DATA LAKE DOCUMENTATION

Version	Creation Date	Last Modified	Author	Description
1.0.0	2025-10-20	2025-10-20	Diego Oliveira de Araújo	Data Lake pipeline using DLT, Python, PostgreSQL, MongoDB and API following the Bronze–Silver–Gold model to ingest, transform, and deliver clean, analytics-ready datasets for business insights.

Berlin, October 20th, 2025

1. DESCRIPTION

The objective of this project is to design and implement a modern, modular Data Lake architecture using open-source tools and Python to enable scalable data ingestion, transformation, and consumption. All the data used here is fictitious and was generated randomly, but in a logical way to support understanding and the practical application of data engineering in real-world scenarios.

It aims to:

- Ingest data from multiple sources (e.g., MongoDB, PostgreSQL, APIs)
- Store raw data in a structured format in a local PostgreSQL database acting as a structured data lake, with support for schema evolution
- Organize the data in three layers:

  Bronze (raw data)

  Silver (cleaned and standardized data)

  Gold (business-ready data for analysis)
- Perform ETL orchestration using Python + DLT (Data Load Tool) pipelines
- Maintain a log file to track ingestion status, errors, timestamps, and pipeline activity, enabling better observability and debugging
- Enable a simple web front-end (e.g., via Streamlit hosted on Replit) to allow users to interact with the data (Sales Registration System).
- Support CSV exports and potential integration with BI tools

The overall goal is to build a flexible, maintainable, and extendable data processing pipeline that supports both batch ingestion and analytical consumption, while applying best practices in data engineering, such as modular pipelines, layer separation, and metadata tracking.

2. DEVELOPMENT

2.1 Frontend

The frontend was developed using Streamlit, a lightweight and interactive Python framework, and deployed on Replit to make the application accessible via the web.

Key aspects of the frontend:

- Built with Streamlit for fast UI prototyping using only Python
- User-friendly interface to:

Select customers and products

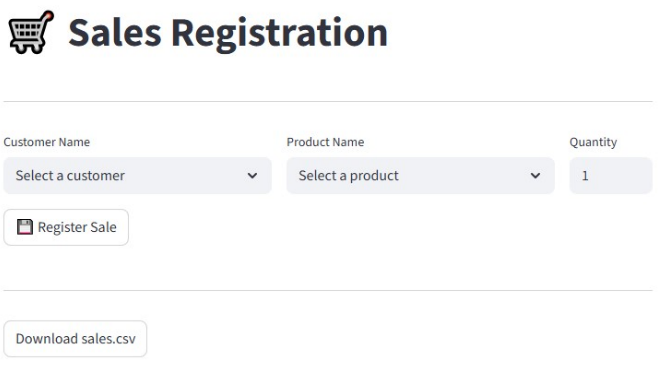
Input sales quantity

Automatically calculate total price and timestamp

Submit and register sales directly into the PostgreSQL database (Render PaaS)

- Dropdown menus populated dynamically from cleaned CSV files (customer.csv and product.csv)

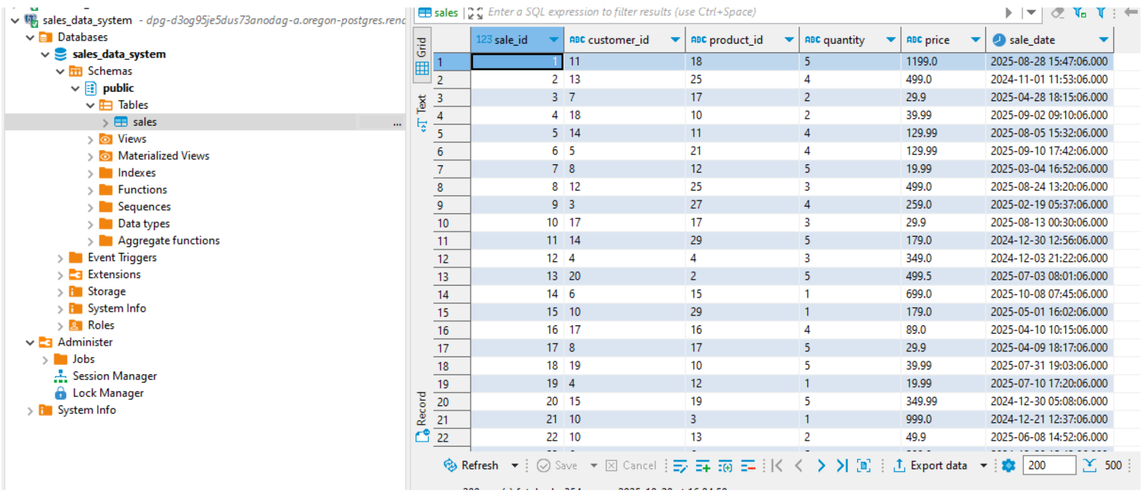
- Form validation to ensure that both customer and product are selected before submission
- Data persistence handled via backend PostgreSQL using SQLAlchemy
- CSV export button for users to download the complete sales data
- Automatic page refresh (st.rerun()) after successful sale registration for a smooth UX



img1: Frontend development

2.1.1 Sale registration database

The sales table is part of the public schema in the sales_data_system database, hosted on Render (as shown in your architecture).
This table records every sale transaction made via the system. It links customers and products together through foreign-like keys (customer_id, product_id) and provides details such as sale quantity, unit price, and timestamp of the sale.



	sale_id	customer_id	product_id	quantity	price	sale_date
1	1	11	18	5	1199.0	2025-08-28 15:47:06.000
2	2	13	25	4	499.0	2024-11-01 11:53:06.000
3	3	7	17	2	29.9	2025-04-28 18:15:06.000
4	4	18	10	2	39.99	2025-09-02 09:10:06.000
5	5	14	11	4	129.99	2025-08-05 15:32:06.000
6	6	5	21	4	129.99	2025-09-10 17:42:06.000
7	7	8	12	5	19.99	2025-03-04 16:52:06.000
8	8	12	25	3	499.0	2025-08-24 13:20:06.000
9	9	3	27	4	259.0	2025-02-19 05:37:06.000
10	10	17	17	3	29.9	2025-08-13 00:30:06.000
11	11	14	29	5	179.0	2024-12-30 12:56:06.000
12	12	4	4	3	349.0	2024-12-03 21:22:06.000
13	13	20	2	5	499.5	2025-07-03 08:01:06.000
14	14	6	15	1	699.0	2025-10-08 07:45:06.000
15	15	10	29	1	179.0	2025-05-01 16:02:06.000
16	16	17	16	4	89.0	2025-04-10 10:15:06.000
17	17	8	17	5	29.9	2025-04-09 18:17:06.000
18	18	19	10	5	39.99	2025-07-31 19:03:06.000
19	19	4	12	1	19.99	2025-07-10 17:20:06.000
20	20	15	19	5	349.99	2024-12-30 05:08:06.000
21	21	10	3	1	999.0	2024-12-21 12:37:06.000
22	22	10	13	2	49.9	2025-06-08 14:52:06.000

img2: Database generated from the sales registration

Field	Data Type	Description
sale_id	Integer	Primary key / unique identifier for each sale transaction
customer_id	Integer	Foreign key reference to the customer who made the purchase
product_id	Integer	Foreign key reference to the product that was purchased
quantity	Integer	Number of units sold in the transaction
price	Decimal/Float	Price per unit of the product at the time of sale
sale_date	Timestamp	Date and time when the sale was recorded (in YYYY-MM-DD HH:MM:SS format)

2.2 API Simutaion and Automation via GitHub Actions
In this project, two JSON files (customer_api.json and product_api.json) are used to simulate external API responses for customer and product data. These files are stored directly in the GitHub repository and treated as structured data sources, mimicking typical RESTful API responses in JSON format.

Key Components:

- Customer_api.json and product_api.json

These files represent mock REST API endpoints.

Each JSON contains a structured list of customers and products, respectively.

Format resembles a real API response, making them suitable for ETL ingestion as if pulling from a live service.

- GitHub Actions Workflow

A workflow is scheduled to run every 2 minutes using GitHub Actions' cron job.

There are 2 workflows: customer.yaml and product.yaml

- Automated ETL-like Process

This setup acts as an automated, serverless ETL using GitHub infrastructure:

No external API is called — JSON simulates the response

No local server needed — everything runs on GitHub

Repeatable and event-driven every 2 minutes

2.2.1 Customer Data (customer_api.json)
This JSON file represents a list of customer records retrieved (or simulated) from an external system, such as a CRM or user management platform. It contains rich user information that can be used for personalized analytics and customer segmentation.

Fields included:

Field	Description
customer_ID	Unique identifier for the customer (e.g., "2")
customer	Full name of the customer (e.g., "Anna Müller”)
Email	Contact email address (e.g., "anna.mueller@example.com”)
gender	Gender of the customer (e.g. “Male”)
birth_date	Date of birth in YYYY-MM-DD format
region	City or administrative region (e.g., "Berlin")
neighborhood	Local neighborhood within the region (e.g., "Kreuzberg”)

2.2.2 Product Data (product_api.json)

This file simulates a product catalog from a product information management (PIM) system or e-commerce backend. It contains various electronic and tech products with relevant specifications.

Fields included:

Field	Description
product_ID	Unique product identifier (e.g., "10")
product	Product name (e.g., "Smartphone")
sku	Stock keeping unit — a unique internal code (e.g. "WH-1001”)
category	General category (e.g., Electronics, Home)
sub_category	More specific grouping (e.g., Audio, Wearables)
brand	Brand name (e.g., Sony, Apple, DJI)
model_year	Year of product release or model (e.g., 2024)

3. MONGODB (STOCK DATA)

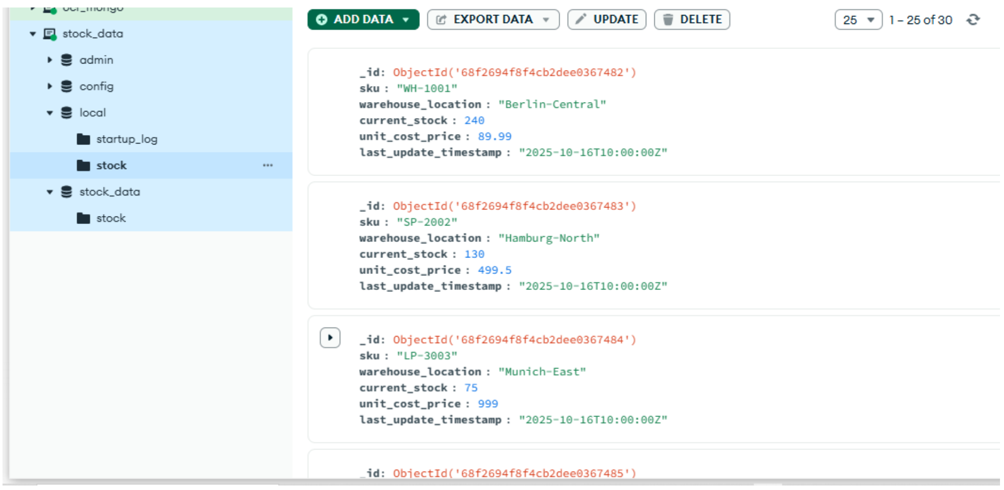
The stock data stored in the local MongoDB database captures detailed information about the current inventory levels of various products across different warehouse locations. Each document in the collection typically includes key attributes such as the product SKU (Stock Keeping Unit), the quantity of units available (current_stock), the specific warehouse location, the unit cost price, and a timestamp indicating the last update.

This data is semi-structured, leveraging MongoDB’s flexible schema to accommodate evolving inventory details without strict schema constraints. Such flexibility allows for real-time updates and seamless integration of additional metadata as needed.

Maintaining this stock data is crucial for accurately monitoring product availability, optimizing supply chain operations, and enabling informed, timely decisions for inventory replenishment and demand forecasting.

Fields included:

Field	**Description**
_id	A unique identifier automatically generated by MongoDB for each document (e.g., "68f2694f8f4cb2dee0367482")
sku	The Stock Keeping Unit, a unique code that identifies a specific product (e.g., "WH-1001")
warehouse_location	The physical location or warehouse where the stock is stored (e.g., "Berlin-Central")
current_stock	The quantity of the product currently available in the warehouse (e.g., "240”)
unit_cost_price	The cost price per unit of the product (e.g., "89.99")
last_update_timestamp	The date and time when the stock information was last updated, in ISO 8601 format (e.g., "2025-10-16T10:00:00Z")



```
_id: ObjectId('68f2694f8f4cb2dee0367482')
sku : "WH-1001"
warehouse_location : "Berlin-Central"
current_stock : 240
unit_cost_price : 89.99
last_update_timestamp : "2025-10-16T10:00:00Z"

_id: ObjectId('68f2694f8f4cb2dee0367483')
sku : "SP-2002"
warehouse_location : "Hamburg-North"
current_stock : 130
unit_cost_price : 499.5
last_update_timestamp : "2025-10-16T10:00:00Z"

_id: ObjectId('68f2694f8f4cb2dee0367484')
sku : "LP-3003"
warehouse_location : "Munich-East"
current_stock : 75
unit_cost_price : 999
last_update_timestamp : "2025-10-16T10:00:00Z"

_id: ObjectId('68f2694f8f4cb2dee0367485')
```

img3: Mongodb stock database

4. ETL PIPELINE

4.1 Bronze Layer

Captures raw data directly from multiple sources, including:

Cloud PostgreSQL (Sales system)

GitHub APIs (customer_api.json, product_api.json)

MongoDB (stock collection)

The data is stored without transformations in a local PostgreSQL instance under the bronze schema.

This layer ensures full traceability and preserves the original structure of each data source, serving as the foundation for downstream cleaning and enrichment in the Silver layer.

4.2 Silver Layer

It transforms and cleanses the raw data from the bronze schema, preparing it for reliable analysis and reporting. It reads data directly from the bronze tables in the local PostgreSQL database.

Transformations Applied:

Table	**Cleaning/Transformation**
customers	Standardizes region capitalization, converts birth_date to datetime, removes duplicates.
products	Renames columns (e.g., sk_u → sku), standardizes category capitalization.
sales	Ensures correct data types for prices and IDs, drops technical columns.
stock	Removes MongoDB _id and DLT metadata columns to normalize the structure.

The pipeline loads the transformed data into the silver schema of the local PostgreSQL.

4.3 Gold Layer

It delivers high-value, business-ready datasets built upon the clean and transformed data in the Silver Layer. It applies advanced logic, aggregations, and joins to support analytics, dashboards, and data-driven decision-making.

Table	**Description**
unsold_products	Identifies products that have never been sold by performing a RIGHT JOIN between sales and products. Helps discover inactive or surplus inventory.
stock	Calculates available stock by subtracting total quantity sold from current warehouse stock, providing an accurate picture of inventory.
sales	Enriched fact table that combines sales, customer, and product data into a single, analytics-ready dataset. Includes calculated fields like amount = quantity × price.

4.4 Automation, Resilience & Logging

The entire ETL pipeline – from data ingestion to transformation and load – was developed in Python using DLT.
It includes:

- Automatic logging of each execution stage (Bronze, Silver, Gold).
- Retry mechanism (up to 3 attempts) for each step in case of:

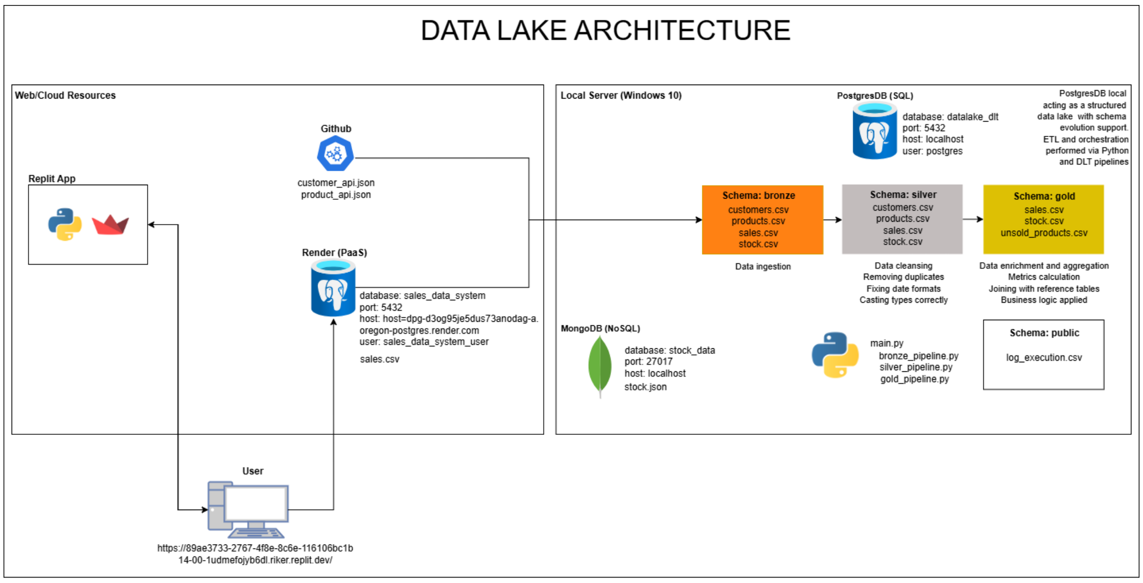
Internet connection failure,

API downtime or inaccessibility,

Any temporary processing issue.

- Log data is stored both locally and in the public schema of the local PostgreSQL database, ensuring visibility, traceability, and auditability of the pipeline runs.

Field	Description
product_id	Primary key of the product (e.g., "1”)
sku	Stock Keeping Unit, unique product code (e.g., "WH-1001”)
product	Name of the product (e.g., "Wireless Headphones”)
category	Main category of the product (e.g., "Electronics”)
brand	Brand or manufacturer of the product (e.g., "Sony”)
model_year	Year of the product model (e.g., "2023”)



img4: Data Lake Architecture

5. RESOURCES

5.1 Endpoints

customer_api: **https://raw.githubusercontent.com/diegooliveiradearaujo/api_json_dlt/refs/heads/main/json/customer_api.json**"

product_api: **https://raw.githubusercontent.com/diegooliveiradearaujo/api_json_dlt/refs/heads/main/json/product_api.json**

5.2 Databases

SGBD	Database	Username	Password	Host	Port
PostgreSQL (Render PaaS)	sales_data_system	sales_data_system_user	VUBLVHWgTvFbBrhmIfEu0ktSkYm1Vvte	dpg-d3og95je5dus73anodag-a.oregon-postgres.render.com	5432
MongoDB	stock_data	xxxx	xxxx	localhost	27017
PostgreSQL (local)	datalake_dlt	postgres	admin	localhost	5432

6. CONCLUSION

By implementing an ETL process, we refine data and generate actionable insights to enhance the business. For example, we can answer questions like: What is the company's revenue? Which region purchases the most? Which products, brands, or categories are top sellers? Which products remain unsold? How do age and gender correlate with purchase behavior? We can also identify strategies to boost sales of low-performing products by aligning with customer interests and needs. Overall, this approach optimizes multiple areas - sales, marketing, logistics, and more. Leveraging Modern Data Stack tools like DLT, integrated with other technologies, enables high-value data-driven decision making.