Academic Year	Module	Assessment Number	Assessment Type
2024	Foundational Data Engineering	1	Report

Summer Class Assignment

Student Id : 2408414

Student Name : Shashank Pandey

Section : L6CG20

Tutor : Gunjan Kumar Mishra

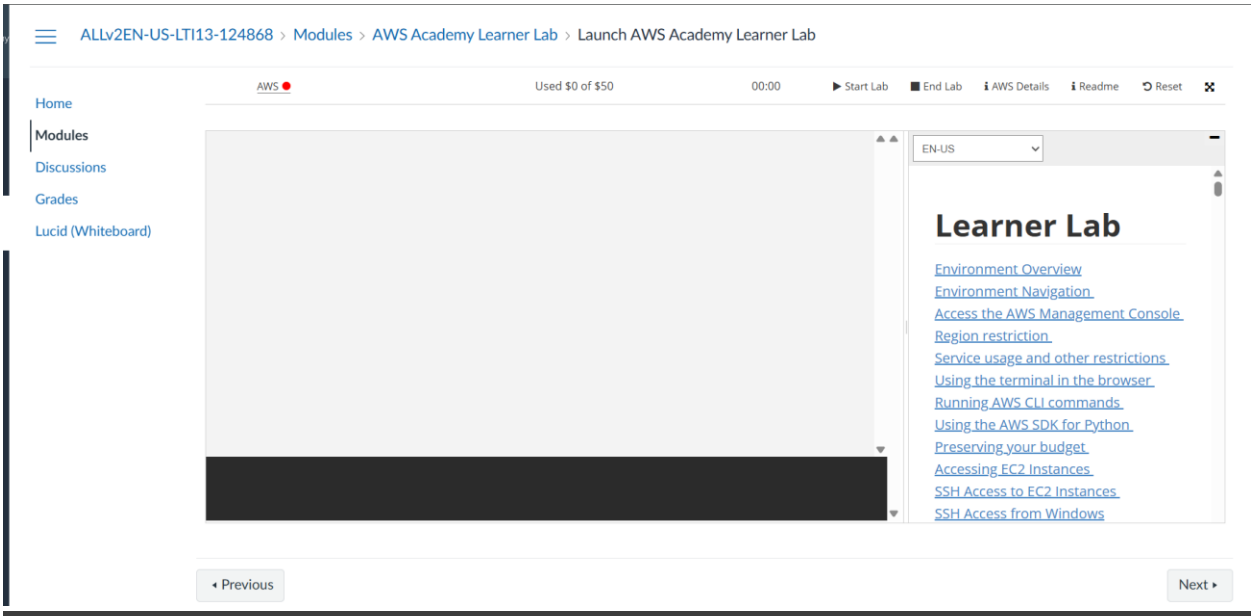
Submitted on : 09-06-2025.

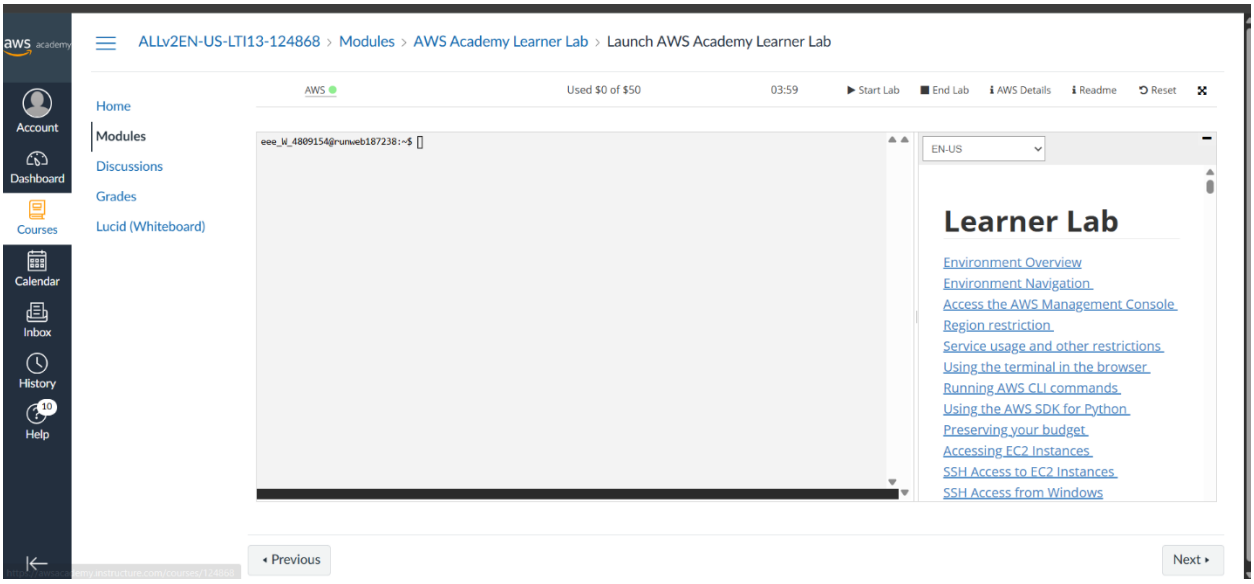
Table of Contents

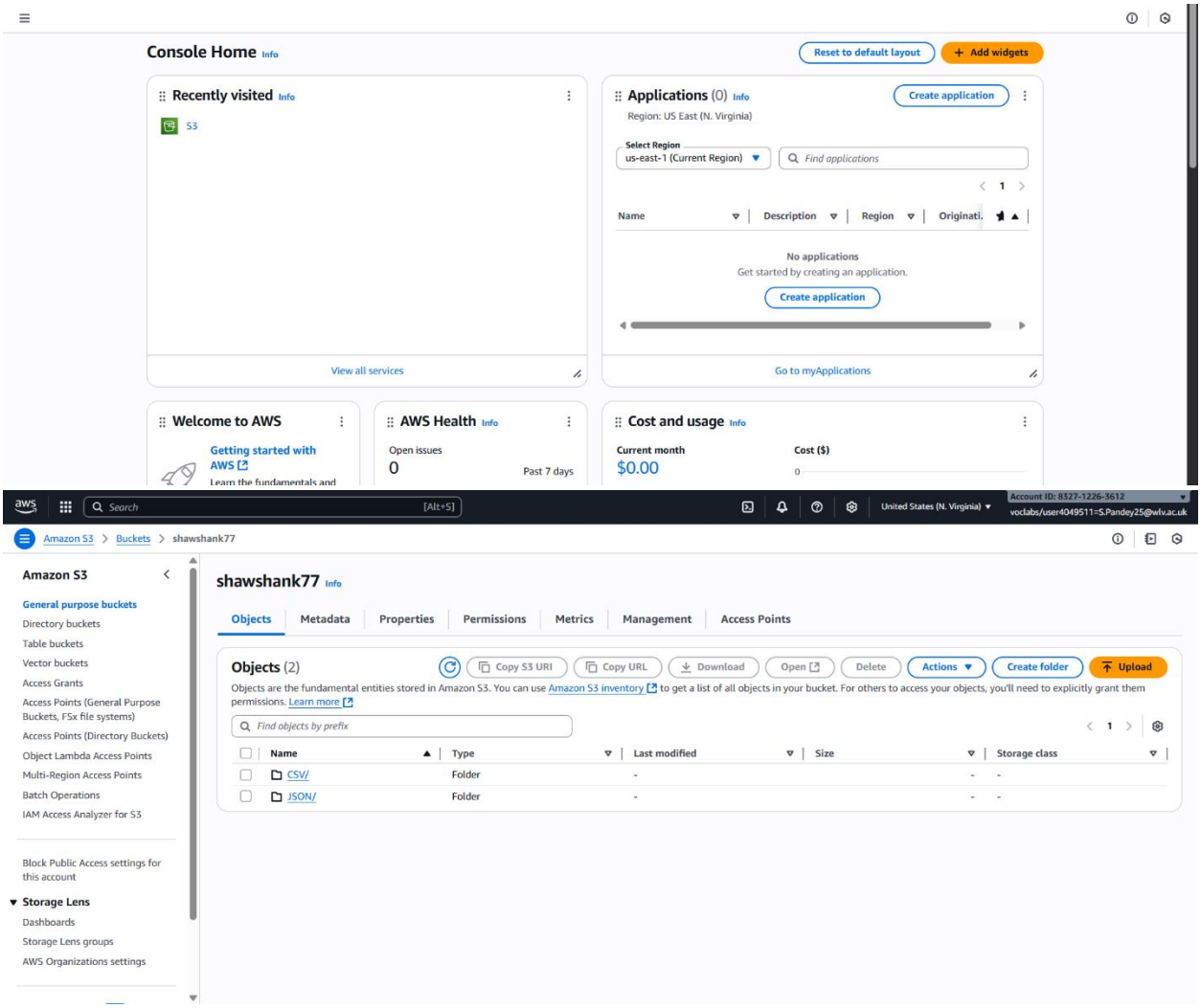
Week 1 – Data Extraction (Architecture Setup)

Step 1: AWS Setup

- Enrolled in AWS Academy and accessed the AWS console.
- Created an S3 bucket to store raw data (CSV & JSON).
- Configured public access for uploaded files (since IAM was restricted in AWS Academy).
- Uploaded raw datasets (CSV + JSON) into separate folders in the S3 bucket.






Console Home Info
Reset to default layout
+ Add widgets
Recently visited Info
S3
View all services
Applications (0) Info
Region: US East (N. Virginia)
Create application
Select Region
us-east-1 (Current Region)
Find applications
Name
Description
Region
Originati.
No applications
Get started by creating an application.
Create application
Go to myApplications
Welcome to AWS
Getting started with AWS
AWS Health Info
Open issues
0
Past 7 days
Cost and usage Info
Current month
$0.00
Cost ($)

Search
[Alt+S]
United States (N. Virginia)
Account ID: 8327-1226-3612
voclabs/user4049511=S.Pandey25@wlv.ac.uk
Amazon S3 > Buckets > shawshank77
Amazon S3
General purpose buckets
Directory buckets
Table buckets
Vector buckets
Access Grants
Access Points (General Purpose Buckets, FSx file systems)
Access Points (Directory Buckets)
Object Lambda Access Points
Multi-Region Access Points
Batch Operations
IAM Access Analyzer for S3
Block Public Access settings for this account
Storage Lens
Dashboards
Storage Lens groups
AWS Organizations settings
shawshank77 Info
Objects
Metadata
Properties
Permissions
Metrics
Management
Access Points
Objects (2)
Copy S3 URI
Copy URL
Download
Open
Delete
Actions
Create folder
Upload
Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more
Find objects by prefix
Name
Type
Last modified
Size
Storage class
CSV/
Folder
JSON/
Folder

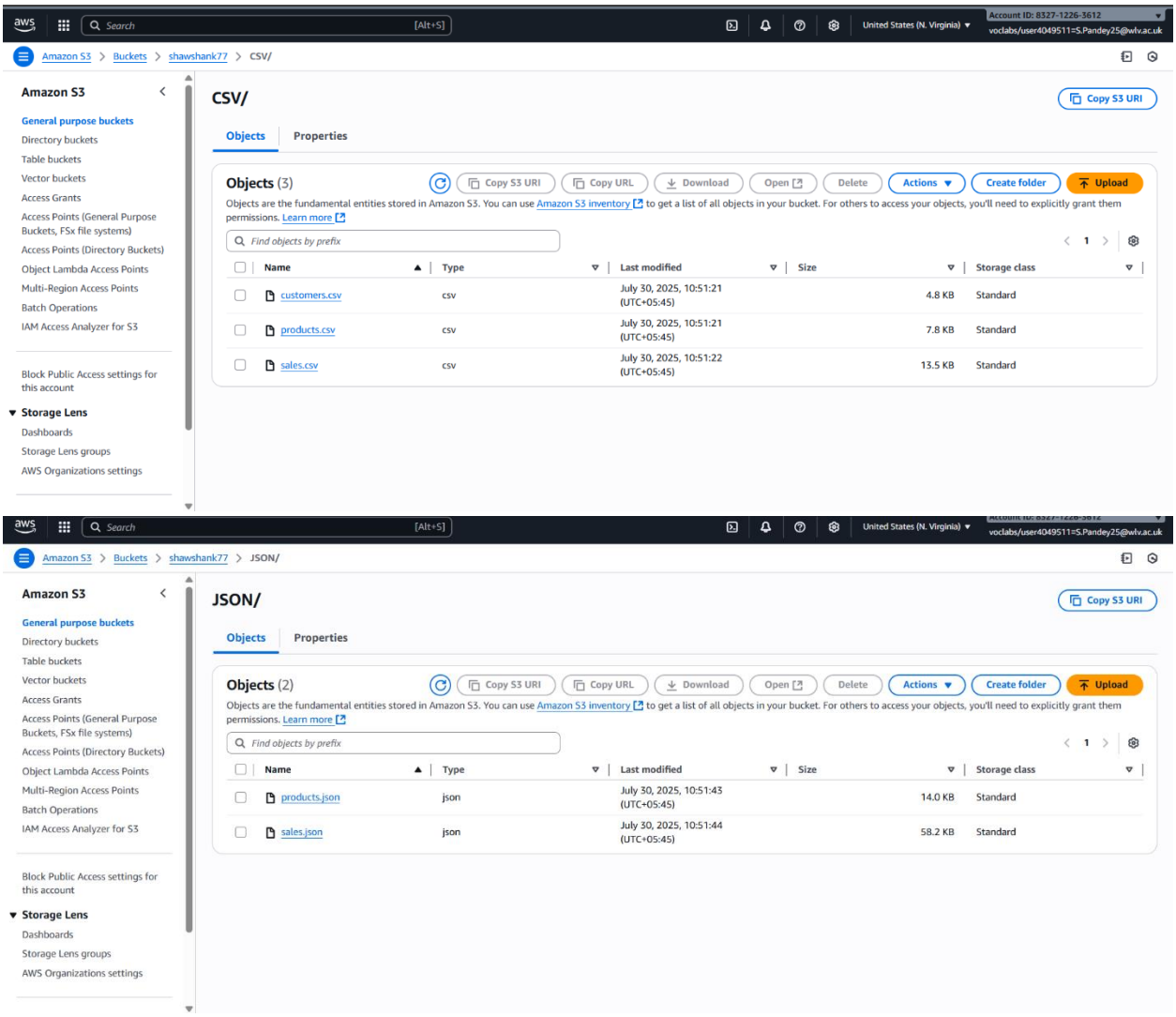


Figure 1 S3 Bucket Creation

Step 2: PostgreSQL & PgAdmin Setup

- Installed PostgreSQL (via installer/Homebrew/apt depending on OS).
- Created a new server group in pgAdmin named FDE.
- Registered PostgreSQL server with username postgres and password set during installation.
- Created a new FDE database with schema landing.

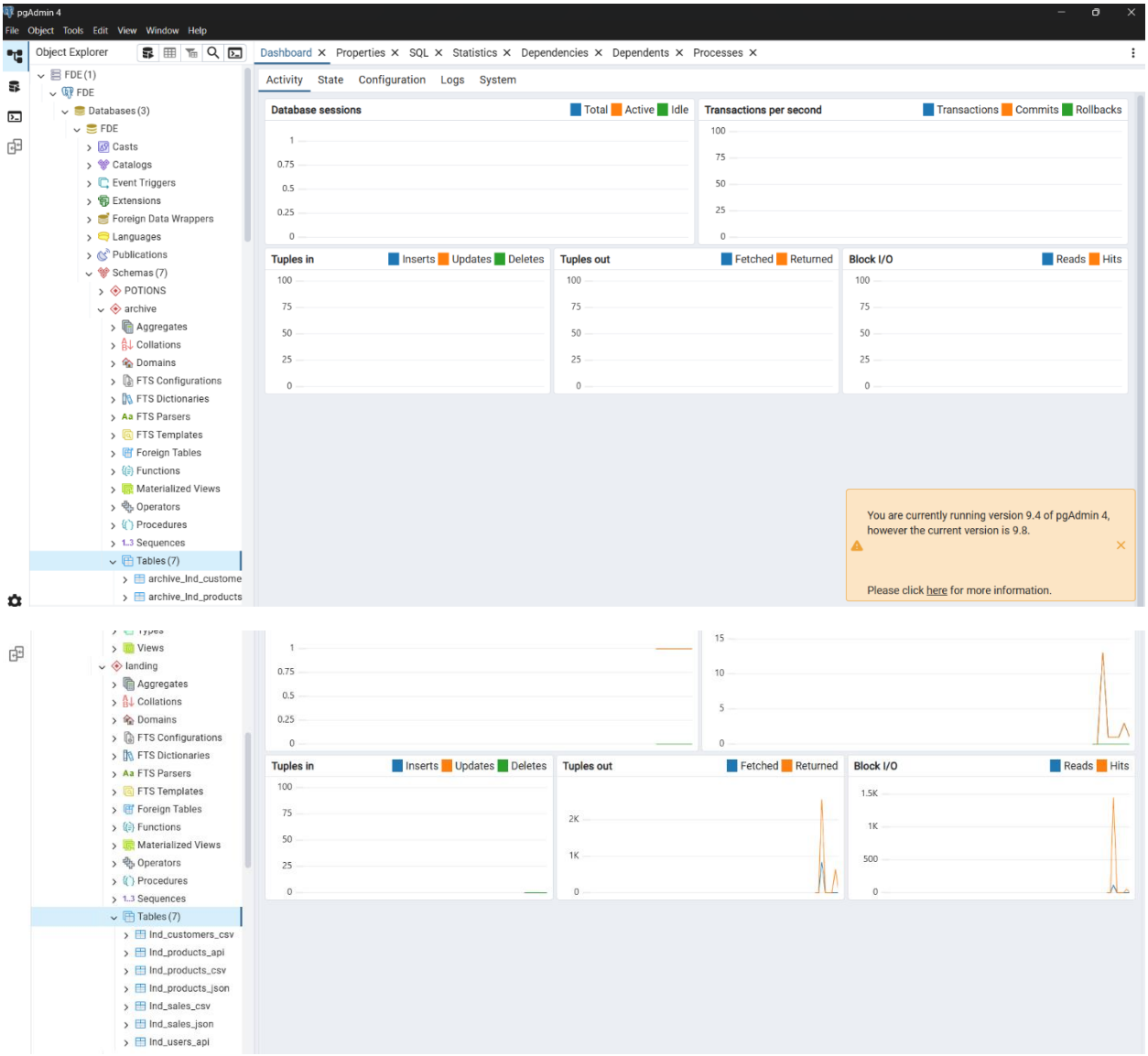


Figure 2 PgAdmin Setup

Week 2 – Data Extraction (Continued)

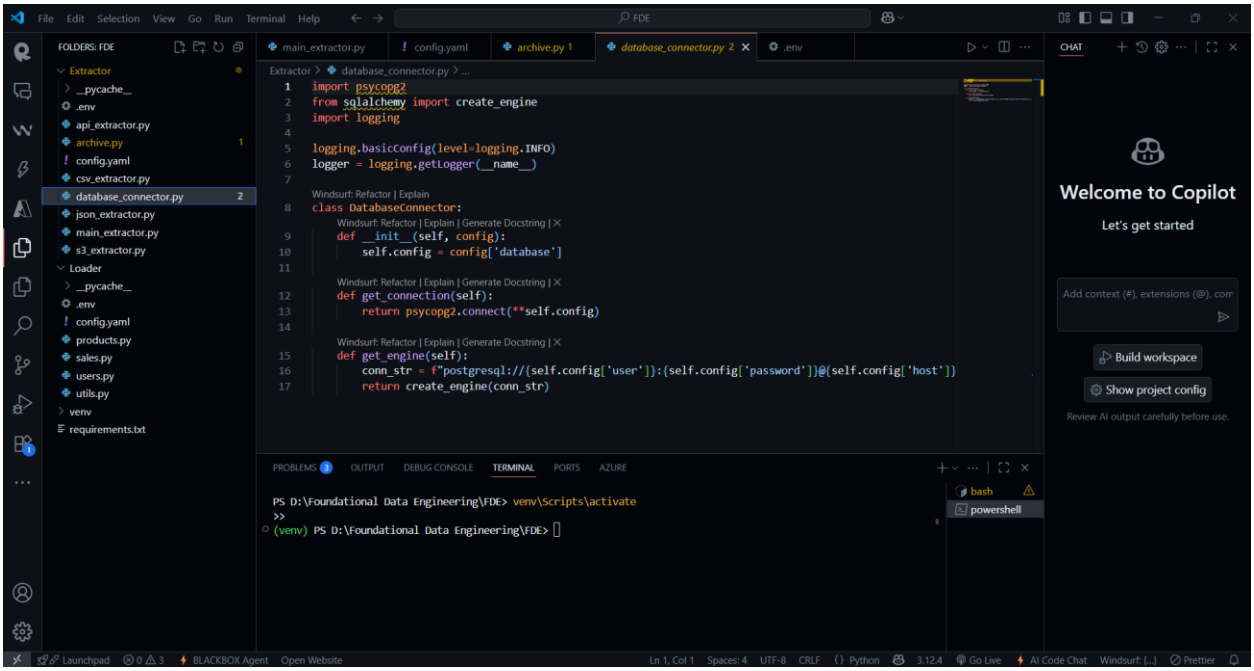
Step 3: Database Connection & Environment Setup

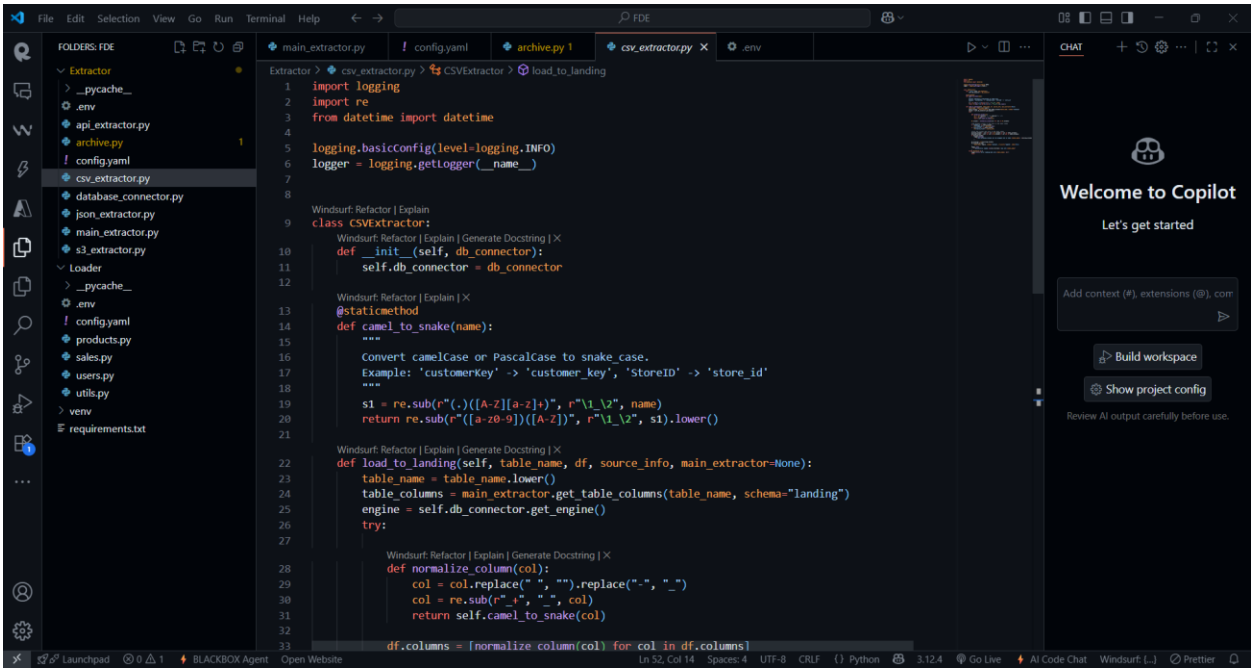
- Created a virtual environment (venv) and activated it.
- Installed dependencies using pip install -r requirements.txt.
- Created database_connector.py using psycopg2 and SQLAlchemy for DB connections.
- Configured logger for monitoring and debugging.

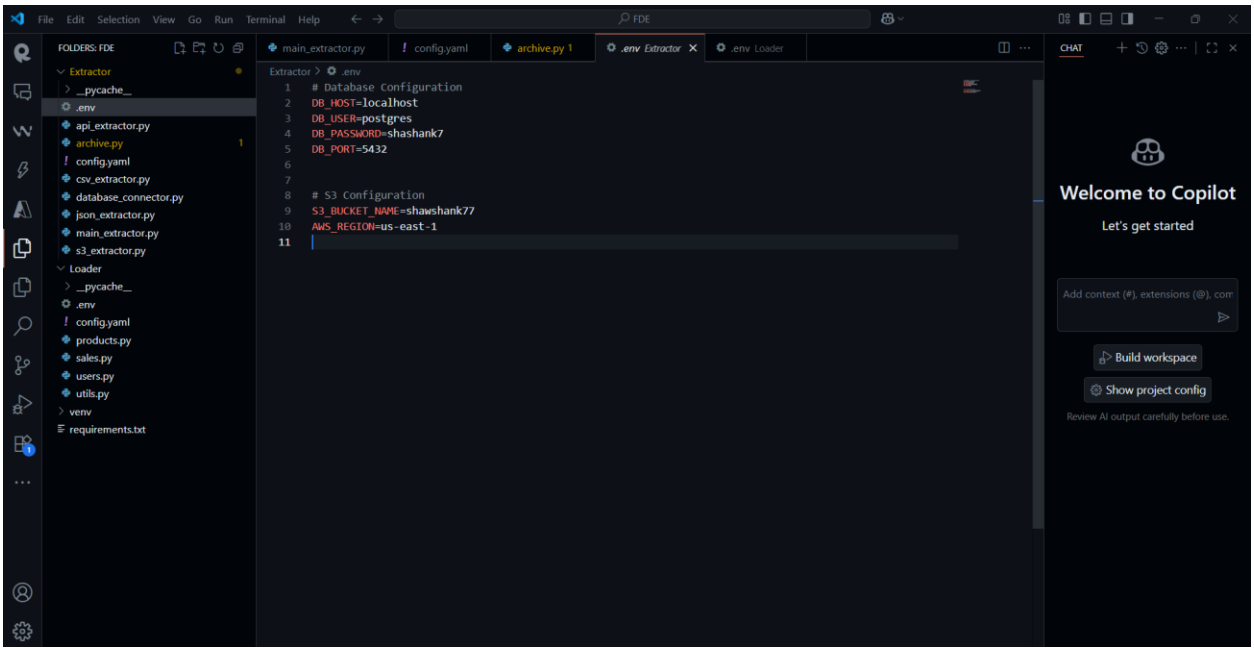
Step 4: Extractor Modules

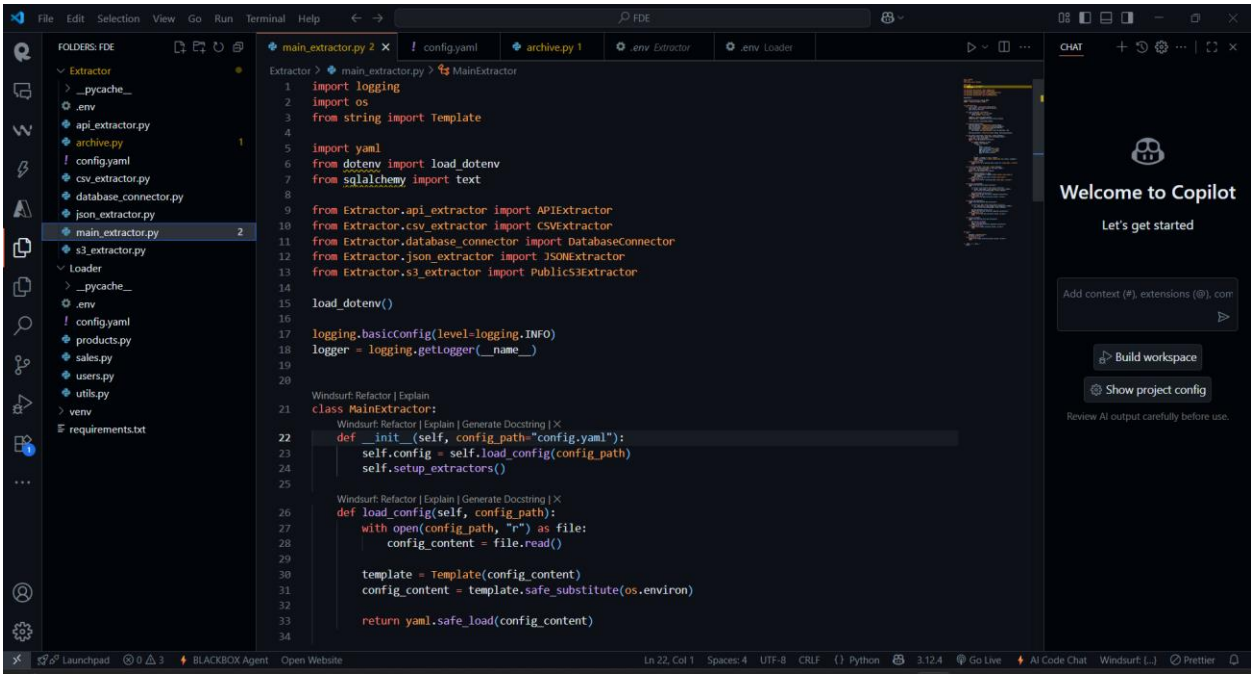
- CSV Extractor: Loaded CSV files from S3 → Landing tables using pandas + SQLAlchemy.
- JSON Extractor: Inserted raw JSON into landing tables using psycopg2.extras.Json.
- Public S3 Extractor: Built get_public_url() and extract_file() to handle file downloads.
- API Extractor: Connected to external APIs, parsed JSON responses, and loaded data.
- Main Extractor: Integrated all extractors, managed configs, handled logging & execution.

Figure 3 venv activation


File Edit Selection View Go Run Terminal Help
FDE
FOLDERS: FDE
Extractor
__pycache__
.env
api_extractor.py
archive.py
config.yaml
csv_extractor.py
database_connector.py
json_extractor.py
main_extractor.py
s3_extractor.py
Loader
__pycache__
.env
config.yaml
products.py
sales.py
users.py
utils.py
venv
requirements.txt
main_extractor.py
config.yaml
archive.py 1
database_connector.py 2
.env
Extractor > database_connector.py > ...
import psycopg2
from sqlalchemy import create_engine
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
Windsurf: Refactor | Explain
class DatabaseConnector:
Windsurf: Refactor | Explain | Generate Docstring |
def __init__(self, config):
self.config = config['database']
Windsurf: Refactor | Explain | Generate Docstring |
def get_connection(self):
return psycopg2.connect(**self.config)
Windsurf: Refactor | Explain | Generate Docstring |
def get_engine(self):
conn_str = f"postgresql://{self.config['user']}:{self.config['password']}@{self.config['host']}
return create_engine(conn_str)
PROBLEMS 3
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
AZURE
bash
powershell
PS D:\Foundational Data Engineering\FDE> venv\Scripts\activate
>>
(venv) PS D:\Foundational Data Engineering\FDE>
CHAT
Welcome to Copilot
Let's get started
Add context (#), extensions (@), com
Build workspace
Show project config
Review AI output carefully before use.
Launchpad
BLACKBOX Agent
Open Website
Ln 1, Col 1
Spaces: 4
UTF-8
CRLF
Python
3.12.4
Go Live
AI Code Chat
Windsurf: (...)
Prettier


File Edit Selection View Go Run Terminal Help
FDE
FOLDERS: FDE
Extractor
__pycache__
.env
api_extractor.py
archive.py
config.yaml
csv_extractor.py
database_connector.py
json_extractor.py
main_extractor.py
s3_extractor.py
Loader
__pycache__
.env
config.yaml
products.py
sales.py
users.py
utils.py
venv
requirements.txt
main_extractor.py
config.yaml
archive.py 1
csv_extractor.py
.env
Extractor > csv_extractor.py > CSVExtractor > load_to_landing
import logging
import re
from datetime import datetime
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
Windsurf: Refactor | Explain
class CSVExtractor:
Windsurf: Refactor | Explain | Generate Docstring |
def __init__(self, db_connector):
self.db_connector = db_connector
Windsurf: Refactor | Explain |
@staticmethod
def camel_to_snake(name):
"""
Convert camelCase or PascalCase to snake_case.
Example: 'customerKey' -> 'customer_key', 'StoreID' -> 'store_id'
"""
s1 = re.sub(r"(.)([A-Z][a-z]+)", r"\1_\2", name)
return re.sub(r"([a-z0-9])([A-Z])", r"\1_\2", s1).lower()
Windsurf: Refactor | Explain | Generate Docstring |
def load_to_landing(self, table_name, df, source_info, main_extractor=None):
table_name = table_name.lower()
table_columns = main_extractor.get_table_columns(table_name, schema="landing")
engine = self.db_connector.get_engine()
try:
Windsurf: Refactor | Explain | Generate Docstring |
def normalize_column(col):
col = col.replace(" ", "").replace("-", "_")
col = re.sub(r"_+", "_", col)
return self.camel_to_snake(col)
df.columns = [normalize_column(col) for col in df.columns]
CHAT
Welcome to Copilot
Let's get started
Add context (#), extensions (@), com
Build workspace
Show project config
Review AI output carefully before use.
Launchpad
BLACKBOX Agent
Open Website
Ln 52, Col 14
Spaces: 4
UTF-8
CRLF
Python
3.12.4
Go Live
AI Code Chat
Windsurf: (...)
Prettier


File Edit Selection View Go Run Terminal Help
FDE
FOLDERS: FDE
Extractor
__pycache__
.env
api_extractor.py
archive.py
config.yaml
csv_extractor.py
database_connector.py
json_extractor.py
main_extractor.py
s3_extractor.py
Loader
__pycache__
.env
config.yaml
products.py
sales.py
users.py
utils.py
venv
requirements.txt
main_extractor.py
config.yaml
archive.py 1
.env Extractor
.env Loader
Extractor > .env
# Database Configuration
DB_HOST=localhost
DB_USER=postgres
DB_PASSWORD=shashank7
DB_PORT=5432
# S3 Configuration
S3_BUCKET_NAME=shawshank77
AWS_REGION=us-east-1
CHAT
Welcome to Copilot
Let's get started
Add context (#), extensions (@), com
Build workspace
Show project config
Review AI output carefully before use.


File Edit Selection View Go Run Terminal Help
FDE
FOLDERS: FDE
Extractor
__pycache__
.env
api_extractor.py
archive.py
config.yaml
csv_extractor.py
database_connector.py
json_extractor.py
main_extractor.py
s3_extractor.py
Loader
__pycache__
.env
config.yaml
products.py
sales.py
users.py
utils.py
venv
requirements.txt
main_extractor.py 2
config.yaml
archive.py 1
.env Extractor
.env Loader
Extractor > main_extractor.py > MainExtractor
import logging
import os
from string import Template
import yaml
from dotenv import load_dotenv
from sqlalchemy import text
from Extractor.api_extractor import APIExtractor
from Extractor.csv_extractor import CSVExtractor
from Extractor.database_connector import DatabaseConnector
from Extractor.json_extractor import JSONExtractor
from Extractor.s3_extractor import PublicS3Extractor
load_dotenv()
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
Windsurf: Refactor | Explain
class MainExtractor:
Windsurf: Refactor | Explain | Generate Docstring |
def __init__(self, config_path="config.yaml"):
self.config = self.load_config(config_path)
self.setup_extractors()
Windsurf: Refactor | Explain | Generate Docstring |
def load_config(self, config_path):
with open(config_path, "r") as file:
config_content = file.read()
template = Template(config_content)
config_content = template.safe_substitute(os.environ)
return yaml.safe_load(config_content)
CHAT
Welcome to Copilot
Let's get started
Add context (#), extensions (@), com
Build workspace
Show project config
Review AI output carefully before use.
Launchpad
BLACKBOX Agent
Open Website
Ln 22, Col 1
Spaces: 4
UTF-8
CRLF
Python
3.12.4
Go Live
AI Code Chat
Windsurf: (...)
Prettier

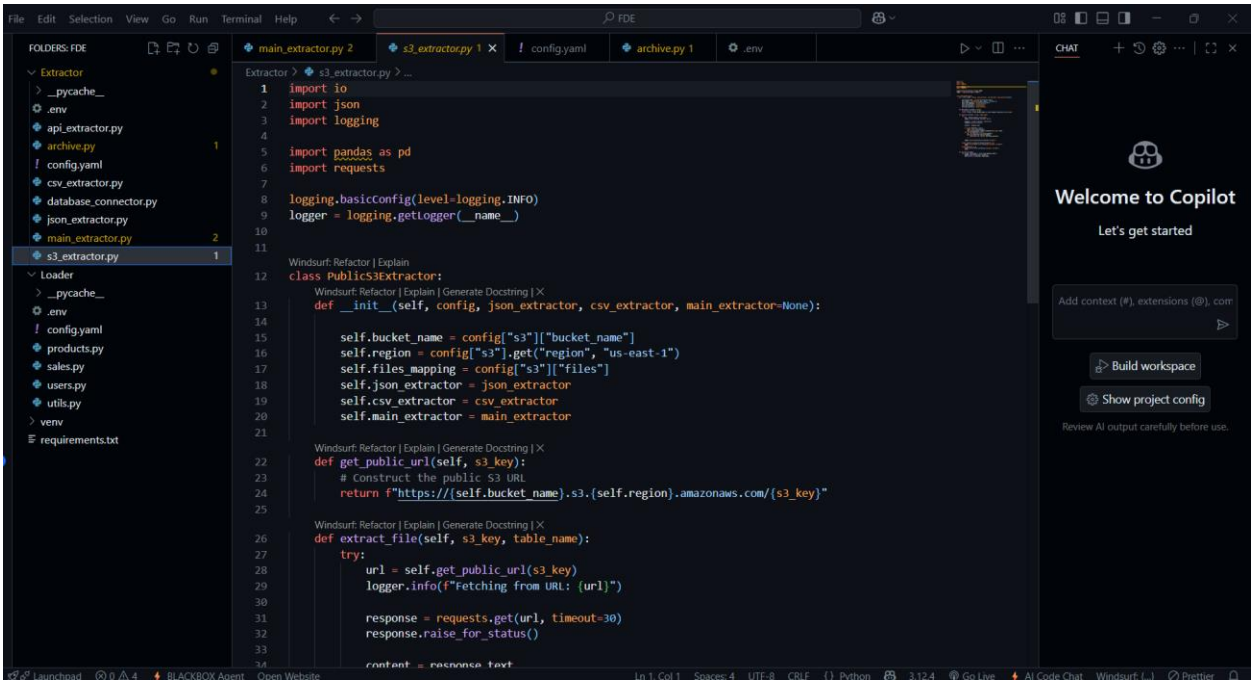


Figure 4 Requirement installation and all the setups

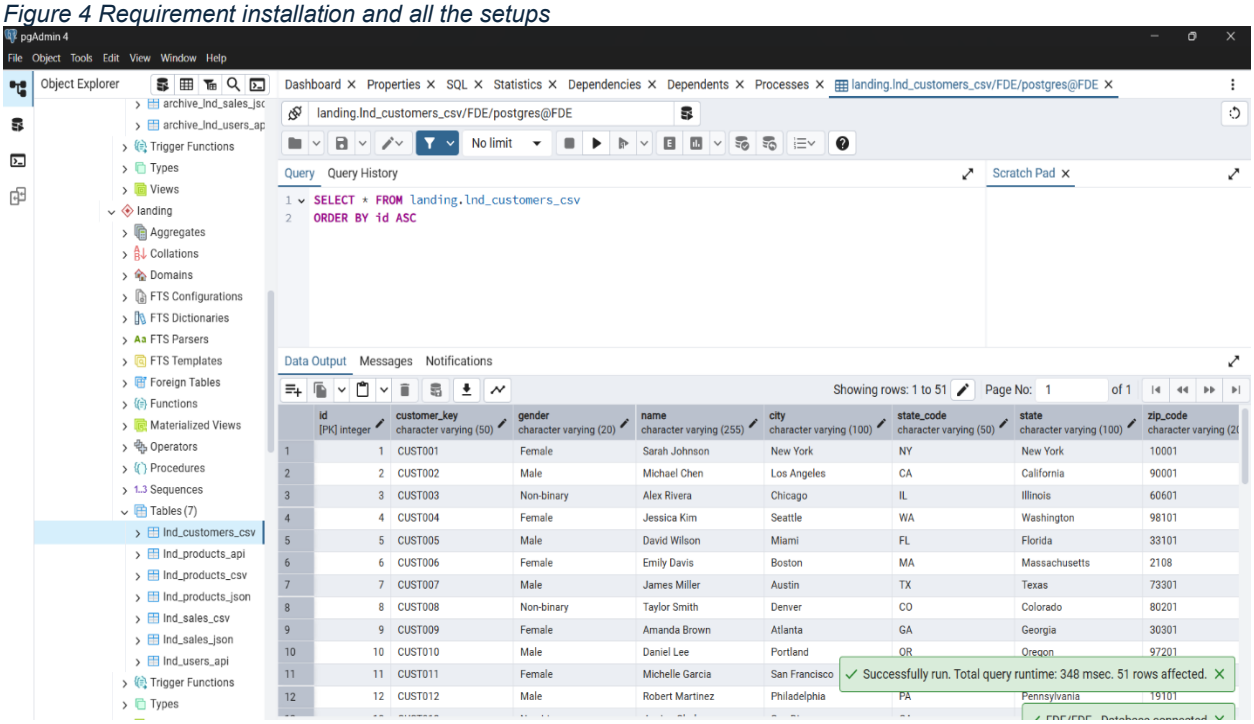


Figure 5 Loaded data in landing table

Week 3- Created all the Schemas left (Staging, Target, Transform)

Step 5: Schemas

- Created all the remaining Schemas i.e. Staging, Target, and Transform
- Loaded the data to the tables

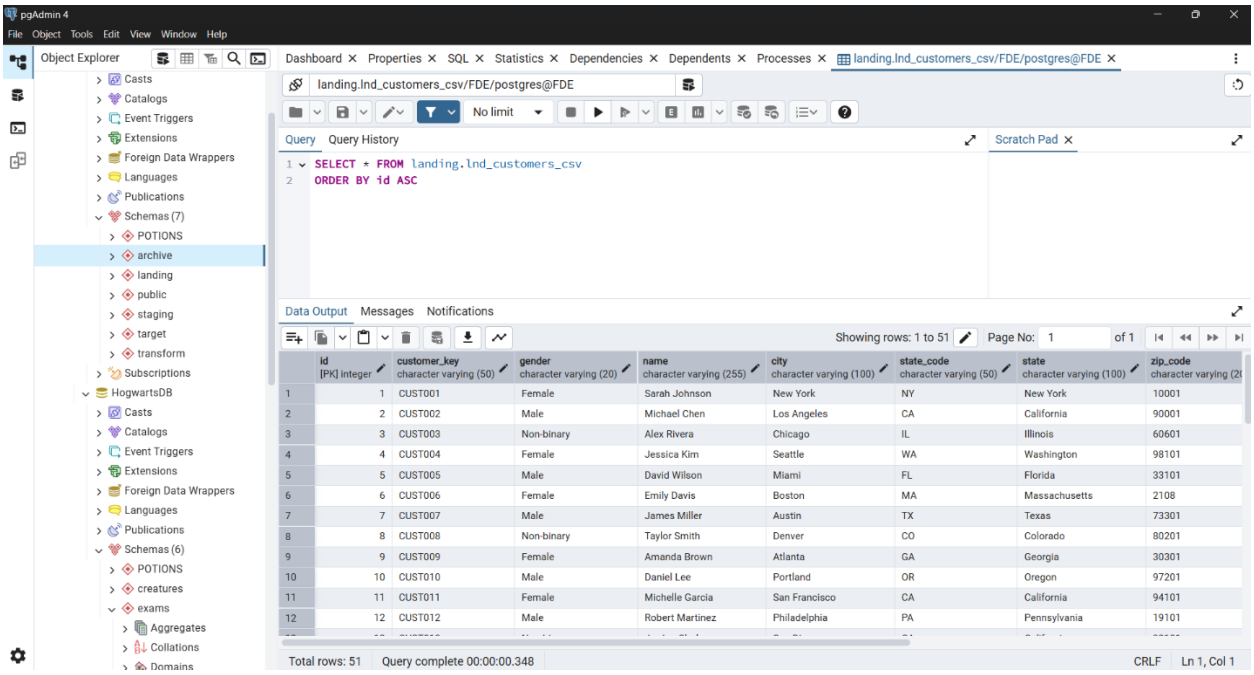


Figure 6 Created the schemas (Staging. Transform, and Target)

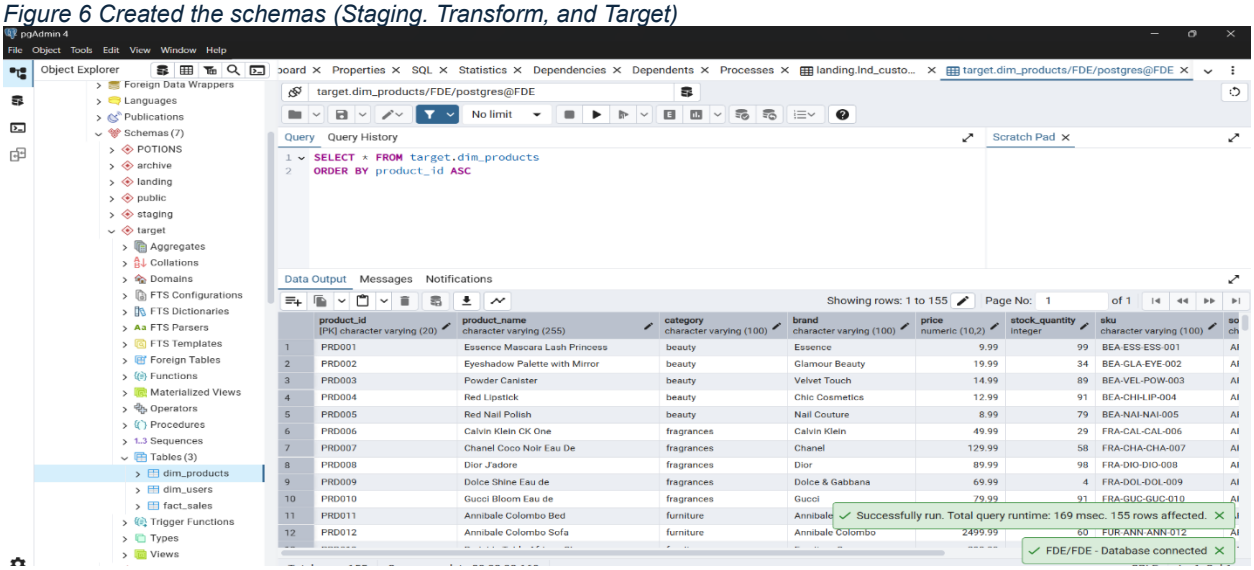


Figure 7 Loaded the data

Week 4- Data Loading

Step 6: Loader Setup

- Created a Loader folder for loading scripts.
- Configured .env file with PostgreSQL credentials.
- Created config.yaml with mappings of schemas (landing, staging, transform, target) and entities (products, users, sales).

Step 7: Utilities (utils.py)

- Implemented load_config(), get_db_connection(), get_schemas(), get_entities(), and execute_query() for DB + config handling.

Step 8: Loading Dimension Tables (products, users)

- Cleared temp tables before each run.
- Inserted required columns from staging views → temp tables.
- Used MERGE (UPSERT) with ON CONFLICT to insert/update target tables.
- Added updated_at timestamp for tracking changes.

Step 9: Loading Fact Table (sales)

- Cleaned missing store_id values (verified with data provider).
- Handled missing unit_price by filling from product table.
- Calculated total_amount = quantity × unit_price.
- Applied deduplication using DISTINCT ON to prevent duplicate sales.

Step 10: Script Execution

- Ran dimension load scripts first (products, users).
- Then ran fact_sales load script (ensuring foreign key references exist).
- Verified loaded data in pgAdmin target tables.

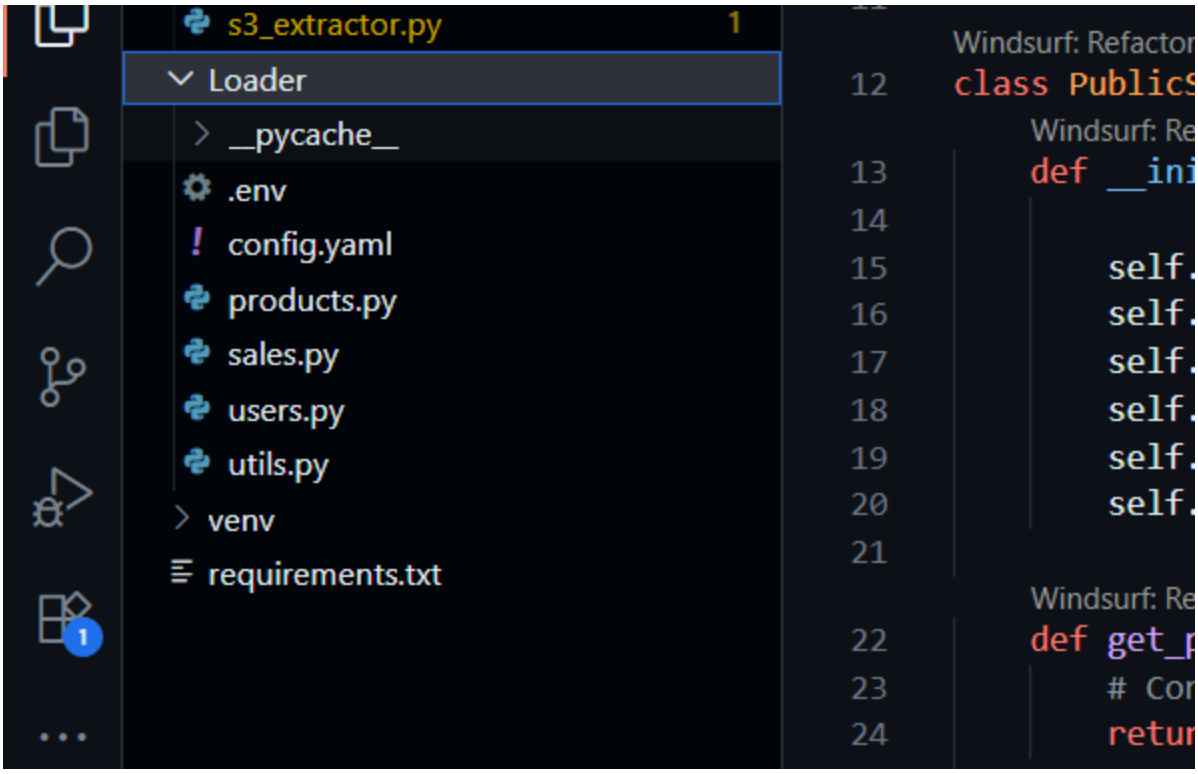


Figure 8 Loader folder created

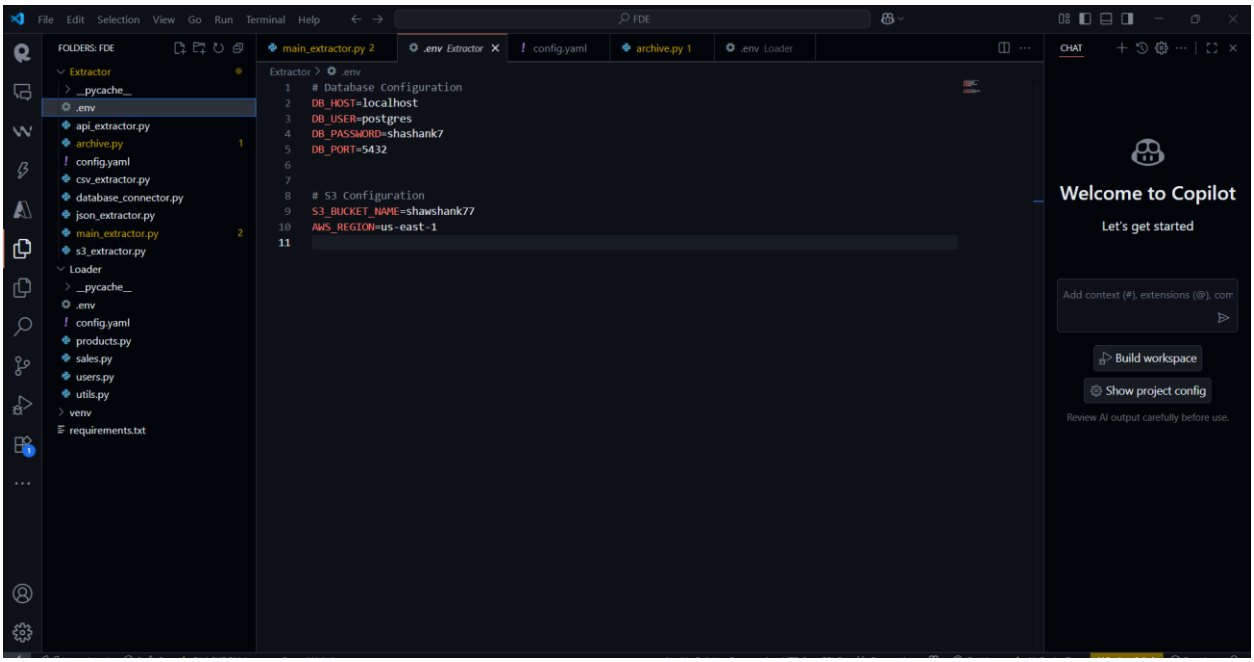


Figure 9 Setup .env

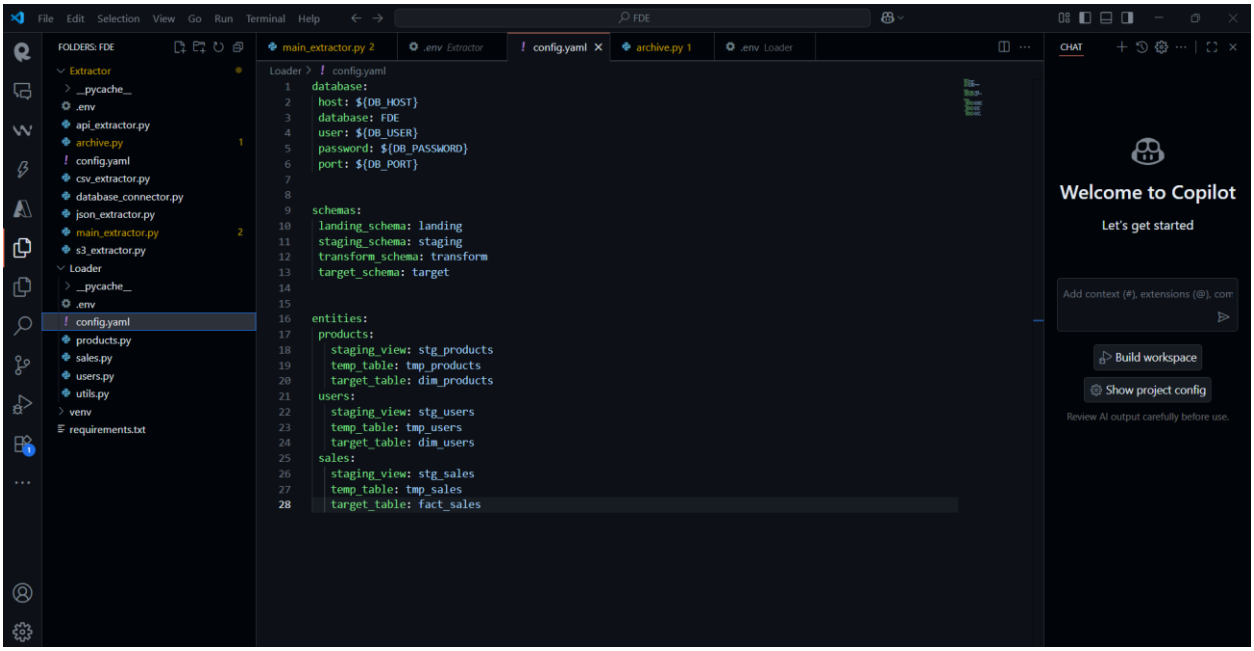
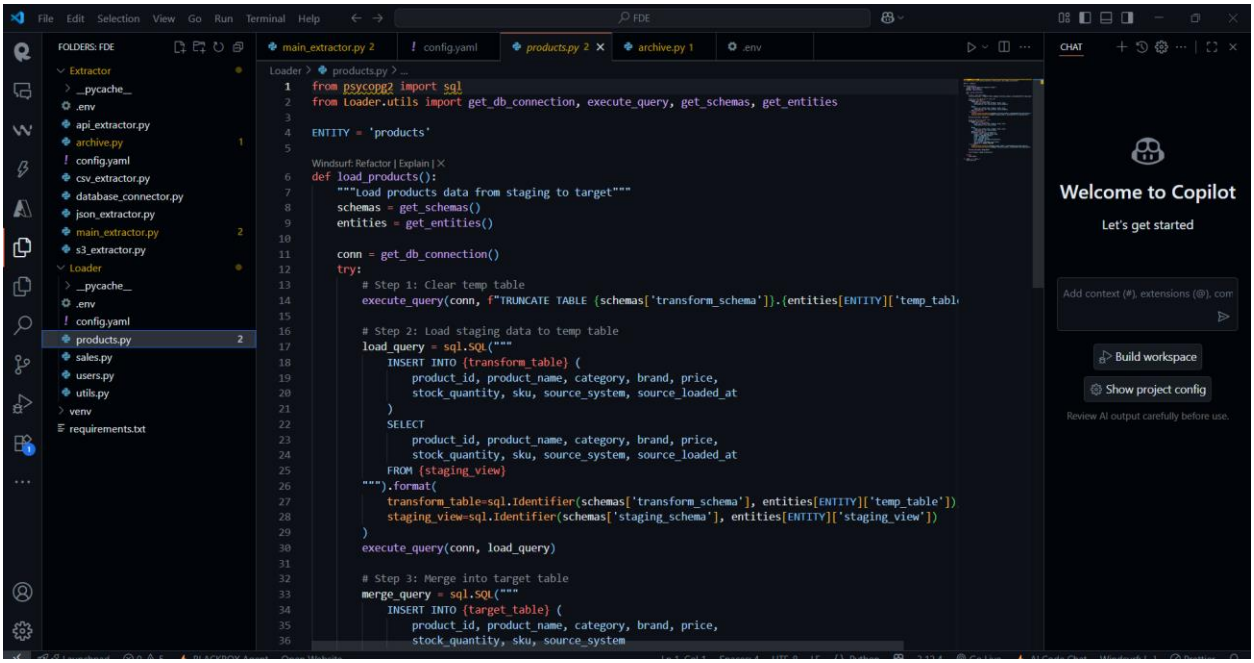
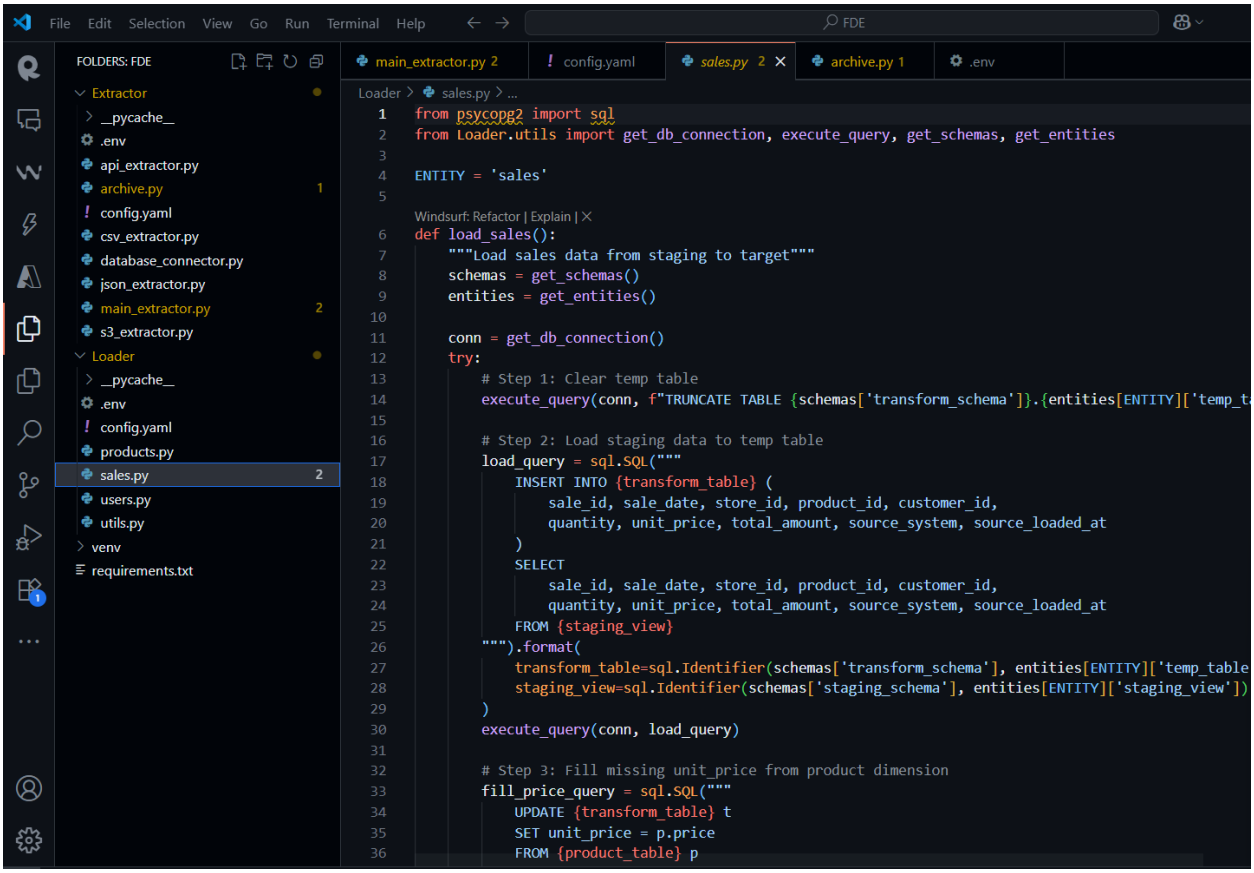
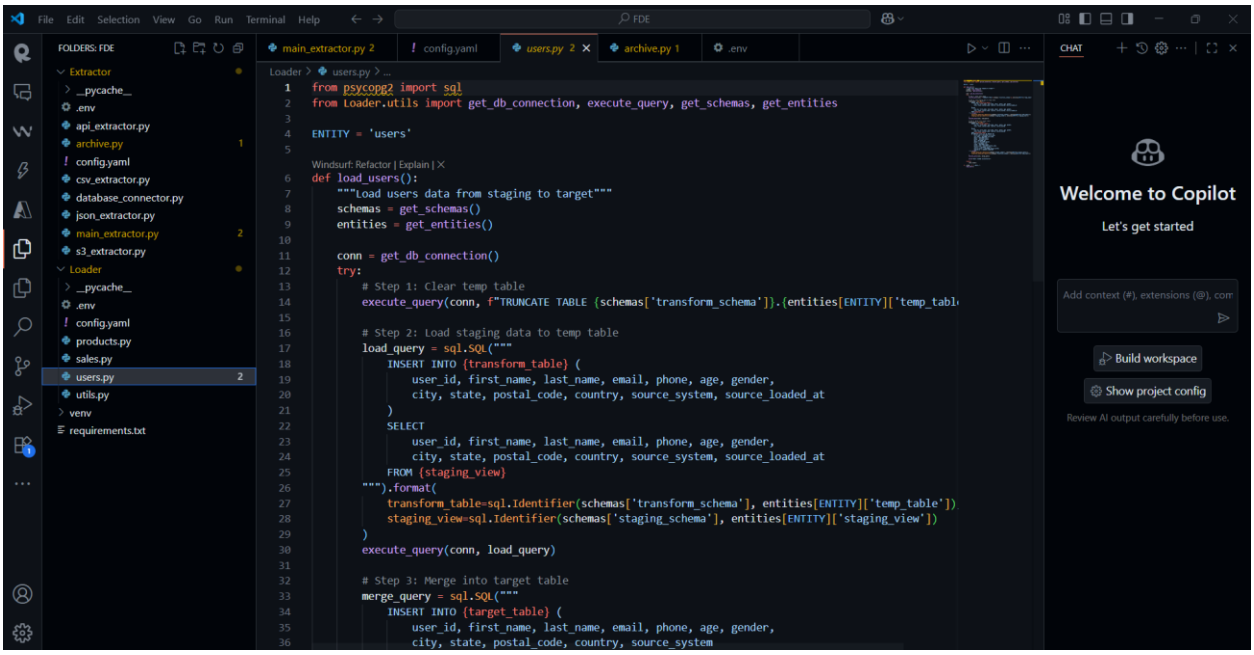


Figure 10 Setup config.yaml




File Edit Selection View Go Run Terminal Help
FDE
FOLDERS: FDE
Extractor
__pycache__
.env
api_extractor.py
archive.py 1
config.yaml
csv_extractor.py
database_connector.py
json_extractor.py
main_extractor.py 2
s3_extractor.py
Loader
__pycache__
.env
config.yaml
products.py
sales.py 2
users.py
utils.py
venv
requirements.txt
main_extractor.py 2
config.yaml
sales.py 2
archive.py 1
.env
Loader > sales.py > ...
from psycopg2 import sql
from Loader.utils import get_db_connection, execute_query, get_schemas, get_entities
ENTITY = 'sales'
Windsurf: Refactor | Explain |
def load_sales():
"""Load sales data from staging to target"""
schemas = get_schemas()
entities = get_entities()
conn = get_db_connection()
try:
# Step 1: Clear temp table
execute_query(conn, f"TRUNCATE TABLE {schemas['transform_schema']}.{entities[ENTITY]['temp_t
# Step 2: Load staging data to temp table
load_query = sql.SQL("""
INSERT INTO {transform_table} (
sale_id, sale_date, store_id, product_id, customer_id,
quantity, unit_price, total_amount, source_system, source_loaded_at
)
SELECT
sale_id, sale_date, store_id, product_id, customer_id,
quantity, unit_price, total_amount, source_system, source_loaded_at
FROM {staging_view}
""").format(
transform_table=sql.Identifier(schemas['transform_schema'], entities[ENTITY]['temp_table
staging_view=sql.Identifier(schemas['staging_schema'], entities[ENTITY]['staging_view'])
)
execute_query(conn, load_query)
# Step 3: Fill missing unit_price from product dimension
fill_price_query = sql.SQL("""
UPDATE {transform_table} t
SET unit_price = p.price
FROM {product_table} p


File Edit Selection View Go Run Terminal Help
FDE
FOLDERS: FDE
main_extractor.py 2
config.yaml
users.py 2
archive.py 1
.env
Loader > users.py > ...
from psycopg2 import sql
from Loader.utils import get_db_connection, execute_query, get_schemas, get_entities
ENTITY = 'users'
Windsurf: Refactor | Explain |
def load_users():
"""Load users data from staging to target"""
schemas = get_schemas()
entities = get_entities()
conn = get_db_connection()
try:
# Step 1: Clear temp table
execute_query(conn, f"TRUNCATE TABLE {schemas['transform_schema']}.{entities[ENTITY]['temp_tabl
# Step 2: Load staging data to temp table
load_query = sql.SQL("""
INSERT INTO {transform_table} (
user_id, first_name, last_name, email, phone, age, gender,
city, state, postal_code, country, source_system, source_loaded_at
)
SELECT
user_id, first_name, last_name, email, phone, age, gender,
city, state, postal_code, country, source_system, source_loaded_at
FROM {staging_view}
""").format(
transform_table=sql.Identifier(schemas['transform_schema'], entities[ENTITY]['temp_table'])
staging_view=sql.Identifier(schemas['staging_schema'], entities[ENTITY]['staging_view'])
)
execute_query(conn, load_query)
# Step 3: Merge into target table
merge_query = sql.SQL("""
INSERT INTO {target_table} (
user_id, first_name, last_name, email, phone, age, gender,
city, state, postal_code, country, source_system
CHAT
Welcome to Copilot
Let's get started
Add context (#), extensions (@), com
Build workspace
Show project config
Review AI output carefully before use.

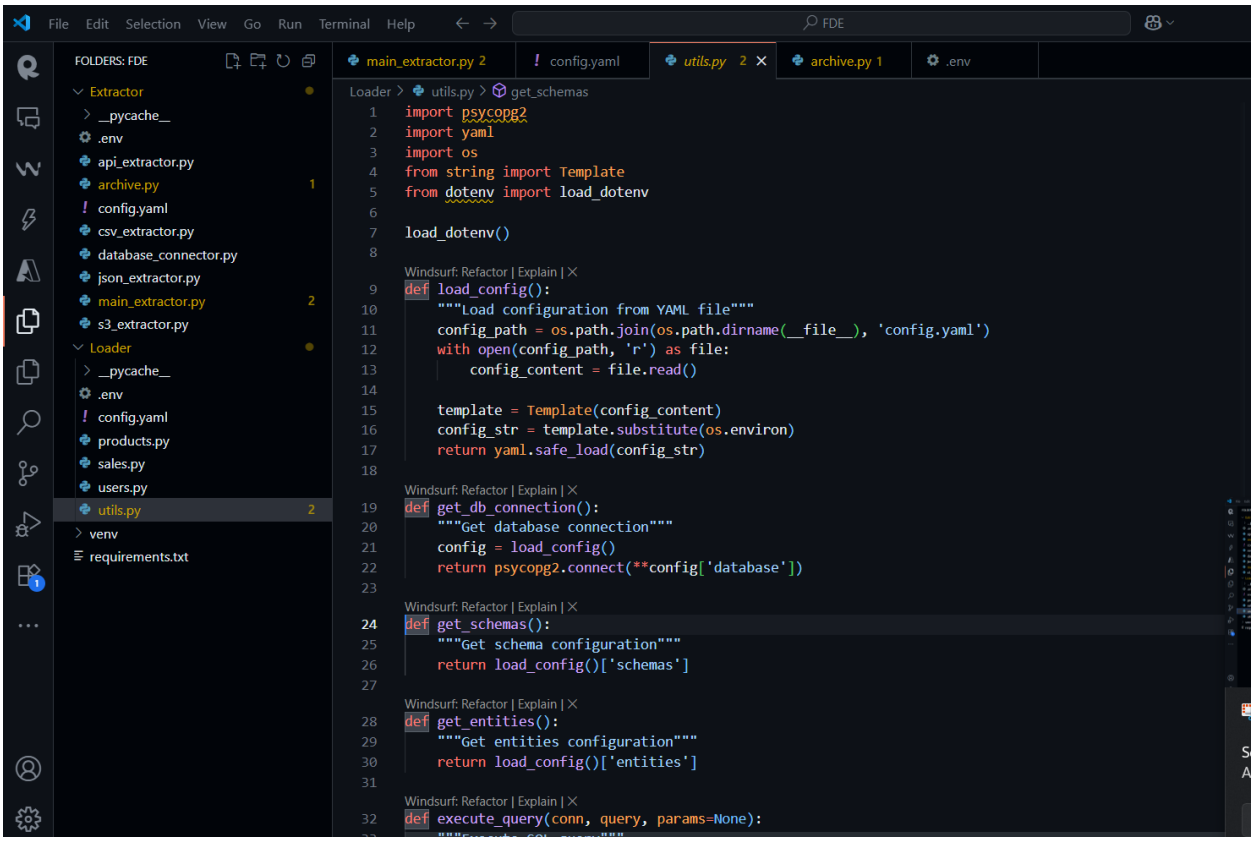


Figure 11 All python files

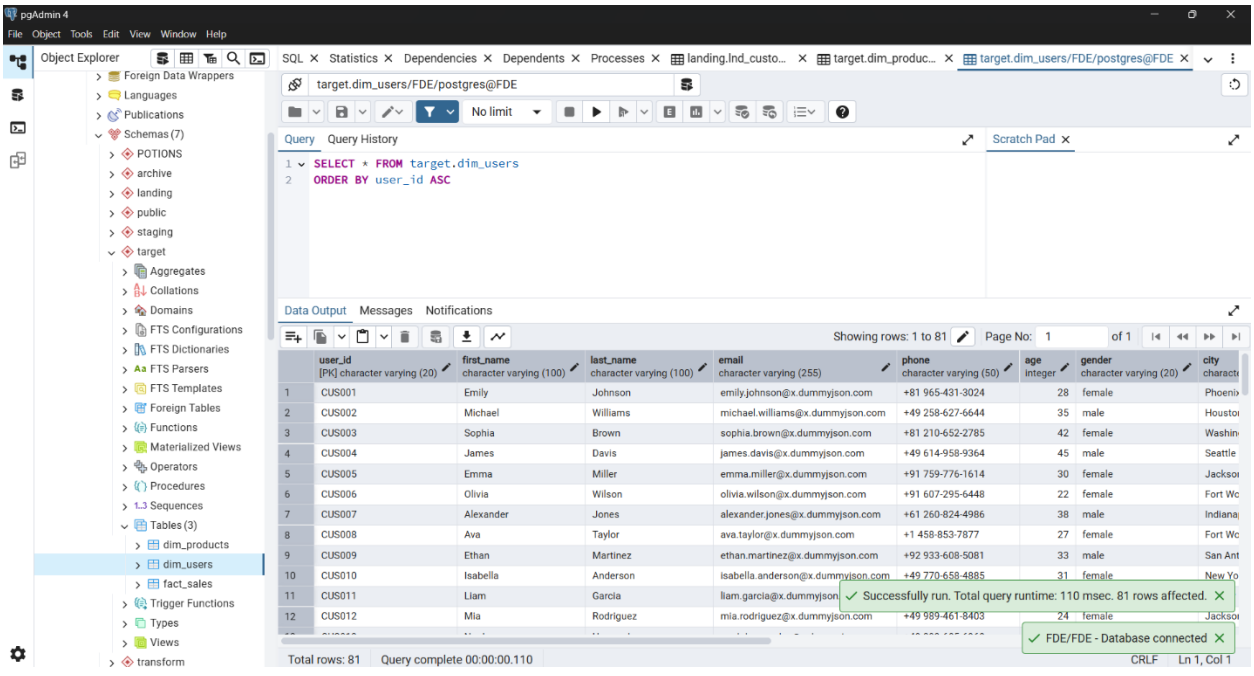


Figure 12 Data after loading

Week 5 – Orchestration with Prefect

Step 11: Orchestration Script (pipeline.py)

- Created orchestration/pipeline.py.
- Imported extract, load, archive tasks from Extractor & Loader modules.
- Configured task retries (3 retries with 30s delay).
- Designed flow sequence: Extract → Load (users & products in parallel) → Load sales → Archive.

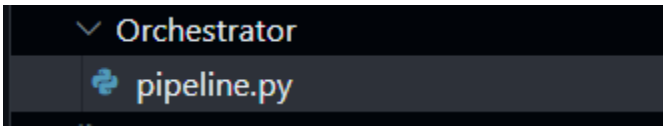


Figure 13 Created Orchestrator and the file

Step 12: Running the Pipeline Locally

- Started Prefect Orion server using:
  → prefect server start
- Set environment variable PREFECT_API_URL.
- Ran pipeline using:
  → python orchestrator/pipeline.py
- Verified execution in Prefect Dashboard (http://127.0.0.1:4200/dashboard).

Step 13: Scheduling the Pipeline

- Deployed pipeline with cron job using:
  → prefect deploy orchestrator/pipeline.py:fde_pipeline --name "fde-pipeline" --cron "13 15 * * *"
- Created a work pool in Prefect.
- Started worker with:
  → prefect worker start --pool 'FDE'
- Verified scheduled deployment in Prefect Deployment tab.
- Observed successful scheduled pipeline run at specified time.

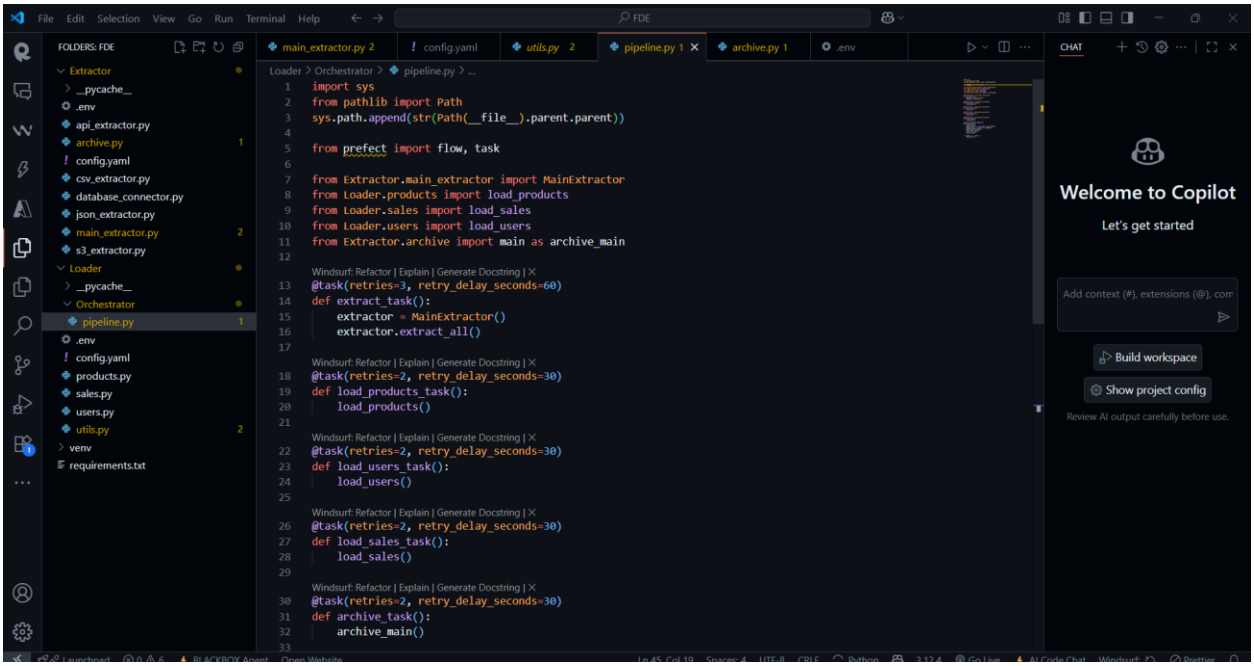


Figure 14 Pipeline.py code

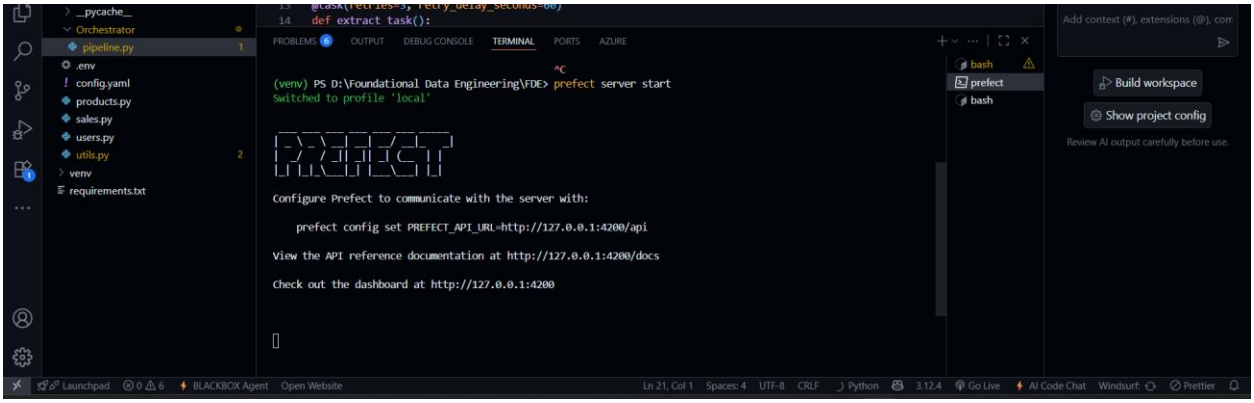
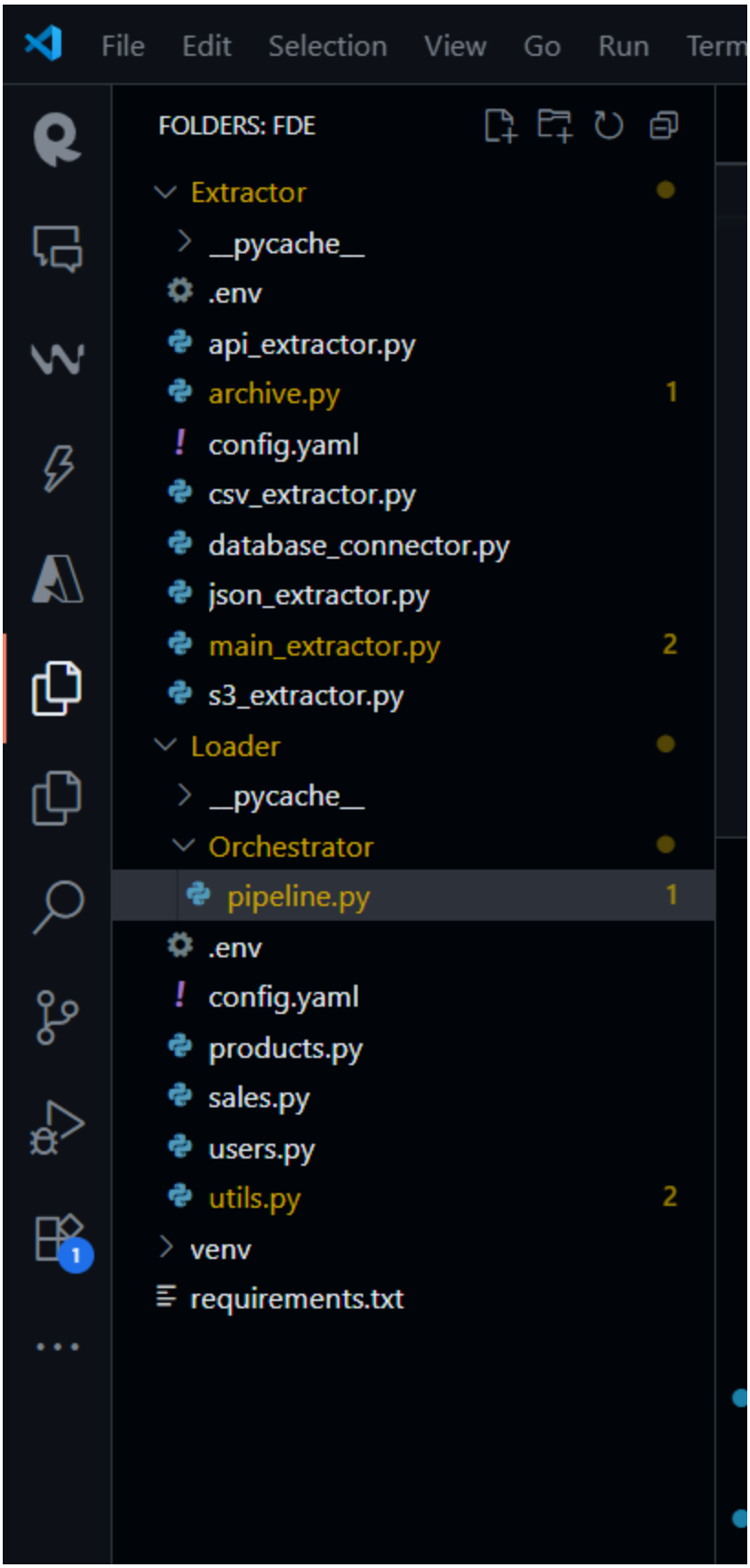


Figure 15 Prefect server start


File
Edit
Selection
View
Go
Run
Term
FOLDERS: FDE
Extractor
__pycache__
.env
api_extractor.py
archive.py 1
config.yaml
csv_extractor.py
database_connector.py
json_extractor.py
main_extractor.py 2
s3_extractor.py
Loader
__pycache__
Orchestrator
pipeline.py 1
.env
config.yaml
products.py
sales.py
users.py
utils.py 2
venv
requirements.txt

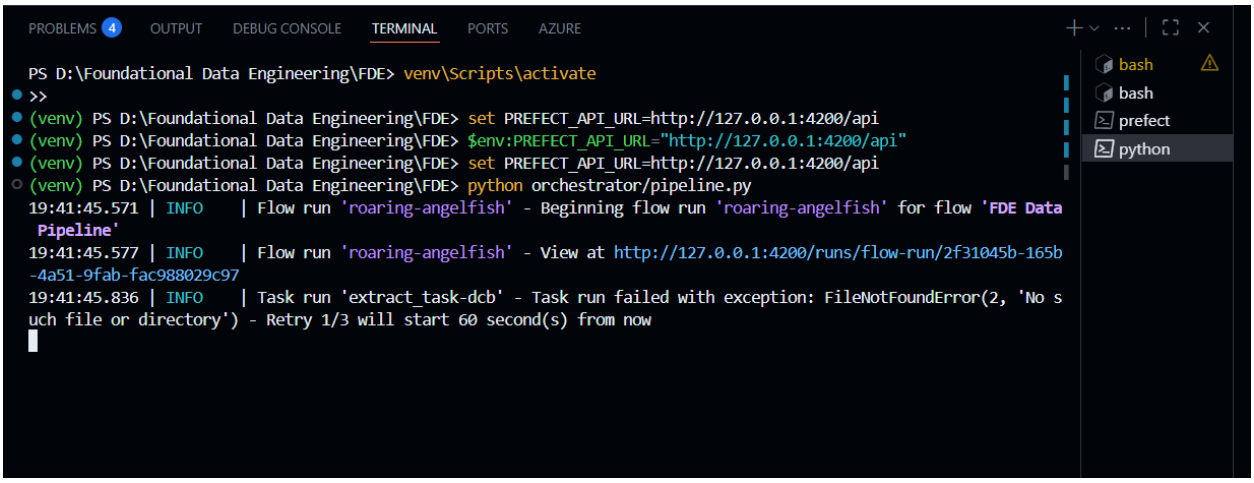


Figure 16 Setting the prefect and then executing

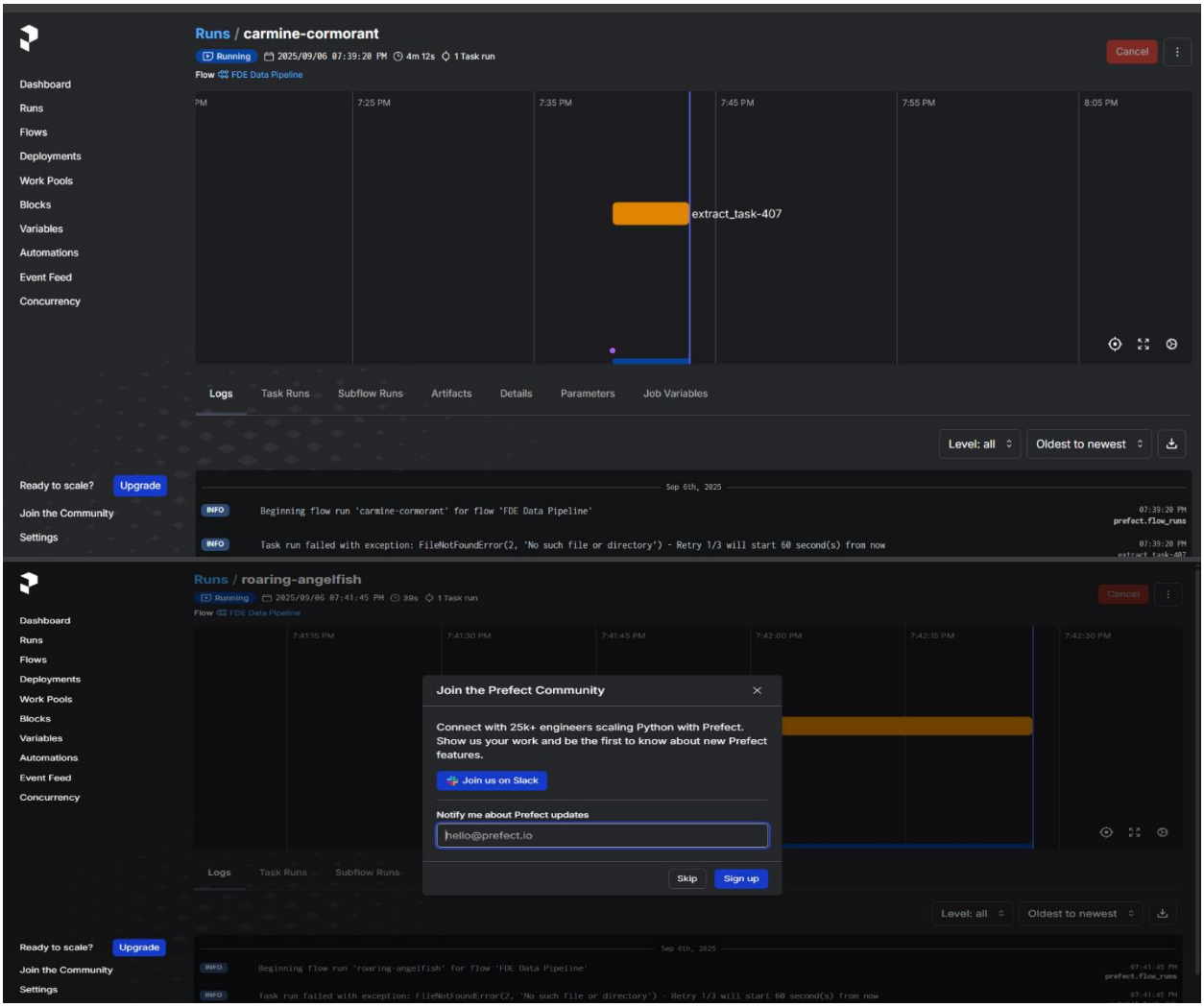


Figure 17 Dashboard

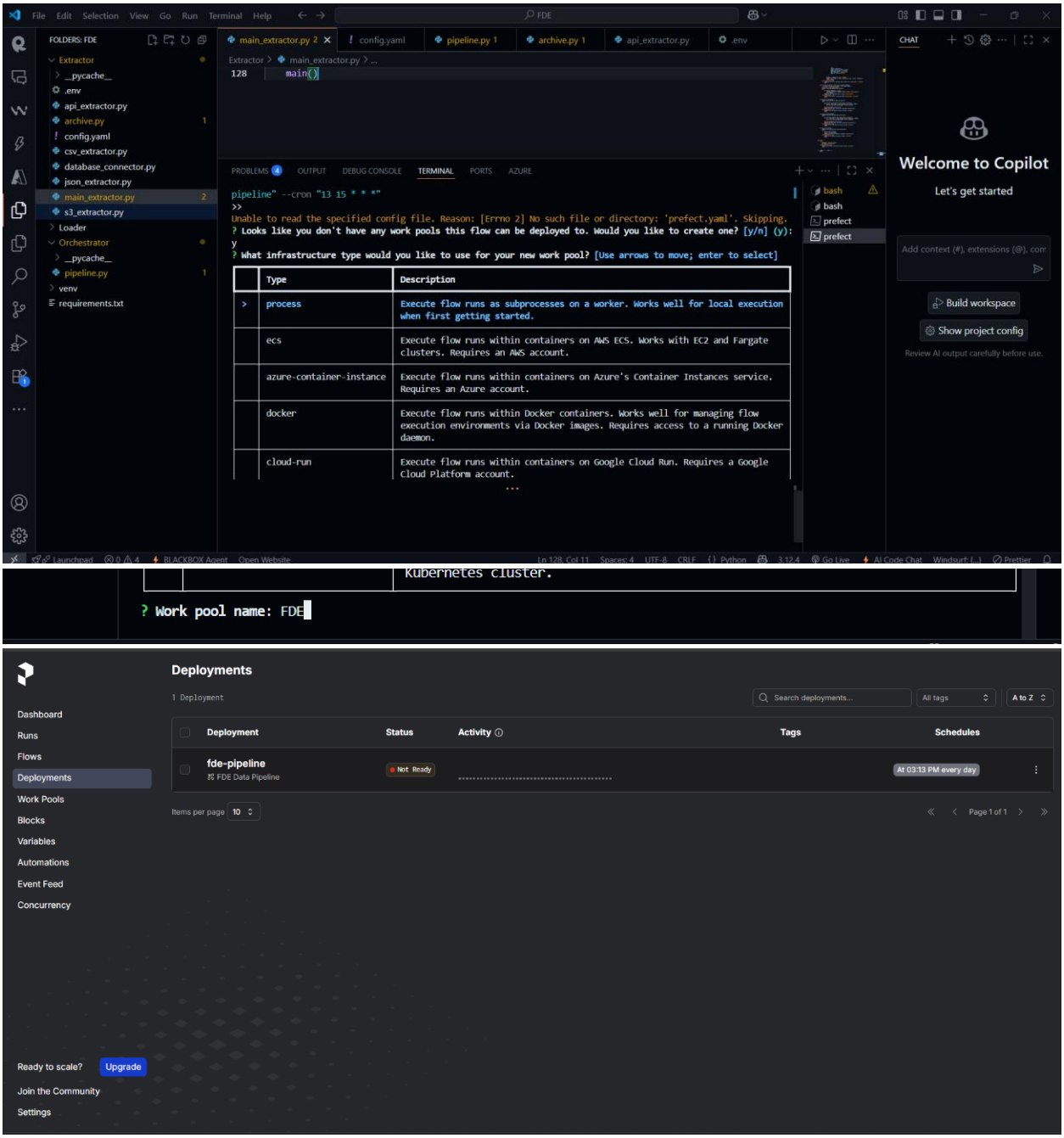


Figure 18 Before not ready

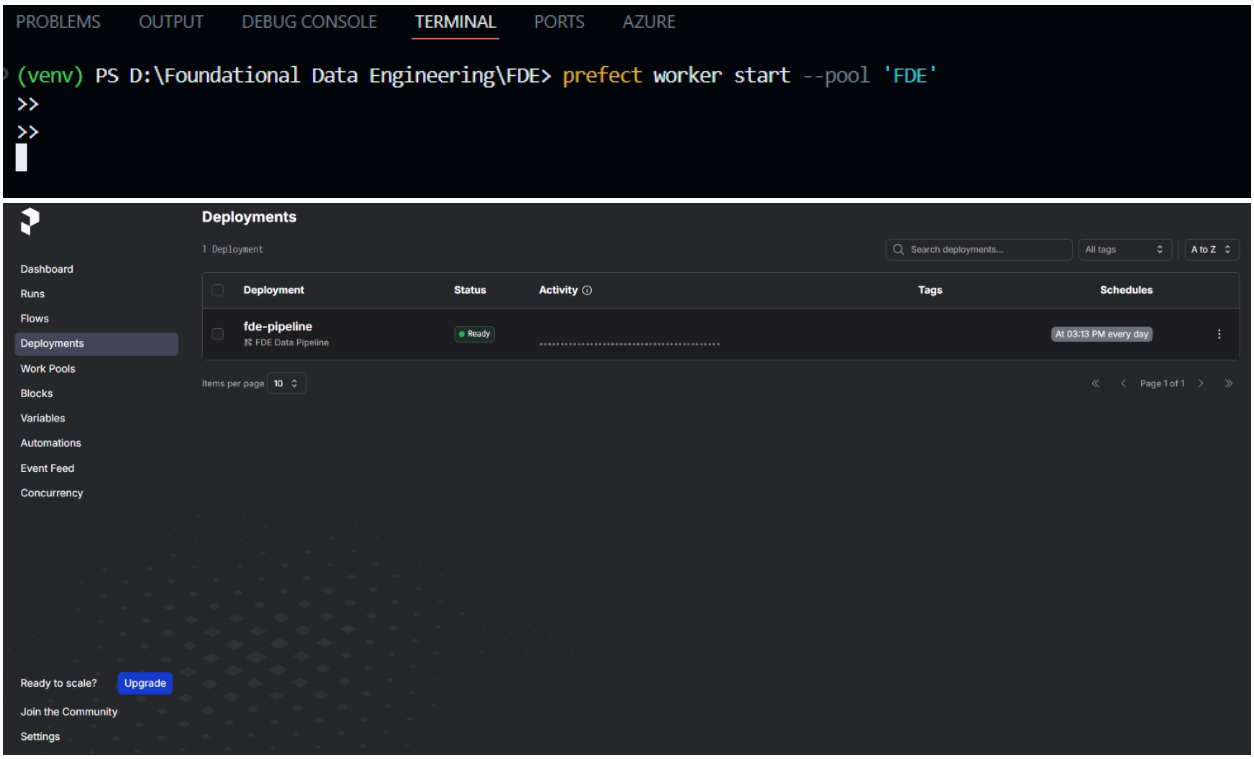
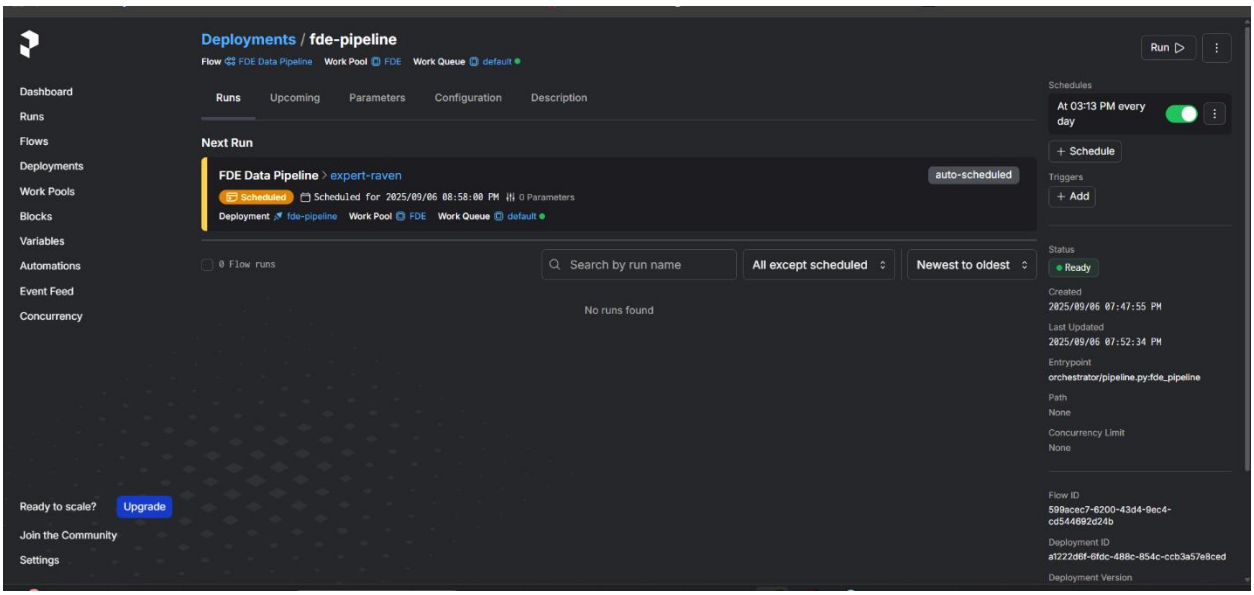


Figure 19 After the deployment



Week 6 – Visualization (Metabase)

Step 14: Metabase Setup

- Installed and configured Metabase.
- Connected PostgreSQL (FDE database) as a data source.

Step 15: Dashboard Creation

- Created charts and visualizations:
- Waste requests by category
- Daily/weekly task completion trend
- Total processed waste vs pending
- Top contributors (citizens/organizations)

Step 16: Final Dashboard

- Combined charts into a Metabase dashboard.
- Took screenshots for submission.

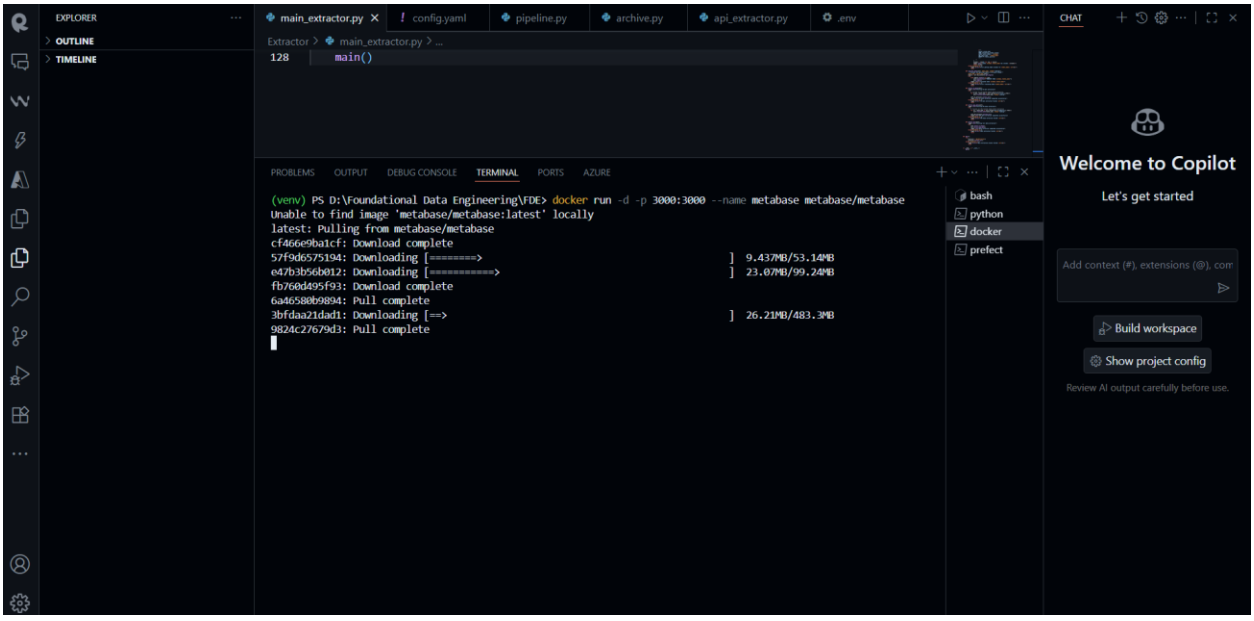


Figure 20 Setting the docker

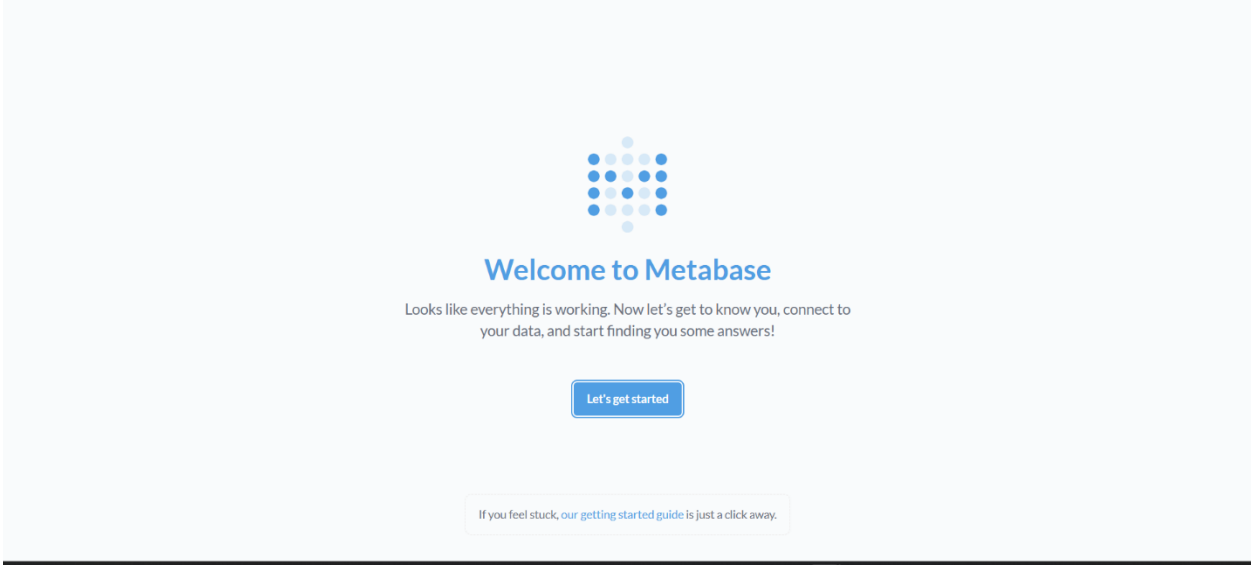


Figure 21 Metabase dashboard

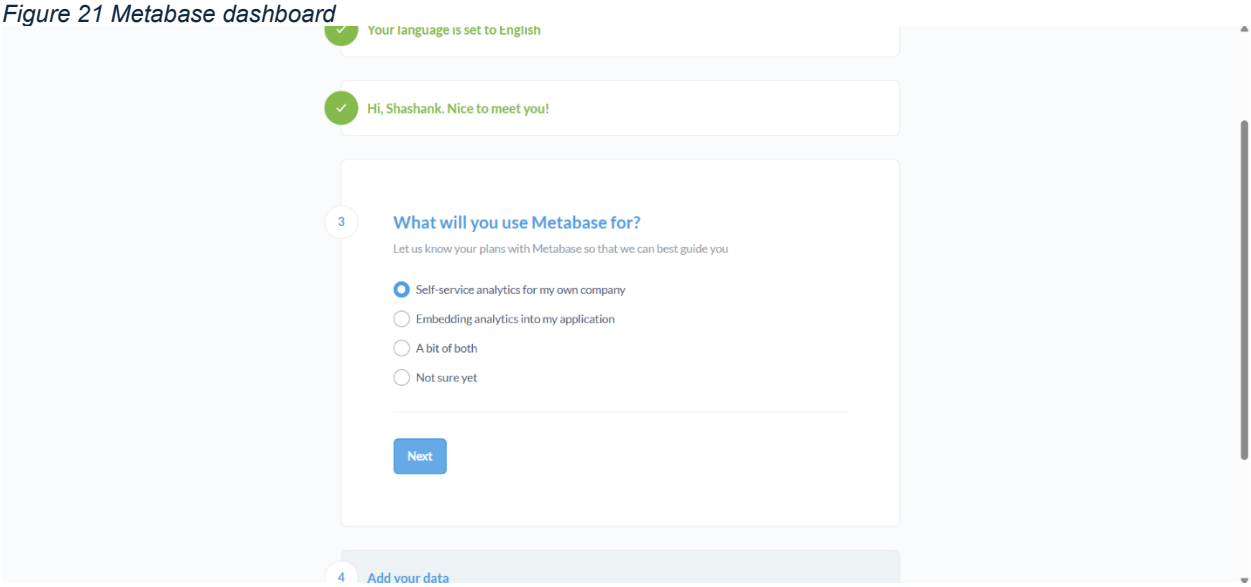
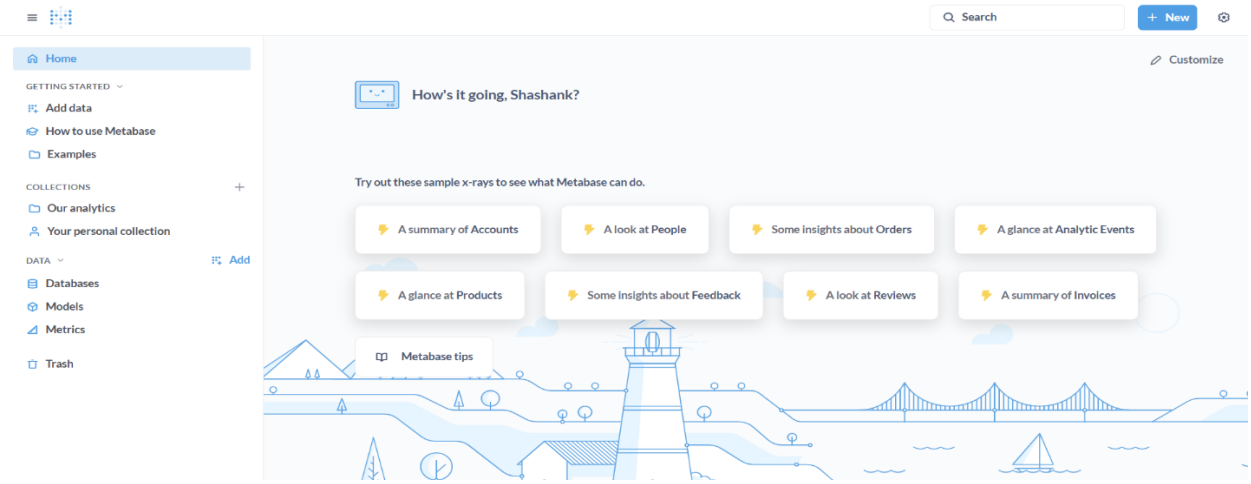


Figure 22 Setting the metadashboard



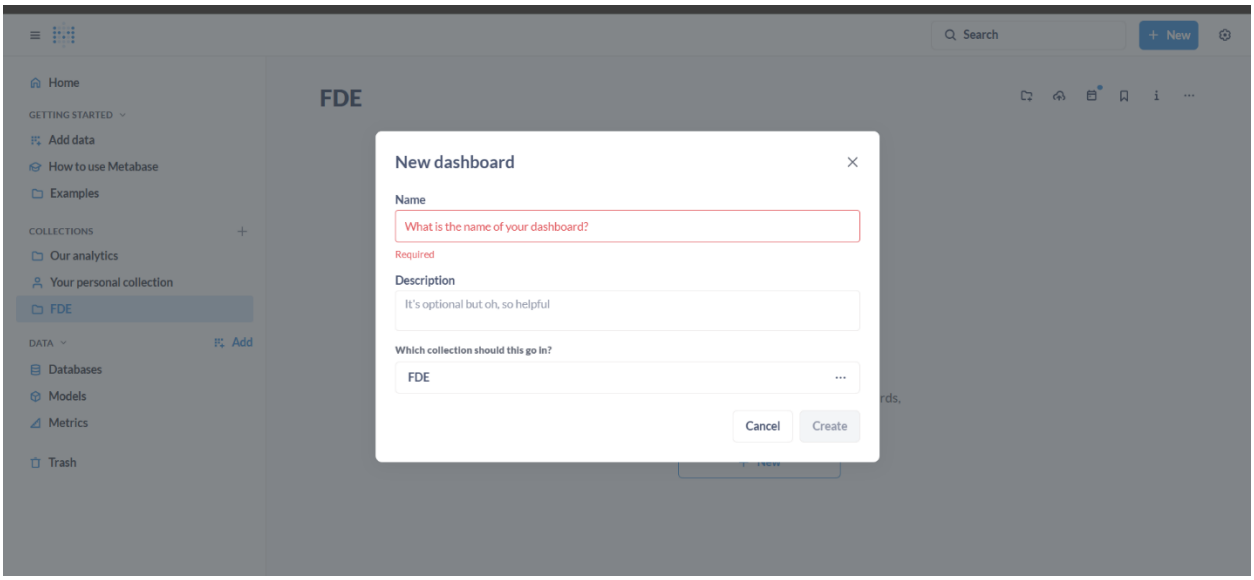


Figure 23 New dashboard for FDE

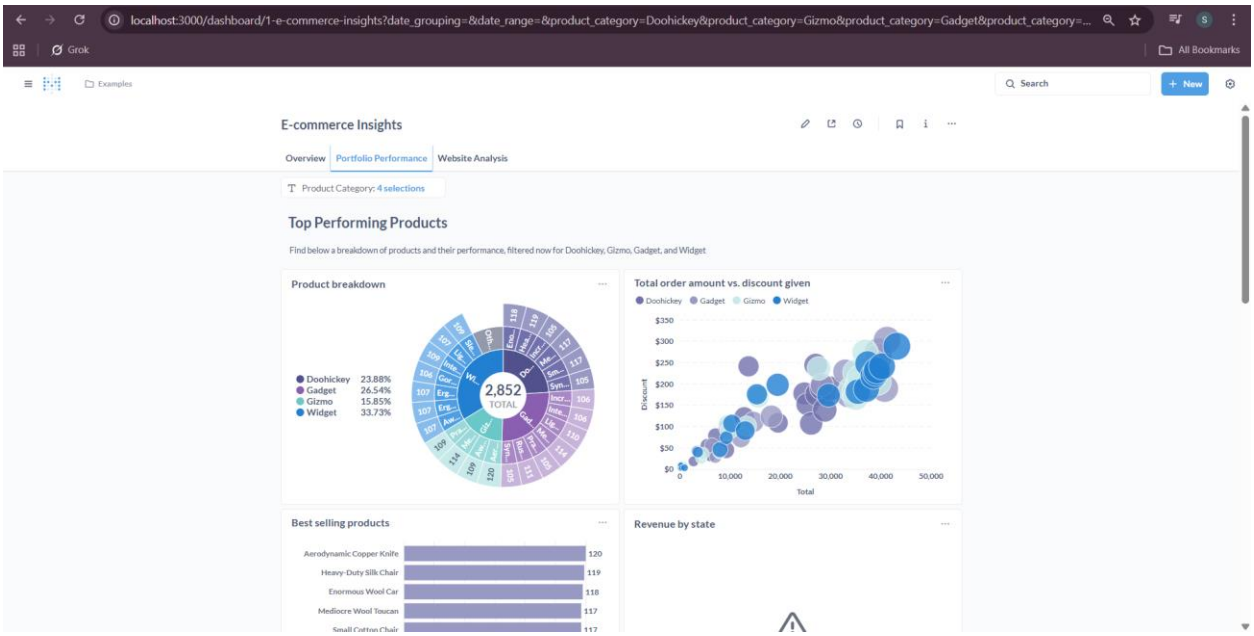


Figure 24 After completion, the graph of the table

Conclusion

The stepwise implementation of the Foundational Data Engineering (FDE) pipeline provided comprehensive learning experience in building an end-to-end data workflow. Beginning with the setup of AWS S3 for raw data ingestion and PostgreSQL as a data warehouse, the project systematically progressed through extraction, transformation, loading, orchestration, and visualization.

In Week 1 and 2, the focus was on establishing the environment, configuring PostgreSQL, and developing extractor modules for CSV, JSON, S3, and API data. This ensured that diverse data sources could be ingested reliably into the landing schema. In Week 4, the emphasis shifted to loading processes, where dimension and fact tables were carefully designed with utilities for data quality checks, dedication, and UPSERT operations. This stage ensured that data was both accurate and analysis ready.

Week 5 introduced orchestration with Prefect, which automated the ETL pipeline, implemented retries for resilience, and scheduled flows for periodic execution. This eliminated manual intervention and aligned the pipeline with industry-standard practices. Finally, in Week 6, the pipeline output was connected to Metabase for visualization. The resulting dashboards transformed raw data into actionable insights, enabling monitoring of trends, performance, and decision-making metrics.

Overall, this project demonstrated how a well-structured data engineering pipeline can transform raw, heterogeneous data into a clean, reliable, and continuously updated analytical resource. Beyond fulfilling the module’s objectives, the experience has built a practical understanding of modern data engineering tools such as PostgreSQL, Python (pandas, SQLAlchemy, psycopg2), Prefect, and Metabase. The project reflects not only technical implementation but also the importance of automation, data quality assurance, and visualization in enabling organizations to derive value from data.