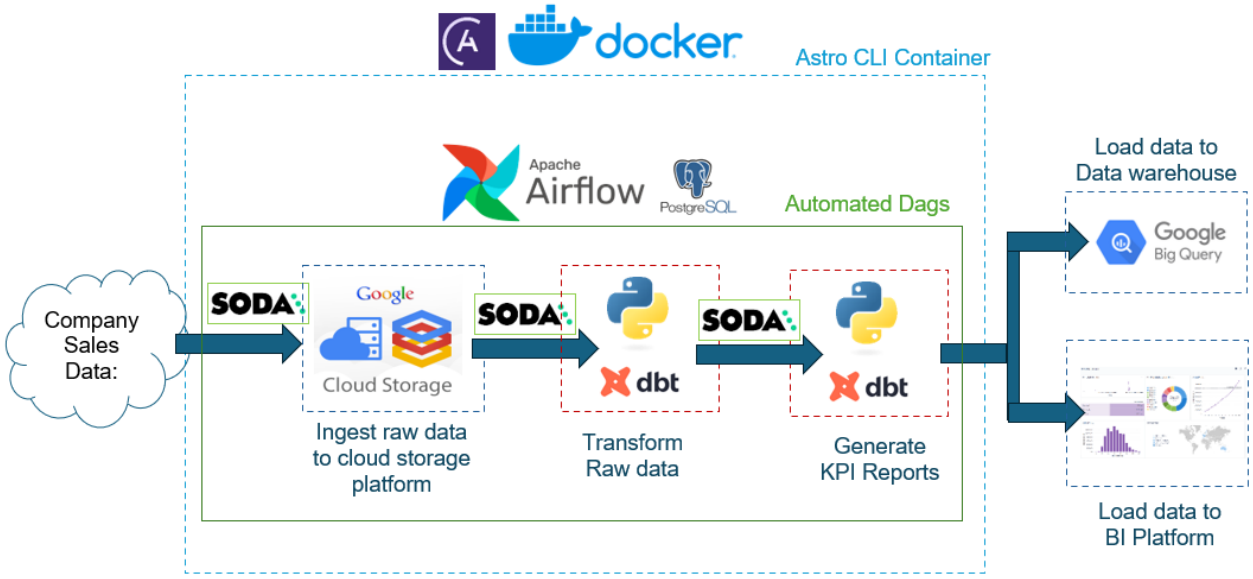
Automated ETL Pipeline

Overview:

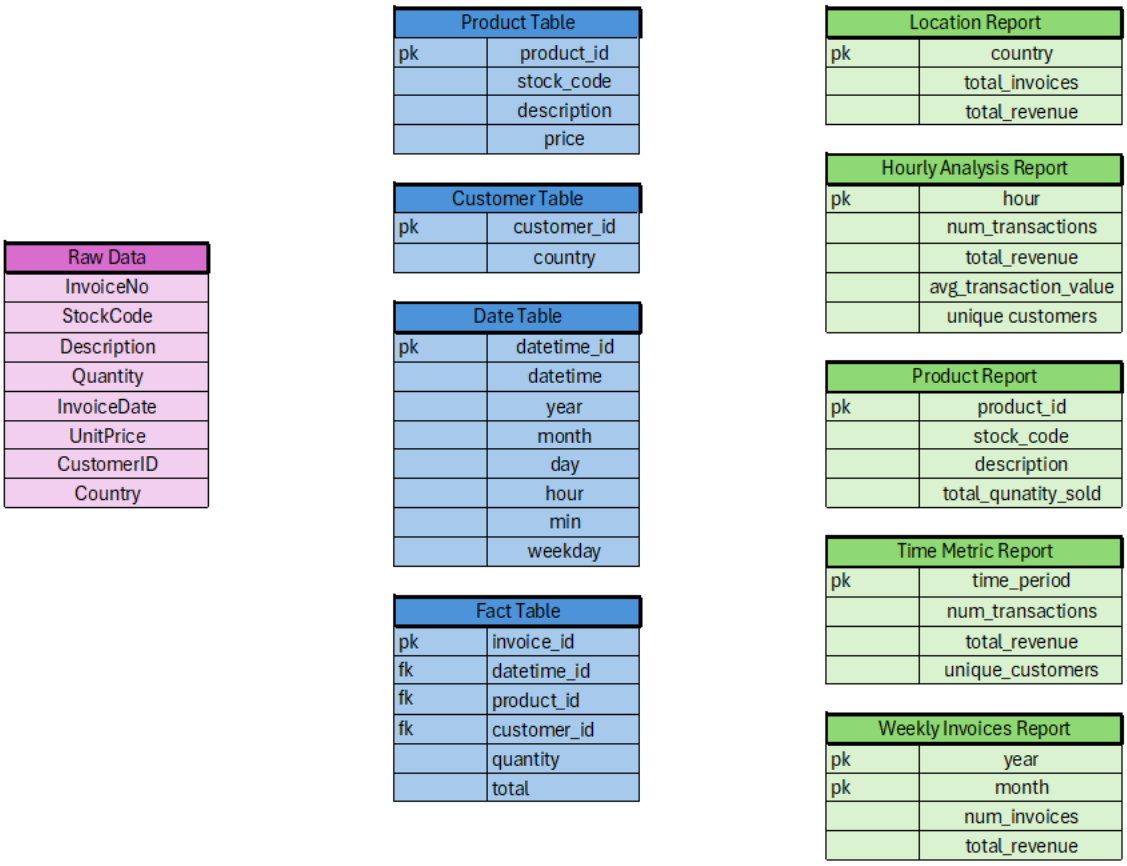
An end-to-end data pipeline that automates the entire process, from ingesting raw data all the way to visualizing transformed structured data on a BI platform. This project was designed for a hypothetical retail merchant with raw invoice data on their internal system. They sought a technology system that could automatically load current and new invoice data to a cloud storage platform, load data into different tables in a cloud-based data warehouse, and have data automatically load to a BI platform that analysts could use to understand and visualize the performance of the business.



Pipeline visualized. Apache Airflow is used to orchestrate each step of the pipeline. Soda Core is used to perform data checks and ensure quality throughout the pipeline.

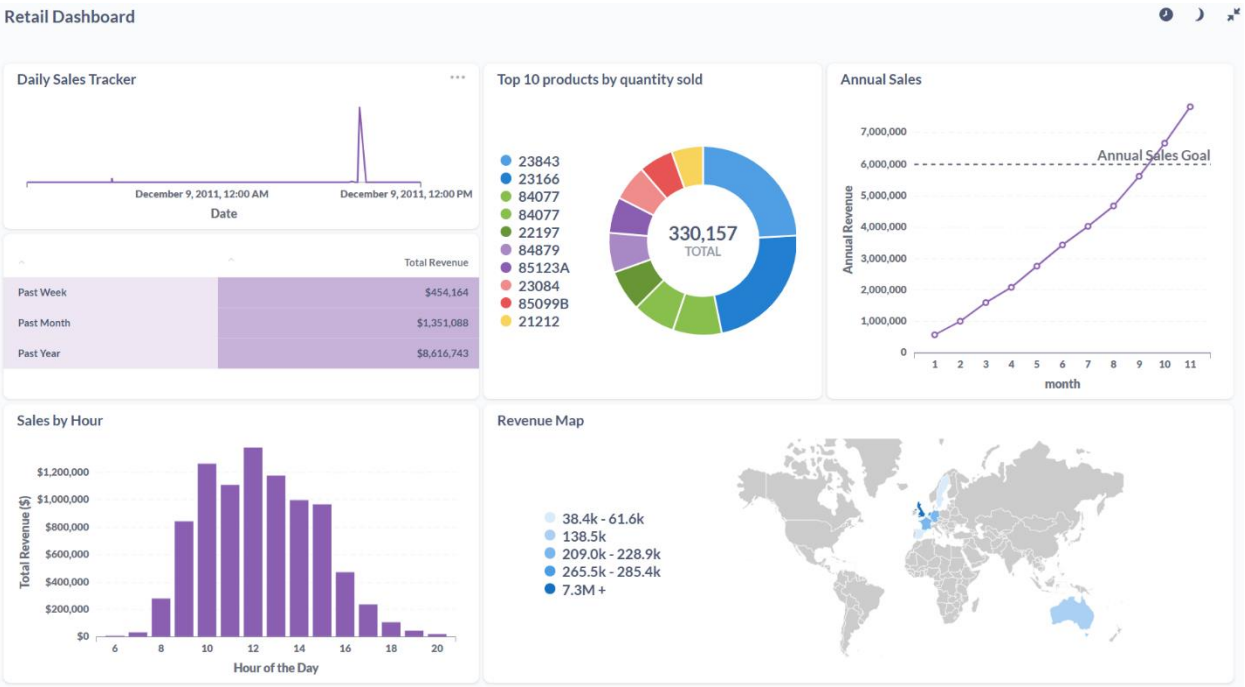
To accomplish this Google Cloud Storage and Google Big Query were selected for their reliability, industry wide adoption, and low-cost adoption. Using GCS and Big Query will provide the customer with a low-cost cloud provider that can easily scale as their demands increase.

A data pipeline was created that would ingest raw data into Google Cloud Storage. Python and DBT would be used to transform the data into structured tables, and then generate reports containing frequently tracked KPI's. Python and DBT were selected for their efficiency and ease of use. This will provide the customer with more ownership of the pipeline and grant them the ability to alter the Python and SQL code in the future if desired. Each time there is a "data handoff" the quality of the data will be checked using Soda. Soda data checks can be easily customized and automatically generates email alerts when data quality checks fail, providing the customer with a cost efficient but effective open-source data quality check mechanism.



DBT is used to transform raw data into the blue structured tables in the BigQuery database. The green reports contain frequently used KPIs and are used to help visualize the data. Due to their frequent use, dbt is configured to automatically generate the report tables each time the pipeline is run.

Once the data has been structured it will be loaded into Big Query to house the database. Big Query is connected to the BI platform Metabase, which is a free, open-source, and user-friendly platform that provides analysts easy real time access to the company's data. Metabase has free visualization tools built into the platform to empower analysts and drive insights.



1Data visualized on metabase. Metabase will automatically grab the latest data from the warehouse, ensuring the dashboard updates as new data is available.

The pipeline is automated using Apache Airflow. Airflow is free, open-source, and allows for scheduling and monitoring the pipeline. Airflow contains many built-in functions which makes interfacing with Cloud Providers, like GCS, easy. The customer can alter how often the pipeline runs to find the right balance between finding insights quickly and minimizing their cloud usage.

Finally, the system is developed in Astro CLI, a data Engineering wrapper built on top of Docker which has many built in functions to help manage install dependencies of the pipeline. Astro CLI greatly reduces the amount of time required to initialize a development environment. Astro CLI and Docker enables quick and easy deployment from a local development environment to a production environment.

Overall, the project sought to provide the customer with a robust and easy maintain automated data pipeline that was as cost-efficient as possible given the customers scale and data complexity needs.

Benefits:

- Automatically and seamlessly upload new sales data to end user's cloud storage platform and data analysis platform.
- Improve end user's speed and efficiency by automatically generating KPI reports that can be quickly leveraged to make business decisions.
- Allow company to track business performance in real time.
- Customize data load frequency.
- Generate email alerts if there is an error in the pipeline, including data entry errors.

Technology Used:

- Astro CLI and Docker: Pre-configured containers with everything needed for Apache Airflow. Greatly speeds up the processes of initializing a data engineering development environment. Containerized approach allows for maintaining multiple development environments on a single computer without creating installed dependency issues. Quick and easy deployment to a production environment. Interfaces well with all major cloud storage providers.
- Google Cloud and Bigquery: Store raw data and transformed tables in the cloud. Google Cloud and Bigquery specifically are very cost-efficient for small applications such as this project and can be easily scaled up as the scope of customer needs increases.
- Apache Airflow: Orchestrates and automates functions in the data pipeline. Easily scalable if the complexity of the pipeline needs to increase. Open source and free. Provider and file structure agnostic. User-friendly visualization of DAGs which help workflow management and error diagnostics.
- DBT: Transforms raw data into structured usable models. Uses easy to write and maintain SQL-based models capable of performing complex queries and optimized transformations. Transformations can be easily automated by Apache Airflow. Open-source and free.
- Soda: Automated data quality monitoring that generates real-time alerts and notifications. Can perform data checks at any point in the pipeline. Capable of identifying issues early in the pipeline to save money later on.
- Metabase: Business Intelligence platform that allows for analysts to easily visualize and explore data. Open-source, free, and extremely easy to deploy. Real-Time access to the Bigquery data warehouse so that analysts are always looking at the latest data available.

Sample Code:

Included below are the different DBT models that were used to generate the KPI reports. For all code contained in the project please visit my github:

https://github.com/aholme1?tab=repositories

DBT Transformation to generate product table:

```
2   -- Find unique products sold and generate a unique identifier for each.
3   -- Stock code is not unique, can have different descriptions or prices
4   SELECT DISTINCT
5       {{ dbt_utils.generate_surrogate_key(['StockCode', 'Description', 'UnitPrice']) }} as product_id,
6                   StockCode AS stock_code,
7       Description AS description,
8       UnitPrice AS price
9   FROM {{ source('retail', 'raw_invoices') }}
10  WHERE StockCode IS NOT NULL
11  AND UnitPrice > 0
```

DBT Transformation to generate customer table:

```sql
-- CTE to generate a unique customer_id using CustomerID and Country, and filter out rows with null CustomerID
-- Use a static country iso table in the datawarehouse to add standard country iso.
WITH customer_cte AS (
        SELECT DISTINCT
            {{ dbt_utils.generate_surrogate_key(['CustomerID', 'Country']) }} as customer_id,
            Country AS country
        FROM {{ source('retail', 'raw_invoices') }}
        WHERE CustomerID IS NOT NULL
)
SELECT
    t.*, -- Select all columns from the CTE (customer_id and country)
        cm.iso
FROM customer_cte t
LEFT JOIN {{ source('retail', 'country') }} cm ON t.country = cm.nicename -- Left join on country table using country names
```

DBT Transformation to generate datetime table:

```sql
--CTE to process different Date time formats, handle 'MM/DD/YYYY HH:MM' and 'MM/DD/YY HH:MM' format. Set data entry errors to null.
--Output table with year, month, day, hour, minute, and weekday to a unique row. Extracted from raw Datetime.
WITH datetime_cte AS (
  SELECT DISTINCT
    InvoiceDate AS datetime_id,
    CASE
      -- handle 'MM/DD/YYYY HH:MM' case
      WHEN LENGTH(InvoiceDate) = 16 THEN
        PARSE_DATETIME('%m/%d/%Y %H:%M', InvoiceDate)
        -- handle 'MM/DD/YY HH:MM' case
      WHEN LENGTH(InvoiceDate) <= 14 THEN
        PARSE_DATETIME('%m/%d/%y %H:%M', InvoiceDate)
      ELSE
       --set data entry errors to null
        NULL
    END AS date_part,
  FROM {{ source('retail', 'raw_invoices') }}
  WHERE InvoiceDate IS NOT NULL
)

-- Extract individual date time components
SELECT
  datetime_id,
  date_part as datetime,
  EXTRACT(YEAR FROM date_part) AS year,
  EXTRACT(MONTH FROM date_part) AS month,
  EXTRACT(DAY FROM date_part) AS day,
  EXTRACT(HOUR FROM date_part) AS hour,
  EXTRACT(MINUTE FROM date_part) AS minute,
  EXTRACT(DAYOFWEEK FROM date_part) AS weekday
FROM datetime_cte
```

DBT Transformation to generate Fact Table:

```sql

-- Generate fact table and calculate the total revenue by multiplying quantitiy sold and unit price
WITH fct_invoices_cte AS (
    SELECT
        InvoiceNo AS invoice_id,
        InvoiceDate AS datetime_id,
        {{ dbt_utils.generate_surrogate_key(['StockCode', 'Description', 'UnitPrice']) }} as product_id,
        {{ dbt_utils.generate_surrogate_key(['CustomerID', 'Country']) }} as customer_id,
        Quantity AS quantity,
        Quantity * UnitPrice AS total
    FROM {{ source('retail', 'raw_invoices') }}
    WHERE Quantity > 0
)
SELECT
    invoice_id,
    dt.datetime_id,
    dp.product_id,
    dc.customer_id,
    quantity,
    total
FROM fct_invoices_cte fi
INNER JOIN {{ ref('dim_datetime') }} dt ON fi.datetime_id = dt.datetime_id
INNER JOIN {{ ref('dim_product') }} dp ON fi.product_id = dp.product_id
INNER JOIN {{ ref('dim_customer') }} dc ON fi.customer_id = dc.customer_id
```

Sample Soda data quality check during product table transformation:

```yaml
 checks for dim_product:
   # Check fails when product_key or english_product_name is missing, OR
   # when the data type of those columns is other than specified
   - schema:
       fail:
         when required column missing: [product_id, description, price]
         when wrong column type:
           product_id: string
           description: string
           price: float64
   # Check fails when customer_id is not unique
   - duplicate_count(product_id) = 0:
       name: All products are unique
   # Check fails when any NULL values exist in the column
   - missing_count(product_id) = 0:
       name: All products have a key
   # Check fails when any prices are negative
   - min(price):
       fail: when < 0
```

Apache Airflow Dag sample. The DBT transform functions are being called, and then the Soda data quality check functions for transforms are being called.

```
70          transform = DbtTaskGroup(
71              group_id='transform',
72              project_config=DBT_PROJECT_CONFIG,
73              profile_config=DBT_CONFIG,
74              render_config=RenderConfig(
75                  load_method=LoadMode.DBT_LS,
76                  select=['path:models/transform']
77              )
78          )
79
80          @task.external_python(python='/usr/local/airflow/soda_venv/bin/python3')
81          # Use python env for soda data check
82          def check_transform(scan_name='check_transform', checks_subpath='transform'):
83              from include.soda.check_function import check
84
85              return check(scan_name, checks_subpath)
86
```