1️⃣ DATA INGESTION

Easy Problem

You are provided with two input files containing customer data:

- `customers_A.csv`
- `customers_B.xlsx`

Both files represent the same logical business entity (customer master records), but **they use different column names** and **may store fields in a different order**.

Your task is to build a Python-based ingestion script that:

1. Reads both files into DataFrames.
2. Standardizes the column names using a JSON configuration file where:

```
{
  "standard_name": "input_column_name"
}
```

3. Combines both files into a single consolidated dataset.
4. Adds a column named `source_file` indicating the filename from which each record originated.
5. Prints **total number of records ingested**.
6. Saves the result as `customers_combined.csv`.

Requirements

- The script must be **scalable** — additional files should be processable **without modifying core logic**.
- Adding a new file type (another CSV / Excel) should require updating only configuration JSON.

Medium Problem

You are provided with three employee datasets in different formats:

- `employees.csv`
- `employees.xlsx`
- `employees.json`

Each contains employee profiles but uses **different schemas** and **different field names**.

Develop a Python ingestion pipeline that:

1. Automatically **detects file type** based on extension.
2. Extracts data from each file.
3. Normalizes datasets into a standard structure using external mapping JSONs.
4. Merges all datasets into one unified DataFrame.
5. Logs ingestion metadata (file name, type, timestamp, row count) and stores it in:
    - `ingestion_log.json` or `ingestion_log.csv`

Requirements

- No hard-coded column names or file paths in logic.
- Entire pipeline must be configuration-driven.
- Adding a new file format (e.g., XML/parquet) should require modifying only config files.

Hard Problem

You must ingest **product pricing** data from:

- A local CSV file
- A local Excel file

- A live REST API endpoint returning JSON

Build a **scheduled updating pipeline** that:

1. Runs every 60 seconds automatically.
2. Fetches data from all three sources.
3. Handles failures (API timeouts, schema mismatch, empty files) without stopping the pipeline.
4. Stores ingestion results into a historical metadata store containing:

   ```
   timestamp, source_type, record_count, status, runtime_ms
   ```

Requirements

- API responses must be saved locally in `raw_api_responses/`.
- The system must be **scalable**, so additional data sources can be added without modifying code — only through configuration.

2️⃣ DATA STORAGE

Easy Problem

Load a customer dataset (`customers.csv`) into SQLite by:

1. Creating a database and a table named `customers`.
2. Automatically detecting and assigning appropriate data types.
3. Running SQL queries via Python:
   - Top customers by spending
   - Customers grouped by city
4. Printing results to console.

Goal: Demonstrate storing and retrieving structured data.

Medium Problem

You are given three files representing relational datasets:

- `customers.csv`
- `orders.csv`
- `products.csv`

Build a Python script that:

1. Automatically loads each file into SQLite as separate tables.
2. Generates table names dynamically from filenames.
3. Creates necessary indexes (e.g., `customer_id`, `product_id`).
4. Measures performance difference between queries with and without indexes.

Goal: Understand indexing and query optimization.

Hard Problem

Implement a **mini Data Warehouse** inside SQLite using Star Schema:

- Fact table: `FactOrders`
- Dimensions: `DimCustomer`, `DimProduct`, `DimDate`

Write a pipeline that:

1. Loads dimension and fact tables from raw CSV sources.
2. Implements **incremental loads** so new records append without rewriting the full dataset.
3. Version controls updates on dimension values using timestamps.

3️⃣ DATA MODELING

Easy Problem

Given a flattened dataset `orders_flat.csv` containing:

```
order_id, order_date, customer_name, customer_email, product_name, price, quantity
```

Normalize into:

- `Customers`
- `Products`
- `Orders`

and create foreign keys correctly.

Medium Problem

Design and build a **Star Schema** for sales analytics:

- `FactSales`
- Dimensions: `DimCustomer`, `DimProduct`, `DimStore`, `DimDate`

Generate surrogate keys and show at least one analytical query.

Hard Problem

Implement **Slowly Changing Dimension (SCD Type 2)** for customer records:

- Track changes to fields like address, phone, email.

- Maintain historical rows using:

```
valid_from, valid_to, is_current
```

4️⃣ ETL / ELT PIPELINE

Easy Problem

Write a script that:

1. Extracts a CSV
2. Cleans it (null handling, type conversion)
3. Saves output to a new file

Medium Problem

Create a step-based ETL system:

```
extract() → transform() → load()
```

Add logging for runtime, row counts, and error handling.

Hard Problem

Create a **fully configuration-driven** ETL system controlled via config JSON:

```
{
  "sources": [...],
  "transformations": [...],
  "destination": "sqlite"
}
```

No code must change when source files change — only configuration.

5️⃣ DATA TRANSFORMATION

Easy Problem

Clean messy ecommerce data by:

- Removing duplicates
- Fixing date formats
- Formatting names consistently

Medium Problem

Merge sales datasets from three regions with different currencies:

- Standardize field names
- Convert currency using exchange rates file
- Merge into a unified dataset

Hard Problem

Build a **transformation engine** using JSON rules:

```
rename, merge_columns, split_column, calculated_fields, type_conversion
```

Pipeline must apply transformations dynamically.

6️⃣ DATA INTEGRATION

Easy Problem

Combine multiple monthly CSV files with identical structure into one master dataset.

Medium Problem

(This is your completed project)

Multiple clients provide multiple files containing similar data but with different column names and formats. Each client also provides a **data dictionary** specifying final table column requirements.

You must:

- Read all client source files
- Use mapping JSON to align their column names to final table column names
- Generate standardized output tables (ft01, ft02, etc.)
- Include a `client` column to track data origin
- Save merged outputs as unified final CSV files

Requirements

- Fully scalable to any number of clients and tables

- No code changes required for adding new clients or new source tables
- Only mappings and config files should change

Hard Problem

Automatically infer schema mapping using similarity/fuzzy matching, and generate a mapping JSON for human approval.

7️⃣ BATCH VS STREAMING

Easy Problem

Read a large CSV in fixed-size chunks (100 rows each) and process incrementally, printing progress counters.

Medium Problem

Simulate a streaming pipeline by:

- Reading rows one-by-one from a file
- Pausing 1 second between reads
- Maintaining running sales totals

Hard Problem

Merge real-time streaming updates with historical batch files into a continuously updated final table.

8️⃣ ORCHESTRATION / SCHEDULING

Easy Problem

You are given a basic ETL workflow consisting of three Python functions:

```
extract()
transform()
load()
```

Write a script that executes them **in a defined sequence**, ensuring that:

- `transform()` executes only after `extract()` has completed successfully.
- `load()` executes only after `transform()` has completed successfully.
- If any function fails, pipeline execution should stop and print a failure message.
- Print a summary of execution order and status.

Goal: Understand how multi-step pipelines are orchestrated sequentially.

Medium Problem

Convert the previous ETL workflow into a **scheduled pipeline** that runs automatically every minute using Python's `schedule` library (or cron).

Requirements:

- Store pipeline execution history in a `pipeline_log.csv` file.
- Log details must include:

```
timestamp, run_id, status(success/failure), execution_time_seconds, rows_proce
```

- Old executions must not be overwritten — append new rows.
- Pipeline must print next scheduled runtime and should not block keyboard interrupt.

Goal: Learn how to automate recurring workloads.

Hard Problem

Build a **task orchestration system** where each task has a dependency configuration defined in JSON, e.g.:

```
{
  "tasks": [
    {"name": "extract", "depends_on": []},
    {"name": "transform", "depends_on": ["extract"]},
    {"name": "load", "depends_on": ["transform"]},
    {"name": "archive", "depends_on": ["load"]}
  ]
}
```

Requirements:

- Run tasks according to dependency graph (DAG-like execution).
- If any task fails, retry **3 times** before marking failure.
- Send a notification message to console when a failure happens.
- Pipeline must be scalable: adding a new task should not require code changes, only JSON edits.

9️⃣ METADATA & DATA LINEAGE

Easy Problem

Create a script that records metadata for every dataset ingested. For each input file, store:

```
file_name, file_format, timestamp, record_count, processing_time_ms
```

Save metadata into a file named `metadata_log.json`.

Goal: Understand metadata capture for audit.

Medium Problem

Build a metadata tracking system that documents lineage between data layers. Assume three storage layers:

- `raw/`
- `staging/`
- `final/`

Each transformation step must create a metadata record documenting:

```
input_file_name, output_file_name, transformation_description, row_count_before, r
```

Generate and save a lineage report showing each file's transformation journey.

Goal: Understand traceability across stages.

Hard Problem

Implement **column-level lineage tracking**. For each output column, record:

```
which input column(s) contributed,
transformation applied,
confidence or reliability notes.
```

Output must be saved to a structured JSON lineage map, example:

```
{
  "final_column_name": {
    "source_columns": ["column1", "column3"],
    "transformation": "merge + upper()"
  }
}
```

Goal: ability to trace what transformations produce which final data fields.

🔟 DATA QUALITY & VALIDATION

Easy Problem

Write a script that performs basic data quality checks on an input file:

- Detect and count duplicate rows
- Detect and count null values per column
- Identify invalid values (e.g., negative price, age > 120)
- Output a summary report in text format

Medium Problem

Create a **rule-based validation system** where rules are stored in a JSON file. Example:

```
{
  "age": {"min": 0, "max": 120},
  "email": {"regex": ".+@.+"},
  "price": {"min": 0}
}
```

Pipeline should:

- Apply rules dynamically
- Generate a `quality_report.csv`
- Include `status: pass/fail` per rule and overall dataset health score

Hard Problem

Build a **data quality firewall**:

- Assign individual weights to validation rules
- Calculate overall quality score (0–100)
- If score < threshold, block ETL process outputs and send an alert
- Store detailed failure report with failing rows in `rejected_records/`

1️⃣1️⃣ DATA GOVERNANCE & SECURITY

Easy Problem

Write a script that masks sensitive information in a dataset:

- Replace phone number middle digits with `XXXXX`
- Partially mask email addresses: `a***@gmail.com`
- Ensure original values are not recoverable

Medium Problem

Implement column-level encryption and decryption for fields such as `email`, `phone`, or `customer_id` using a secure symmetric encryption key stored in a `.env` file.

Constraints:

- Encryption code must not store the key inside source code
- Produce encrypted output and verify decryptibility

Hard Problem

Implement **row-level access control**:

- Admin should see full records
- Analyst should see masked records
- Intern should see aggregated data only
- Determine access based on a `user_role` parameter

1️⃣2️⃣ MONITORING & OBSERVABILITY

Easy Problem

Add execution time measurement logging into an existing ETL pipeline. Print:

```
start_time, end_time, total_execution_seconds
```

Medium Problem

Write structured log records for each ETL step including:

```
timestamp, step, status, rows, memory_usage, errors
```

Store logs in `pipeline_logs.log` formatted JSON-like.

Hard Problem

Build a **real-time terminal dashboard** that shows live:

- Current status of each ETL step
- Execution times
- Failure alerts
- SLA progress bar (e.g., should finish within 30 sec)

1️⃣3️⃣ SCALABILITY & PERFORMANCE

Easy Problem

Process a 1M-row CSV using:

- Standard Pandas load
- Pandas chunk loading (e.g., chunksize=50k)

Compare performance.

Medium Problem

Parallelize processing across multiple files using Python multiprocessing. Measure processing time vs. sequential.

Hard Problem

Split a single large dataset into chunks, process each chunk in separate processes, merge results efficiently and validate correctness.

1️⃣4️⃣ VERSION CONTROL

Easy Problem

Initialize a Git repo for ETL project and commit initial scripts.

Medium Problem

Implement Git branching strategy:

- `main`, `dev`, feature branches
- Pull requests and merge workflow

Hard Problem

Add semantic versioning to pipeline releases:

- Format: `major.minor.patch` (e.g. `2.1.5`)
- Maintain CHANGELOG.md describing every release
- Rollback pipeline version if errors detected

🎉 Completed