Ejercicio: Orquestación dinámica de un proceso de carga y análisis de datos con Airflow (usando bases de datos locales y APIs)

Enunciado:

Tu empresa quiere automatizar el proceso de carga, análisis y reporte de datos de ventas desde un archivo CSV local y una API de ventas simulada. El proceso de datos incluye obtener registros desde un archivo CSV, cargar los datos a un almacenamiento local (simulando una carga en S3), y luego realizar transformaciones o análisis sobre esos datos. Finalmente, se genera un reporte de los resultados y se guarda en un archivo local.

Requerimientos:

1. **DAG Dinámico:**
    - El DAG debe ser **dinámico** y permitir la adición de nuevas fuentes de datos (por ejemplo, diferentes archivos CSV o nuevas APIs).
    - Usa **variables de Airflow** para almacenar la configuración de las fuentes de datos y la ubicación del archivo de salida.
    - Usa **plantillas Jinja** para permitir la creación dinámica de tareas y la personalización de los nombres de los archivos o rutas de almacenamiento.
2. **Extracción de Datos desde CSV y API:**
    - Debes crear un **PythonOperator** para simular la extracción de datos desde un archivo CSV local.

- Debes crear un **HttpSensor** o un **PythonOperator** para obtener datos de una **API REST** simulada, que entregue los resultados en formato JSON.
- Los datos extraídos del archivo CSV o la API deben cargarse en un "almacenamiento" local (simulando un bucket S3), para lo cual puedes usar el **PythonOperator** para mover los archivos a una carpeta local.

3. **Transformación y Análisis de Datos:**

   - Después de cargar los datos, debes usar un **PythonOperator** para realizar algunas transformaciones en los datos (por ejemplo, limpieza, agregar nuevas columnas o filtrar los registros).
   - Genera un reporte que contenga estadísticas o el estado de los datos transformados (por ejemplo, cantidad de registros procesados o errores encontrados).
   - El reporte debe ser guardado en un archivo local (simulando un proceso de carga a un servidor FTP).

4. **Flujo Condicional de Tareas:**

   - Usa el **PythonBranchOperator** para tomar decisiones basadas en los resultados de las transformaciones. Por ejemplo, si el número de registros procesados es menor a un umbral, se podría decidir realizar un proceso de validación adicional.
   - Si es necesario realizar procesamiento adicional, el DAG debe redirigir las tareas a un conjunto de operadores de transformación (como **PythonOperator** o **BashOperator**).

5. **XComs:**

   - Utiliza **XComs** para compartir el estado de las transformaciones entre las tareas. Por ejemplo, después de que se extraigan los datos, puedes almacenar en XCom el número de registros procesados para que luego se use en la bifurcación con el **PythonBranchOperator**.
   - También puedes usar **XComs** para pasar el nombre del archivo generado con el reporte, para que una tarea posterior lo utilice.

6. **Uso de Pools:**

    - Simula la gestión de recursos limitados utilizando **Pools** para limitar el número de tareas concurrentes que manipulan archivos grandes o requieren un uso intensivo de recursos.
    - Asigna un **Pool** a las tareas que consumen más recursos, como las de procesamiento de datos, para evitar que el DAG se sobrecargue.

7. **Plugins:**

    - Si es necesario, crea un **Plugin** personalizado para realizar alguna tarea específica que no se pueda hacer con los operadores estándar. Por ejemplo, un operador que lea o escriba en archivos locales con formatos específicos.
    - El Plugin debe ser reutilizable y modular.

8. **Fernet Key:**

    - Utiliza una **Fernet Key** para cifrar la información sensible, como las claves de API o configuraciones de conexión. Cifra esta información en las **Variables de Airflow** y asegúrate de que se desencripte durante el proceso de ejecución del DAG.

Estructura del DAG:

1. **Extracción de Datos:**

    - Extrae los datos desde un archivo CSV local y desde una API REST simulada. Para la API, puedes hacer una llamada a un servicio local que simule la respuesta.

2. **Cargar los Datos en Almacenamiento Local (Simulando S3):**

    - Usa un **PythonOperator** para guardar los archivos procesados en un directorio local.

3. **Transformación de Datos:**

    - Realiza transformaciones básicas en los datos extraídos (por ejemplo, filtrar registros o agregar nuevas columnas).

4. **Generar el Reporte:**
    - Usa un **PythonOperator** para generar un reporte en un archivo de texto o CSV con el estado de los datos.
5. **Condicional con PythonBranchOperator:**
    - Si el número de registros procesados es bajo, ejecuta un proceso adicional de validación o limpieza de datos.
6. **Guardar el Reporte:**
    - Guarda el archivo de reporte generado en un directorio local, simulando la carga a un servidor FTP.

Tareas y Operators a usar:

- **PythonOperator** (para procesar y transformar los datos, generar el reporte)
- **HttpSensor** o **PythonOperator** (para simular la extracción de datos desde una API)
- **XCom** (para compartir el estado del procesamiento entre tareas)
- **PythonBranchOperator** (para tomar decisiones condicionales dentro del flujo)
- **Pool** (para limitar la concurrencia de ciertas tareas)
- **Plugin** (para crear un operador personalizado si es necesario)
- **Fernet Key** (para cifrar información sensible)

Instrucciones para el ejercicio:

1. **Crea un DAG** llamado `data_pipeline_local_dag` que siga la estructura mencionada.
2. **Usa variables de Airflow** para configurar las ubicaciones del archivo CSV y la API.

3. **Crea las tareas dinámicamente** usando **Jinja templating** para poder agregar nuevas fuentes de datos (como nuevos archivos CSV o nuevas APIs).

4. **Aplica la bifurcación** con el **PythonBranchOperator** y usa **XComs** para compartir información entre las tareas.

5. **Gestiona la concurrencia** de las tareas con **Pools**.

6. Si es necesario, **crea un Plugin** que realice una tarea específica dentro del DAG, como un operador personalizado para procesar datos locales o interactuar con una API.

7. Asegúrate de **cifrar las credenciales** y la configuración sensible usando **Fernet Key**.