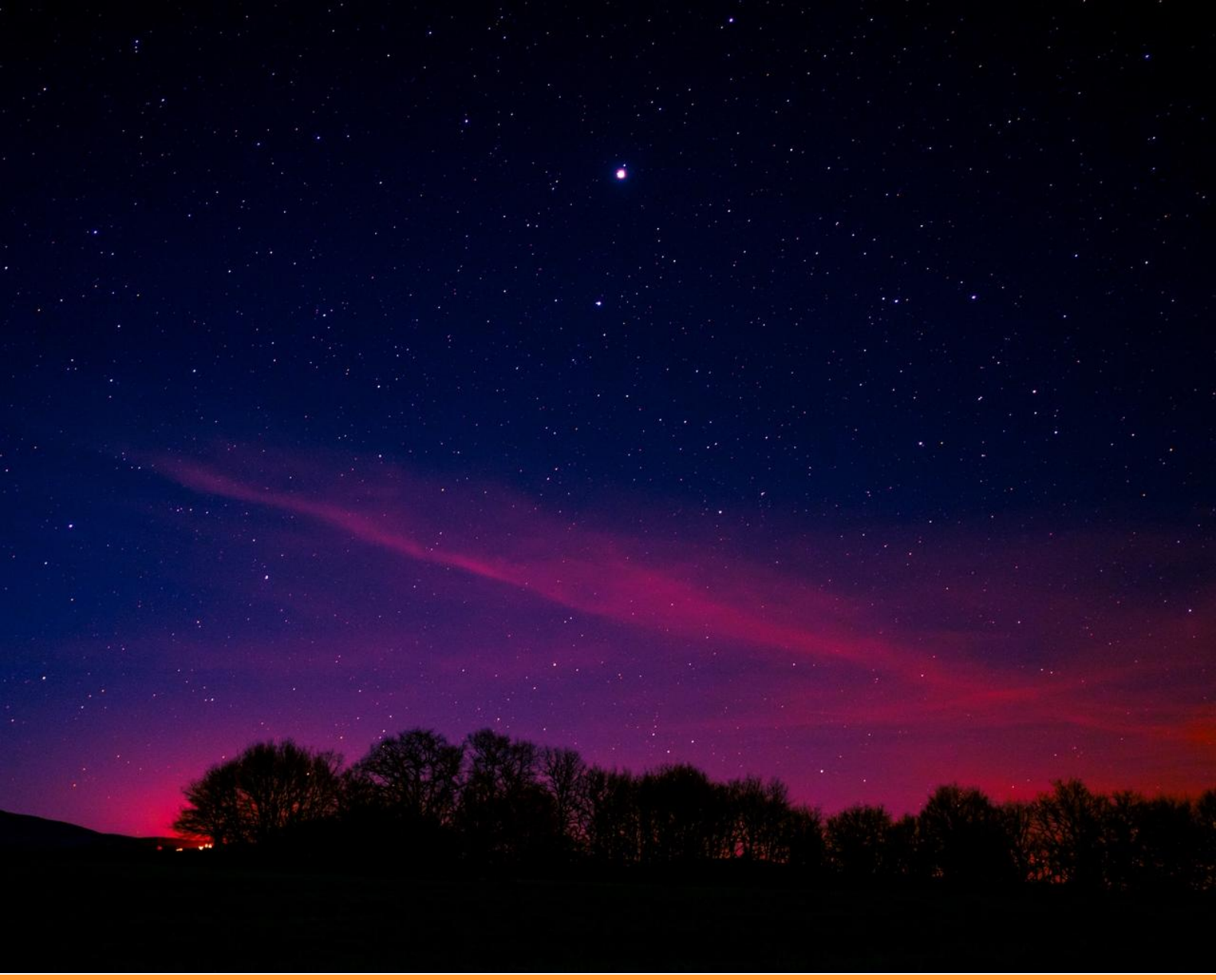


Practicum: Análisis y Gobierno de Datos ENSANUT 2018

Julio Cesar Velázquez Corona

Universidad Anáhuac Campus Norte

Enero - Mayo 2026

Asesora: Dra. María del Carmen Villar Patiño

Índice general

Capítulo 1 Preparación

1.1 Requisitos de Software

Un proyecto de ciencia de datos requiere diferentes tipos de software para llevarse a cabo y en el caso de nuestro proyecto, es necesario contar con los siguientes:

- Docker: es una plataforma para empaquetar y ejecutar aplicaciones en contenedores que llevan consigo las dependencias para que corran igual en distintas máquinas.
- Ubuntu 20.04.6 LTS: es un entorno Linux dentro de Windows que corre dentro de WSL (Windows Subsystem for Linux). Es básicamente el backend de Docker.
- OpenMetadata: es una plataforma unificada para el descubrimiento, la observabilidad y la gobernanza, impulsada por un repositorio central de metadatos, linaje detallado y una colaboración fluida del equipo.

1.1.1 Instalación de Docker

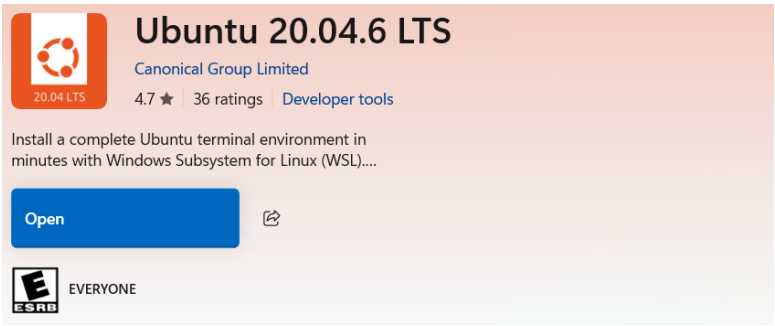
Accedemos a la página de Docker y damos click en `Download Docker Desktop`, luego aparecerá la opción de elegir el instalador que queremos dependiendo de nuestro SO. Ya con el instalador descargado, lo ejecutamos y seguimos los pasos que nos muestra. Cuando se indique, asegurarse de utilizar WLS 2 en lugar de Hyper-V. Por último, es necesario asignar como mínimo 6 GiB de memoria y 4 vCPUs. Para asegurarnos de que todo se instaló bien podemos correr en Powershell los siguientes comandos:

```
> docker --version
```

```
> docker compose version
```

1.1.2 Instalación de Ubuntu 20.04.6 LTS (Solo para Windows)

Puede descargarse directamente desde la Microsoft Store.



Ya instalado lo corremos y pedirá crear un usuario y una contraseña, luego entramos a Ubuntu e instalamos lo siguiente:

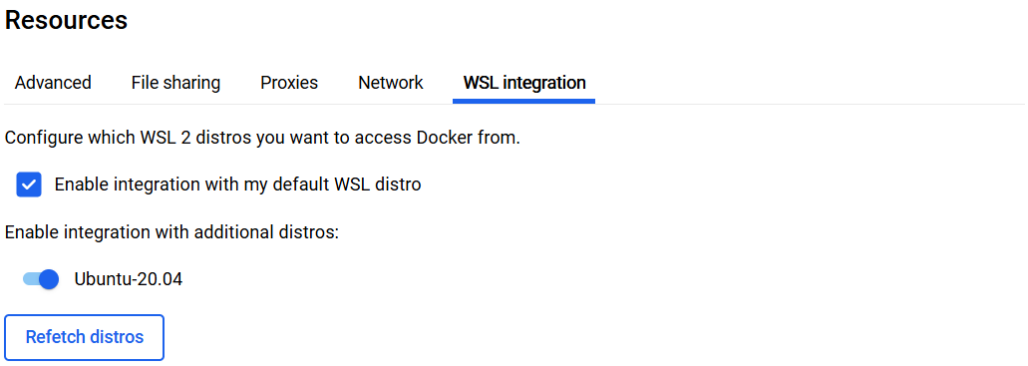
```
> sudo apt update
> sudo apt install python3-pip python3-venv
```

1.1.3 Instalación de OpenMetadata

En nuestro entorno Ubuntu en el caso de Windows o en la Docker CLI en el caso de Mac, colocar los siguientes comandos:

```
> mkdir openmetadata-docker && cd openmetadata-docker
> curl -sL -o docker-compose-postgres.yml https://github.com/open-metadata/
OpenMetadata/releases/download/1.11.3-release/docker-compose-postgres.yml
```

En el caso de Windows, entrar a Docker Desktop e ir a Settings > Resources > WSL integration y asegurarse de que todo esté activado de la siguiente manera:



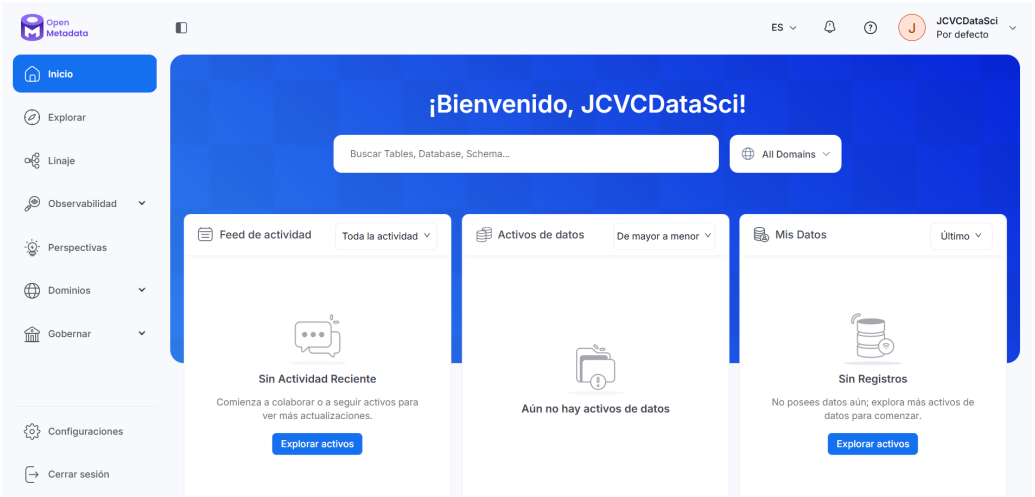
Colocamos el siguiente comando y al correrlo, reiniciamos la terminal:

```
> sudo usermod -aG docker $USER
```

Al volver a abrir la terminal corremos lo siguiente:

```
> docker compose -f docker-compose-postgres.yml up --detach
```

Al correr lo anterior se levanta el servicio de Docker Compose para OpenMetadata y podemos acceder al servicio desde `http://localhost:8585`. Ya dentro se nos pedirá crear una cuenta y ya podemos usar OpenMetadata.



Si posteriormente queremos volver a utilizar OpenMetadata hay que tener corriendo Docker Desktop en el fondo y correr los comandos que se presentarán a continuación:

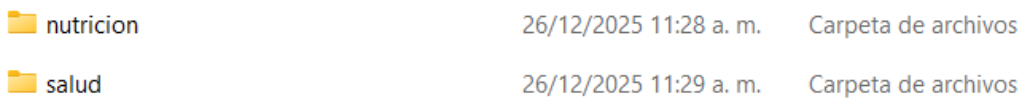
```
> cd openmetadata-docker/
> docker compose -f docker-compose-postgres.yml down
> docker compose -f docker-compose-postgres.yml up -d
```

Finalmente acceder nuevamente a http://localhost:8585.

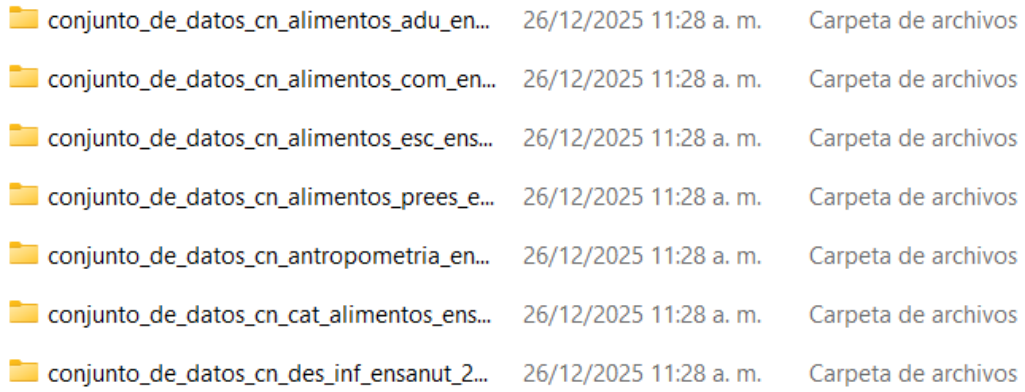
1.2 Desarrollo Previo de Automatización de Creación y Llenado de Tablas

El proceso que se debe llevar a cabo es el siguiente: **Automatizar la creación y llenado de las tablas, a una base de datos PostgreSQL y Oracle**. Se va a plantear cómo se tendría que realizar este proceso y de ahí se creará un diagrama de flujo para facilitar la reproducibilidad.

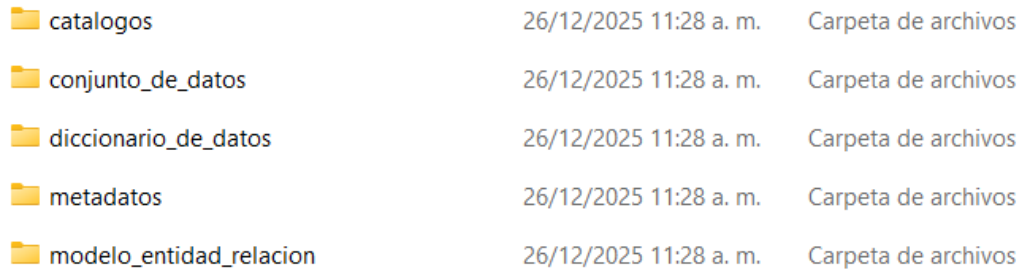
Nuestros datos se dividen en dos carpetas:



Cada una de esas carpetas tiene carpetas raíz con la siguiente estructura:



Cada carpeta raíz contiene lo siguiente:



La carpeta `diccionario_de_datos` contiene un archivo CSV con una descripción de cómo está estructurada la tabla que tiene los datos, la idea es que desde ahí se empiece la automatización de creación de tablas y por lo tanto nuestro primer paso es leer los diccionarios de datos.

1.2.1 Lectura de un Diccionarios de Datos

La información leida debe de almacenarse y asignarse para cada carpeta raíz, mi idea para eso es que se cree un programa que busque, almacene y mande esa información estructurada correctamente a un archivo JSON. De esa forma, las tablas que queremos crear automáticamente podrán usar ese archivo como referencia. Para convertir de un CSV a JSON con Python, encontré tres opciones:

- Usar los módulos csv y json: es la forma más básica, estandar y es útil para procesos pequeños y datasets pequeños.
- Usar pandas: esta librería de manipulación de datos es un buen candidato.
- Usar jsonlines: esta librería escribe JSON línea por línea y es bastante útil para datasets grandes.

Las opciones que tengo como tentativa a usar son los módulos o jsonlines. Cabe destacar que la ruta de donde se obtendrán los datos será la que se utiliza localmente en mi computadora, sin embargo, en la documentación se elegirá una ruta genérica. A continuación se presenta un ejemplo básico con jsonlines de lo que se busca hacer:

```
# Extraccion del primer diccionario de datos

import csv
import jsonlines
from pathlib import Path
import re

# Path del primer CSV de salud
csv_path = 'C:\\Personal\\practicum\\rawdata\\salud\\conjunto_de_datos_cs_ensanut_2018_csv
\\conjunto_de_datos_cs_act_fis_ado_ensanut_2018\\diccionario_de_datos
\\diccionario_datos_cs_act_fis_ado_ensanut_2018.csv'

# Path para que se guarde en la carpeta actual
out_path = out_path = Path(__file__).resolve().parent / 'output.jsonl'

# Nombre de la tabla
m = re.search(r'diccionario_datos_(.+?)_ensanut_2018', csv_path)
nombre = m.group(1) if m else None

# Generamos JSON
with open(csv_path, mode='r', newline='', encoding='utf-8-sig') as csvfile, \
    jsonlines.open(out_path, mode='w') as writer:
    for row in csv.DictReader(csvfile):
        writer.write(row)
```

Este programa funciona, se guarda un JSON con toda la información del diccionario de datos y tengo planeado que para tener más orden y facilitar una automatización, se cree un JSON para cada tabla, por lo tanto, **nos conviene utilizar los módulos en lugar de jsonlines.**

La estructura que se manejó en el JSON es demasiado plana y no conviene para que pueda ser utilizada por PostgreSQL u Oracle, se tiene que corregir eso. Vamos a enfocarnos por el momento en PostgreSQL, en ese software existe una función llamada `JSON_TABLE` la cual realiza una query en un JSON para poder crear una tabla a partir de esos datos. Se debe ajustar la estructura del JSON para que la función pueda trabajar bien con los datos.

```
# Extraccion de diccionario de datos con el primer conjunto de datos de la carpeta salud
```

```
import csv
import json
import re
from pathlib import Path

# Ruta donde se ubica el archivo .csv
csv_path = 'C:\\Personal\\practicum\\rawdata\\salud\\conjunto_de_datos_cs_ensanut_2018_csv
\\conjunto_de_datos_cs_act_fis_ado_ensanut_2018\\diccionario_de_datos\\
diccionario_datos_cs_act_fis_ado_ensanut_2018.csv'

# Extraccion del nombre de la tabla
m = re.search(r'diccionario_datos_(.+?)_ensanut_2018', csv_path)
nombre = m.group(1) if m else 'tabla'

# Para que el JSON se guarde en la misma carpeta donde esta este programa
out_path = Path(__file__).resolve().parent / f'{nombre}.json'

# Funcion anonima que convierte numeros de caracter a numeros int
to_int = lambda x: int(x) if (x := (x or '').strip()).isdigit() else None

# Funcion anonima que convierte los '' a null
to_none = lambda x: (x := (x or '').strip()) or None

# Lectura del CSV
with open(csv_path, 'r', encoding='utf-8-sig', newline='') as f:
    campos = []
    for row in csv.DictReader(f):
        # Quitamos un espacio extra en los encabezados si es que hay
        row = {k.strip(): v for k, v in row.items()}
        # Extraemos informacion con nuestras funciones anonimas
        campos.append({
            'nombre_campo': to_none(row.get('nombre_campo')),
            'longitud': to_int(row.get('longitud')),
            'tipo': to_none(row.get('tipo')),
            'nemonico': to_none(row.get('nemónico')) or to_none(row.get('nemonico')),
            'catalogo': to_none(row.get('catálogo')) or to_none(row.get('catalogo')),
            'rango_claves': to_none(row.get('rango_claves')),
        })

# Creamos un 'contenedor' tipo JSON donde se guardara la informacion
payload = {'nombre_tabla': nombre, 'campos': campos}

# Creamos JSON
with open(out_path, 'w', encoding='utf-8') as f:
    # Cuidamos por si hay caracteres mexicanos
    json.dump(payload, f, ensure_ascii=False, indent=2)
```

El programa funcionó bien y ya tenemos listo el algoritmo que se utilizará para obtener la estructura de las demás tablas.

1.2.2 Automatización de Lectura de Diccionarios de Datos

Lo que tenemos que hacer ahora es darle retoques al programa que hicimos anteriormente para que almacene la estructura de los demás diccionarios de datos de la carpeta llamada salud. Mi idea para hacerlo fue la siguiente:

- Obtener la ruta donde se encuentran nuestros datos.
- Crear nuestras funciones anónimas.
- Sacamos los directorios donde estan todos los diccionarios para después alimentarlo al programa.
- Extraemos los nombres otra vez para asignarlos a cada JSON respectivamente.
- Indicamos que queremos que la carpeta se cree en el directorio donde está el programa.
- Recorremos nuestra variable con los directorios, sacamos los CSV, los leemos.
- Creamos un payload y metemos la información ahí, todo dentro de un ciclo.

```
# Extracción de todos diccionario de datos de la carpeta salud
import csv
import json
import re
from pathlib import Path

# Ruta donde se ubican las carpetas raíz
root_path =
↪  Path('C:\\Personal\\practicum\\rawdata\\salud\\conjunto_de_datos_cs_ensanut_2018_csv')

# Función anónima que convierte números de caracter a números int
to_int = lambda x: int(x) if (x := (x or '').strip()).isdigit() else None

# Función anónima que convierte los '' a null
to_none = lambda x: (x := (x or '').strip()) or None

# Sacar directorios con un CSV que empiece con 'diccionario'
dirs_with_csv = sorted({f.parent for f in root_path.rglob('diccionario*.csv')})
for d in dirs_with_csv:
    print(d)

# Extraemos el nombre de la tabla desde el nombre del archivo
nombres = []
pat = re.compile(r'diccionario_datos_(.+?)_ensanut_2018')
for d in dirs_with_csv:
    f = sorted(d.glob('diccionario*.csv'))[0]
    m = pat.search(f.stem)
    nombres.append(m.group(1))

# Indicamos que queremos que la carpeta con los jsons se creen aqui
out_dir = Path(__file__).resolve().parent / 'jsons'
# Pero se crea si no existe la carpeta
out_dir.mkdir(exist_ok=True)
```

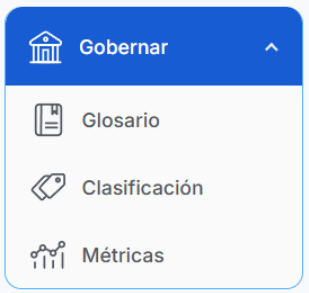
```
# Recorremos nuestra variable con los directorios
for d, nombre in zip(dirs_with_csv, nombres):
    # Guardamos los CSV por orden alfabetico para mantener orden
    in_csv = sorted(d.glob('diccionario*.csv'))[0]
    # Leemos cada CSV
    campos = []
    with open(in_csv, 'r', encoding='utf-8-sig', newline='') as f:
        for row in csv.DictReader(f):
            # Quitamos un espacio extra en los encabezados si es que hay
            row = {k.strip(): v for k, v in row.items()}
            # Extraemos información de cada CSV con nuestras funciones anónimas
            campos.append({
                'nombre_campo': to_none(row.get('nombre_campo')),
                'longitud': to_int(row.get('longitud')),
                'tipo': to_none(row.get('tipo')),
                'nemonico': to_none(row.get('nemónico')) or to_none(row.get('nemonico')),
                'catalogo': to_none(row.get('catálogo')) or to_none(row.get('catalogo')),
                'rango_claves': to_none(row.get('rango_claves')),
            })
    # Contenedor de como debe estar la estructura
    payload = {'nombre_tabla': nombre, 'campos': campos}
    # Ruta de salida del JSON y asignamos a cada uno el nombre indicado
    out_path = out_dir / f'{nombre}.json'
    # Creamos JSON
    with open(out_path, 'w', encoding='utf-8') as f:
        json.dump(payload, f, ensure_ascii=False, indent=2)
    print('JSON creado:', out_path)
```

Ahora, nos queda hacer interacción con PostgreSQL.

1.2.3 Creación de Tablas en PostgreSQL a partir de Archivos JSON

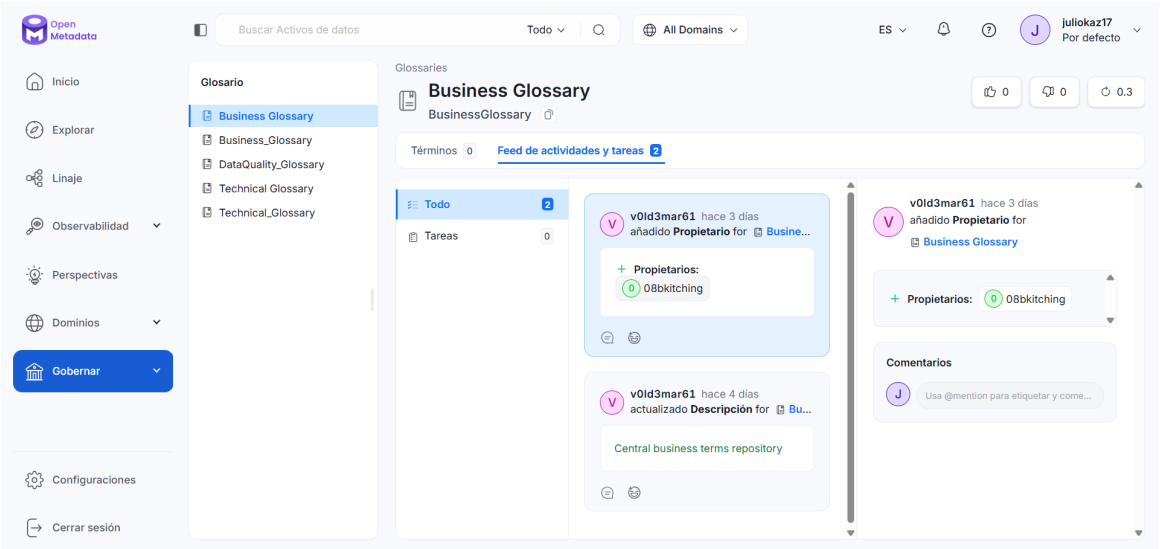
1.3 Gobernanza de Datos con OpenMetadata

El gobierno de datos es el conjunto de decisiones, reglas y prácticas con las que definimos y hacemos cumplir cómo se crean, se describen, se usan, se comparten, se protegen y se mantienen los datos dentro de una organización, para que sean confiables, seguros, consistentes y útiles para el negocio. En OpenMetadata tenemos tres artefactos que nos permiten llevar a cabo lo anterior y se encuentran en la sidebar del menú principal.



1.3.1 Glosario

En `Glosario` es donde se aloja el repositorio de los conceptos de los datos, eso quiere decir que ayuda a definir qué significa cada dato y también conectarlos con activos como tablas o columnas. En la sandbox nos encontramos con lo siguiente:



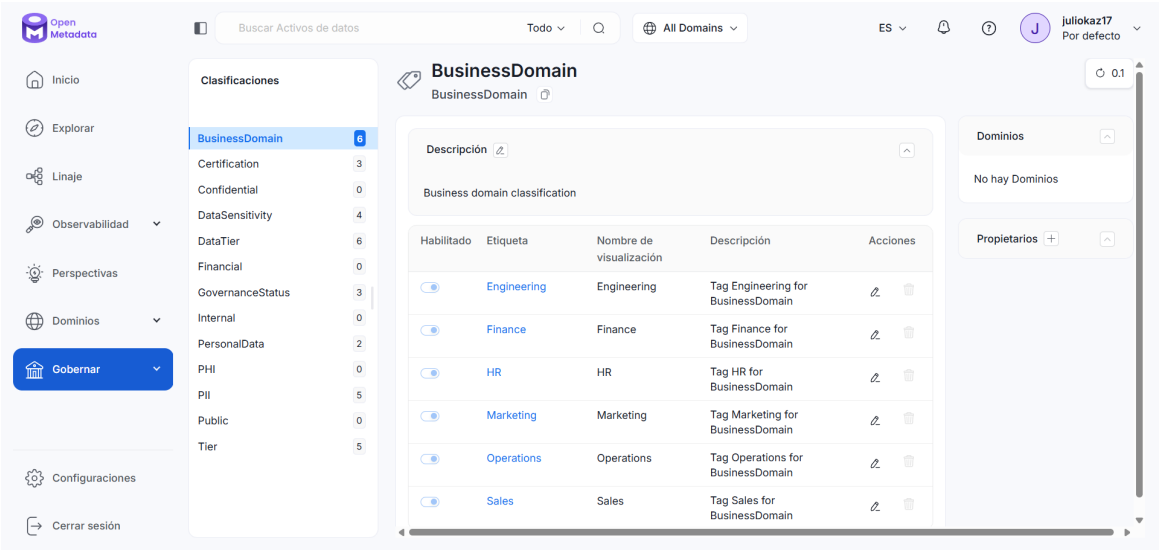
En el panel izquierdo podemos ver que hay diferentes glosarios y en este caso tenemos seleccionado el `Business Glosary`. A la derecha aparece nuestro título y vemos que en la parte de abajo dice `Términos 0`, eso quiere decir que aún no tenemos definiciones como tal de los datos. En la otra pestaña llamada `Feed de actividades y tareas` hay dos eventos registrados:

- El usuario `v0ld3mar61` añadió ”Propietario” a nuestro glosario y el propietario asignado es `08bkitching`.
- Ese mismo usuario actualizó la descripción del glosario y la dejó como: *Central business terms repository.*

En la parte de la derecha aparece el cambio seleccionado con una sección de comentarios por si se quiere hablar del cambio.

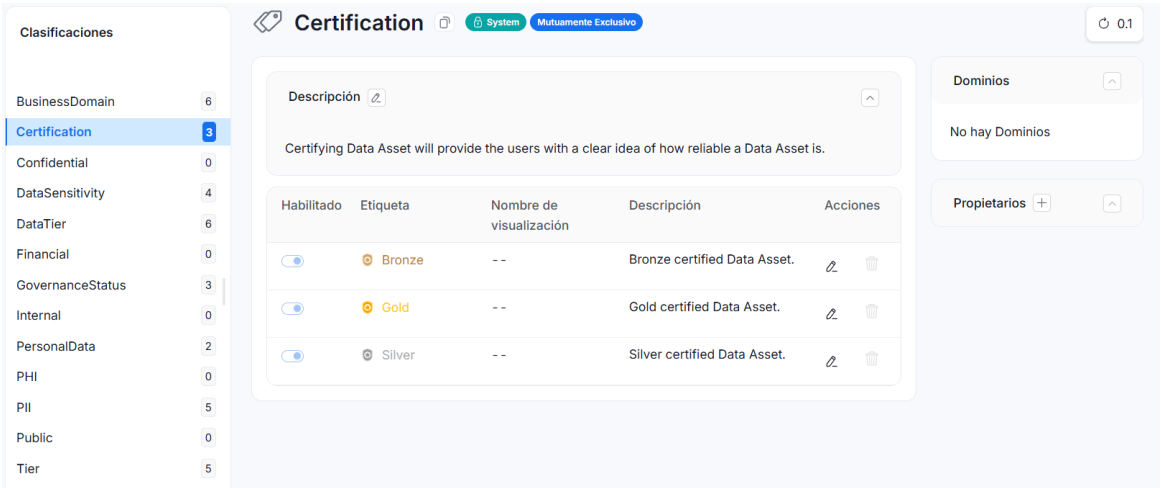
1.3.2 Clasificación

El siguiente artefacto es `Clasificación` y sirve para definir y administrar los datos que tenemos. No definimos el significado como tal, más bien se define qué tipo de dato es, cómo debe tratarse, si es PII o no PII, etc. La sandbox lo muestra de la siguiente forma:

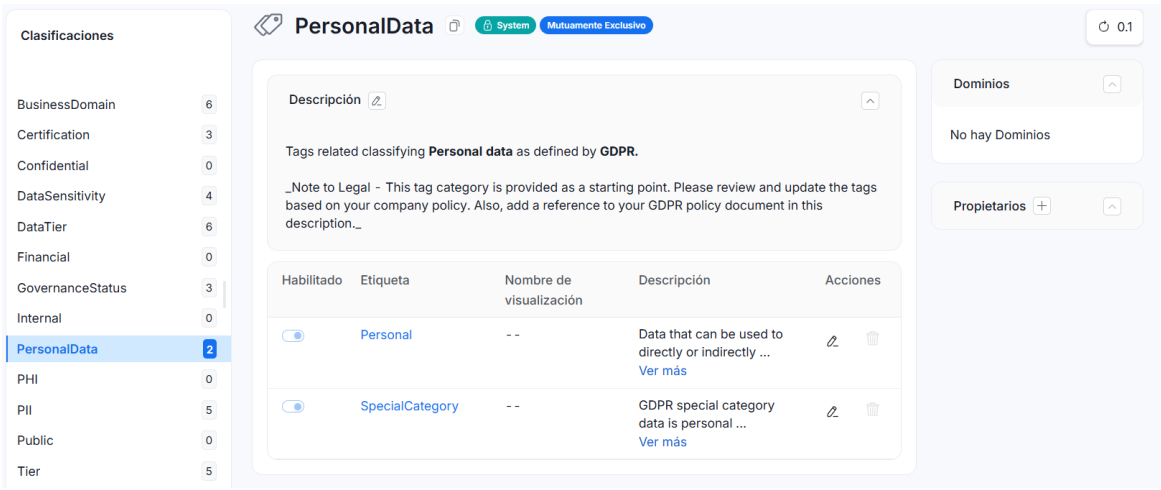


En el sidebar debajo de `Clasificaciones` es donde se encuentra lo importante: son las etiquetas en las que pueden entrar nuestros datos. Algunas son creadas por default en el sistema y otras son creadas por usuarios. Para nuestro interes vale la pena describir las etiquetas creadas por el sistema:

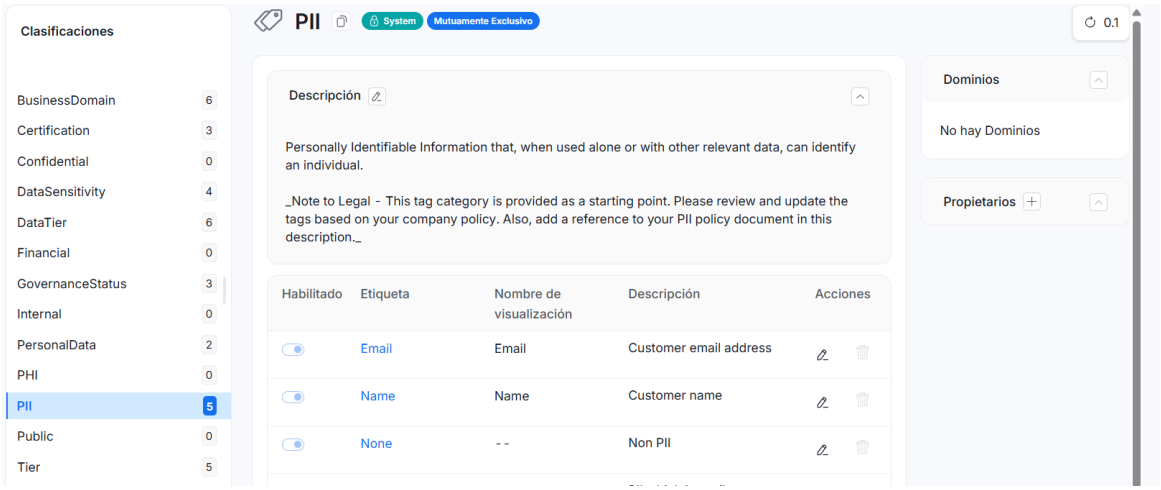
- Certification: aquí se nos muestra si un dato es confiable o no.



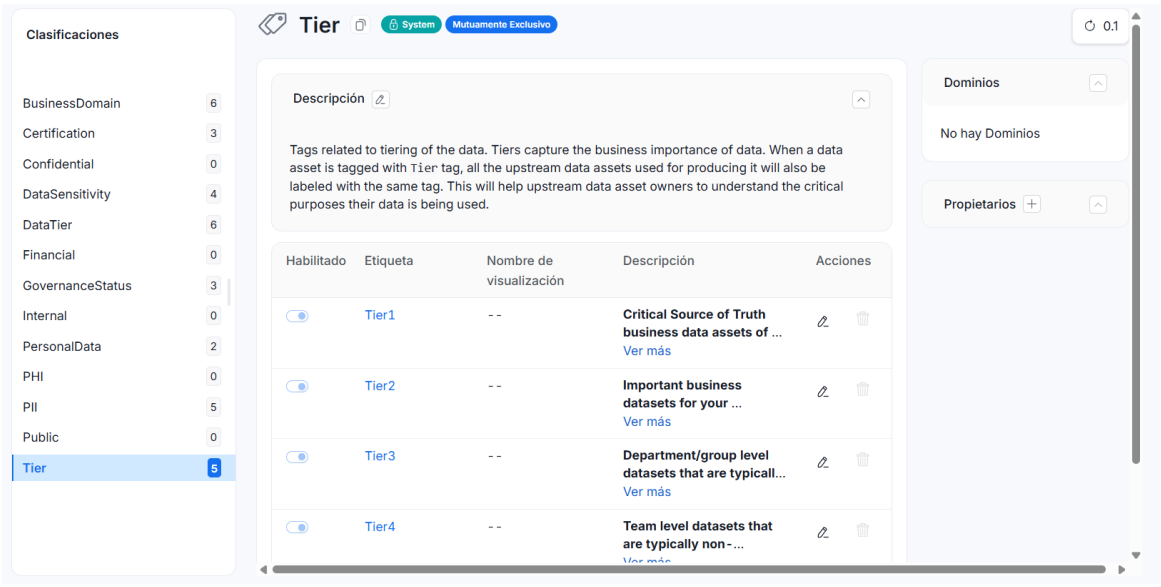
- PersonalData: aquí se marcan datos que son personales.



- PII: aquí se etiqueta información personal que es capaz de identificar a una persona.



- Tier: en esta etiqueta se clasifican nuestros activos de datos en capas que van del `Tier1` al `Tier5` y mientras más alta es la capa, más importante es el dato.

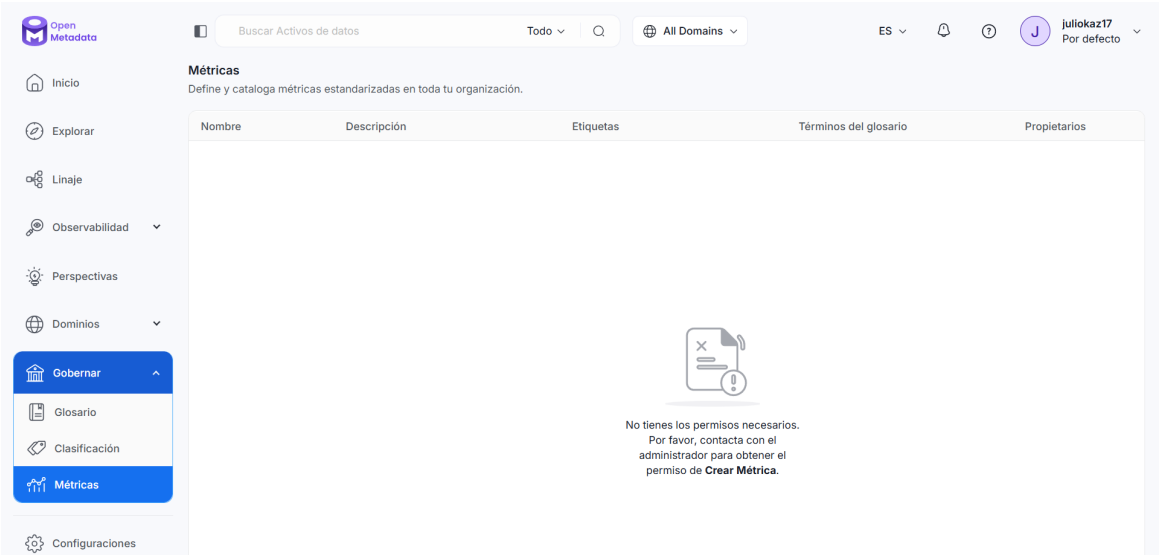


1.3.3 Métricas

Por último, en `métricas` es donde se define explícitamente con qué lógica opera una métrica definida. Por ejemplo: si tenemos un KPI llamado `monthly_arpu` (ingreso promedio por usuario al mes), podemos definir lo siguiente:

- Nombre o ID.
- Descripción.
- Tipo o unidad.
- Granularidad.
- Responsable.
- Glosarios vinculados.
- Tags.

En la sandbox desafortunadamente no podemos observar cómo es que se utiliza este artefacto porque no hay métricas definidas y aparte aparezco registrado como un usuario que no cuenta con los permisos necesarios para crear una.



1.4 OpenMetadata y otros Softwares

Hasta ahora, hemos descrito bastante OpenMetadata. Sin embargo, es de nuestro interes estar conscientes de que existe una competencia y que OpenMetadata no es la única opción. Los dos softwares más relevantes para lo que queremos y a mi parecer, los más ”sonados” son Amazon Datazone y Oracle Cloud Infrastructure Data Catalog. A continuación se mostrará un cuadro comparativo de interes:

Software	**OpenMetadata**	**Amazon Datazone**	**OCI Data Catalog**
Instalación	Sí se instala y se mantiene con base de datos, plataforma y con un componente que hace las ingestas	No se instala porque ya viene como servicio pero sí hay que configurar acceso	No se instala pero hay que crear una instancia y configurar permisos
Experiencia recomendada	No mucha, solo en operación de plataformas y un poco en gobierno de datos	Más o menos, se requiere saber manejar el servicio IAM y un poco más de gobierno de datos	No mucha, solo estar envuelto en el entorno de Oracle
Ventajas	Más flexibilidad y control, podemos usar las herramientas que queramos	Está administrado por AWS y facilita la entrega de los datos	Está administrado por Oracle y brilla mucho en la cosecha (harvesting) de metadatos
Desventajas	Hay que mantenerlo constantemente y hay que hacer ajustes en el caso de actualizar algo	Disponibilidad solo en ciertas regiones y algunas cuotas en los servicios (pésimo)	Tiende a tener errores (bugs) según los known issues y también cuenta con algunas limitaciones

Glosario

AWS IAM Es un servicio que permite controlar de forma segura quién tiene acceso a qué recursos de AWS.

DDL Son las sentencias SQL para definir la estructura de la base de datos. O sea, crear, modificar y borrar objetos.

FKs Las llaves foráneas son restricciones que hacen que un campo en una tabla apunte a una fila existente en otra tabla, para mantener integridad referencial.

Harvesting Es el proceso de extraer, traer y actualizar metadatos desde las fuentes de datos hacia un catálogo o herramienta de gobierno.

JSON Es un tipo de archivo que utiliza la JavaScript Object Notation para almacenar información.

Metadato Información que describe a un dato para que se pueda entender.

Bibliografía

[1] Oxycon Blog. *Leveraging JSON in PostgreSQL 17*. https://blog.oxyconit.com/leveraging-json-in-postgresql-17, 2026.

[2] crunchydata. *Convert JSON into Columns and Rows*. https://www.crunchydata.com/blog/easily-convert-json-into-columns-and-rows-with/json_table, 2026.

[3] GeeksforGeeks. *Convert CSV to JSON*. https://www.geeksforgeeks.org/python/convert-csv-to-json-using-python, 2026.

[4] OpenMetadata. *OpenMetadata Documentation*. https://docs.open-metadata.org/latest, 2025.

[5] C. V. Patiño. *Apuntes de Machine Learning*. Universidad Anáhuac México Campus Norte, 2025.

[6] PostgreSQL. *JSON Functions and Operators*. https://www.postgresql.org/docs/current/functions-json.html, 2026.

[7] PostgreSQL. *Populating a Database*. https://www.postgresql.org/docs/current/populate.html, 2025.

[8] Python. *pathlib, Object-oriented filesystem paths*. https://docs.python.org/3/library/pathlib.html, 2026.