Conestoga College

School of Applied Computer Science & Information Technology

SENG8081 - Case Studies

Big Data Solution Architecture

Section 1

**Online Retail Sales Analysis**

A person using a computer to buy products

AI-generated content may be incorrect.

**Team 5**

**Team Members:**

Jyoti Maske

Hiral Parekh

Yash Khamar

Neel Patel

**Abstract**

This project focuses on analyzing online retail sales data by integrating historical sales data from a Kaggle dataset and enriching product information using an external API. The goal is to create a reliable, well-structured data pipeline and storage system to support downstream analytics such as customer behavior analysis, product trends, and sales forecasting. Python scripts were developed for data collection, cleaning, transformation, enrichment, and loading into a SQL Server database.

**For code and team contribution please refer to the Git Hub**

**Git Hub Repository**

<https://github.com/SENG8081/SENG8081-S25-Team5>

Table of Contents

[**Introduction 4**](#_Toc201452955)

[**System Overview 4**](#_Toc201452956)

[**Data Research and Integration 4**](#_Toc201452957)

[**Types of data: 4**](#_Toc201452958)

[**Data Sources 5**](#_Toc201452959)

[**Data Collection 5**](#_Toc201452960)

[**Data Storage and Maintenance 11**](#_Toc201452961)

[**Database: SQL Server (Sales\_Analysis) 11**](#_Toc201452962)

[**Data Loading 13**](#_Toc201452963)

[**Storage Needs 15**](#_Toc201452964)

[**Storage Solutions 15**](#_Toc201452965)

[**Data Retention Policies 16**](#_Toc201452966)

[**Data Backup and Disaster Recovery 16**](#_Toc201452967)

[**Data Quality 17**](#_Toc201452968)

[**Data Cleaning Methods 17**](#_Toc201452969)

[**Data Exploration 17**](#_Toc201452970)

[**Handling Missing Values 17**](#_Toc201452971)

[**Data Transformation 17**](#_Toc201452972)

[**Data Validation and Verification 17**](#_Toc201452973)

[**Validation Rules 17**](#_Toc201452974)

[**Cross-Validation 18**](#_Toc201452975)

[**Data Accuracy Assurance 18**](#_Toc201452976)

[**Error Detection 18**](#_Toc201452977)

[**Data Cleaning Tools 18**](#_Toc201452978)

[**Data Documentation and Metadata Management 18**](#_Toc201452979)

[**Metadata Creation 18**](#_Toc201452980)

[**Data Dictionary 18**](#_Toc201452981)

[**Project Timeline: 19**](#_Toc201452982)

[**References 20**](#_Toc201452983)

# **Introduction**

In modern e-commerce, understanding customer purchases, product trends, and pricing behavior is critical to gaining a competitive edge. This project builds a robust data pipeline for retail sales analysis by using structured historical data and enriched metadata for each product (e.g., category, brand, rating, image).

The Python pipeline performs end-to-end normalization, enrichment, and loading into a SQL Server database. This enables high-quality analysis, visualizations, and reporting in the next phase using tools such as Power BI or Tableau.

# **System Overview**

**System Components**

* **Python scripts** for ETL (Extract, Transform, Load)
* **SQL Server** for data storage and query execution
* **Kaggle Dataset**: Historical online retail sales transactions
* **FakeStoreAPI**: Used to enrich product information

# **Data Research and Integration**

The online retail dataset contains multiple dimensions, including transactional sales records, customer demographics, and enriched product attributes. These are systematically cleaned, normalized, and stored in a relational database for efficient querying and analysis. This multi-dimensional data enables our team to perform advanced forecasting, segment analysis, and customer behavior modeling, essential for driving business insights and decision-making.

## **Types of data:**

| **Data Type** | **Description** | **Purpose** |
| --- | --- | --- |
| **Sales Data** | Historical transaction data (e.g., orders, order details, revenue) | Identify seasonal trends, product performance, high-value customers |
| **Customer Data** | Includes customer ID, location (country), and potentially segmentation information | Understand geographic distribution, loyalty, churn rate |
| **Product Data** | Product ID, name, brand, category, rating, image URLs | Analyze product mix, popularity, and gaps in catalog |
| **Brand Data** | Unique brand names associated with products | Assess brand performance and customer preferences |
| **Inventory Data** *(optional extension)* | Product availability, stock turnover rate | Manage restocking and avoid stockouts or overstocking |
| **Behavioral Data** *(future extension)* | Customer interaction history, cart activity, clickstream data | Enable recommendation engines and improve user experience |
| **Promotional/Discount Data** *(future extension)* | Historical campaigns and pricing strategies | Evaluate effectiveness of discounts and marketing efforts |

## **Data Sources**

* **Kaggle Online Retail Dataset**
* **(**[**https://www.kaggle.com/datasets/carrie1/ecommerce-data**](https://www.kaggle.com/datasets/carrie1/ecommerce-data)**)** :
  + It contains historical sales records with customer ID, invoice number, product codes, quantity, price, date, and location.
  + Key Tables derived: Orders, OrderDetails, Customers, Products
* **FakeStoreAPI (**[**https://fakestoreapi.com/products**](https://fakestoreapi.com/products)**)**:
  + It used to enrich products with additional metadata: Category, Brand, Rating, and ImageURL.
  + This helps simulate a real-world catalog with diversified product types and associated attributes.

# **Data Collection**

Python file for Data Collection from different sources and data cleaning

import pandas as pd

import numpy as np

import requests

import random

# --------------------------

# STEP 1: Read Kaggle Flat File

# --------------------------

kaggle\_data = pd.read\_csv("C:\\Users\\jyoro\\Downloads\\data.csv", encoding="ISO-8859-1")

# Drop missing important values

kaggle\_data = kaggle\_data.dropna(subset=["CustomerID", "Description", "InvoiceNo", "InvoiceDate", "StockCode", "Quantity", "UnitPrice"])

# Remove unwanted countries

kaggle\_data = kaggle\_data[~kaggle\_data["Country"].isin(["Unspecified", "European Community"])]

# --------------------------

# STEP 2: Normalize Tables

# --------------------------

# Orders Table

orders = kaggle\_data[["InvoiceNo", "InvoiceDate", "CustomerID"]].drop\_duplicates()

orders.columns = ["OrderID", "OrderDate", "CustomerID"]

orders["OrderDate"] = pd.to\_datetime(orders["OrderDate"], errors="coerce")

def random\_date(start\_year=2020):

    start = pd.Timestamp(f"{start\_year}-01-01")

    end = pd.Timestamp.today()

    return start + pd.to\_timedelta(np.random.randint(0, (end - start).days), unit='D')

orders["OrderDate"] = [random\_date() for \_ in range(len(orders))]

orders = orders.dropna(subset=["OrderID", "OrderDate", "CustomerID"]).drop\_duplicates(subset=["OrderID"])

# OrderDetails Table

order\_details = kaggle\_data[["InvoiceNo", "StockCode", "Quantity", "UnitPrice"]]

order\_details.columns = ["OrderID", "ProductID", "Quantity", "UnitPrice"]

order\_details = order\_details[(order\_details["Quantity"] > 0) & (order\_details["UnitPrice"] > 0)].drop\_duplicates()

# Customers Table

customers = kaggle\_data[["CustomerID", "Country"]].drop\_duplicates()

customers = customers.drop\_duplicates(subset=["CustomerID"])

customers = customers.dropna(subset=["CustomerID", "Country"])

customers["Country"] = customers["Country"].str.strip()

customers = customers[~customers["Country"].isin(["Unspecified", "European Community"])]

# Products Table (Initial)

products = kaggle\_data[["StockCode", "Description"]].dropna()

products.columns = ["ProductID", "ProductName"]

products = products[products["ProductName"].str.strip() != ""].drop\_duplicates(subset=["ProductID"])

# --------------------------

# STEP 3: Enrich Products with FakeStoreAPI

# --------------------------

try:

    response = requests.get("https://fakestoreapi.com/products?limit=150")

    response.raise\_for\_status()

    api\_data = response.json()

    brand\_dict = {

        "electronics": ["Apple", "Samsung", "Sony", "Dell"],

        "jewelery": ["Tiffany", "Swarovski", "Cartier"],

        "men's clothing": ["Zara", "H&M", "Nike"],

        "women's clothing": ["Gucci", "Chanel", "Prada"]

    }

    api\_df = pd.DataFrame([{

        "API\_ProductName": p["title"],

        "Category": p["category"],

        "BrandName": random.choice(brand\_dict.get(p["category"], ["GenericBrand"])),

        "Rating": p["rating"]["rate"],

        "ImageURL": p["image"]

    } for p in api\_data])

    # Sample to match number of products

    api\_sample = api\_df.sample(n=len(products), replace=True).reset\_index(drop=True)

    products = products.reset\_index(drop=True)

    products["ProductName"] = api\_sample["API\_ProductName"]

    products["Category"] = api\_sample["Category"]

    products["BrandName"] = api\_sample["BrandName"]

    products["Rating"] = api\_sample["Rating"]

    products["ImageURL"] = api\_sample["ImageURL"]

except Exception:

    print("Failed to fetch from API. Using fallback enrichment.")

    categories = ["electronics", "books", "home decor", "fashion", "fitness"]

    products["Category"] = np.random.choice(categories, size=len(products))

    products["BrandName"] = products["Category"]  # fallback

    products["Rating"] = np.random.uniform(3.0, 5.0, size=len(products)).round(1)

    products["ImageURL"] = "https://via.placeholder.com/150"

# --------------------------

# STEP 4: Generate Brands Table with BrandID and Category

# --------------------------

brands = products[["BrandName", "Category"]].drop\_duplicates().reset\_index(drop=True)

brands["BrandID"] = brands.index + 1  # simulate sequential BrandID

# Final products table as per schema

final\_products = products[["ProductID", "ProductName", "Category", "BrandName", "Rating", "ImageURL"]]

# --------------------------

# STEP 5: Save Cleaned + Enriched Tables

# --------------------------

orders.to\_csv("C:\\Users\\jyoro\\Downloads\\clean\_orders.csv", index=False)

order\_details.to\_csv("C:\\Users\\jyoro\\Downloads\\clean\_order\_details.csv", index=False)

customers.to\_csv("C:\\Users\\jyoro\\Downloads\\clean\_customers.csv", index=False)

final\_products.to\_csv("C:\\Users\\jyoro\\Downloads\\clean\_products.csv", index=False)

brands[["BrandID", "BrandName", "Category"]].to\_csv("C:\\Users\\jyoro\\Downloads\\clean\_brands.csv", index=False)

print("All cleaned and enriched tables saved as CSV.")

**Step-by-Step Process:**

1. **Raw Data Ingestion**:
   * Kaggle .csv file read using pandas.
   * Encoded in "ISO-8859-1" to support special characters.
2. **Data Cleaning**:

**1. Orders Table Cleaning**

**Source Columns**: InvoiceNo, InvoiceDate, CustomerID

**Steps Applied:**

* + **Null Removal**: Rows missing any of these fields were dropped using dropna(subset=["InvoiceNo", "InvoiceDate", "CustomerID"]).
* **Date Conversion**: InvoiceDate was converted to datetime format using pd.to\_datetime(errors='coerce').
* **Duplicate Removal**: Duplicates based on all three columns were dropped using .drop\_duplicates() to ensure only unique orders.
* **Column Renaming**: Columns were renamed to OrderID, OrderDate, and CustomerID for consistency.

**Result**: Cleaned Orders table with consistent date format and no missing or duplicate order IDs.

**2. OrderDetails Table Cleaning**

**Source Columns**: InvoiceNo, StockCode, Quantity, UnitPrice

**Steps Applied:**

* **Column Renaming**: Columns were renamed to OrderID, ProductID, Quantity, and UnitPrice.
* **Null Removal**: Rows with missing values in any of the columns were dropped.
* **Quantity Check**: Filtered out rows where Quantity <= 0.
* **UnitPrice Check**: Filtered out rows where UnitPrice <= 0.
* **Duplicate Removal**: Removed duplicate entries for the same OrderID and ProductID combination using .drop\_duplicates().

**Result**: Valid transactional records only, with positive quantity and price values.

**3. Customers Table Cleaning**

**Source Columns**: CustomerID, Country

**Steps Applied:**

* **Null Removal**: Dropped rows where CustomerID was missing using dropna().
* **Duplicate Removal**: Applied .drop\_duplicates(subset=["CustomerID"]) to ensure no repeated customers.
* **Data Type Fix**: Ensured CustomerID is treated as integer where applicable.

**Result**: Clean and unique set of customers with country-level granularity.

**4. Products Table Cleaning**

**Source Columns**: StockCode, Description

**Steps Applied:**

* **Column Renaming**: Renamed StockCode to ProductID and Description to ProductName.
* **Null & Empty Check**: Removed rows with null ProductID or where ProductName was empty (str.strip() != "").
* **Duplicate ProductIDs**: Ensured ProductID uniqueness by keeping the first occurrence using .drop\_duplicates(subset=["ProductID"]).

**Result**: A distinct list of valid products with non-empty names.

**5. Product Enrichment Cleaning**

**API Columns**: title, category, rating, image

**Steps Applied:**

* **API Integration**: Enriched products using https://fakestoreapi.com/products.
* **Sampling**: Randomly sampled matching rows to map onto ProductIDs.
* **Column Mapping**: Mapped API title -> ProductName, category -> Category & Brand, rating.rate -> Rating, and image -> ImageURL.
* **Fallback Handling**: If API fails, fallback to randomly generated data for:
  + - **Category**: Random choice from predefined list
    - **Brand**: Copied from Category
    - **Rating**: Random float between 3.0 and 5.0
    - **ImageURL**: Placeholder image

**Result**: Enriched Products table with consistent metadata for use in sales analysis.

**6. Brands Table Cleaning**

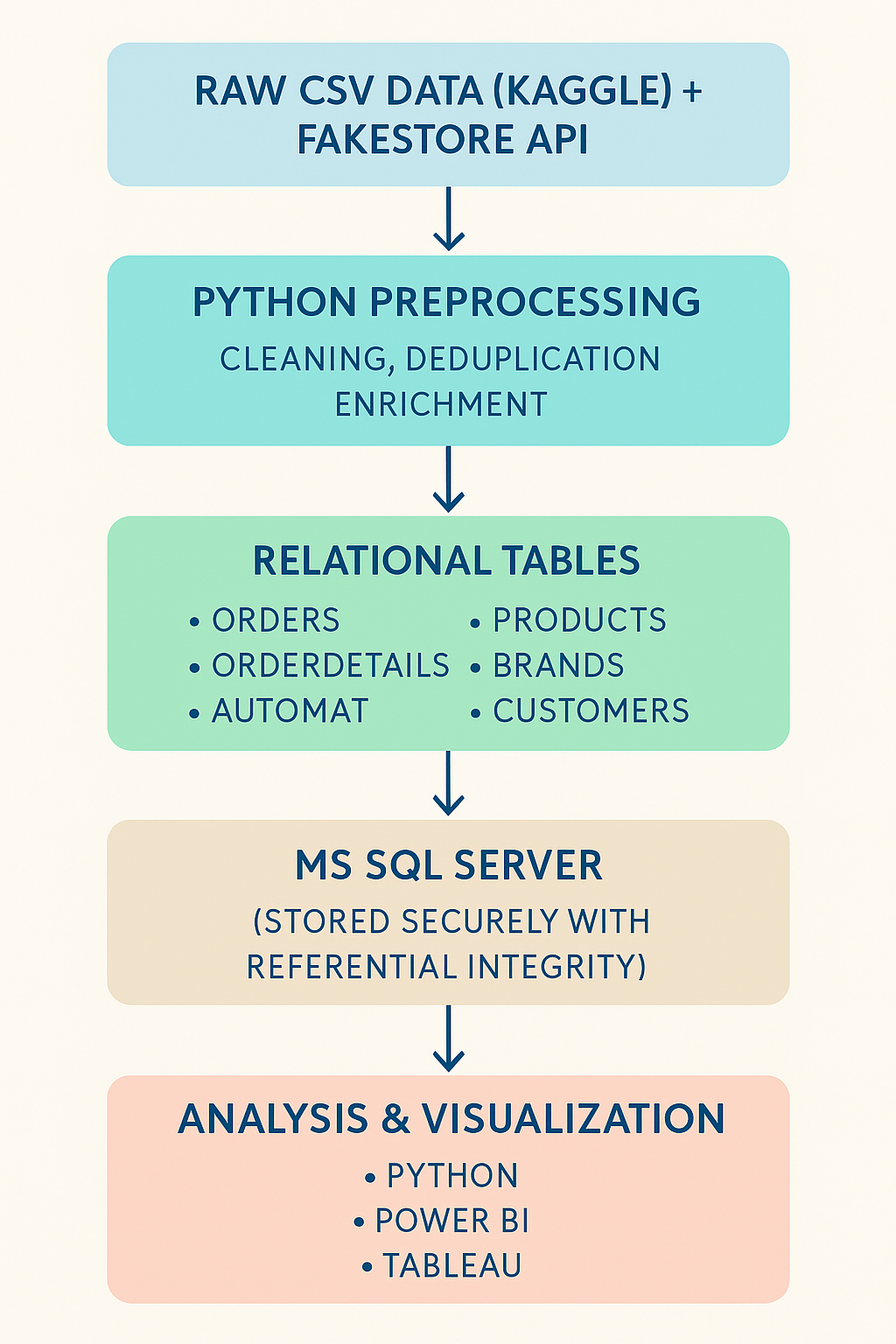
**Derived From**: Enriched Brand column in Products table

**Steps Applied:**

* **Unique Extraction**: Used .drop\_duplicates() on Brand column.
* **Null Removal**: Dropped missing or blank values.
* **Renaming**: Set final column name to BrandName.

**Result**: Cleaned and unique brand list used for foreign key mapping in the Products table.

1. **Table Normalization**:
   * Split flat file into 4 main normalized tables:
     + Orders: unique invoice numbers with timestamps and customer ID
     + OrderDetails: quantity and price per product per order
     + Customers: unique customer IDs and their country
     + Products: unique product IDs and names
2. **Enrichment via API**:
   * For each product, additional attributes were fetched from FakeStoreAPI.
   * If API fails, fallback logic uses randomly generated values from predefined categories.
3. **De-duplication & Final Checks**:
   * Removed any remaining duplicate ProductID, CustomerID, or OrderID entries before export.
4. **CSV Export**:
   * Cleaned tables exported to:  
     clean\_orders.csv, clean\_order\_details.csv, clean\_customers.csv, clean\_products.csv, clean\_brands.csv



# **Data Storage and Maintenance**

## **Database: SQL Server (Sales\_Analysis)**

**Tables Created** using the below SQL script:

* **Brands** (BrandID, BrandName)
* **Customers** (CustomerID, Country)
* **Products** (ProductID, ProductName, Category, BrandID, BrandName, Rating, ImageURL)
* **Orders** (OrderID, OrderDate, CustomerID)
* **OrderDetails** (OrderDetailID, OrderID, ProductID, Quantity, UnitPrice)

1. Brands Table

CREATE TABLE Brands (

BrandID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incremented primary key

BrandName VARCHAR(255) NOT NULL UNIQUE, -- Brand name must be unique

Category VARCHAR(100) NOT NULL

);

1. Create Customers Table

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Country VARCHAR(100)

);

1. Create Products Table

CREATE TABLE Products (

ProductID VARCHAR(50) PRIMARY KEY,

ProductName VARCHAR(255),

Category VARCHAR(100),

BrandID INT,

Brand VARCHAR(255) NOT NULL,

Rating FLOAT CHECK (Rating BETWEEN 1 AND 5),

ImageURL VARCHAR(500),

FOREIGN KEY (Brand) REFERENCES Brands(BrandName)

);

1. Create Orders Table

CREATE TABLE Orders (

OrderID VARCHAR(50) PRIMARY KEY,

OrderDate DATETIME,

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

1. Create OrderDetails Table

CREATE TABLE OrderDetails (

OrderDetailID INT IDENTITY(1,1) PRIMARY KEY,

OrderID VARCHAR(50),

ProductID VARCHAR(50),

Quantity INT,

UnitPrice FLOAT,

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID));

## **Data Loading**

* Used python file to connect and insert into SQL Server.
* Mapped Brand from product CSV to BrandID via lookup dictionary.
* Applied error handling for missing brands, type conversions, and duplicate entries.
* Integrity constraints maintained using primary and foreign keys.

import numpy as np

import pandas as pd

import pyodbc

# --------------------------

# STEP 1: Connect to SQL Server

# --------------------------

def connect\_to\_db():

    try:

        conn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};'

                            'SERVER=Jyoti\\SQLEXPRESS;'

                            'DATABASE=Sales\_Analysis;'

                            'Trusted\_Connection=yes;')

        return conn

    except pyodbc.Error as e:

        print("Database connection failed:", e)

        return None

# --------------------------

# STEP 2: Load CSV Data

# --------------------------

brands = pd.read\_csv("C:\\Users\\jyoro\\Downloads\\clean\_brands.csv")

customers = pd.read\_csv("C:\\Users\\jyoro\\Downloads\\clean\_customers.csv")

products = pd.read\_csv("C:\\Users\\jyoro\\Downloads\\clean\_products.csv")  # uses 'Brand' column

orders = pd.read\_csv("C:\\Users\\jyoro\\Downloads\\clean\_orders.csv")

order\_details = pd.read\_csv("C:\\Users\\jyoro\\Downloads\\clean\_order\_details.csv")

# --------------------------

# STEP 3: Insert Data into SQL Server

# --------------------------

def insert\_all\_data():

    conn = connect\_to\_db()

    if conn is None:

        return

    cursor = conn.cursor()

    # DELETE from child to parent to avoid FK conflicts

    try:

        cursor.execute("DELETE FROM OrderDetails")

        cursor.execute("DELETE FROM Orders")

        cursor.execute("DELETE FROM Products")

        cursor.execute("DELETE FROM Customers")

        cursor.execute("DELETE FROM Brands")

        cursor.execute("DBCC CHECKIDENT ('Brands', RESEED, 0);")

        conn.commit()

    except Exception as e:

        print("Error during deletion:", e)

    # Insert unique brands first and build mapping of BrandName -> BrandID

    brand\_ids = {}

    for \_, row in brands.iterrows():

        brand\_name = str(row["BrandName"]).strip()

        category = str(row["Category"]).strip()

        try:

            cursor.execute("""

                IF NOT EXISTS (SELECT 1 FROM Brands WHERE BrandName = ? AND Category = ?)

                INSERT INTO Brands (BrandName, Category) VALUES (?, ?)

            """, brand\_name, category, brand\_name, category)

        except Exception as e:

            print("Error inserting brand:", brand\_name, e)

    # Commit so we can fetch BrandIDs

    conn.commit()

    # Fetch all brand IDs

    cursor.execute("SELECT BrandID, BrandName FROM Brands")

    for row in cursor.fetchall():

        brand\_ids[row.BrandName.strip()] = row.BrandID

    # Insert into Products table

    for \_, row in products.iterrows():

        brand\_name = str(row['BrandName']).strip()

        brand\_id = brand\_ids.get(brand\_name, None)

        # Clean rating

        try:

            rating = float(row['Rating'])

        except:

            rating = 0.0

        try:

            cursor.execute("""

                INSERT INTO Products (ProductID, ProductName, Category, Brand, Rating, ImageURL)

                VALUES (?, ?, ?, ?, ?, ?)

            """, row['ProductID'], row['ProductName'], row['Category'], brand\_name, rating, row['ImageURL'])

        except Exception as e:

            print("Error inserting product:", row['ProductID'], e)

    # Insert customers

    for \_, row in customers.iterrows():

        cursor.execute("INSERT INTO Customers (CustomerID, Country) VALUES (?, ?)",

                    int(row["CustomerID"]), row["Country"])

    # Insert orders

    for \_, row in orders.iterrows():

        cursor.execute("INSERT INTO Orders (OrderID, OrderDate, CustomerID) VALUES (?, ?, ?)",

                    row["OrderID"], row["OrderDate"], int(row["CustomerID"]))

    # Insert order details

    for \_, row in order\_details.iterrows():

        cursor.execute("""

            INSERT INTO OrderDetails (OrderID, ProductID, Quantity, UnitPrice)

            VALUES (?, ?, ?, ?)""",

            row["OrderID"], row["ProductID"], int(row["Quantity"]), float(row["UnitPrice"]))

    conn.commit()

    cursor.close()

    conn.close()

    print("All tables loaded into SQL Server successfully.")

if \_\_name\_\_ == "\_\_main\_\_":

    insert\_all\_data()

## **Storage Needs**

* In the online retail sales, storage must support a high volume of transaction data, including product catalogues, customer information, order histories, and real-time inventory updates.
* It is essential to ensure data integrity, availability, and security, especially when dealing with sensitive customer data such as names, locations, and purchase behavior.
* The storage solution must offer fast access for analytics and reporting, as well as support scalability to handle seasonal sales spikes (e.g., Black Friday, holiday season).

## 

## **Storage Solutions**

* The collected and cleaned data from the Kaggle dataset and enriched external sources is stored in a relational database (MS SQL Server).
* Data is first normalized into structured tables: Orders, OrderDetails, Customers, Products, and Brands.
* Before storing the data, pre-processing steps are performed in Python:
  + Handling missing values
  + Removing duplicate entries
  + Formatting date and numeric types
* Integrating enrichment data from APIs (e.g., FakeStoreAPI for product details)
* This pre-processed data ensures accurate analytics and reporting once stored in the SQL Server.

## **Data Retention Policies**

* As a best practice, data retention policies must be established:
  + Customer orders and transaction logs may need to be retained for 5–7 years for auditing and legal compliance.
  + Product and inventory data can be updated periodically, with version control or archiving in place.
  + User activity data (like purchase trends or clickstream behavior) may be kept longer for predictive modeling and business intelligence.
* Policies should be designed based on the nature of data:
  + Transactional data for finance/reporting
  + Customer data for support or loyalty analysis
  + Marketing interaction data for trend forecasting

## **Data Backup and Disaster Recovery**

* A **daily or real-time backup system** is essential to protect business-critical retail data from:
  + **System crashes**
  + **Ransomware or malicious attacks**
  + **Human errors or accidental deletion**
* A **disaster recovery plan** must be in place to:
  + Restore SQL Server databases from backups
  + Re-deploy customer-facing systems like dashboards and APIs
  + Notify internal teams and mitigate impact on ongoing sales
* Regular **system maintenance** should ensure:
  + Database indexes are optimized
  + Storage drives are not nearing full capacity
  + Latest **security patches and updates** are applied to both the database and the hosting infrastructure.

# **Data Quality**

## **Data Cleaning Methods**

### Data Exploration

* Begin by exploring datasets like Orders, OrderDetails, Products, Customers, and Brands using pandas profiling, .info(), .describe(), and .value\_counts().
* Identify:
* **Missing values** in key columns like CustomerID, ProductName, or OrderDate.
* **Outliers** in numeric fields like Quantity, UnitPrice, or Rating.
* **Duplicates** especially in OrderID, CustomerID, or ProductID.
* **Inconsistencies** in text fields like Country, Brand, or Category.

### Handling Missing Values

* Implement strategies for handling missing values, such as imputation (replacing missing values with calculated estimates) or deletion (removing rows or columns with missing values).
* Choose imputation techniques like K-nearest Neighbors (KNN) Imputation, Forward Fill/Backward Fill baes on the data's nature and the extent of missingness.

### Data Transformation

* Apply transformations such as normalization or standardization to ensure consistency and comparability across variables.
* Transform categorical variables into numerical representations using techniques like one-hot encoding or label encoding.

## **Data Validation and Verification**

### Validation Rules

* **Range checks**:
  + Rating should be between 0 and 5
  + UnitPrice and Quantity must be non-negative
* **Format validation**:
  + OrderID and ProductID must follow unique, consistent format
* **Referential integrity**:
  + CustomerID in Orders must exist in Customers
  + ProductID in OrderDetails must exist in Products

### Cross-Validation

* Cross-validate total sales across:
* Orders vs OrderDetails
* OrderDate ranges vs Customer activity
* Use groupby operations to check if customer segments are logical

## **Data Accuracy Assurance**

### Error Detection

* Use data profiling techniques column analysis, pattern analysis and descriptive statistics to detect errors or anomalies in the dataset.
* Look for patterns of inconsistency or unusual values that may indicate data quality issues.

### Data Cleaning Tools

* **pandas** for manual cleaning (.dropna(), .fillna(), .duplicated())
* **OpenRefine** for bulk text cleaning and clustering
* **Pyjanitor** or **Dataprep** for chaining transformation tasks

## **Data Documentation and Metadata Management**

### Metadata Creation

* Document metadata describing the dataset's structure, variables, and data quality issues.Included information on data sources, collection methods, and any transformations applied to the data.

### Data Dictionary

| **Variable Name** | **Definition** | **Data Type** | **Permissible Values** |
| --- | --- | --- | --- |
| OrderID | Unique order identifier | String | Alphanumeric, non-null |
| CustomerID | Unique customer identifier | Integer | Positive integers |
| UnitPrice | Price per unit for each product | Float | ≥ 0 |
| Rating | User rating for products | Float | 0 to 5 |
| Category | Type of product category | String | Electronics, Home, Clothing, etc. |
| Brand | Brand name of product | String | Unique, matched to Brands table |
| OrderDate | Date of order | DateTime | Valid timestamp |

# **Project Timeline:**

|  |  |  |
| --- | --- | --- |
| **Date** | **Deliverable** | **Responsible** |
| May 29 | Resource finding (API and dataset) | Yash, Neel, Hiral |
| Jun 06 | Collected data from API and planned, cleaning | Hiral, Jyoti |
| Jun 20 | Stored data in the database | Jyoti |
| Jun 22 | Midterm report | Jyoti, Hiral, Yash, Neel |
| Jul 05 | Further cleaning of data | Jyoti |
| Jul 20 | Unit Testing for Database Connection and Cleaning | Jyoti |
| Jul 29 | Job Scheduling Process | Yash |
| Jul 29 | Report Modification | Hiral |
| Jul 29 | Tableau Connection and Data Loading | Jyoti |
| Jul 30 | System Diagram | Hiral |
| Jul 30 | Visualization and Analysis | Jyoti, Hiral, Yash, Neel |
| Jul 30 | Testing for Tableau | Neel |
| Jul 31 | Power Point Presentation | Jyoti, Hiral, Yash, Neel |
| Jul 31 | Final Report | Jyoti, Hiral, Yash, Neel |

# **References**

* Clark, C. (2019). *Ecommerce Data*. Kaggle. <https://www.kaggle.com/datasets/carrie1/ecommerce-data>
* Microsoft. (2023). *SQL Server documentation*. Microsoft Docs. Retrieved from <https://learn.microsoft.com/en-us/sql/sql-server>
* Pandas Development Team. (2023). *Pandas documentation*. Retrieved from <https://pandas.pydata.org/docs>
* Numpy Developers. (2023). *NumPy documentation*. Retrieved from <https://numpy.org/doc/>
* PyODBC. (2023). *pyodbc - A Python DB API module*. GitHub. Retrieved from <https://github.com/mkleehammer/pyodbc>
* Fake Store API. (n.d.). *Fake Store API Documentation*. Retrieved from <https://fakestoreapi.com/docs>
* Canva. (2024). *Infographic & Visual Design Tool*. Retrieved from <https://www.canva.com>
* IBM. (2023). *Data cleaning and data preparation for analytics*. IBM Cloud Learn Hub. Retrieved from <https://www.ibm.com/cloud/learn/data-cleaning>
* DataCamp. (2023). *Data cleaning in Python: The ultimate guide*. Retrieved from <https://www.datacamp.com/tutorial/data-cleaning-python>