BOSCH

Project Documentation : Vehicle Data ETL Pipeline

Proposed solution for the **EM/BDO-P Data Engineer Assessment**
to extract, process, and load data from public sources.

Francisco Pereira

18th September 2025

Introduction

This project is a modular Python pipeline that automates the extraction, transformation, and loading (ETL) of multiple vehicle-related datasets into an Azure SQL database.

It fetches data from three external sources:

Source	Description
Fuel Economy	Vehicle details, emissions, and MPG (mile per gallon) summary and detailed information per vehicle
National Highway Traffic Safety Administration	Safety ratings, recalls, complaints, inspection stations
Alternative Fuel Stations	Alternative fuel stations across the US

The pipeline processes the data into structured pandas DataFrames, writes intermediate results into CSV files for testing, and finally loads the processed data into SQL staging tables.

Folder Structure

The project is structured around a **main.py** file, which coordinates the execution of objects and logic defined in multiple supporting modules. The diagram shows how these files are organized within the project, along with the additional folders that are created and manipulated during execution.

```
bosch-challenge/
│
├── extracted_data/             # Raw CSVs saved after extraction
├── processed_data/             # Cleaned CSVs saved after processing
├── extracted_data_schemas/     # Auto-generated JSON schemas for extracted files
├── processed_data_schemas/     # Auto-generated JSON schemas for processed files
├── sql_scripts/                # CREATE TABLE scripts for Azure SQL
│
├── utils/                      # Python modules for each stage
│   ├── fuel_economy_async.py         # Defines FuelEconomyETL
│   ├── highway_safety_admin_async.py # Defines SafetyAdministrationETL
│   ├── alternative_fuel_async.py     # Defines AlternativeFuelETL
│   ├── data_processing.py            # Defines Processing
│   ├── data_loading.py               # Defines Loading
│   ├── schema_producer.py            # Generates schemas for files
│   └── alternative_fuel_schema.json  # Schema for Alternative Fuel API
│
├── main.py                     # Entrypoint to orchestrate ETL pipeline
├── requirements.txt            # Project dependencies
├── connection_config.json      # Database credentials/config
```

Each module defines a class for one stage of the pipeline, and all classes share a common structure: a **run_all** method that orchestrates the operations of that stage.

Setup and Running the Pipeline

To execute the program, it is required to install some python libraries defined in the requirement.txt file:

1) Open Command Prompt and navigate to your project directory:

```
cd C:\Users\.....bosch-challenge
```

2) Create a virtual environment ('bosch'):

```
python -m venv bosch
```

3) Activate the environment (Windows):

```
bosch\Scripts\activate
```

4) Install dependencies:

```
pip install -r requirements.txt
```

Once these are completed, you may execute the project by running the command `python main.py`

Pipeline Overview

The pipeline runs in four main stages:

Stage 1 - Extraction

Asynchronous API calls are used to fetch datasets from FuelEconomy, NHTSA, and Alternative Fuel sources. This stage was particularly complex because the first two APIs are hierarchical: to reach the vehicle details it was necessary to call several endpoints in sequence (years → makes → models → vehicle IDs → details). To manage this, additional helper classes named `Model` and `Vehicle` were created. These made the code more structured and reusable but also added complexity to the extraction phase.

During the extraction phase there is also a step where the response of certain endpoints is split into multiple DataFrames. This was necessary because some API responses contained fields that were themselves JSON objects or arrays, not just simple scalar values. For example, in the Fuel Economy API, the endpoint https://fueleconomy.gov/ws/rest/vehicle/{vehicleid} returns both vehicle attributes and structured emissions data. Instead of flattening everything into a single table, the program writes two different DataFrames for the same endpoint: one for the vehicle-level information and another for emissions.

The same strategy applies to other sources. In the Alternative Fuel API, fields such as federal_agency or related_stations are records, while fields such as lpg_nozzle_types, ev_connector_types, hy_pressures, and hy_standards are arrays. Each of these complex types can originate a separate DataFrame. In practice, only related_stations, ev_connector_types, hy_pressures, and hy_standards produced DataFrames in the extracted results, because the others were empty for the specific parameters used. This explains why multiple CSV files appear under each extraction output folder, and also why there could be more in the future. For now, the extraction parameters for this source are fixed, but a future improvement could allow them to be dynamically specified in a configuration file, enabling more flexible dataset generation.

Pipeline Overview

The pipeline runs in four main stages:

Stage 1 - Extraction

NOTE: For simplicity, the API key used in the Alternative Fuel extraction module is currently hardcoded in the code. This is not a best practice: in a real-world project, API keys should never be committed to source control. Instead, they should be stored securely in environment variables, configuration files excluded from version control, or managed through a secret manager such as Azure Key Vault.

Pipeline Overview

The pipeline runs in four main stages:

Stage 2 – Schema Generation

Schema generation in this project happens in two distinct moments, and this decision was deliberate. The first set of JSON schema files is created immediately after the extraction stage and saved under extracted_data_schemas folder. These schemas are necessary because the raw CSVs coming from the APIs often contain values that are ambiguous with respect to their datatypes. For example, a field such as mpgData may contain values like `"N"` or `"Y"`, which would normally be treated as strings, but the schema allows the processing stage to cast them reliably into boolean fields. In short, the extracted schemas act as a contract that ensures all extracted files are interpreted with the correct data types when entering the processing stage.

A second set of schema files is generated after the processing stage and saved under processed_data_schemas. At this point, the datasets have been cleaned, standardized, and in some cases enriched with new fields. One important example is the creation of additional boolean columns such as mpgData_bool, which do not exist in the raw extraction but are introduced as part of the transformation logic. Because the structure of the processed files differs from the extracted ones, new schemas are required to accurately describe them. These processed schemas are later used as the blueprint for generating SQL `CREATE TABLE` scripts and ensure that the tables created in Azure SQL match the transformed structure of the datasets.

This two-stage approach avoids inconsistencies: the first schemas guarantee correct type conversion during processing, while the second schemas guarantee that the database structure reflects the final processed state of the data.

Pipeline Overview

The pipeline runs in four main stages:

Stage 3 – Processing

Cleans and normalizes the extracted data.

Null values are fixed, booleans standardized, columns renamed, and duplicates removed. At this stage, CSVs are written into processed_data folder.

This writing of intermediate files was originally designed to simplify testing, allowing each stage to be validated independently without rerunning the entire pipeline.

However, the ideal design would eliminate this overhead by passing DataFrames directly as class attributes from one stage to the next.

Pipeline Overview

The pipeline runs in four main stages:

Stage 4 – Loading

Processed data is loaded into an Azure SQL staging schema (`stg`). CREATE TABLE scripts are generated from the processed schemas to ensure table structures match the transformed data. The data is then inserted into the staging tables.

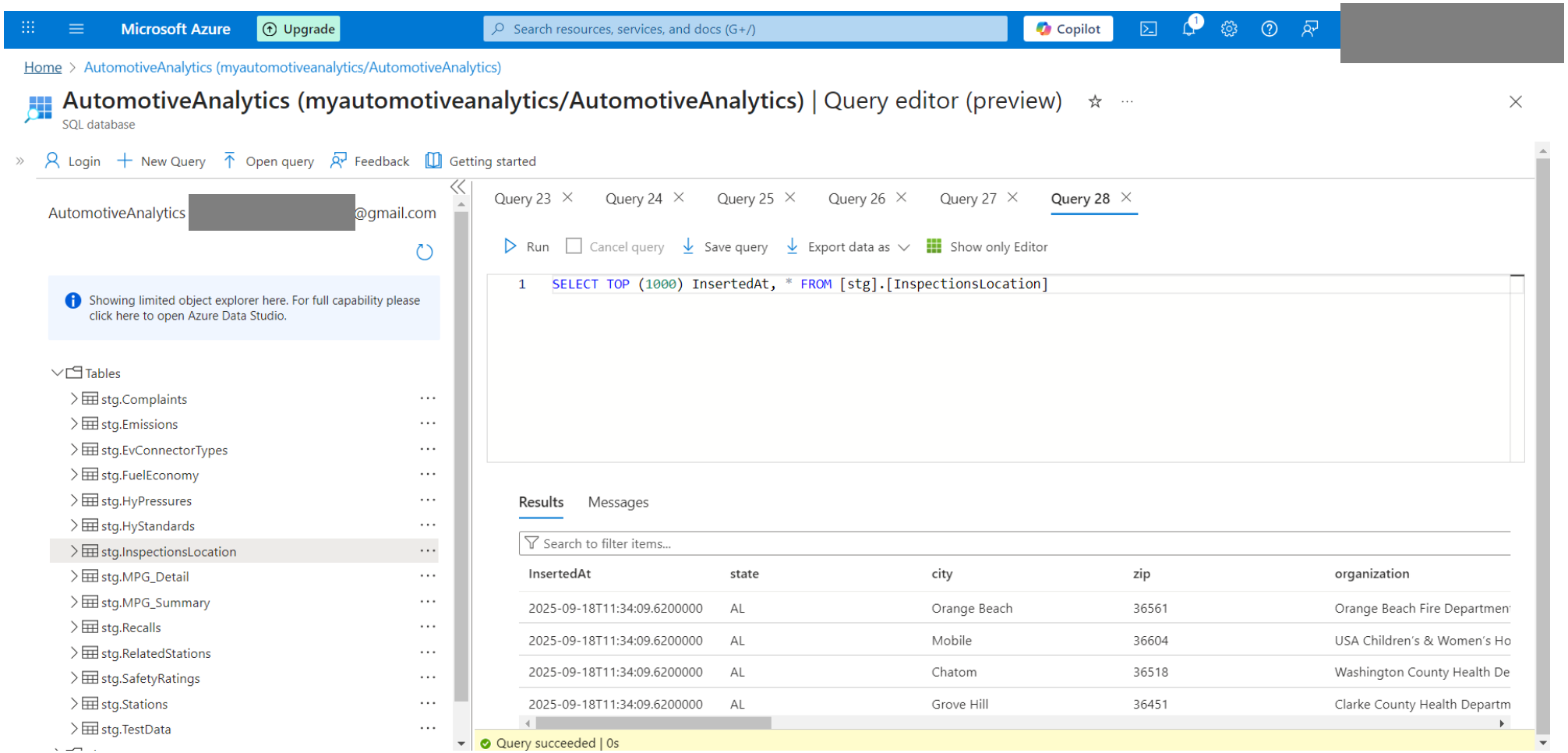
This stage was not strictly required but was fully implemented. It involved creating a new Azure account with free credits, provisioning an Azure SQL Database service, and configuring firewall rules for local machine access. It also required installing the ODBC Driver 18 for SQL Server to enable data writing into the database and adapting the SQL scripts when certain fields exceeded the initially defined character lengths.

Pipeline Overview

The pipeline runs in four main stages:

Stage 4 – Loading

Here is an example of a basic select statement to one of the tables written in the Azure SQL database:



Module Documentation – main.py

The **main.py** module acts as the central orchestrator, bringing together all stages of the pipeline and managing their execution from start to finish.

Flow:

1. Runs `FuelEconomyETL`, `SafetyAdministrationETL`, and `AlternativeFuelETL`.
2. Generates schemas using `schema_producer.py`.
3. Cleans extracted data using `Processing`.
4. Loads processed data into Azure SQL with `Loading`.

Inputs:

- ETL parameters (`num_years`, `concurrency`)
- DB credentials from `connection_config.json`

Outputs:

- Extracted CSVs in `extracted_data/`
- Processed CSVs in `processed_data/`
- JSON schema files created by `schema_producer.py`
    - Extracted schemas → `extracted_data_schemas/`
    - Processed schemas → `processed_data_schemas/`
- Tables created in Azure SQL (schema `stg`)

Module Documentation – Extraction modules

The **fuel_economy_async.py** module defines the FuelEconomyETL class, responsible for extracting vehicle and emissions data from the Fuel Economy API.

- **Inputs:** `num_years`, `concurrency`
- **Outputs:** CSVs in `extracted_data/FuelEconomy/` (`FuelEconomy_*.csv`, `Emissions_*.csv`, `MPG_Summary_*.csv`, `MPG_Detail_*.csv`)
- **Notes:**
  Complex JSON fields originate new DataFrames instead of exploding into a single one. Required creating helper classes (`Model`, `Vehicle`) to handle API hierarchy.

The **highway_safety_admin_async.py** module defines the SafetyAdministrationETL class, which extracts safety ratings, recalls, complaints, and inspections data from the NHTSA API.

- **Inputs:** `num_years`, `concurrency`
- **Outputs:** CSVs in `extracted_data/NHTSafetyAdministration/` (`SafetyRatings_*.csv`, `Recalls_*.csv`, `Complaints_*.csv`, `InspectionsLocation_*.csv`)
- **Notes:**
  Filters out invalid years (`9999`, `2027`). Reused skeleton from FuelEconomy classes with tuning for differences in JSON fields. The API structure required multiple endpoint calls: first fetch vehicle IDs, then details.

The **alternative_fuel_async.py** module defines the AlternativeFuelETL class, designed to extract alternative fuel station data and handle complex fields like records and arrays.

- **Inputs:** `concurrency`
- **Outputs:** CSVs in `extracted_data/AlternativeFuel/` (`Stations_*.csv`, `RelatedStations_*.csv`, `EvConnectorTypes_*.csv`, `HyPressures_*.csv`, `HyStandards_*.csv`)
- **Notes:**
  Handles arrays and records as separate DataFrames. Uses pagination (limit + offset).

Module Documentation – Schema Generation

The **schema_producer.py** module generates JSON schema files that describe dataset structures, ensuring consistency for both processing and loading stages.

- **Role:** Generates JSON schema definitions for CSVs.
- **Inputs:** Latest extracted or processed CSVs.
- **Outputs:** JSON schema files saved in `extracted_data_schemas/` and `processed_data_schemas/`.

Module Documentation – Processing

The **data_processing.py** module defines the Processing class, which cleans and standardizes the extracted data before it is loaded into the database.

The processing flow is structured around a series of methods applied sequentially to each DataFrame:

1) fix_null_values(df) - Identifies placeholder strings such as "unknown" or "Not Rated" and converts them into proper NULL values

2) convert_columns_based_on_schema(df, dataset, schema_file, decimals=3) - Applies type conversions to ensure columns match the definitions in the corresponding JSON schema file. This also handles numeric precision by rounding decimals when required, and importantly, this stage includes string-to-boolean conversion, where values like "N", "No", or "n" are mapped to False, and values like "Y", "Yes", or "YES" are mapped to True.

3) convert_columns_to_camel_case(df) - Standardizes column names into camelCase format for consistency across datasets

4) lower_first_letter(df) - Further normalizes column names by converting the first letter to lowercase

5) **drop_duplicates(df) -** Prints the DataFrame shape before and after removing duplicates to validate the cleaning process

- **Role:** Cleans and standardizes extracted CSVs.
- **Inputs:** Mapping of datasets to CSVs (from `produce_schemas`).
- **Outputs:** Processed CSVs under `processed_data/<dataset>/`.

This pipeline ensures all datasets share a consistent structure and datatypes, preparing them for reliable loading into SQL. The design is intentionally modular: any additional processing requirements can be included easily by adding new methods to the class and integrating them into the flow.

Module Documentation – Loading

The **data_loading.py** module defines the Loading class, which loads processed data into an Azure SQL staging schema and manages table creation.

Currently, the loading is append-only, but in future versions there could be new versions of these tables on a different schema that would hold the more recent and not all the entire history of the extractions.

- **Role:** Loads processed data into Azure SQL staging schema.
- **Inputs:**
  - DB credentials (`server`, `database`, `username`, `password`) from `connection_config.json`
  - Latest processed CSVs (from `produce_schemas`)
- **Outputs:**
  - `CREATE TABLE` scripts saved in `sql_scripts/`
  - Data loaded into staging tables (`stg` schema) in Azure SQL

Scheduling and Merge Strategy

Distinguishing between dimensions and facts is key to defining how data is refreshed and maintained.

Some datasets in this project behave more like dimensions. For example, vehicle information (make, model, fuel type, MPG, etc.) is relatively stable but may be updated occasionally. FuelEconomy table includes a modifiedOn column, which provides a natural way to detect when data has changed. A merge strategy could be applied here: for each vehicle, keep the most recent record based on the modifiedOn timestamp. If this dimensional data does not change frequently, a weekly schedule could be sufficient, although a daily refresh would still be safe and keep the pipeline consistent.

Other datasets behave more like facts. For example, complaints data represents individual events wgere each record is a new complaint submitted by a user. This type of data typically grows over time without overwriting existing records. Here, a scheduling strategy of daily ingestion makes the most sense, as it ensures new complaints are quickly available while avoiding unnecessary reloads of older data.

Considering both sides, the most favorable scheduling option is to run the pipeline once per day for all datasets. This provides fresh data for fact tables, while also ensuring that dimension tables remain current without the overhead of multiple runs per day. Over time, logs and update patterns could be monitored to fine-tune the frequency.

It is also important to note that I need more time to fully understand the data model and the context of each dataset. While I focused heavily on the technical aspects of extraction, processing, and loading, the merging and scheduling strategies are critical to keeping the data accurate, up to date, and reliable for downstream use. A deeper analysis of business requirements and dataset semantics would allow me to design more robust strategies for these stages.