RÉPUBLIQUE DU SÉNÉGAL

Un Peuple - Un But - Une Foi

Ministère de l'Économie, du Plan et de la Coopération

****

Agence nationale de la Statistique et de la Démographie (ANSD)

Ecole nationale de la Statistique et de L'Analyse économique Pierre NDIAYE (ENSAE)

BIGDATA ET CLOUD COMPUTING:

DOCUMENTATION SUR LA CREATION D'UN DATA WAREHOUSE POUR L'ANALYSE DES VENTES

Rédigé par :

Judicaël Oscar Gandwende KAFANDO

Ramatoulaye NDEYE NDOYE FALL

Djerakei MISTALENGAR

Ben Idriss Diloma SOMA

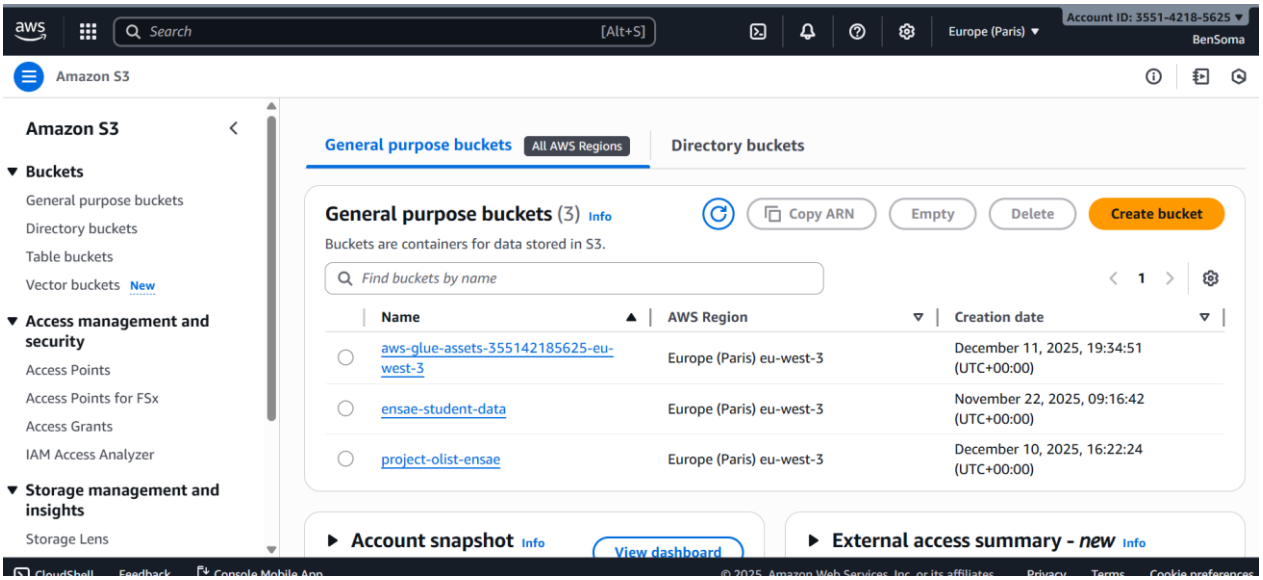
Sous la supervision de :

Mme Mously DIAW

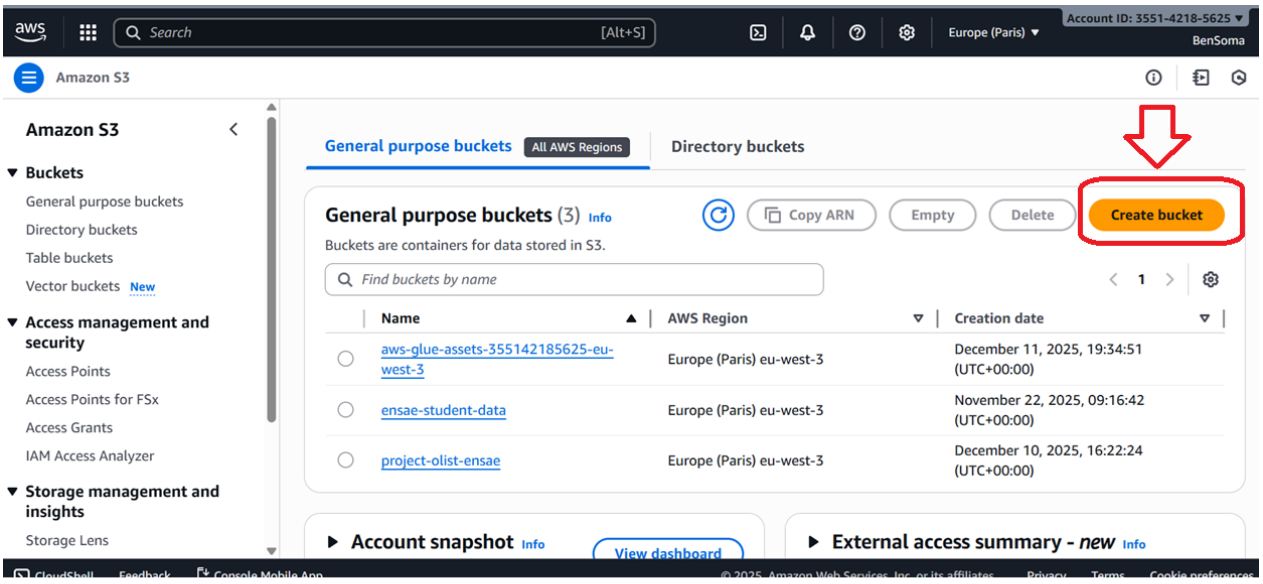
Table des matières

I. Création d'un data lake avec amazon S3

On part sur AWS , ensuite on recherche le service S3 qui se présente comme ça :

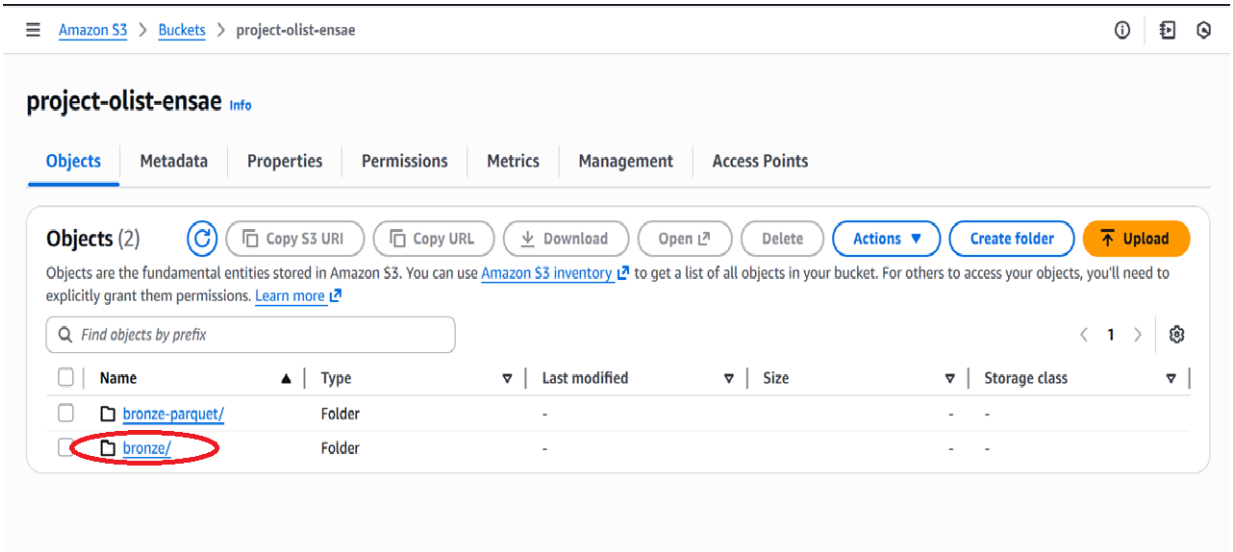


On crée un compartiment en créant sur le bouton create bucket

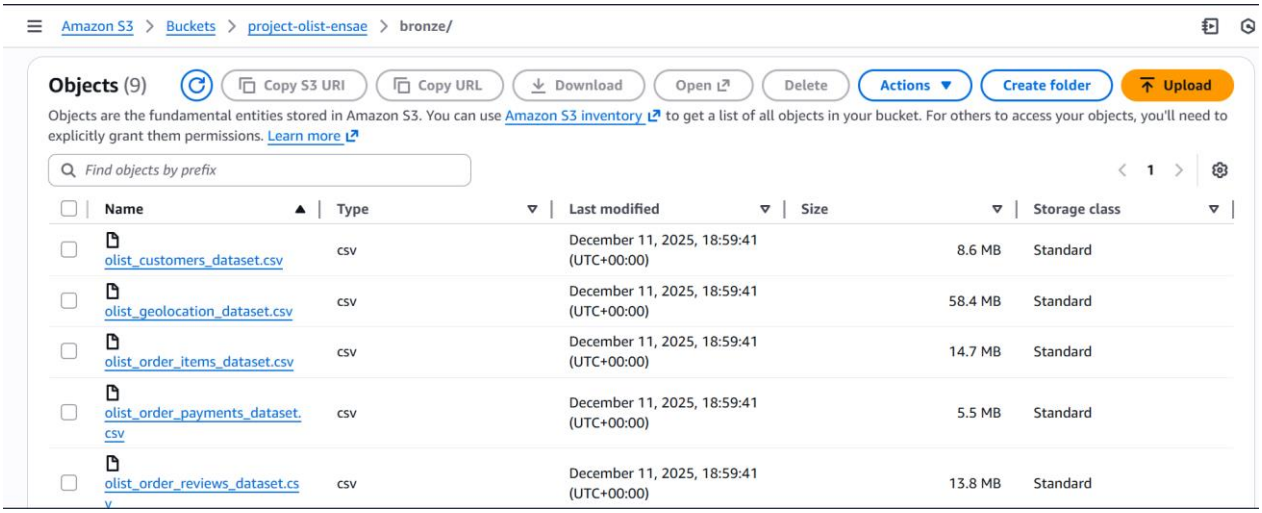


Après la création du bucket, les tables de la base de données sont chargées dans celui-ci.

Le bucket, déjà créé, a été nommé **projet-olist-ensae**. En cliquant sur projet-olist-ensae, on accède à cette section où la base de données est visible, les différentes tables se trouvant dans le dossier bronze.

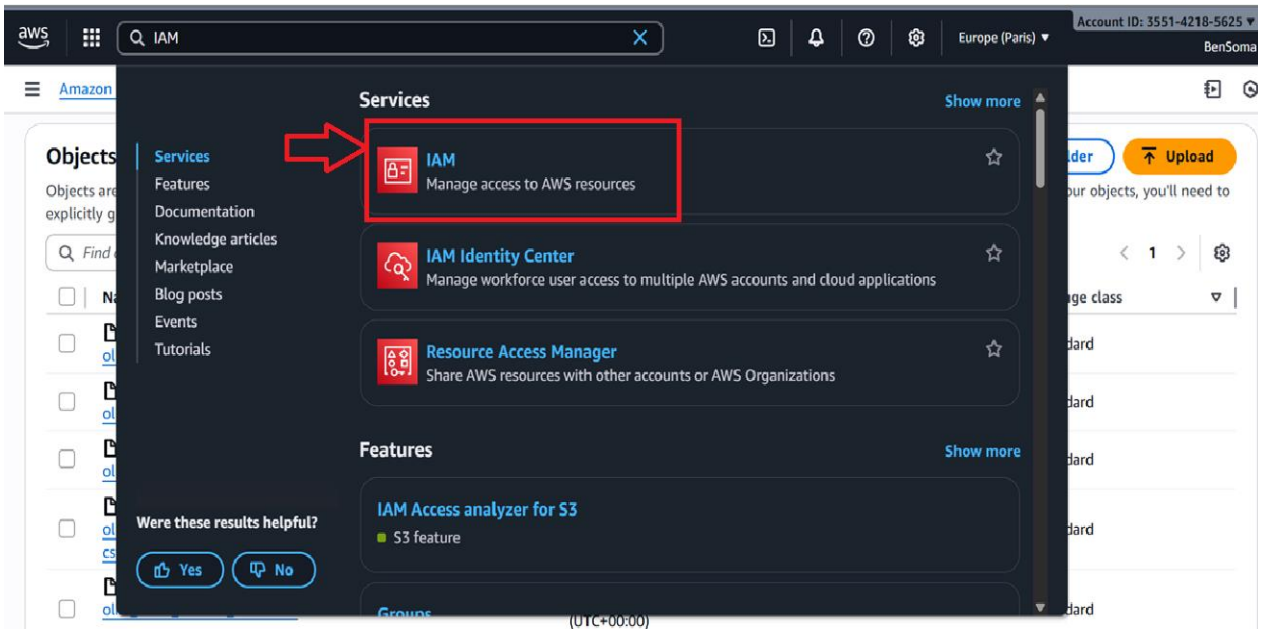


Voici comment les différentes tables se présentent :

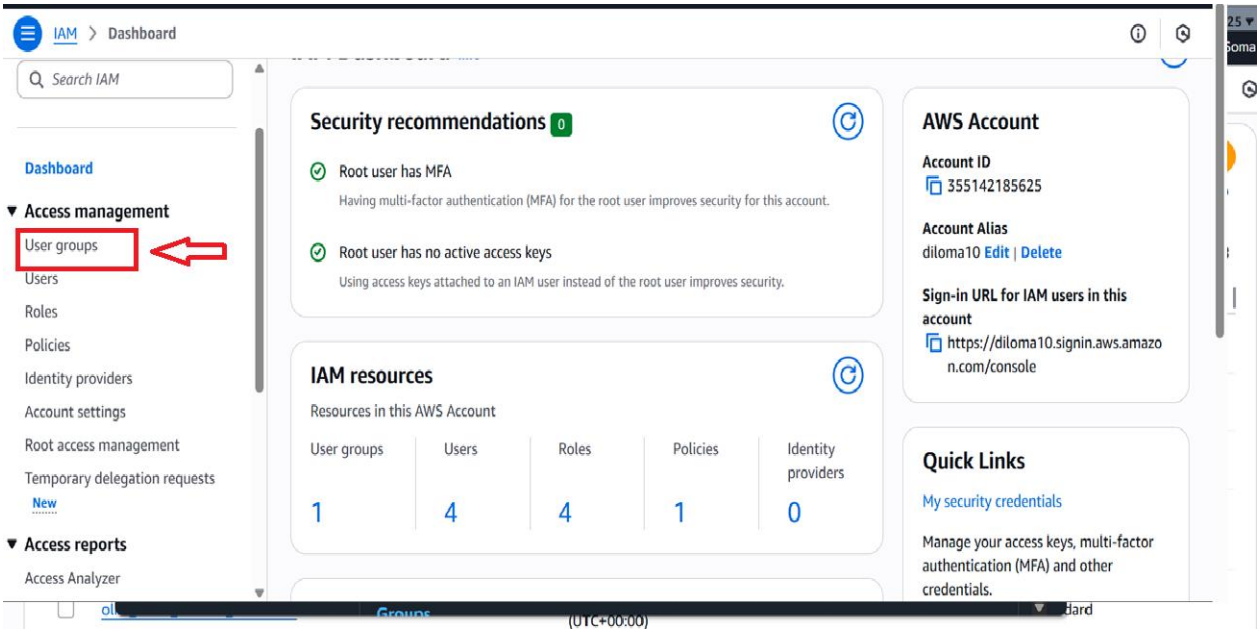


I.1 Création d'un groupe et de plusieurs IAM ID (5 au total)

Partir au niveau de la barre de recherche de la console et rechercher **IAM**



On clique sur **IAM**

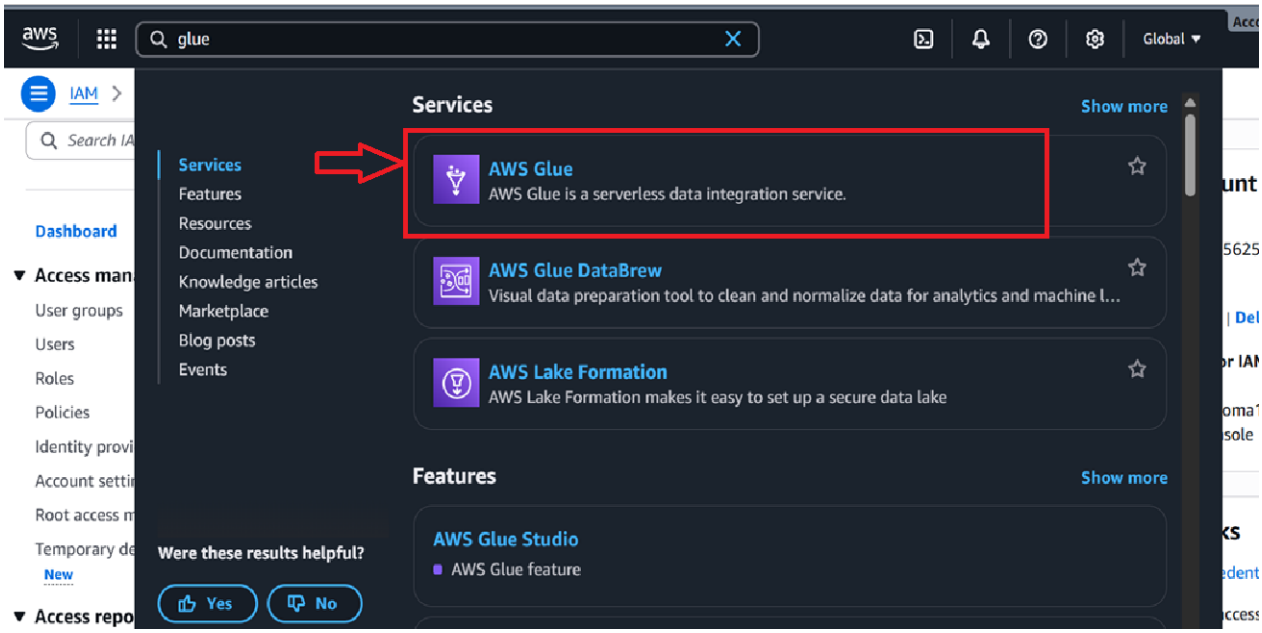


À présent, il convient de cliquer sur **User_groups** afin de créer les différents utilisateurs **IAM** disposant d'un accès complet aux ressources.

La création de ces utilisateurs permettra aux membres de l'équipe d'accéder à l'ensemble des travaux et d'y apporter des modifications.

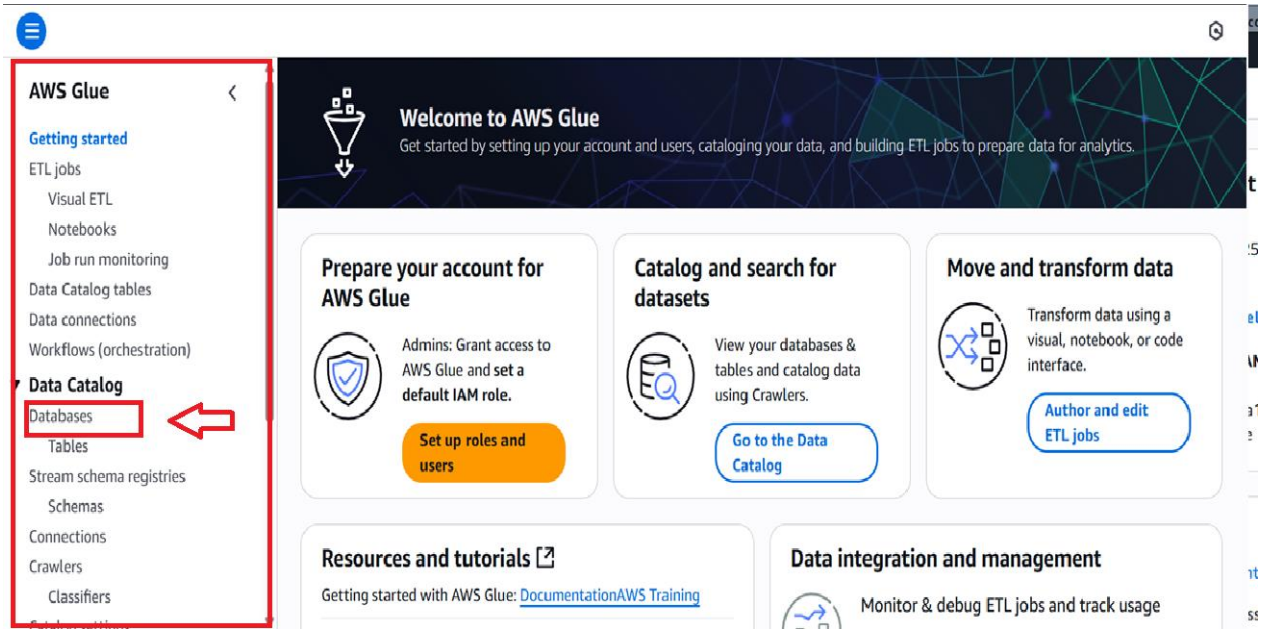
I.2 Création d'un bucket sur AWS Glue

Rechercher **Glue** dans la barre de recherche



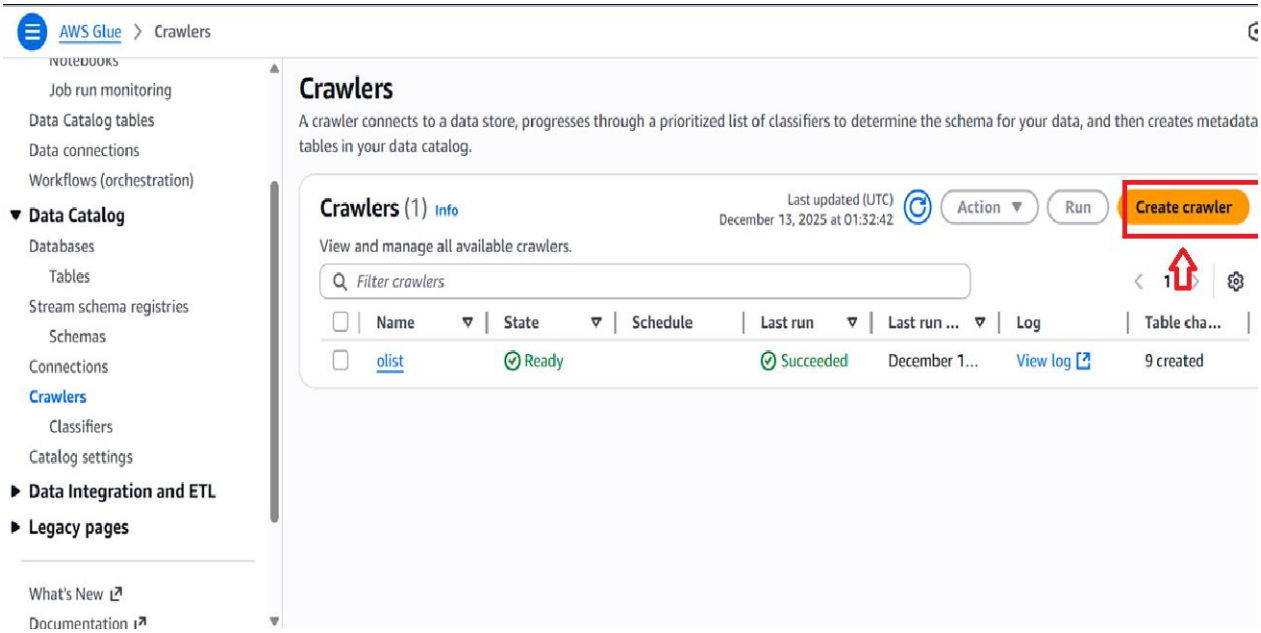
Il faut cliquer sur **AWS Glue**.

Après avoir cliqué sur ce service, on se rend dans l'onglet **Databases** situé à droite.



Il convient ensuite de cliquer sur « **Add database** » afin de créer la base de données sur AWS Glue, ce qui permettra de charger les tables depuis S3.

On crée un CRAWLER en cliquant sur **CRAWLER** dans la barre latérale



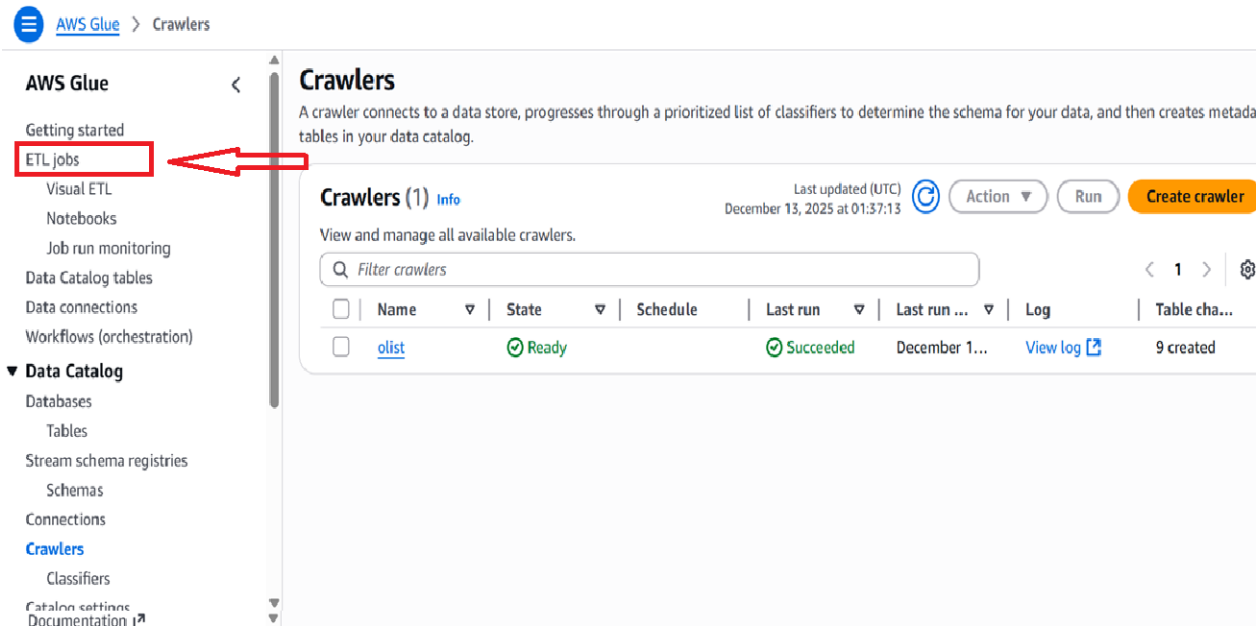
Après la creation de notre crawler, cliquer sur « **run a crawler** » et apres ca nous verrons les différentes tables qui ont été importées sur **GLUE** .

La base a eté crée et se nomme olis-database on clique sur olist-database

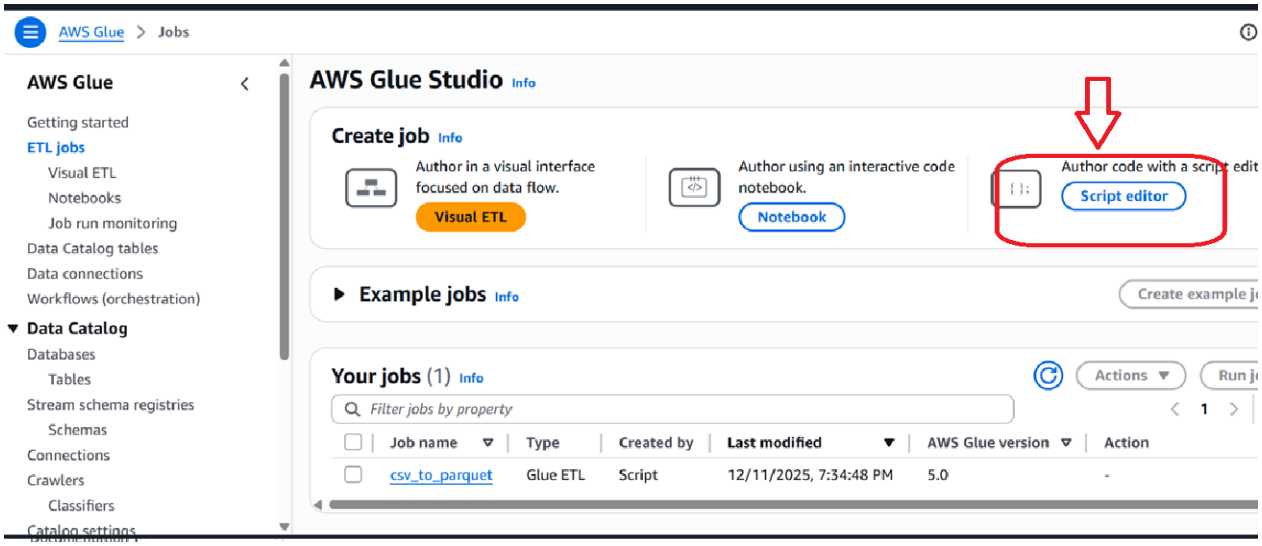
I.3 Transformation des tables CSV en parquet

Pourquoi parquet ?

- Pour pouvoir gérer de grandes bases de données
- Pour faire ceci, il faut aller au niveau de **GLUE** dans la barre latérale a gauche nous cliquons sur « **ETL JOB** »



Dans **ETL Jobs** on clique sur script Editor



On choisit l’option spark pour le code

Un code a eté utilisé pour transformer les fichiers CSV en parquet : il se trouve en annexes.

Lorsque l'on se rend dans S3, il est possible de constater qu'un nouveau dossier a été créé et qu'il contient les fichiers parquet. Pourquoi des tables sous format parquet ?

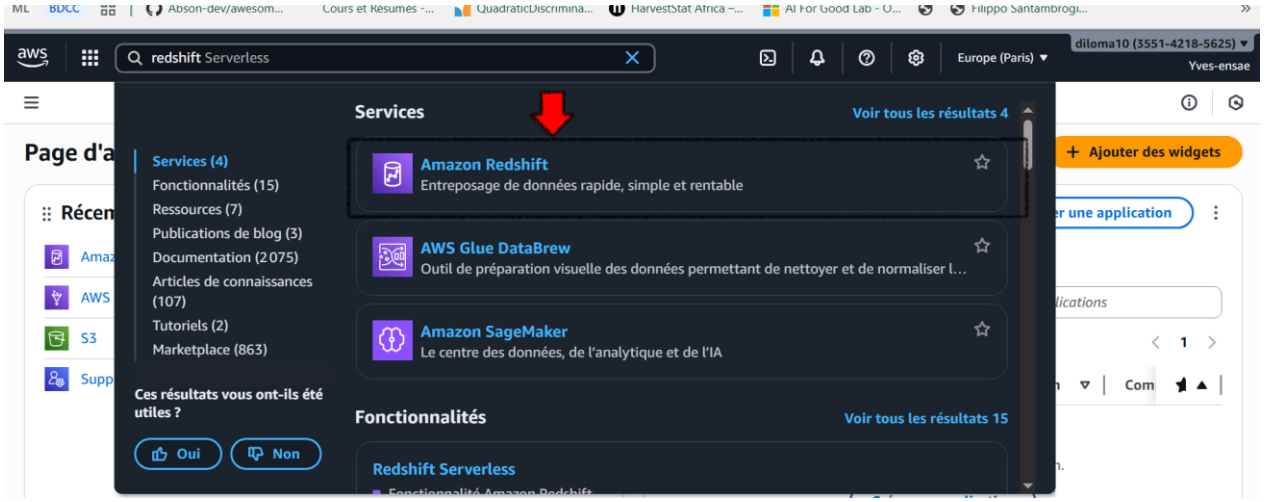
Parce que le format Parquet est plus performant que le CSV : il est orienté colonnes, permet une **compression efficace**, réduit l’espace de stockage sur S3, **accélère les lectures avec Spark**, conserve le **schéma des données** et est mieux adapté aux **analyses Big Data**.

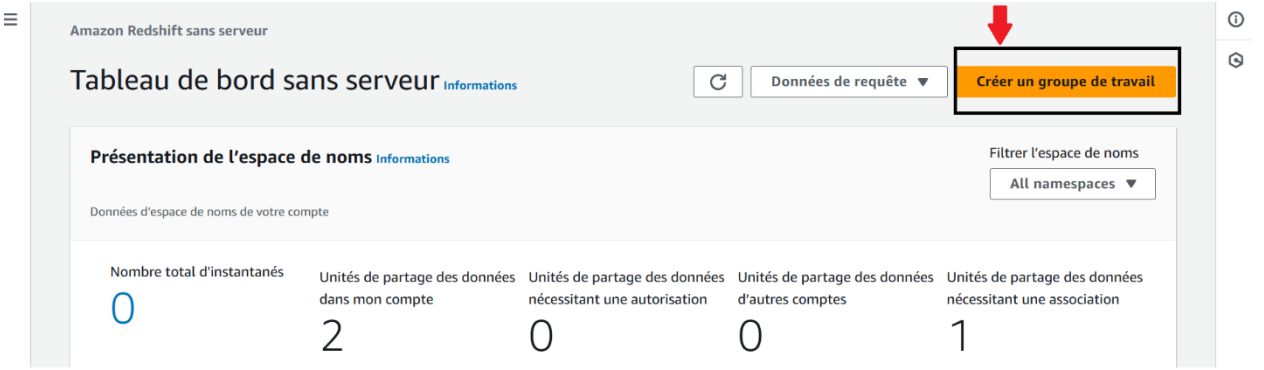
Tableau 1: Résumé des services utilisés et leur rôle jusqu’à présent

Service AWS	Utilité dans le projet
Amazon S3	Stockage scalable et sécurisé des données brutes et transformées.
IAM	Gestion des accès et des permissions pour les utilisateurs et services.
AWS Glue	Catalogage des données, crawling automatique, et transformation ETL serverless.
Glue Data Catalog	Catalogue centralisé des métadonnées pour interroger les données avec Athena/Redshift.
Glue Crawler	Découverte automatique des schémas et mise à jour du catalogue.
Glue ETL Job	Exécution de scripts PySpark pour transformer les données (CSV → Parquet).
Parquet	Format de stockage optimisé pour l’analytique Big Data.

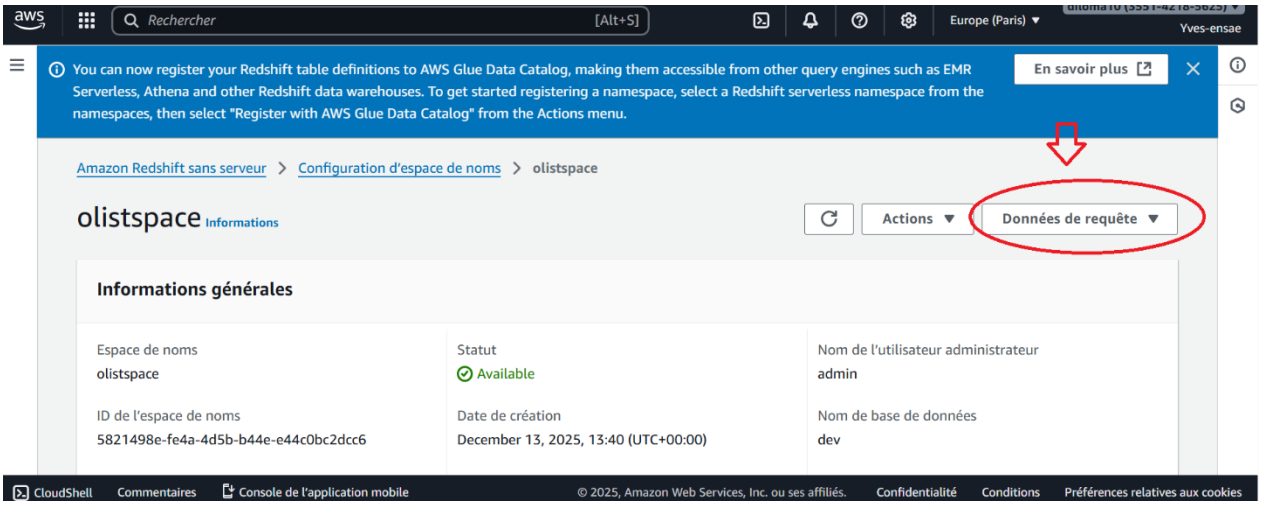
Source : calculs des auteurs

Après ces différentes étapes, l'accès à Amazon Redshift est effectué.





Après la création du groupe il faut cliquer sur le nom dans l'espace de groupes



Ensuite, il faut se rendre dans **Query Editor** afin d'ajouter des scripts, d'importer des tables et d'effectuer les transformations nécessaires.

Dans cette étape, de nombreuses transformations sont réalisées sur la base de données.

Avant toute transformation, il est nécessaire d'importer les données de S3 vers Redshift. Les codes correspondant à cette tâche figurent en annexe.

Une fois l'importation terminée, les données sont transformées, puis stockées dans un dossier **silver**. Les codes utilisés pour ces transformations sont également disponibles en annexe.

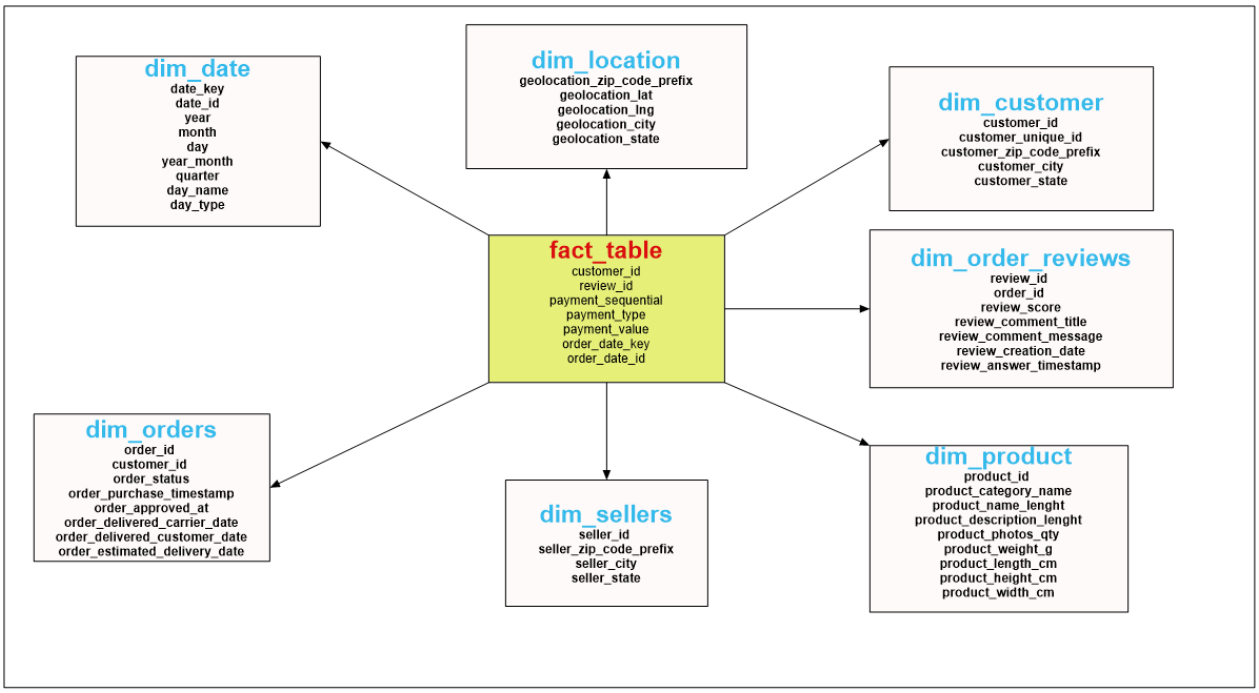
Le schéma du data warehouse est ensuite défini.

Après cela, un dossier **dw** est créé pour stocker les tables de dimensions et de faits, selon une modélisation en **étoile**.

Pourquoi le modèle étoile ?

Nous avons utilisé le **modèle en étoile** car il permet une **modélisation simple et lisible**, optimise les **performances des requêtes analytiques**, facilite les **jointures entre tables de faits et dimensions**, et est particulièrement adapté aux **outils de BI** comme Power BI pour l'analyse et la visualisation des données.

Figure 1: Modèle en étoile



Source : auteurs

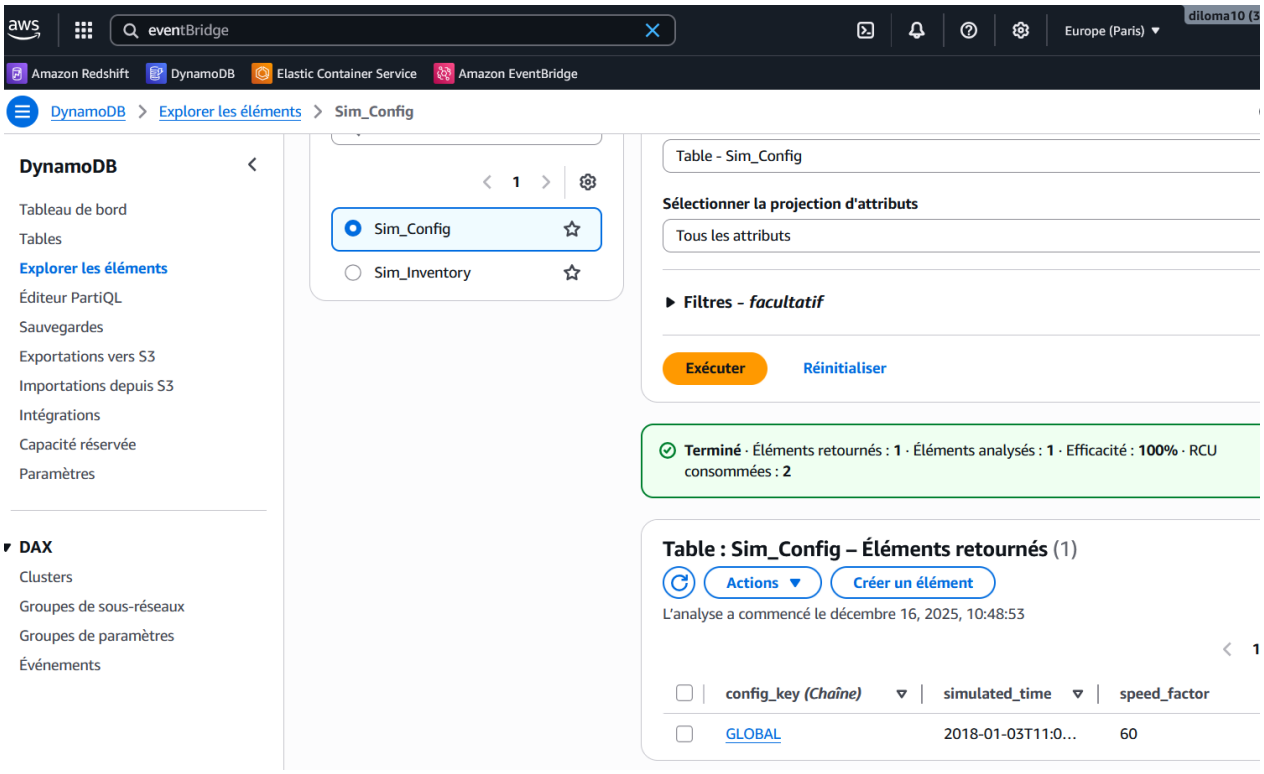
II. PARTIE TEMPS REELLE DU PROJET

Dans le cadre de la construction de notre Data Warehouse basé sur le dataset Olist, nous avons mis en place une architecture capable de gérer les données en temps réel. Il est important de préciser que, dans notre contexte, nous distinguons deux approches du "temps réel". La première est une approche **"Batch rapide"**, où les données arrivent par intervalles de quelques minutes, et la seconde est une version purement streaming, de l'ordre de la seconde ou de la milliseconde. Cette section de la documentation se concentre spécifiquement sur l'implémentation de la partie Batch, simulant un flux de commandes continu qui alimente notre entrepôt de données.

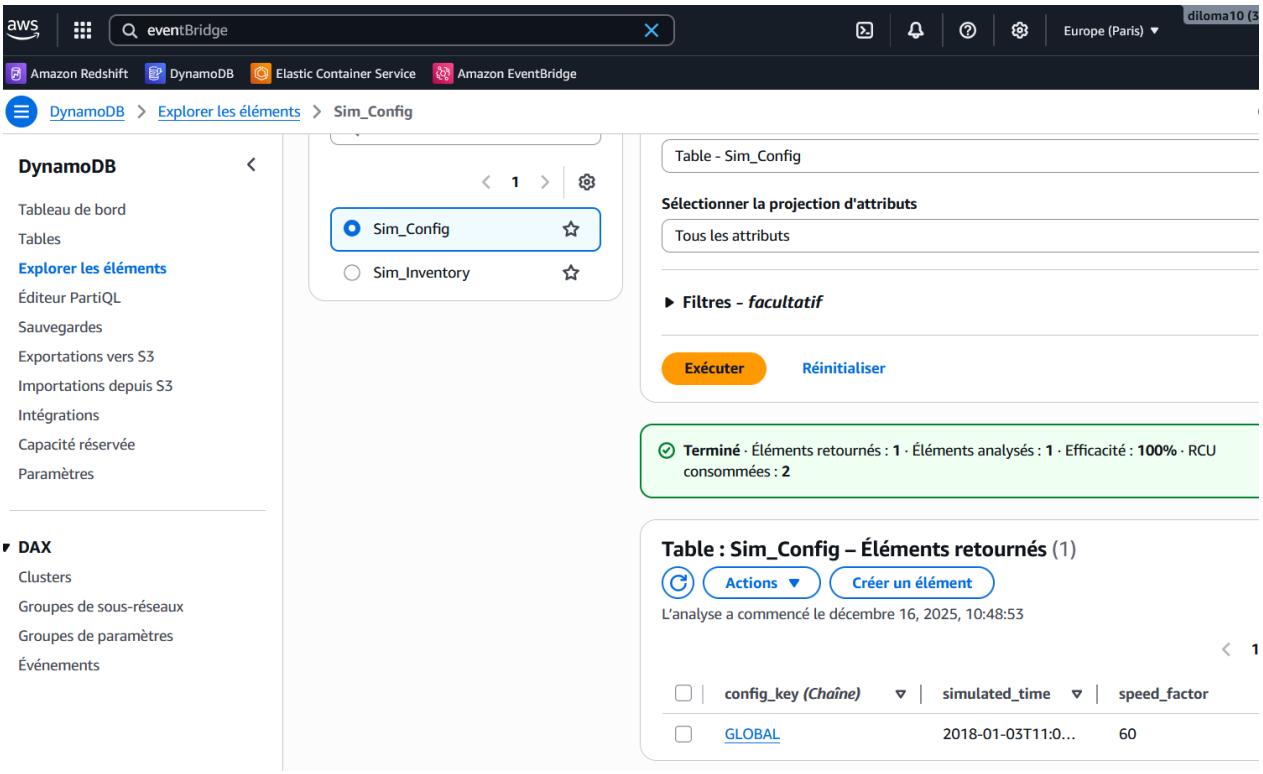
II.1 1. Le moteur de simulation et la gestion des stocks (DynamoDB)

Pour simuler un environnement de production réaliste, nous ne pouvions pas simplement injecter des données statiques. Nous avons donc créé un moteur de simulation basé sur Amazon

DynamoDB. La première composante est la table Sim_Config. Cette table agit comme le chef d'orchestre temporel de notre simulation. Elle contient une clé globale qui stocke l'heure virtuelle (simulated_time) et un facteur d'accélération (speed_factor). Cela nous permet de faire avancer le temps beaucoup plus vite que la réalité (par exemple, une minute réelle équivaut à une heure simulée), testant ainsi la robustesse du pipeline sur de longues périodes virtuelles en peu de temps.



En parallèle, nous avons mis en place la table Sim_Inventory pour gérer les stocks de produits. Plutôt que de reprendre l'intégralité du dataset Olist, nous avons sélectionné un échantillon pertinent pour démontrer la faisabilité du concept. Cette table contient les identifiants produits (product_id), ainsi que leur niveau de stock actuel (stock_level), la catégorie et le prix. Nous avons utilisé un script Python, exécuté via l'AWS CLI, pour peupler initialement cette table. Ce script prépare le terrain pour que notre générateur de commandes puisse interagir avec des données concrètes.

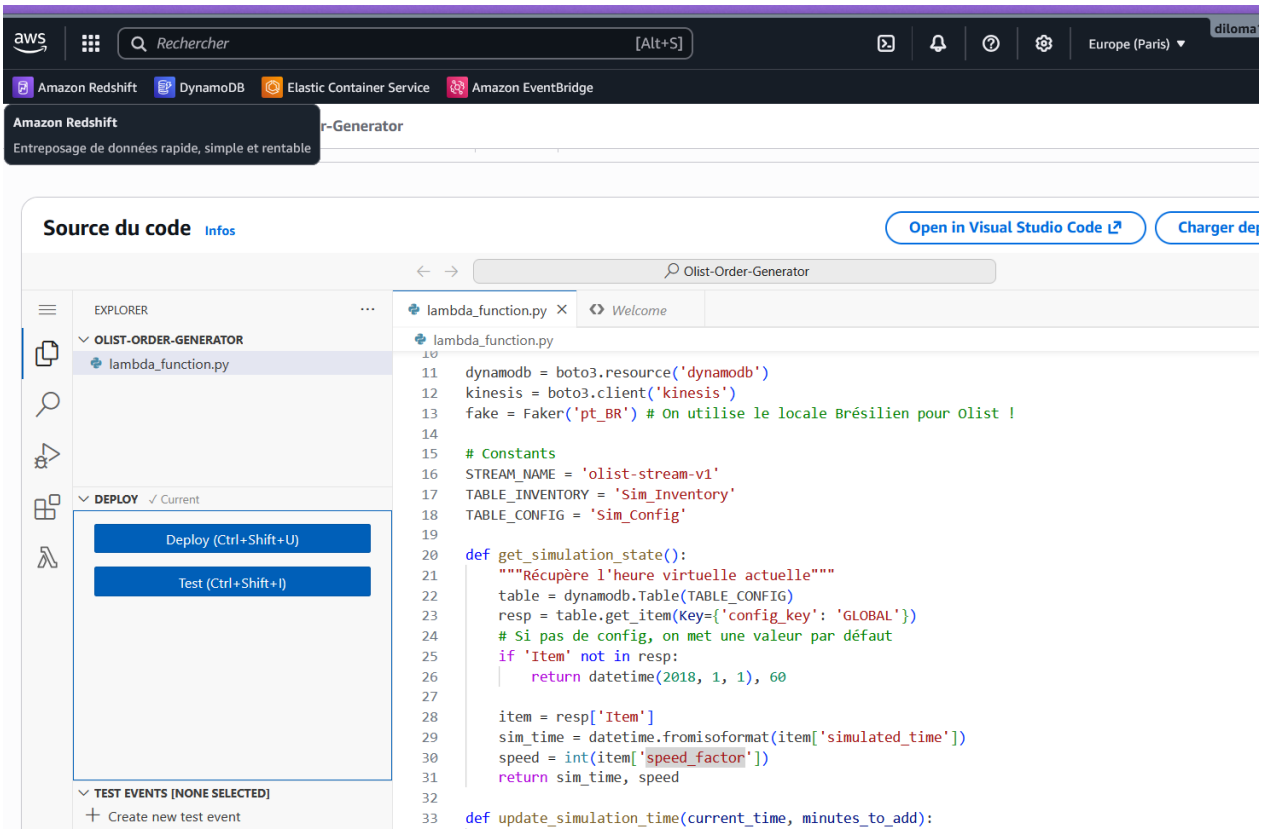


II.2 Génération de commandes via AWS Lambda

Le cœur de notre ingestion de données réside dans une fonction AWS Lambda nommée "Order Generator". Cette fonction est déclenchée automatiquement toutes les minutes grâce à un planificateur Amazon EventBridge. La logique du code est conçue pour être performante et réaliste. Au démarrage, la fonction consulte la table Sim_Config pour savoir "quelle heure il est" dans la simulation. Pour éviter de surcharger la base de données et réduire les coûts, nous avons implémenté un système de mise en cache mémoire qui charge une liste limitée d'IDs de produits une seule fois par cycle, évitant ainsi de scanner toute la table d'inventaire à chaque commande.

La fonction génère ensuite un nombre aléatoire de commandes (entre 3 et 20) en utilisant la librairie Faker localisée pour le Brésil, afin de rester fidèle au contexte Olist. Point crucial de notre architecture : la gestion de la cohérence des données. Avant de valider une commande, le script effectue une transaction atomique sur DynamoDB pour décrémenter le stock. Si le stock est insuffisant, la commande est annulée, ce qui empêche de vendre des produits inexistants. Une fois la commande validée, elle est convertie en JSON et poussée vers notre flux de données.l'evenement peux etre actité ou désactivé ,cela n'impacte pas le temps similé et nous cela nous économise de l'argent sans qu'il y ait des troue. Firehose stockes le format parquet en format

année/mois/heure/jour/munites en se referant au temps de la vrai vie , pour changer cela il vas faloir appeler une nouvelle function lambda et c'est couteux et comme nous ne nous servons pas directement de cette information (par contre si on voulais stocker dans ce style les data statique et les similé ensemble alors la oui) donc nous avons laissé intacte.

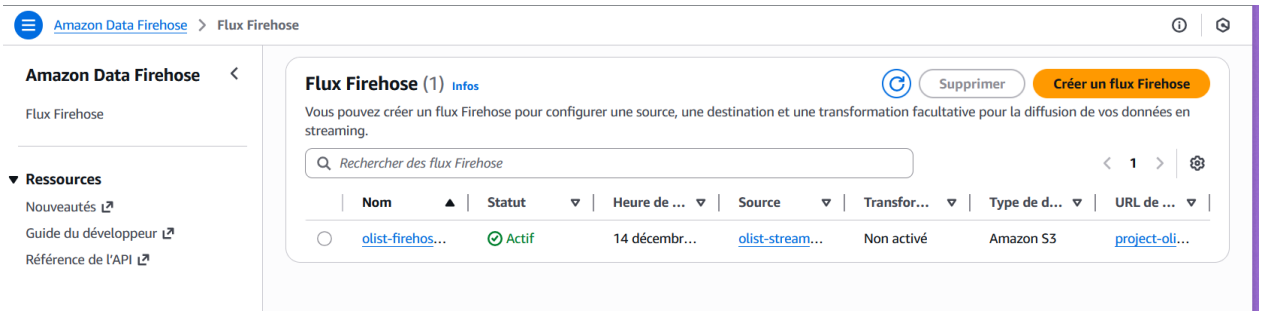


II.3 Ingestion, Buffering et Stockage

Une fois les commandes générées, elles entrent dans le pipeline d'ingestion via Amazon Kinesis Data Stream. Ce flux continu est ensuite consommé par Kinesis Firehose. Nous avons configuré Firehose avec des paramètres de tampon spécifiques : il accumule les données soit pendant 3 minutes, soit jusqu'à atteindre 5 Mo. Ce mécanisme de "micro-batch" est essentiel pour éviter de créer une multitude de trop petits fichiers, ce qui nuirait aux performances ultérieures.

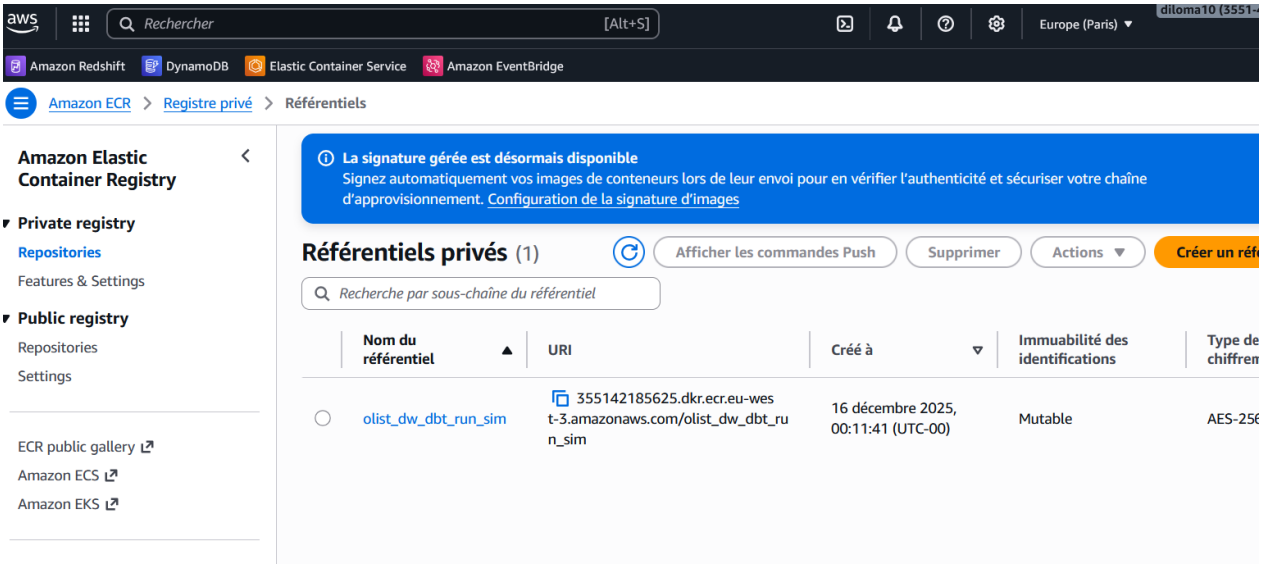
Durant ce transit dans Firehose, une transformation majeure a lieu. Grâce à son intégration avec AWS Glue Data Catalog, Firehose convertit automatiquement les données JSON brutes au format Parquet avant de les écrire moyenant la creation d'un data catalogue conforme au format json

attendu. Le format Parquet, étant en colonnes et compressé, est beaucoup plus performant pour l'analytique. Les fichiers résultants sont déposés dans notre bucket S3, dans un dossier désigné comme notre couche "Bronze" (sim-bronze).



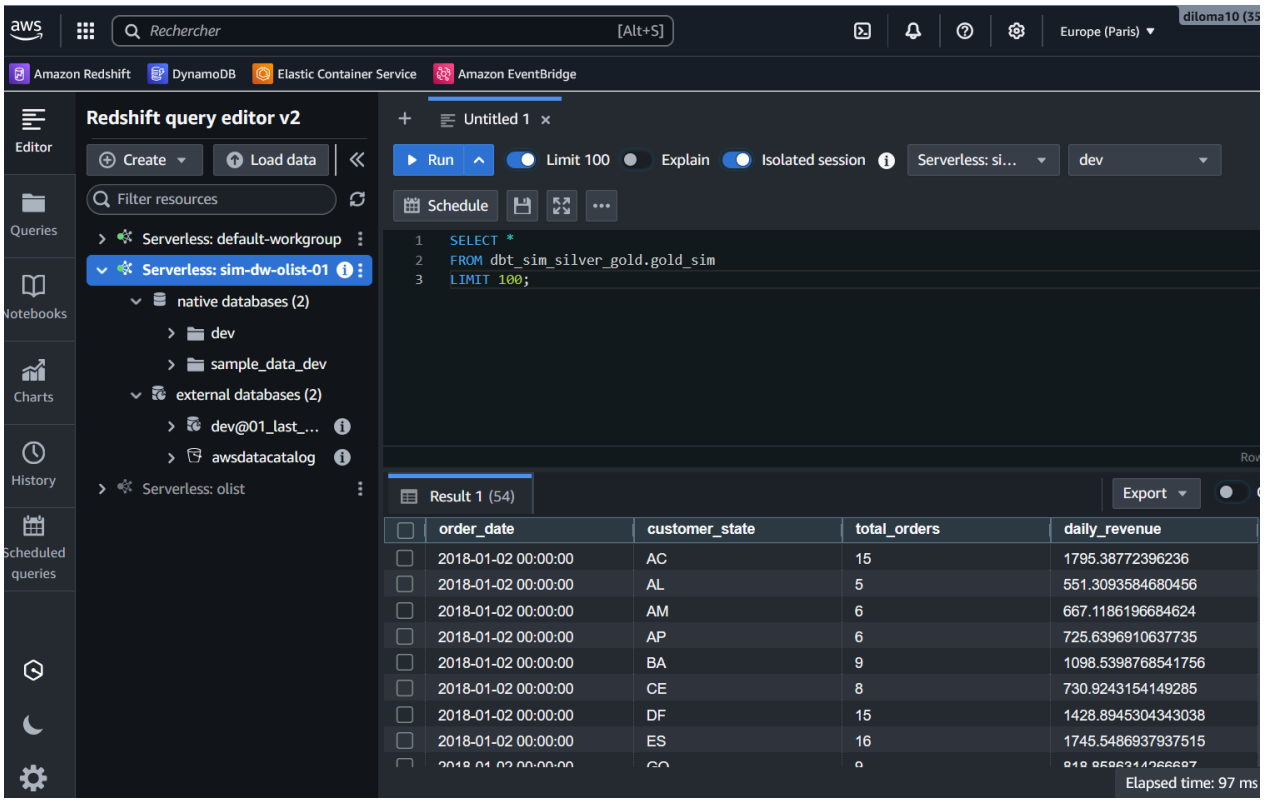
II.4 Intégration dans le Data Warehouse et Transformation (Redshift & dbt)

Pour rendre ces données accessibles à l'analyse, nous utilisons Amazon Redshift Serverless couplé à Redshift Spectrum. Grâce au schéma externe défini dans AWS Glue, Redshift peut lire directement les fichiers Parquet stockés dans S3 sans avoir besoin de les charger physiquement dans son propre stockage interne. Cela nous offre une flexibilité énorme et sépare les coûts de stockage des coûts de calcul.



La transformation de la donnée brute (Bronze) vers des modèles nettoyés (Silver) et agrégés (Gold) est assurée par dbt (data build tool). Pour automatiser cette étape, nous avons conteneurisé notre projet dbt et stocké l'image dans Amazon ECR. L'orchestration est gérée par AWS Step Functions. Le processus est entièrement événementiel : dès qu'EventBridge détecte qu'un nouveau fichier a

été déposé dans le dossier S3 Bronze, il déclenche la Step Function. Celle-ci lance alors le conteneur ECR qui exécute les transformations dbt. Pour sécuriser l'accès à la base de données, le conteneur récupère les identifiants de connexion de manière sécurisée via AWS Secrets Manager.

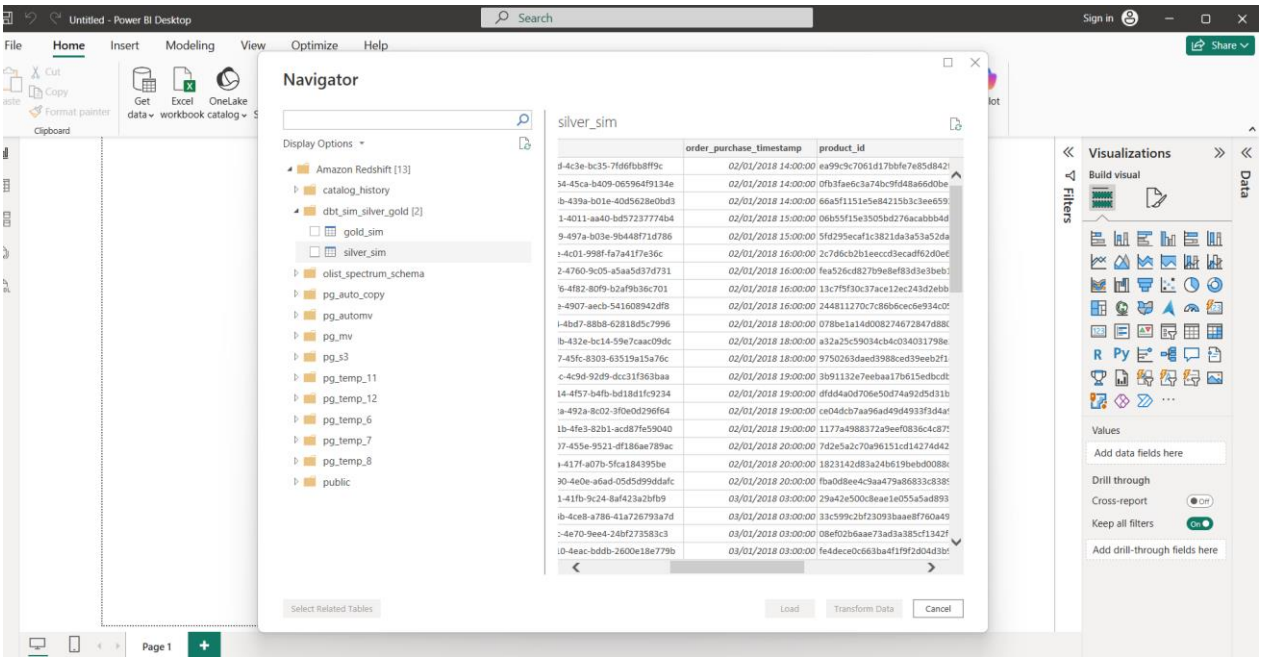


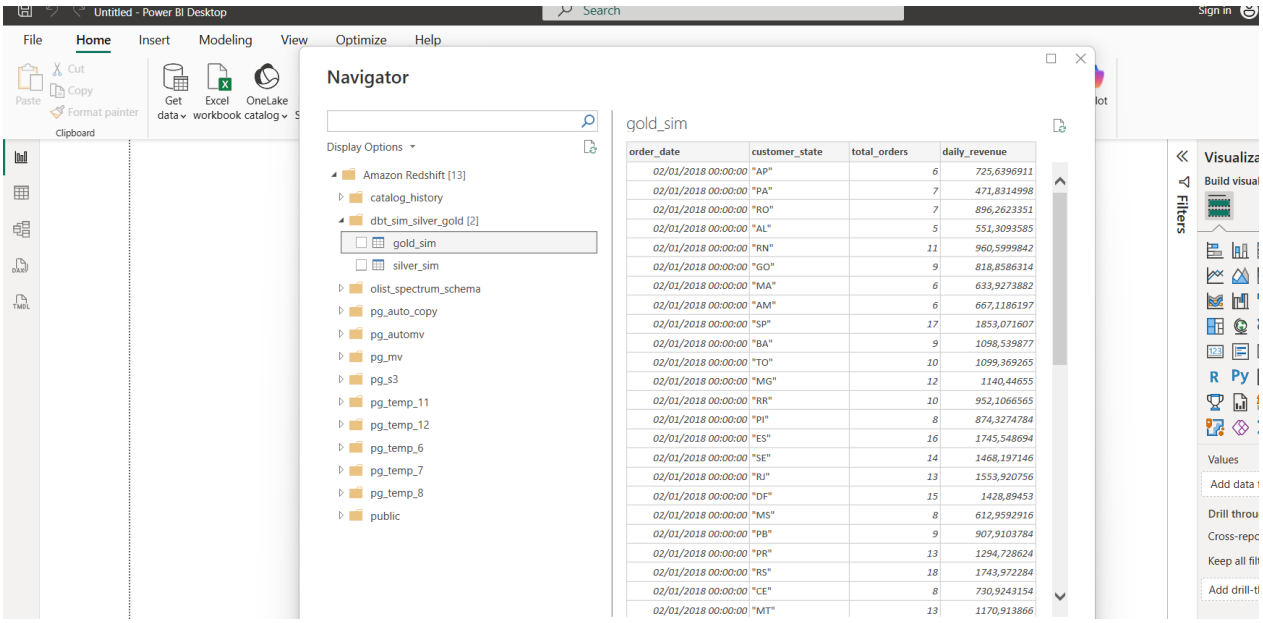
II.5 Visualisation (Power BI)

La finalité de ce pipeline est la visualisation des données. Nous avons connecté Power BI directement à notre cluster Redshift. Pour permettre cette connexion, nous avons dû configurer un utilisateur dédié sur Redshift et ajuster les paramètres du VPC (Virtual Private Cloud) et des groupes de sécurité pour autoriser le trafic entrant depuis Power BI. Cela permet aux parties prenantes de suivre l'évolution des commandes et des stocks avec une latence de seulement quelques minutes par rapport à la génération des données.

Les transformation pour avoir le silver et le gold data sont dans le dossier models/ sur https://github.com/Ben10-som/Data-warehouse-for-sales-analysis-

Dans la prochaine version nous passerons par Cloudformation ou terraform pour refaire le projet et y inclure la partie temps réelle.





II.6 Architecture Streaming (Temps Réel Pur) et Perspectives

Comme évoqué dans l'introduction, notre analyse distinguait deux niveaux de temporalité : le micro-batch (actuellement en place) et le streaming pur (latence de l'ordre de la seconde). Pour cette itération du projet, nous avons fait le choix conscient de ne pas implémenter la branche de streaming pur. Cette décision a été dictée par des contraintes pragmatiques de temps de développement et de budget (optimisation des coûts des services Cloud), l'architecture actuelle suffisant à démontrer la robustesse de l'ingestion de données.

Toutefois, l'architecture cible pour ce besoin spécifique est déjà théorisée. L'objectif serait de réduire drastiquement la latence en éliminant l'étape de stockage intermédiaire sur Amazon S3 ("Sim-Bronze"). Dans cette configuration, nous utiliserions l'intégration native Streaming Ingestion d'Amazon Redshift. Concrètement, le flux Kinesis Firehose ne déposerait plus de fichiers Parquet, mais injecterait les données directement dans les tables Redshift en temps réel. Cela permettrait de contourner les délais liés à la bufferisation des fichiers et aux triggers événementiels, offrant ainsi une vision quasi instantanée des transactions pour des cas d'usage critiques.

Annexe: *Code pour transforme des bases csv en format parquet*

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job

 # Initialisation du job Glue

args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)
#Emplacements S3
input_path = "s3://your-bucket-name/bronze/"          # tes CSV Olist
output_path = "s3://your-bucket-name/bronze-parquet/" # dossier Parquet à créer

# Lecture des CSV

df_csv = (
   spark.read
       .option("header", "true")
       .option("inferSchema", "true")   # (optionnel)
       .option("sep", ",")
       .csv(input_path)
)

print("Nombre de lignes chargées :", df_csv.count())
print("Schéma détecté :")
df_csv.printSchema()

# Écriture en Parquet
(
   df_csv.write
       .mode("overwrite")           # écrase si le dossier existe
       .option("compression", "snappy")
       .parquet(output_path)
)

print("Transformation CSV → Parquet terminée.")

job.commit()
```