Generative AI for Automated Business Reports: Insights from the Olist E-Commerce Dataset

Sacred Heart University
Final Project

Presented by
Eliza Paul Ganta
gantae@mail.sacredheart.edu

1. Abstract

Business executives depend on recurring reports to monitor sales, customer behavior, logistics performance, and overall business health. Traditionally, analysts spend significant time extracting, cleaning, and joining datasets, computing KPIs, and writing narrative summaries. This project demonstrates how **Generative AI** can automate a major part of that reporting process by converting structured KPIs into natural language business reports.

Using the **Olist Brazilian E-Commerce Dataset** from Kaggle, the project builds an end-to-end pipeline that:

1. Ingests raw CSV files from Olist.
2. Cleans and integrates them into analytical data marts.
3. Computes key performance indicators (KPIs) at monthly and product-category levels.
4. Exposes these outputs through an interactive **Streamlit web app**.
5. Generates a compact **factsheet JSON** summarizing the last three months of performance.
6. Uses a **Large Language Model (LLM)** (Google Gemini via `google-generativeai`) to transform the factsheet into an executive-style narrative.

The final deliverables include the **Streamlit app** (`streamlit_app.py`), a **requirements file** listing dependency, a **README** describing the generated assets, and this written report explaining the overall design, methodology, and real-world implications.

2. Introduction

2.1 Background and Motivation

Organizations generate large volumes of transactional data across sales, logistics, and customer interactions. Business leaders need recurring reports that distill this raw data into:

- Monthly revenue trends and growth
- Customer retention and churn behavior
- Logistics performance (delivery times, late deliveries)
- Customer satisfaction and review sentiment

Traditionally, this workflow looks like:

1. Data extraction from source systems or CSV files.
2. Cleaning and joining disparate tables.
3. KPI calculation and dashboard preparation.
4. Manual writing of "Executive Summary" sections for each report.

This process is **time-consuming, repetitive, and error-prone**. As data volume and reporting frequency increase, the bottleneck shifts from data availability to the human capacity to interpret and narrate results.

Recent advances in **Generative AI**—especially LLMs such as GPT and Gemini—make it possible to automatically convert structured data into clear, human-readable text. This project explores that idea in a realistic setting: the **Olist Brazilian E-Commerce Dataset**, a well-known benchmark for online retail analytics.

2.2 Project Goal

The main goal is to build a proof-of-concept **AI-powered reporting system** that behaves like a **"junior business analyst"**:

- It reads prepared KPIs instead of raw text.
- It organizes the numbers into a structured factsheet.
- It generates business summaries and recommendations automatically.

This is implemented as a **Streamlit web app** named "📦 Olist AI Reports — One-Click ETL, KPIs, Charts & Factsheet" and supported by a lightweight data engineering stack and an LLM integration layer.

3. Dataset Description

The project uses the **Olist Brazilian E-Commerce Dataset** from Kaggle, which consists of multiple csv tables that collectively represent the end-to-end e-commerce lifecycle:

- **Orders** (`olist_orders_dataset.csv`) – order IDs, timestamps, statuses (delivered, canceled, etc.).
- **Order Items** (`olist_order_items_dataset.csv`) – products per order, prices, freight values.
- **Payments** (`olist_order_payments_dataset.csv`) – payment types and amounts.
- **Customers** (`olist_customers_dataset.csv`) – customer IDs, unique IDs, location metadata.
- **Sellers** (`olist_sellers_dataset.csv`) – seller IDs and locations (optional in this app).
- **Products** (`olist_products_dataset.csv`) – product-related metadata including category.
- **Reviews** (`olist_order_reviews_dataset.csv`) – customer review scores and review text.
- **Geolocation** (`olist_geolocation_dataset.csv`) – ZIP-code-level coordinates (optional here).
- **Category Translation** (`product_category_name_translation.csv`) – mapping from Portuguese to English product category names.

The app expects all these as inputs; it uses them to build consolidated **analytical marts** and later to compute **monthly and category-level KPIs**.

4. System Architecture Overview

At a high level, the system follows the **CRISP-DM** methodology proposed in the project proposal: Business Understanding → Data Understanding → Data Preparation → Modeling → Evaluation → Deployment.

Technically, the solution consists of four main layers:

1. **Data Ingestion Layer**
   - Users upload the nine Olist CSVs through the **Streamlit UI** (`st.file_uploader`).
   - The app checks that all required files are present using a `REQUIRED_FILES` mapping.
2. **Data Preparation & Mart Layer (`build_marts`)**
   - Deduplication and basic integrity checks (e.g., ensuring order IDs match across tables).
   - Casting IDs to string, numeric columns to floats, and parsing timestamps.
   - Joining tables into:
     - A **monthly KPI table** (`monthly`)
     - A **category performance table** (`category_perf`)
     - An order-level **fact table** (`delivered` / `fact_orders`)
     - Dimension tables `dim_customer` and `dim_product`
3. **Analytics & Visualization Layer**
   - Compute KPIs: monthly revenue, orders, average order value (AOV), average delivery days, late delivery ratio, and average review score.
   - Create line charts for trends over time (`line_plot` function).
   - Prepare category-level breakdowns (e.g., revenue by product category per month).
4. **AI Reporting Layer**
   - Build a compact **factsheet** for the last three months using `build_factsheet`.
   - Serialize it to JSON (`factsheets/monthly_facts.json`).
   - Call **Gemini** via `google-generativeai` using the `call_gemini` helper to generate text summaries based on the factsheet.

The overall outputs are organized into a structured folder (as described in the README):

- `data_raw/` – original CSV copies.
- `data_clean/` – cleaned dimensions like `dim_customer.csv`, `dim_product.csv`.
- `marts/` – analytical tables (`monthly_kpis.csv`, `category_perf.csv`, `fact_orders.csv`).
- `factsheets/` – `monthly_facts.json` used as LLM input.
- `plots/` – PNG charts suitable for slide decks.

5. Methodology and Implementation

5.1 Data Ingestion and Interface (Streamlit)

The file `streamlit_app.py` configures the app with:

```
st.set_page_config(page_title="Olist AI Reports", layout="wide")
st.title("📦 Olist AI Reports — One-Click ETL, KPIs, Charts & Factsheet")
st.caption(
    "Upload the nine Olist CSVs (from Kaggle) and get ready-to-use marts, 
plots, "
    "a compact factsheet, and an optional LLM report."
)
```

The function **`load_inputs()`**:

- Displays a section titled **"1) Upload your CSVs"**.
- Uses `st.file_uploader` for each required file defined in `REQUIRED_FILES`.
- Reads each CSV into a Pandas DataFrame via a helper `_read_csv()` that handles optional `parse_dates` arguments for timestamp columns.
- Validates that all required uploads are present; otherwise, it shows an info message asking the user to upload all CSVs.

At the end, the function returns a dictionary of DataFrames (orders, items, payments, reviews, products, customers, translation, etc.) that feed into the next step.

5.2 Data Cleaning and Deduplication

To ensure data consistency, the app defines a helper:

- **`dedup(df: pd.DataFrame, name: str)`**
    - Computes the number of rows **before** and **after** `drop_duplicates()`.
    - If duplicates were removed, it informs the user via `st.warning` for transparency.
    - Returns the deduplicated DataFrame.

Common ID columns such as `order_id`, `customer_id`, `customer_unique_id`, `product_id`, and `seller_id` are explicitly cast to `str` across the relevant DataFrames. This avoids issues when joining tables that might have numeric IDs in some files and string IDs in others.

Numeric columns like `price`, `freight_value`, and `payment_value` are converted using `pd.to_numeric(..., errors="coerce")`, so any non-numeric entries become `NaN` and are handled safely.

5.3 Integrity Checks

The code uses an internal **`integrity_report`** helper (partially visible in the file) to validate join keys between tables. Typical checks include:

- Ensuring every row in `items` has a corresponding `order_id` in `orders`.
- Verifying payments and reviews also align with existing orders.
- Confirming that orders refer to valid customers.

If referential integrity is broken (e.g., an order appears in items but not in orders), the helper can surface counts of missing parent keys, helping the analyst spot serious data issues.

5.4 Building Analytical Marts (`build_marts`)

The core of the data preparation logic resides in:

```
def build_marts(dfs: Dict[str, pd.DataFrame]):
    """
    Return (monthly, category_perf, fact_orders, dim_customer, dim_product).
    """
```

Key steps inside this function:

1. **Dimension Tables**
    - `dim_product`: Built from `products` joined with `translation` so that each product has both the original and English category name (`category_en`).
2. **Order-Level Features**
    - A `line_revenue` column is defined on order items, initially equal to `price`.
    - Aggregation at `order_id` level computes:
        - `revenue` (sum of item prices)
        - `items_cnt` (number of order lines)
    - Freight costs can be aggregated separately and then merged back.
3. **Delivery Time and Lateness**
    - Orders are filtered to those in **delivered** status.
    - `delivery_days` is computed as the difference (in days) between `order_delivered_customer_date` and `order_purchase_timestamp`.
    - A boolean `is_late` flag is derived based on estimated delivery dates or empirical distribution (e.g., marking as late when `delivery_days` exceeds a high percentile for orders without explicit estimates).
4. **Monthly KPIs**
   The snippet in the code shows:

   ```
   monthly = delivered.groupby("order_month").agg(
       revenue=("revenue", "sum"),
       orders=("order_id", "nunique"),
       avg_delivery_days=("delivery_days", "mean"),
       late_ratio=("is_late", "mean"),
   ).reset_index()
   monthly["aov"] = monthly["revenue"] / monthly["orders"]
   ```

Here:

- **`order_month`** is derived from `order_purchase_timestamp` converted to a monthly period string (e.g., `"2017-03"`).
- **`revenue`**: total revenue per month.
- **`orders`**: distinct delivered orders per month.
- **`avg_delivery_days`**: average time from purchase to delivery.
- **`late_ratio`**: fraction of delivered orders flagged as late.
- **`aov`**: average order value = `revenue / orders`.

5. Monthly Review KPIs

```
revx = reviews.merge(delivered[["order_id", "order_month"]],
on="order_id", how="left")
    review_month = revx.groupby("order_month").agg(
        avg_review_score=("review_score", "mean"),
        reviews=("order_id", "count"),
    ).reset_index()
    monthly = monthly.merge(review_month, on="order_month", how="left")
```

This adds:

- **`avg_review_score`**: average 1–5 rating per month.
- **`reviews`**: number of reviews per month.

6. Category Performance Mart

```
   items_cat = (
       items.merge(products[["product_id", "category_en"]], on="product_id",
how="left")
       .merge(orders[["order_id", "order_purchase_timestamp",
"order_status"]], on="order_id", how="left")
   )
   items_cat = items_cat[items_cat["order_status"] == "delivered"].copy()
   items_cat["order_month"] =
items_cat["order_purchase_timestamp"].dt.to_period("M").astype(str)

   category_perf = items_cat.groupby(["order_month", "category_en"]).agg(
       revenue=("price", "sum"),
       items=("order_item_id", "count"),
       avg_freight=("freight_value", "mean"),
   ).reset_index()
```

This creates an analytical table of **category** × **month** with:

- Total **revenue** per category per month.
- Number of **items** sold.
- Average **freight cost**.

7. Returned Objects

The function finally returns:

- monthly – monthly KPIs (saved later as monthly_kpis.csv)
- category_perf – category-level KPIs (category_perf.csv)
- delivered – order-level fact table for delivered orders (fact_orders.csv)
- dim_customer – dimension table for customers
- dim_product – dimension table for products

These are persisted under the /marts and /data_clean folders described in the README.

5.5 Visualization Layer

For quick exploratory analysis directly in the app, a helper **line_plot(x, y, title, ylabel)** wraps Matplotlib:

```
def line_plot(x, y, title, ylabel):
    fig = plt.figure()
    plt.plot(x, y, marker="o")
    plt.title(title)
    plt.xlabel("Month")
    plt.ylabel(ylabel)
    plt.xticks(rotation=45)
    plt.tight_layout()
    st.pyplot(fig)
```

In the UI, the app typically displays:

- **Revenue over time** – highlights seasonality and growth patterns.
- **Orders over time** – shows demand trends.
- **Average order value (AOV)** – indicates shifts in basket size or pricing.
- **Average delivery days** – reveals improvements or deterioration in logistics.
- **Late delivery ratio** – monitors operational risk and potential dissatisfaction.
- **Average review score** – tracks customer sentiment month by month.

These charts are also saved under plots/ as PNG files, ready for inclusion in PowerPoint slides and written reports.

6. LLM Factsheet and Automated Reporting

6.1 Factsheet Construction (build_factsheet)

To keep prompts structured and concise, the project uses a **factsheet** abstraction rather than sending raw CSVs into the LLM:

```
def build_factsheet(monthly: pd.DataFrame) -> Dict:
    m = monthly.sort_values("order_month").tail(3).copy()
    facts = {
        "periods": m["order_month"].tolist(),
        "revenue": [float(x) if pd.notnull(x) else None for x in
m["revenue"].round(2)],
        "orders": [int(x) if pd.notnull(x) else None for x in m["orders"]],
        "aov": [float(x) if pd.notnull(x) else None for x in
m["aov"].round(2)],
        "avg_delivery_days": [float(x) if pd.notnull(x) else None for x in
m["avg_delivery_days"].round(2)],
        "late_ratio": [float(x) if pd.notnull(x) else None for x in
m["late_ratio"].round(3)],
        "avg_review_score": [
            float(x) if pd.notnull(x) else None for x in
m["avg_review_score"].round(2)
        ],
    }
    return facts
```

This dictionary is then:

- Displayed in the Streamlit app as **"Factsheet JSON (last 3 months)"**.
- Downloadable as a JSON file (e.g., via `st.download_button`).
- Stored in the `factsheets/` folder as `monthly_facts.json`.

The design choice to use the **last three months** aligns with how executives often review recent performance and identify short-term trends.

6.2 LLM Integration (`call_gemini` and Prompting)

The app integrates with **Google Gemini** via `google-generativeai`:

```
def call_gemini(prompt: str, model: str = "gemini-1.5-flash", temperature:
float = 0.2, api_key: str = None) -> str:
    """
    Call Google Gemini using google-generativeai.
    Returns the text content or raises on error.
    """
    # Priority: explicit key → GEMINI_API_KEY → GOOGLE_API_KEY
    api_key = api_key or os.environ.get("GEMINI_API_KEY") or
os.environ.get("GOOGLE_API_KEY")
    if not api_key:
        raise RuntimeError(
            "No Gemini API key provided. Set GEMINI_API_KEY/GOOGLE_API_KEY or
enter it in the sidebar."
        )

    genai.configure(api_key=api_key)
    ...
```

Key points:

- The API key can be provided in the environment (`GEMINI_API_KEY` / `GOOGLE_API_KEY`) or via the Streamlit sidebar.

- A helper like `make_llm_prompt(facts)` (in the file) formats the facts into a **natural-language prompt** instructing Gemini to behave like a **business analyst** and produce an **executive summary, key insights, and recommendations**.

Although the full prompt text is partially truncated in the visible snippet, a typical prompt includes:

- A brief **role description** ("You are a senior business analyst for Olist").
- The **factsheet JSON** embedded in the prompt.
- Instructions to:
    - Summarize financial performance.
    - Comment on logistics (delivery days, late ratio).
    - Reflect on customer sentiment (review scores).
    - Provide 3–5 **actionable recommendations**.

The result is then rendered in the app as a rich text report, allowing the user to:

- Copy/paste into email or slides.
- Compare with their own interpretation or dashboards.

7. Results and Discussion

Because the app is dataset-agnostic (as long as the Olist CSVs follow the expected schema), results will vary depending on the specific subset and time window used. However, some **typical patterns** that the app reveals include:

1. **Seasonal Revenue Fluctuations**
   Monthly revenue charts highlight peaks and troughs in sales, which can often be associated with seasonal events or promotions.
2. **Order Growth vs. AOV**
   There may be periods where the number of orders grows faster than revenue, signaling a decline in average basket size—or the opposite pattern, where fewer but higher-value orders dominate.
3. **Delivery Performance and Late Ratio**
   `avg_delivery_days` and `late_ratio` help identify whether growth is stressing logistics. An increasing late ratio alongside rising revenue can signal capacity constraints.
4. **Customer Sentiment Trends**
   Average review scores per month show whether operational changes are improving or hurting customer satisfaction.
5. **Category-Level Winners and Losers**
   The `category_perf` mart makes it possible to identify product categories with:
    - Strong revenue growth
    - High freight costs relative to revenue
    - Volume-driven vs. value-driven performance

The LLM-generated report usually mirrors what a human analyst might write, but it does so **automatically** using only the structured factsheet. The narrative typically contains:

- A concise summary of revenue and orders for the last three months.
- Commentary on improvement or deterioration in delivery times and late deliveries.
- A quick interpretation of review score trends.
- Tactical recommendations such as:
    - Optimizing freight in high-cost categories.
    - Focusing marketing on consistently high-growth categories.
    - Addressing causes of negative customer feedback.

8. Evaluation

8.1 Technical Evaluation

Accuracy

- Numeric accuracy relies on the `build_marts` and `build_factsheet` functions. Since the LLM sees the already-aggregated metrics, the risk of miscalculating KPIs is minimized.
- The LLM can still occasionally misinterpret or over-emphasize minor changes, which is a known limitation of generative models.

Performance and Scalability

- All computations are done in memory using Pandas, which is sufficient for the Olist dataset size.
- For production-scale data, a migration to a SQL-based warehouse or distributed environment would be recommended.

Robustness

- Deduplication and integrity checks improve data quality.
- The app provides warnings if required files are missing or if API keys are not set.

8.2 Business Evaluation

Time Savings

- Once the pipeline is configured, a new reporting run is essentially **one click**: upload fresh CSVs → generate marts, charts, and AI narrative.
- This significantly reduces the manual effort spent on recurring monthly or quarterly reports.

Consistency

- The factsheet structure ensures that the LLM always sees metrics in a **consistent format**, leading to stable reporting style and coverage of key dimensions (financial, logistics, sentiment).

Actionability

- The LLM outputs are not just descriptive but also **prescriptive**, providing recommendations.
- Decision makers still need to validate and prioritize actions, but the AI accelerates the initial interpretation.

8.3 Limitations

- The model only sees **aggregated KPIs**, not raw records or detailed segments; deeper drill-down analyses remain the analyst's responsibility.
- The quality of the AI report is sensitive to prompt engineering and the clarity of the factsheet.
- The prototype currently uses Gemini; using multiple models or adding regression/forecasting models for future months would enhance predictive capabilities.

9. Conclusion and Future Work

This project shows how **Generative AI** can be combined with **classical data engineering and BI** to create an automated business reporting pipeline. Using the Olist e-commerce dataset, the system:

- Ingests nine relational CSV files.
- Cleans and integrates them into well-defined marts.
- Computes core KPIs across financial, logistics, and customer sentiment dimensions.
- Visualizes trends through a Streamlit dashboard.
- Summarizes the most recent three months into a factsheet JSON.
- Uses an LLM (Gemini) to produce executive-style reports with insights and recommendations.

Future Enhancements

Potential directions for extending this work include:

1. **Forecasting and What-If Analysis**
    - Integrate time-series models (e.g., Prophet, ARIMA) to forecast revenue and orders.
    - Ask the LLM to explain forecast scenarios in natural language.
2. **Segmented Reporting**
    - Build separate factsheets for key customer segments or regions, enabling more granular AI-generated reports.
3. **Real-Time or Near-Real-Time Data Streams**

  - Connect the app to a live data source (e.g., database or API) and schedule automatic report generation.
4. **Multi-Model Comparison**
   - Experiment with different LLM providers (OpenAI, Gemini, etc.) and compare clarity, accuracy, and style of generated reports.
5. **Integration with BI Tools**
   - Combine this system with **Tableau** or **Power BI** dashboards, using the factsheet narrative as a textual overlay embedded directly in BI reports.

10. Implementation Files and Environment

- **`streamlit_app.py`**
  - Main application script implementing the Streamlit UI, mart building, plotting, factsheet creation, and LLM integration.
- **`requirements.txt`**
  Contains Python dependencies:
  - `streamlit` – web app framework
  - `pandas`, `numpy` – data handling
  - `matplotlib` – charting
  - `openai` – (optional) for alternative LLM integration
  - `pyarrow` – efficient I/O and parquet/feather support
  - `google-generativeai` – Gemini integration
- **`README.md`**
  - Documents the output directory structure: `data_raw/`, `data_clean/`, `marts/`, `factsheets/`, `plots/`.
  - Helps users understand where to find the generated CSVs, JSON factsheet, and figures.