Level 7 - AI & Machine Learning Analyst

Module 14

Data Pipelines

PROJECTING SUCCESS

SAFEGUARDING

DEPUTY DESIGNATED SAFEGUARDING OFFICER: Jackie Collins

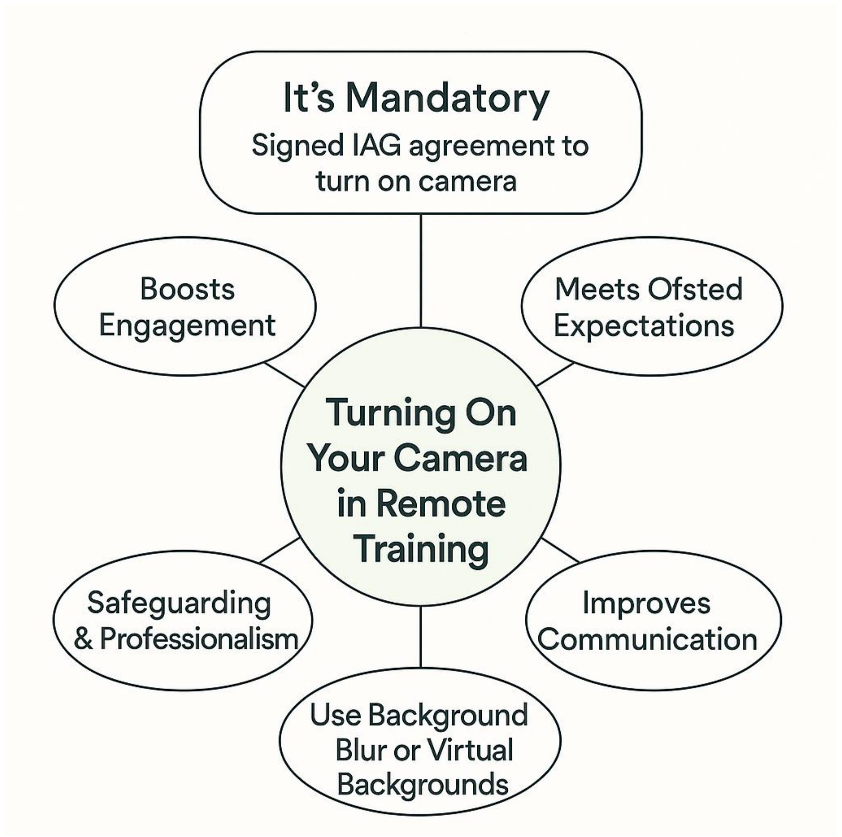
EMAIL: JackieC@projectingsuccess.co.uk

DEPUTY DESIGNATED SAFEGUARDING OFFICER: Scott Owens

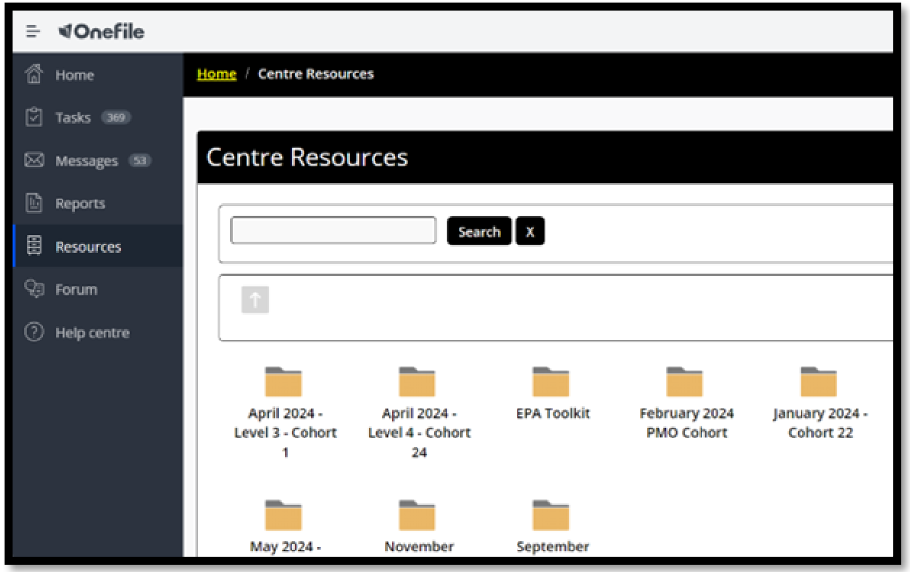
EMAIL: ScottO@projectingsuccess.co.uk

SAFEGUARDING FORM: Click Here

Cameras


It's Mandatory
Signed IAG agreement to turn on camera
Boosts Engagement
Meets Ofsted Expectations
Turning On Your Camera in Remote Training
Safeguarding & Professionalism
Improves Communication
Use Background Blur or Virtual Backgrounds

Resources - OneFile & Teams



Centre Resources

April 2024 - Level 3 - Cohort 1

April 2024 - Level 4 - Cohort 24

EPA Toolkit

February 2024 PMO Cohort

January 2024 - Cohort 22

May 2024 -

November

September

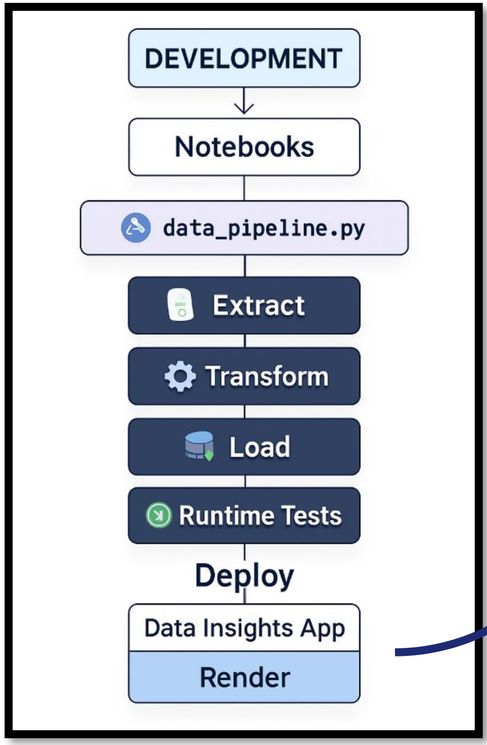
OneFile > Resources > Your cohort > Module X

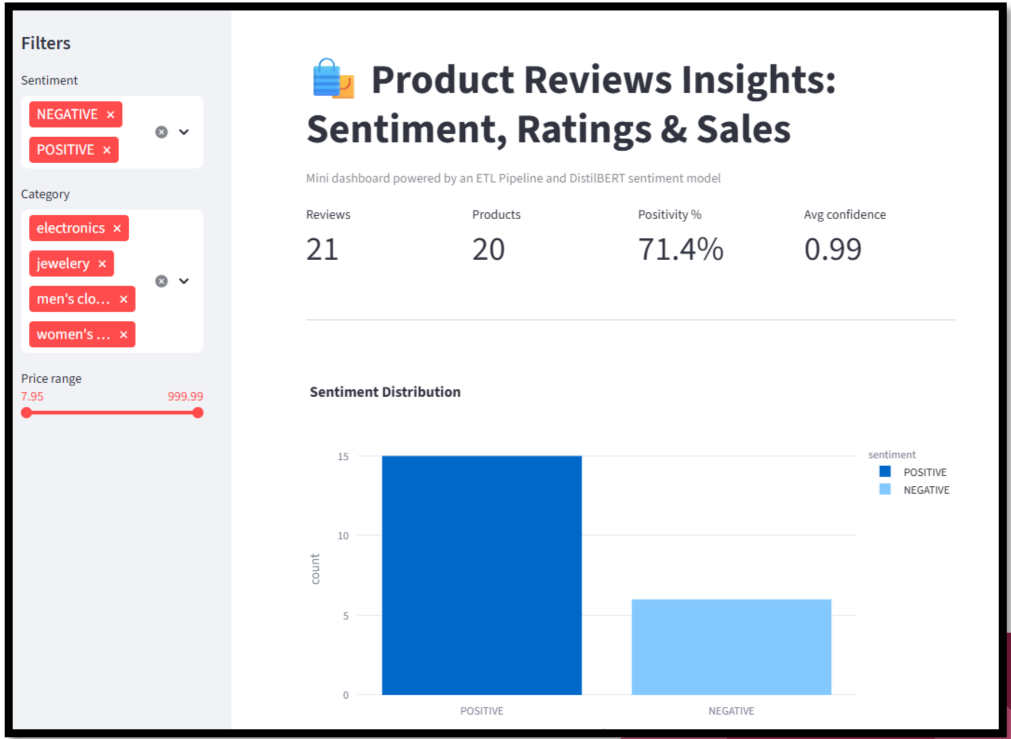
Learning objectives

At the end of this module, you should be able to:

- Explain and implement each stage of the ETL lifecycle
- Combined and integrate data from different sources
- Build a basic ML pipeline on top of the ETL output

Data Pipeline Project: ETL for Retail Analytics





Filters

Sentiment: NEGATIVE ×, POSITIVE ×

Category: electronics ×, jewelery ×, men's clo… ×, women's … ×

Price range: 7.95 – 999.99

Product Reviews Insights: Sentiment, Ratings & Sales

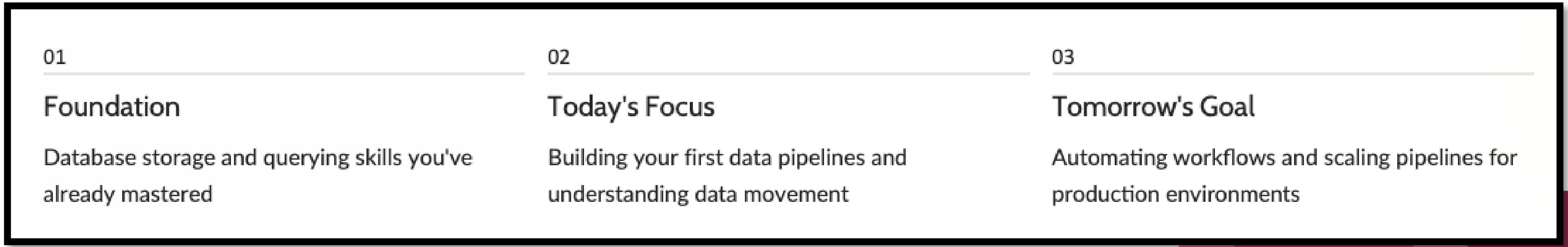
Mini dashboard powered by an ETL Pipeline and DistilBERT sentiment model

Reviews	Products	Positivity %	Avg confidence
21	20	71.4%	0.99

Sentiment Distribution

From Databases to Data Pipelines

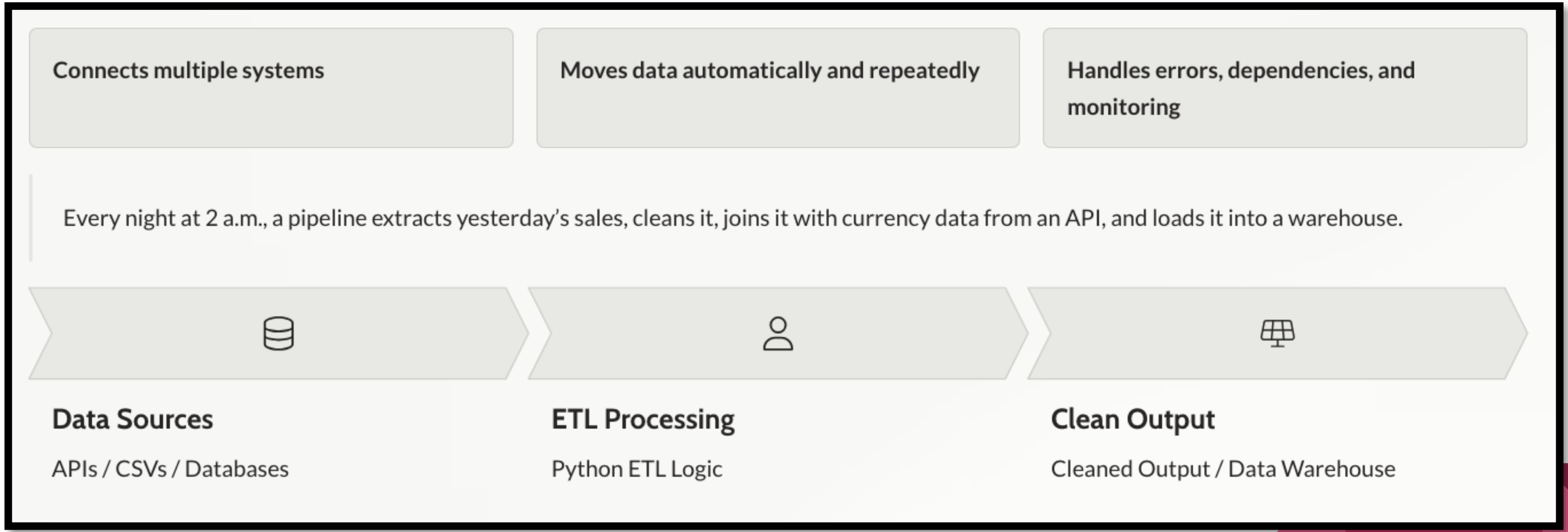
- Data pipelines are the backbone of modern data systems, they are the invisible infrastructure that powers everything from business intelligence dashboards to machine learning models. Without reliable pipelines, even the most sophisticated database becomes isolated and limited in its usefulness.



01	02	03
Foundation	Today's Focus	Tomorrow's Goal
Database storage and querying skills you've already mastered	Building your first data pipelines and understanding data movement	Automating workflows and scaling pipelines for production environments

What is a Data Pipeline?

A Data pipeline moves data from Source to Destination



Connects multiple systems

Moves data automatically and repeatedly

Handles errors, dependencies, and monitoring

Every night at 2 a.m., a pipeline extracts yesterday's sales, cleans it, joins it with currency data from an API, and loads it into a warehouse.

Data Sources

APIs / CSVs / Databases

ETL Processing

Python ETL Logic

Clean Output

Cleaned Output / Data Warehouse

A data pipeline automates the flow of data, ensuring it's consistent, clean, and available when needed

ETL Is the Heart of a Data Pipeline

Step	Purpose	Example
Extract	Get data from source	Read CSV, API, or DB
Transform	Clean & enrich	Fix missing values, merge, compute metrics
Load	Save for use	Write to database or file

Hands-On Project: Building your first production pipeline

Project Goal

Design and implement a data pipeline that combines product, sales, and customer review data to produce a curated dataset suitable for analytics or AI model input.

Practice 1: Data pipeline Hands-on Lab

Open script: data_pipeline_prototype.ipynb

From Prototype to Production: Enterprise Ready Data Solutions

- While agile environments as Jupyter notebooks are invaluable for rapid prototyping and data exploration, transitioning to production demands a robust, structured approach. Enterprise data solutions require pipelines to operate with unwavering reliability, repeatability, and collaborative efficiency, underpinned by well-organised repositories

- **Future proof scalability**

Moving beyond "one big script" to modular, well-defined components ensures your data infrastructure can effortlessly scale with growing data volumes and evolving business needs, supporting long-term strategic growth.

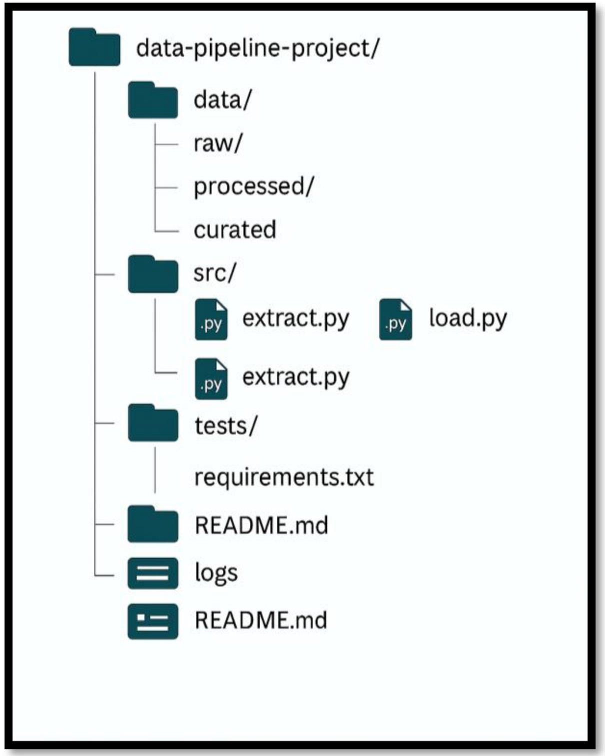
.

Clarity, Collaboration, and Control

A clear repository layout ensures:

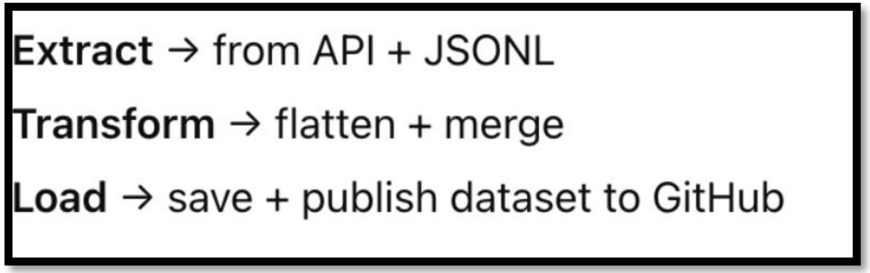
- Everyone knows where code, data, and configs live
- Changes can be tracked with Git
- Components can be reused and tested independently
- Without structure, pipelines become “one big script”, hard to debug, automate, or hand over.

Data pipeline repository structure



```
data-pipeline-project/
├── data/
│   ├── raw/
│   ├── processed/
│   └── curated
├── src/
│   ├── extract.py    load.py
│   └── extract.py
├── tests/
│   requirements.txt
├── README.md
└── logs
    README.md
```

Src/



Extract → from API + JSONL

Transform → flatten + merge

Load → save + publish dataset to GitHub

Extract Data

In your previous notebook, you originally wrote all your extraction steps inline.

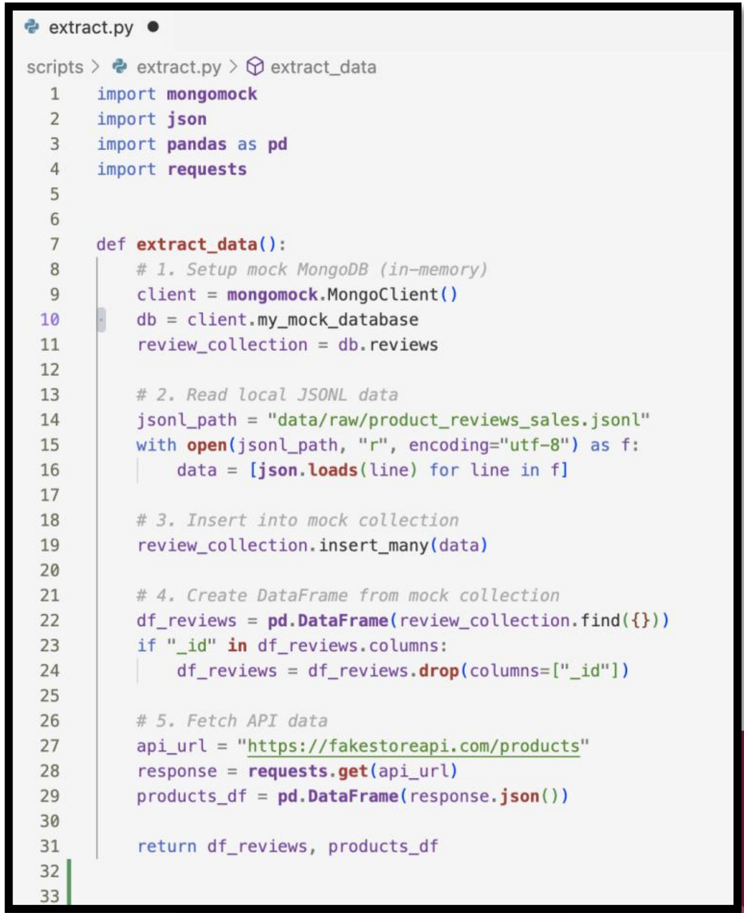
I have refactored that notebook logic into a reusable Python module called extract.py.

Refactoring the notebook code into extract.py gives us:

-Cleaner scripts. Notebooks stay focused on testing, debugging, and explaining, not holding all pipeline logic.

-Reusable extraction logic. The same function (extract_data()) can now be used by: our pipeline script, automated jobs, tests.

-A real project structure. This mirrors what professional data engineering teams do: **organise ETL steps into separate, testable modules.**


extract.py ●

scripts > extract.py > extract_data

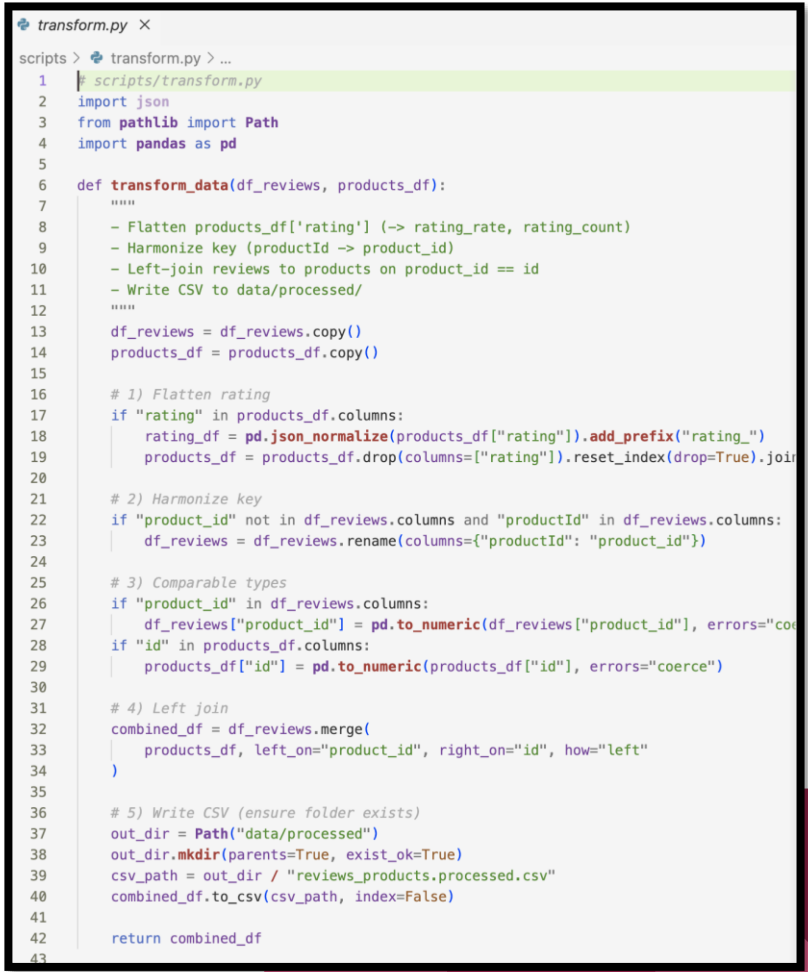
```python
import mongomock
import json
import pandas as pd
import requests


def extract_data():
    # 1. Setup mock MongoDB (in-memory)
    client = mongomock.MongoClient()
    db = client.my_mock_database
    review_collection = db.reviews

    # 2. Read local JSONL data
    jsonl_path = "data/raw/product_reviews_sales.jsonl"
    with open(jsonl_path, "r", encoding="utf-8") as f:
        data = [json.loads(line) for line in f]

    # 3. Insert into mock collection
    review_collection.insert_many(data)

    # 4. Create DataFrame from mock collection
    df_reviews = pd.DataFrame(review_collection.find({}))
    if "_id" in df_reviews.columns:
        df_reviews = df_reviews.drop(columns=["_id"])

    # 5. Fetch API data
    api_url = "https://fakestoreapi.com/products"
    response = requests.get(api_url)
    products_df = pd.DataFrame(response.json())

    return df_reviews, products_df

```

Transform Data

- The transform.py script cleans and prepares both datasets (reviews and API products) so they can be reliably joined.

- It flattens nested JSON fields (such as product ratings), standardises column names and types, ensures product_id keys match across sources, and merges everything into one combined dataset.

- Finally, it outputs a clean, processed CSV ready for analysis or loading into the next pipeline stage.

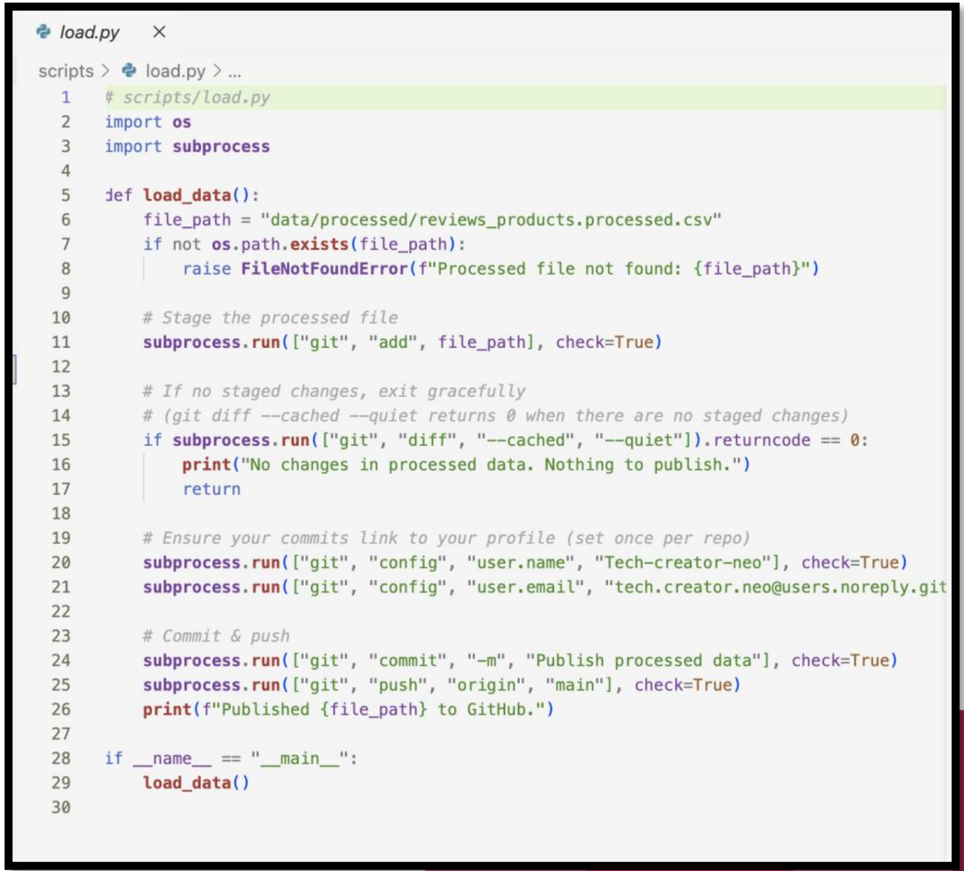


```python
# scripts/transform.py
import json
from pathlib import Path
import pandas as pd

def transform_data(df_reviews, products_df):
    """
    - Flatten products_df['rating'] (-> rating_rate, rating_count)
    - Harmonize key (productId -> product_id)
    - Left-join reviews to products on product_id == id
    - Write CSV to data/processed/
    """
    df_reviews = df_reviews.copy()
    products_df = products_df.copy()

    # 1) Flatten rating
    if "rating" in products_df.columns:
        rating_df = pd.json_normalize(products_df["rating"]).add_prefix("rating_")
        products_df = products_df.drop(columns=["rating"]).reset_index(drop=True).joi

    # 2) Harmonize key
    if "product_id" not in df_reviews.columns and "productId" in df_reviews.columns:
        df_reviews = df_reviews.rename(columns={"productId": "product_id"})

    # 3) Comparable types
    if "product_id" in df_reviews.columns:
        df_reviews["product_id"] = pd.to_numeric(df_reviews["product_id"], errors="coe
    if "id" in products_df.columns:
        products_df["id"] = pd.to_numeric(products_df["id"], errors="coerce")

    # 4) Left join
    combined_df = df_reviews.merge(
        products_df, left_on="product_id", right_on="id", how="left"
    )

    # 5) Write CSV (ensure folder exists)
    out_dir = Path("data/processed")
    out_dir.mkdir(parents=True, exist_ok=True)
    csv_path = out_dir / "reviews_products.processed.csv"
    combined_df.to_csv(csv_path, index=False)

    return combined_df
```

Load Data

- The load.py script handles the loading and publishing stage of the pipeline. It checks that the processed CSV exists, stages the file with Git, commits it with a message, and pushes it to GitHub.

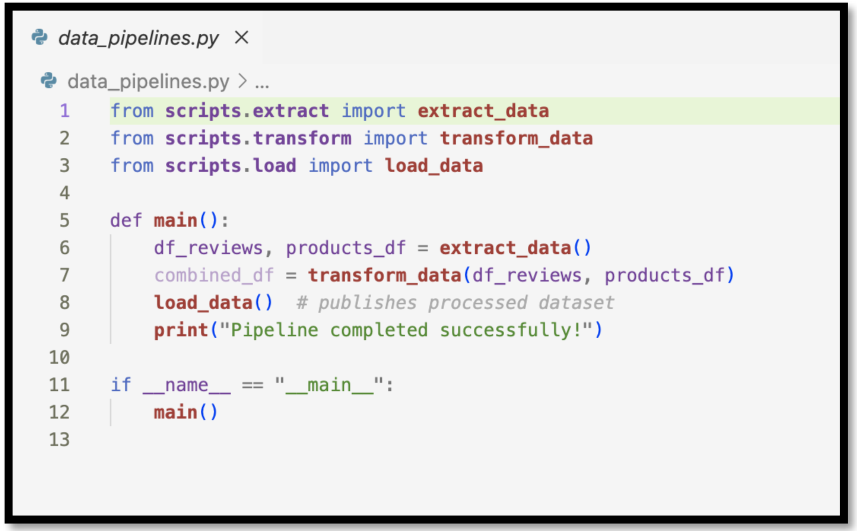
- This simulates a real-world deployment step, where cleaned data is automatically published to a repository or downstream system. **It completes the final step of the ETL workflow by making the processed dataset available for others to consume.**



```python
# scripts/load.py
import os
import subprocess

def load_data():
    file_path = "data/processed/reviews_products.processed.csv"
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"Processed file not found: {file_path}")

    # Stage the processed file
    subprocess.run(["git", "add", file_path], check=True)

    # If no staged changes, exit gracefully
    # (git diff --cached --quiet returns 0 when there are no staged changes)
    if subprocess.run(["git", "diff", "--cached", "--quiet"]).returncode == 0:
        print("No changes in processed data. Nothing to publish.")
        return

    # Ensure your commits link to your profile (set once per repo)
    subprocess.run(["git", "config", "user.name", "Tech-creator-neo"], check=True)
    subprocess.run(["git", "config", "user.email", "tech.creator.neo@users.noreply.git

    # Commit & push
    subprocess.run(["git", "commit", "-m", "Publish processed data"], check=True)
    subprocess.run(["git", "push", "origin", "main"], check=True)
    print(f"Published {file_path} to GitHub.")

if __name__ == "__main__":
    load_data()

```

- **data_pipeline.py** orchestrates the entire ETL pipeline by linking all three modular steps into one runnable workflow. It calls:

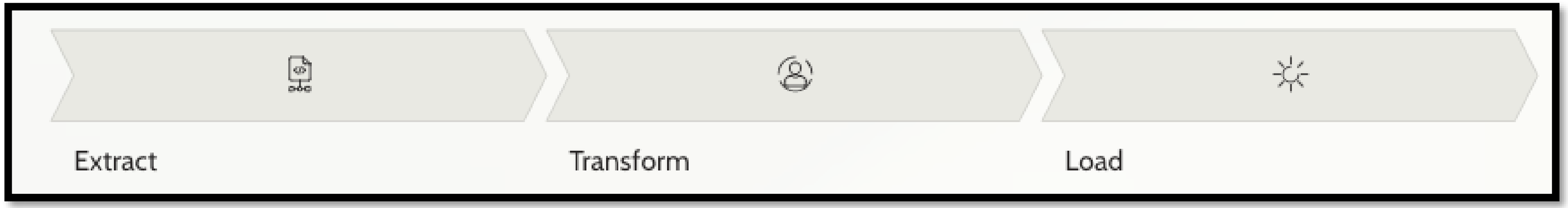
1. **extract_data()** → to pull data from the API and local files
2. **transform_data()** → to clean and merge the datasets
3. **load_data()** → to publish the final processed output



```python
from scripts.extract import extract_data
from scripts.transform import transform_data
from scripts.load import load_data

def main():
    df_reviews, products_df = extract_data()
    combined_df = transform_data(df_reviews, products_df)
    load_data()  # publishes processed dataset
    print("Pipeline completed successfully!")

if __name__ == "__main__":
    main()
```

This script acts as the pipeline controller, showing a realistic example of how ETL components are stitched together into a single executable process.
It simulates what would later be scheduled by Airflow, cron, or CI/CD.

Adding Prediction Model to ETL Pipeline



Where Prediction model fits in the pipeline?

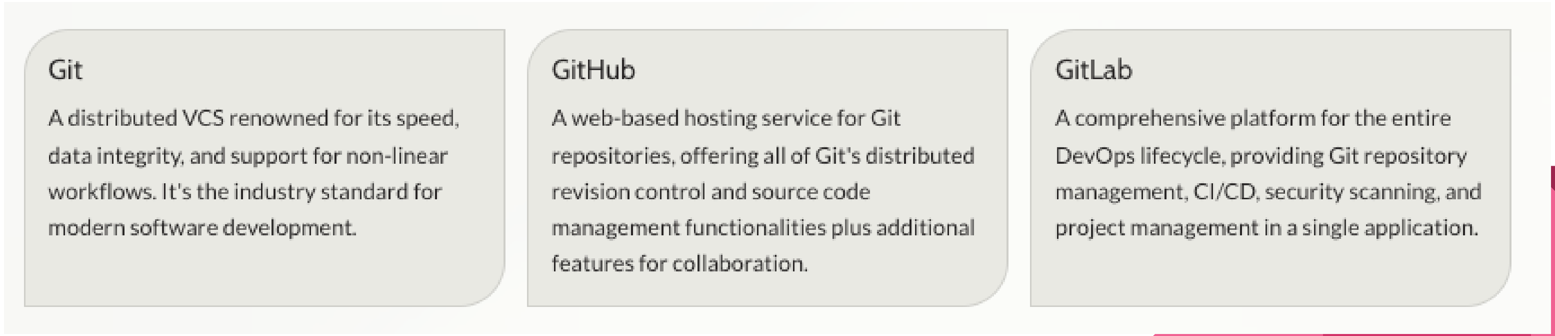
Version Control: Enterprise Code Management & Collaboration

- Version Control Systems (VCS) are indispensable tools that manage changes to documents, source code, websites, and other collections of information. For enterprises, VCS transcends mere change tracking; it's about safeguarding intellectual property, streamlining development workflows, and ensuring consistent, high-quality product delivery.

What is Version Control?

- A system that records changes to a file or set of files over time so that you can recall specific versions later. It's the "undo button" for your entire project, providing a complete history of every change.

- **Popular Enterprise Version Control Systems**



Git

A distributed VCS renowned for its speed, data integrity, and support for non-linear workflows. It's the industry standard for modern software development.

GitHub

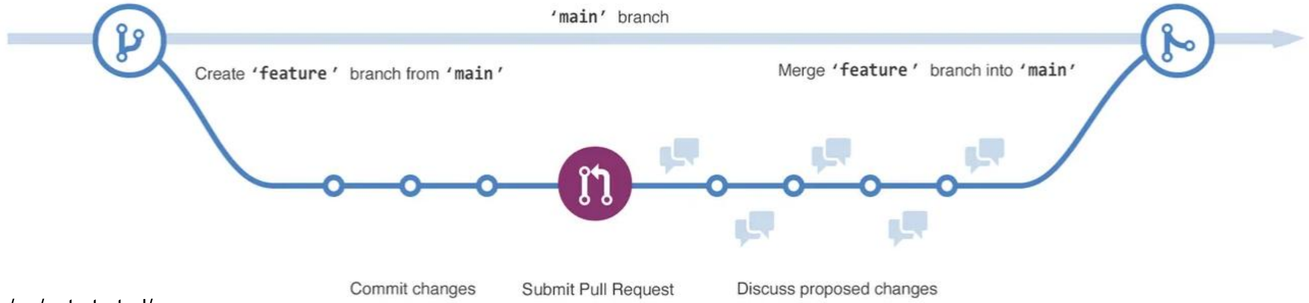
A web-based hosting service for Git repositories, offering all of Git's distributed revision control and source code management functionalities plus additional features for collaboration.

GitLab

A comprehensive platform for the entire DevOps lifecycle, providing Git repository management, CI/CD, security scanning, and project management in a single application.

Why use Git?

- **Track changes:** every edit you make is saved as a version (commit).
- **Collaboration:** multiple people can work on the same project at once without overwriting each other's work.
- **Backup:** your project can exist in multiple repositories (local and remote).
- **Branching & merging:** safely experiment with new features without affecting the main code.



https://docs.github.com/en/get-started/

Commits

- On GitHub, **saved changes are called commits**.

- Each commit has an associated commit message, which is a description explaining why a particular change was made. Commit messages capture the history of your changes so that other contributors can understand what you have done and why.

Pull requests

- Pull requests are the heart of collaboration on GitHub.
- When you open a pull request, **you are proposing your changes and requesting that someone review and pull in your contribution and merge them into their branch.**
- Pull requests show differences, of the content from both branches. The changes, additions, and subtractions are shown in different colours.

Cloning

- git clone is used to copy (download) an existing Git repository from a remote source (like GitHub, GitLab, or Bitbucket) onto your local machine.

- It does two things:

1. Downloads all the repository files (code, commits, branches, etc.).
2. Sets up a link to the remote repo so you can pull, push, and fetch changes later.

Summary steps

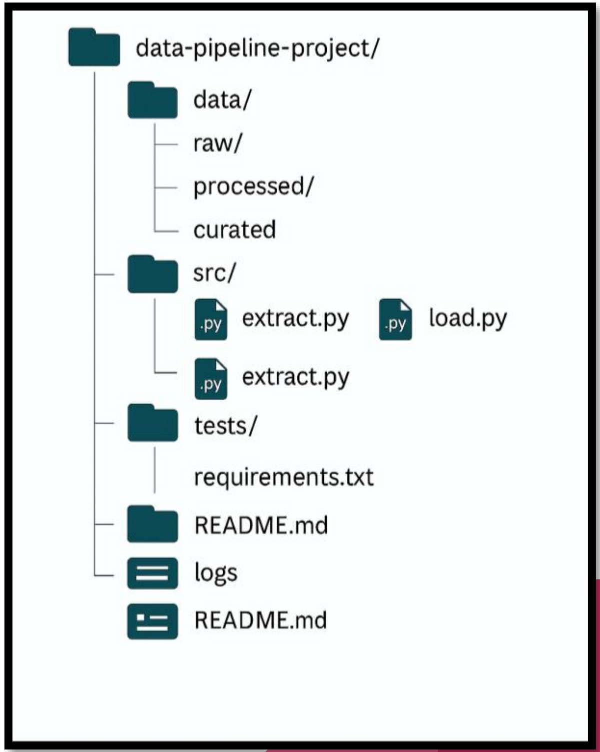
Step	Action	Purpose
1	Create feature branch	Work independently without breaking main code
2	Commit changes	Save and document progress
3	Submit PR	Request to merge work into main
4	Discuss changes	Review and improve collaboratively
5	Merge to main	Deploy or release stable code

Practice 2: Practice Exercise

Open the document: practice2_exercise.pdf

From Prototype to Production

- Now we have a working prototype and set up GitHub, we can start building the data pipeline repository structure.



Practice 3: Guided practice

Follow your instructor to build the Data pipeline repository

Note: this is a step-by-step process, don’t jump ahead otherwise you might break the flow of the pipeline.

Adding Data Insights

- Now we have build an ETL pipeline ready for production. We will add on top of our ETL backend, a clean, one-page dashboard showing how customers feel about products and how that relates to price, category, and ratings.

- This dashboard will be built based on the data you got from your pipeline.

Practice 4: Independent practice

Open the document: practice4_dataInsights.pdf

Why every data team needs ETL pipelines?

- Ensures data quality and consistency
- Enables automation and repeatability
- Feeds clean data to AI, ML, and BI systems
- Reduces manual data wrangling

Thank you!