.load_string(..) # call the load string method
```

Utiliser le S3Hook et la méthode load_string pour mettre à jour la tâche afin qu'elle créer l'objet voulu sur S3

Vérifier ensuite que la tâche tourne bien, et que le fichier est présent sur S3.

9. L'API de 7timer fonctionne de façon similaire, avec l'URL :

```
http://www.7timer.info/bin/api.pl?lon=***&lat=***&product=civil&output=json&unit=Metric
```

Créer une deuxième tâche qui

- Tourne en parallèle de la première
- Récupère les résultats de 7timer
- Stock le résutlat dans un objet S3 sous `s3://esgi-lyon-iabd-m2-cloud/{UserName}/weather/bronze/7timer/`
- Le nom du fichier doit contenir le timestamp d'exécution du DAG grâce à un argument templaté
- La fonction doit renvoyer le chemin de l'objet créé

Vérifier ensuite que la tâche tourne bien, et que le fichier est présent sur S3.

10. Mainteant que nous récupérons périodiquement des données brutes, on peut les fusionner dès lors que les deux tâches sont terminées. Pour des raisons de maintenance, on souhaite faire cela via une Lambda (qui pourrait donc être exécuter dans un contexte autre qu'Airflow par exemple).

Créer une Lambda python `EsgiIabdM2{UserName}AirflowWeather` avec

- Le rôle `EsgiIabdM2Lambda{UserName}`

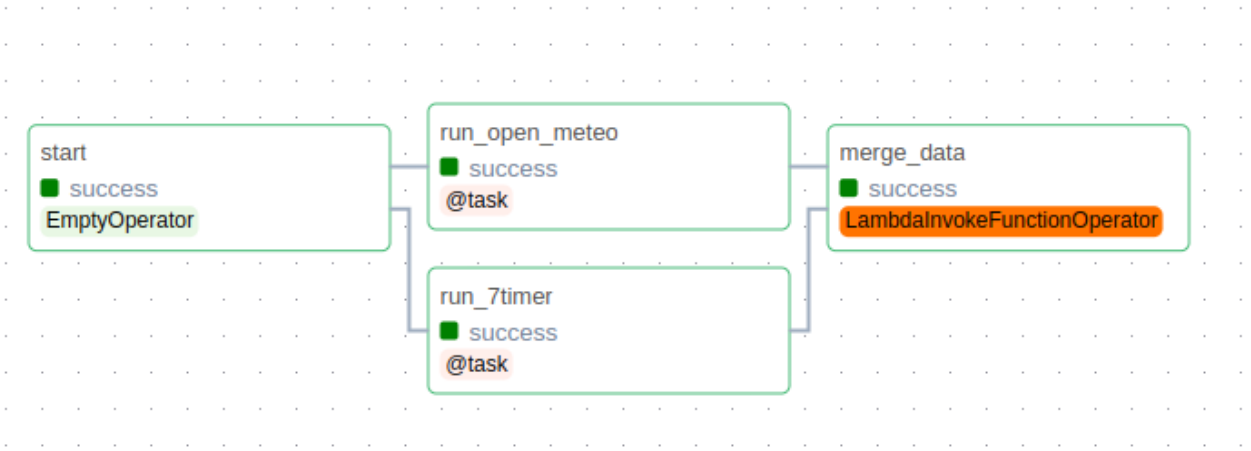
- Les tags habituels (`school = esgi, promotion = iabd-m2, user = {UserName}`)

Pour l'instant, laissons le code par défaut de la lambda

Pour exécuter une Lambda depuis Airflow, on peut utiliser l'operateur LambdaInvokeFunctionOperator

Ajouter un opérateur pour appeler la lambda une fois que les deux autres tâches sont terminées

Déployer et vérifier que le graphe ressemble bien à ça :



11. La lambda va avoir besoin du chemin S3 des deux fichiers brutes météo. Lorsqu'une fonction créée avec `@task` renvoie un résultat, il peut être utilisée ensuite dans le DAG :

```python
@task
def some_function():
    # ...
    return 42

with DAG(...) as dag:
    some_function_output = some_function()

    print(some_function_output) # 42
```

Ajouter un payload à l'operator Lambda pour passer en argument les deux URIs S3

12. **Implémenter la Lambda pour qu'elle**

- Lise les deux fichiers
- À partir des données openmeteo on garde la température et la vitesse du vent
- À partir des données 7timer on garde des informations uniquement de la premire data serie : la direction du vent et l'argument *weather*
- Upload le résultat sous forme de fichier json sous `s3://esgi-lyon-iabd-m2-cloud/{UserName}/weather/silver/`
- Le nom du fichier doit être similaire au deux autres

Exemple de code pour la lambda :

```python
import boto3
import json


def python_handler(event, context):
    # Get input arguments
    openmeteo_key = event["openmeteo_key"]
    seventimer_key = event["7timer_key"]

    # Read data
    s3 = boto3.client("s3")
    openmeteo_data = json.loads(
        s3.get_object(Bucket="esgi-lyon-iabd-m2-cloud", Key=openmeteo_key)
["Body"].read().decode()
    )
    seventimer_data = json.loads(
        s3.get_object(Bucket="esgi-lyon-iabd-m2-cloud", Key=seventimer_key)
["Body"].read().decode()
    )

    # TODO: Transformer et fusionner le sdonnées
    data = ...

    # Upload data
    s3.put_object(Bucket="esgi-lyon-iabd-m2-cloud", Key=..., Body=json.dumps(data))

    return {"status_code": 200}
```

Vérifier que le DAG tourne bien

12. Créer une table sur Glue qui permet de requêter les données
13. Ajouter un opérateur AthenaOperator dans le DAG pour compter le nombre de lignes dans la table après chaque mise à jour et print ce résultat.

Exercice 2

Dans cet exercice, on se met dans une situation d'une entreprise de streaming audio. Chaque jour, des données d'écoute sont collectées et stockées de façon assez brut sur S3 au format json. Chaque donnée correspond à un évènement (soit le début d'une écoute, soit la fin)

Les données se trouvent sur S3, avec des nouvelles données chaque jour dans un nouveau préfixe :

```
s3://esgi-lyon-iabd-m2-cloud/{UserName}/audio/bronze/{yyyy-MM-dd}/
```

Le schema est le suivant :

- `uid` : L'identifiant du user qui a écouté un morceau. (Exemple `uid_42`)
- `session` : L'identifiant de la session d'écoute (exmple `42`). Un même user peut écouter plusieurs morceaux dans une journée (chaque écoute correspond à une session)
- `event` : Peut valoir "start" ou "stop" si la ligne correspond au début ou à la fin d'uneé coute
- `time` : Le timestamp correspondant à l'évènement (start / stop)

- `artist` : Le nom de l'artiste écouté par le user
- `song` : Le nom de la chanson
- `genre` : Le genre de la chanson

Exemple :

uid	session	event	time	artist	song	genre
uid_42	1	start	1705323600	Taylor Swift	Shake it off	Pop
uid_12	3	start	1705323720	Coldplay	Everglow	Pop
uid_12	3	stop	1705323780	Coldplay	Everglow	Pop
uid_42	1	stop	1705323900	Taylor Swift	Shake if off	Pop

En plus, nous possédons des informations sur le consentement de chaque personne ici :

```
s3://esgi-lyon-iabd-m2-cloud/public/td5/consent/
```

Sous la forme

- `uid` : Le user id
- `analytics` (true/false) : Si la personne a consentit à l'analytics
- `personalization` (true/false) : Si la personne a consentit à la personalisation
- `adtargeting` (true/false) : Si la personne a consentit à la publicité ciblée

Exemple :

uid	analytics	personalization	adtargeting
uid_1	true	true	false
uid_2	false	false	false
uid_3	true	true	true

Le but de cet exercice est de construire progressivement un DAG qui capable :

- De se lancer uniquement lorsque les données brutes sont disponible sur S3 en utilisant un sensors
- D'exécuter un job spark sur EMR qui prend les données brutes pour reconstruire une session complète et ajouter le consentement
- Mettre à jour les partitions Glue ensuite
- Envoyer une notification en cas de succès/échec

1. Comme pour l'exercice 1, **créer un DAG** `audio` avec :

- Un EmptyOperator *start*
- Une start date correspondant aux plus vieilles données disponibles sur S3
- L'option `catchup` à `True` cette fois pour traiter toutes les données que nous possédons
- Périodicité : `"@daily"` (un argument compris par Airflow pour lancer le DAG tous les jours à minuit)

2. On souhaite désormais écrire un script PySpark qui devra s'exécuter sur EMR Serverless (comme lors du TD2 - Exercice 3).

Le but est de construire un dataframe avec, pour chaque couple uid/session :

- uid

- session
- start_time : Le temps du début de la session (en utilisant la fonction to_timestamp pour convertir le timestamp présent sous forme de nombre dans les données brutes)
- end_time : Le temps de fin de la session
- artist
- song
- genre

En plus, on souhaite garder *uniquement les données des personnes ayant consentit à l'analytics*

Exemple :

uid	session	start_time	end_time	artist	song	genre
uid_42	1	2024-01-15 14:00:00	2024-01-15 14:05:00	Taylor Swift	Shake it off	Pop
uid_12	3	2024-01-15 14:02:00	2024-01-15 14:03:00	Coldplay	Everglow	Pop

Les données doivent être enregistrer au format parquet sur S3, avec une partition par date, sous le chemin :

```
s3://esgi-lyon-iabd-m2-cloud/{UserName}/audio/silver/date={yyyy-MM-dd}/
```

Le job spark doit prendre un argument en entrée correspondant à la date au format `yyyy-MM-dd`

💡 Voici un exemple de code pyspark à compléter :

```python
import sys
import pyspark
from pyspark.sql import SparkSession
import pyspark.sql.functions as f

spark = SparkSession.builder.getOrCreate()

date = sys.argv[1]

raw_data = spark.read.json("...")
consent_data = spark.read.json("...")

raw_consented_date = raw_data.join(...)

start_events = raw_consented_date.where(...)
end_events = raw_consented_date.where(...)


# We need to use aliases to select precise column
clean_data = (
    start_events.alias("start")
    .join(stop.alias("stop"), ...)
    .select(
        ...
    )
)
```

```
clean_data.write.mode("overwrite").parquet("...")
```

Créer le script et le déposer sur S3 sous le préfixe `{UserName}/`

3. **Modifier le DAG pour ajouter un operateur** EmrServerlessStartJobOperator

Toutes les informations nécessaire (nom du role, id de l'application, etc.) peuvent être trouvé dans l'énoncé du TD2 exercice 3 pour certains et sur la console AWS pour d'autres.

Attention le job EMR s'attend à un argument en entrée correspondant à la date des données (information disponible avec les arguments templaté d'Airflow - voir CM)

Déployer et faire marcher le DAG pour les 4 jours de données

4. On souhaite pouvoir requêter les données créer via Athena. **Créer manuellement une table Glue** qui se base sur les données

```
s3://esgi-lyon-iabd-m2-cloud/{UserName}/audio/silver/
```

Voir TD2 exercice 3 question 8 si besoin.

5. Pour que les données soient requêtables, il faut déclarer à Glue les partitions où se trouvent les données (voir CM 2 si besoin d'un rappel sur les partitions). On souhaite que les partitions soient mise à jour dans le DAG, après que les données aient été mise à jour sur S3. Pour cela, on peut exécuter une requête Athena de la forme :

```
ALTER TABLE my_db.my_table ADD PARTITION (partition_column = partition_value) LOCATION
's3://...'
```

Ajouter un opérateur AthenaOperator en aval de l'opérator EMR pour mettre à jour les partitions

Vérifier que le DAG tourne bien et que les données sont requêtable via Athena

6. Le DAG est programmé pour se lancer tous les jours à minuit, mais rien ne garantit que les données brutes seront disponible à minuit. Grâce aux *sensors*, on peut créer une tâche qui se met en attente d'une condition.

Ajouter un sensor S3KeySensor qui vérifie que la présence des données brutes sur S3 en amont du job EMR.

7. Pour vérifier le bon fonctionnement du sensor, supprimer un fichier source sur S3, vérifier que le S3 se met attente (en regardant ses logs), puis remettre le fichier sur S3 et le voir passer au vert.

8. Nous allons maintenant améliorer la résilience de ce DAG.

- On sait que l'exécution de requêtes Athena peut parfois planter si trop de requêtes sont envoyées en même temps. Dans ce cas, il suffit de relancer la requête
- Dans le cas d'EMR, si on sait que les données sources sont présentes sur S3, une erreur du job nécessitera forcément une action manuelle, et il n'est pas nécessaire de faire une relance (sauf pour avoir plus de coûts)

Ajouter un mécanisme de *retry* pour prendre ces éléments en compte

> *Quelques docs utiles :*
>
> - *Code source du S3KeySensor*

> - *Documentation boto3*