E-Commerce Analytics Pipeline & Dashboard

End-to-End Data Engineering and Visualization Project

Analysis of 1.8M Amazon Purchases from 5,000+ U.S. Users with Linked Demographics

Author: Alda Çadri

Date: August 2025

Technologies: Snowflake · dbt · Python · Streamlit · GitHub · VS Code

Table of Contents

1. Project Overview

This document provides a comprehensive overview of the foundational steps in the E-commerce Analytics project. The project demonstrates an end-to-end analytics solution using real-world e-commerce data from Amazon, integrating multiple file formats (CSV, JSON), enriching with demographic APIs, and transforming it using Snowflake and dbt.

You will understand why each dataset was chosen, how the Snowflake environment was set up, the ELT ingestion of large CSV files (and workaround for the UI limit), and how I enriched the pipeline with external JSON data via Python.

The results are served via a responsive and interactive Streamlit dashboard.

Key Technologies:
Snowflake · dbt · Python · Streamlit · GitHub · VS Code

Primary Goals:

- Load and cleanse raw survey and transaction data.
- Enrich transactions with external U.S. Census data.
- Apply star schema modeling (fact/dimension) using dbt.
- Create curated marts for business reporting.
- Build a visually engaging dashboard to derive actionable insights.

2. Datasets and Rationale

To simulate a production-grade pipeline and showcase multi-format ingestion, I selected:

- **amazon-purchases.csv** (≈300 MB)
  • 1.85 million+ Amazon orders from 5 027 U.S. users (2018–2022)
  • Fields: Survey_ResponseID, Order_Date, Shipping_Address_State, Purchase_Price_Per_Unit, Quantity, ASIN/ISBN_Prod_Code, Title, Category
  Reason: Real-world order volume and complexity, perfect for demonstrating bulk loading and error handling.

- **survey.csv** (5 027 rows)
  • Demographics and consumer-behaviour survey responses keyed by Survey_ResponseID
  Reason: Adds rich user attributes: age, gender, income bracket, life events for advanced segmentation analyses.

- **fields.csv** (~30 rows)
  • Metadata mapping survey field names to human-readable labels and data types
  Reason: Ensures clear documentation of every survey question and aids DBT model definitions.

- **ACS 5-Year State Demographics JSON** (2018–2023)
  • Median household income and total population for 52 geographies (50 states + DC + PR)

Reason: Demonstrates JSON ingestion and enriches order data with socioeconomic context.

3. Environment & Tools

- **Snowflake**: Central data warehouse for raw landing, ELT transformations, and analytics compute.
- **dbt**: SQL modeling framework enforcing raw - staging - refinement - delivery layers, with built-in testing and documentation.
- **GitHub**: Source control, collaboration, and CI/CD for both DBT and Python scripts.
- **Python**: Ingestion of external APIs (requests + Snowflake Python Connector).
- **VS Code**: Development environment for Dashboard scripts
- **Streamlit:** Dashboard (live Snowflake queries)

3.1 Project Environments & Dependency Management

To maintain clear separation between the backend transformations and the frontend dashboard application, this project uses two isolated Python virtual environments:

Environment 1: `dashboard_app/.venv`

- **Purpose**: Powers the Streamlit dashboard application.
- **Dependencies**: Includes libraries like `streamlit`, `altair`, `pandas`, `snowflake-snowpark-python`, and visualization tools.
- **requirements.txt**: Saved in `dashboard_app/requirements.txt`.

Environment 2: `scripts/.venv`

- **Purpose**: Dedicated to ETL and API interaction scripts, such as the `load_census.py` that ingests U.S. Census data into Snowflake.
- **Dependencies**: Lightweight—includes only necessary packages like `requests`, `python-dotenv`, and `snowflake-connector-python`.
- **requirements.txt**: Saved in `scripts/requirements.txt`.

This modular approach ensures:

- Faster dependency resolution and smaller virtual environments.
- Lower risk of dependency conflicts.
- Cleaner reproducibility of the dashboard vs. backend jobs.

Secrets Management

To avoid exposing sensitive information (Snowflake credentials, API keys), the script folder uses an `.env` file that is excluded via `.gitignore`. Variables are loaded securely using `python-dotenv`.

3.2 Repository Structure

The repository is organized to separate transformation logic, dashboard code, and supporting scripts, making it easier to maintain and navigate.

Folder & File Breakdown:

```
ECOM_ANALYTICS_PIPELINE/
│
├── dashboard_app/          # Streamlit dashboard application
│   ├── app.py              # Main dashboard script
│   ├── requirements.txt    # Dashboard dependencies
│   └── .streamlit/         # Local Snowflake secrets (excluded from GitHub)
│
├── models/                 # dbt models by layer
│   ├── source/             # Raw source references
│   ├── staging/            # Cleaning & type casting of raw data
│   ├── refinement/         # Star schema models (fact & dimensions)
│   └── delivery/           # Aggregated marts for BI & dashboard
│
├── scripts/                # Python ETL & API ingestion scripts
│
├── .gitignore              # Ignore rules (e.g., target/, secrets)
├── dbt_project.yml         # dbt project configuration
└── README.md               # Overview & run instructions
```

3.3 Project Configuration

The *dbt_project.yml* file defines the structure, configuration, and default build settings for the dbt models in this project. It ensures that each layer of the pipeline is materialized in the correct schema and with the appropriate performance trade-offs.

Purpose in This Project:

- Enforces clear schema separation: staging, refinement, and delivery.
- Optimizes performance by materializing transformation-heavy layers (staging and refinement) as **tables** for faster downstream queries.
- Keeps the delivery layer lightweight as **views**, ensuring dashboards always access the latest data without full rebuilds.
- Maintains a consistent folder structure aligned with dbt best practices.

View file in GitHub: dbt_project.yml

3.4 Job Execution – dbt Build

A manual **dbt build job** has been set up to run the complete transformation pipeline in a single command. This process compiles all models, executes transformations, and validates data quality through automated tests.

Execution Flow:

1. **Compile Models** – Translates Jinja and SQL from `/models` into executable Snowflake queries.
2. **Run Models** - Builds tables/views in the correct dependency order (raw → staging → refinement → delivery).
3. **Run Tests** - Executes all tests defined in `schema.yml` files (e.g., not null, unique, accepted values).
4. **Generate Documentation** - Running `dbt docs generate` after build creates browsable model and lineage docs.

Usage in This Project:

- Triggered manually via terminal for full refreshes and QA checks before dashboard updates.
- Ensures that all delivery layer marts feeding the dashboard are fully rebuilt and tested in one run.

4. Snowflake Setup

All objects were created in a dedicated database **ecom_analytics_db** using warehouse **ecom_analytics_wh**. Roles were provisioned to enable secure development and ingestion.

Steps:

- Created a dedicated Snowflake warehouse (`XSMALL`, autosuspend, autoresume).
- Created `ecom_analytics_db` and `raw_data` schema.
- Granted developer role access to warehouse, database, schema, and internal stages.

View setup SQL scripts in GitHub → WH, DB creation and grants.sql

5. Data Ingestion

5.1 Internal Stage for CSV Ingestion

To support data ingestion from local CSV files, an internal Snowflake stage was created. This stage served as a temporary holding area for files uploaded via the **SnowSQL CLI** and **SnowSight** before loading into raw tables. Files are first PUT to this stage, then COPY-INTO tables.

5.2. Loading CSVs & Overcoming the 250 MB UI Limit

The Snowflake Web UI is capped at 250 MB per upload. I bypassed this by using the SnowSQL CLI:

1. **Install SnowSQL** and configure ~/.snowsql/config with a connection named myconn.
2. **Upload** the large CSV files:

```
snowsql -c myconn -q "PUT file:///C:/Users/aldap/Desktop/e comm
project/data/amazon-purchases.csv @raw_data.csv_stage
AUTO_COMPRESS=FALSE;"
```

3. **COPY INTO** raw tables to complete ingestion.

5.3 Creating & Populating Raw Tables

Raw tables were created to mirror the original schema. Data was ingested using COPY INTO, with ON_ERROR='CONTINUE' for robustness.

Example tables:

- amazon_purchases_raw
- survey_raw
- fields_raw

View table creation scripts in GitHub→ creating_raw_tables

5.4. Ingesting ACS Demographics via Python

U.S. Census API was used to enrich the dataset with state-level:

- **Median household income**
- **Population**

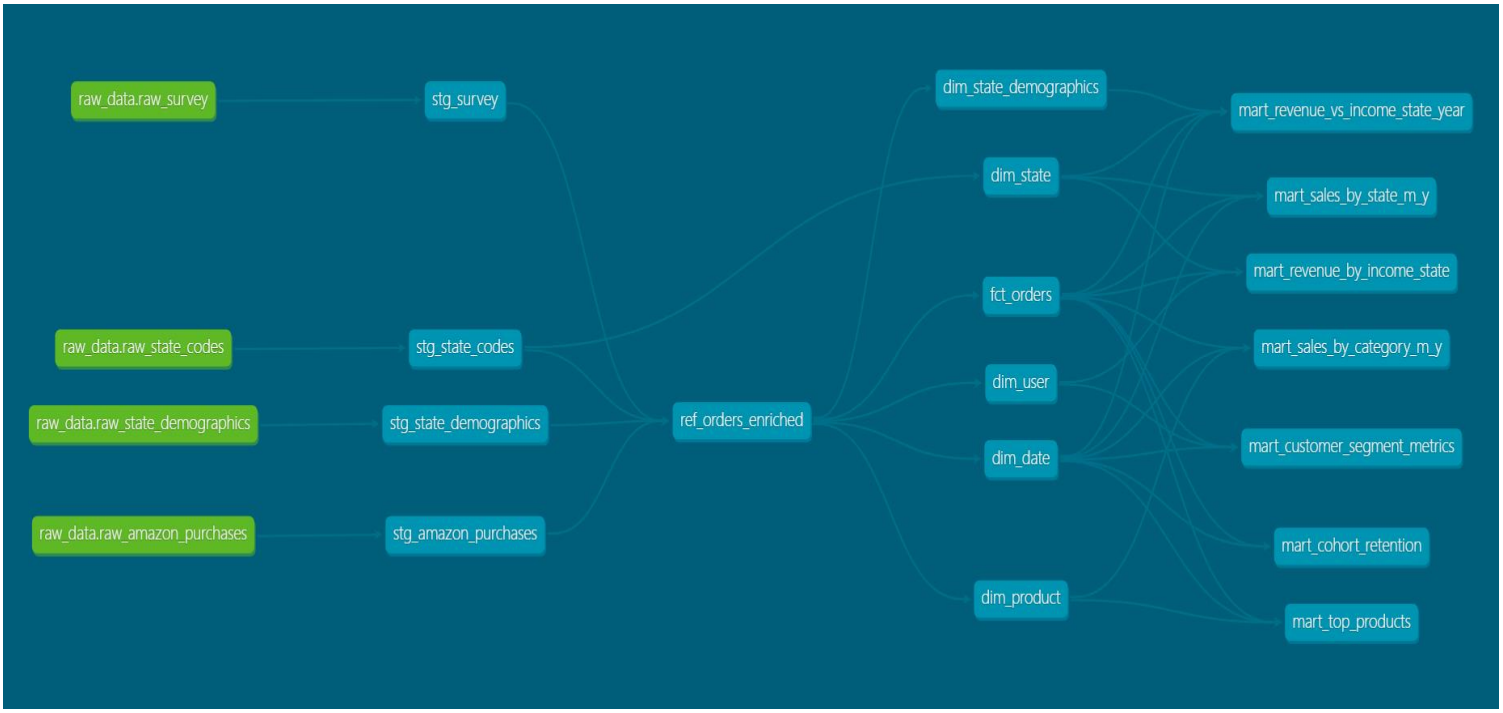
Steps:

- Python script (load_census.py) fetched and inserted JSON into state_demographics_raw
- Resulting JSON payload was transformed via dbt in the staging layer

View Python script in GitHub→ load_census.py

6. Pipeline DAG

The Pipeline DAG shows the end-to-end flow from raw data ingestion to final analytics marts. Four raw sources: transactions, survey demographics, state codes, and U.S. Census data are first cleaned and standardized in the staging layer. These feed into ref_orders_enriched, which is transformed into core fact and dimension tables in the refinement layer. Finally, specialized delivery marts are built for sales trends, customer segmentation, retention, product performance, and income–revenue analysis, powering the Streamlit dashboard.



7. DBT transformations

7.1 Staging Layer Documentation

7.1.1 Staging: Amazon Purchases (stg_amazon_purchases)

The stg_amazon_purchases model applies light structural cleaning to raw Amazon order data and prepares it for further transformations in the business logic layers (refinement and delivery).

Key Transformations Applied:

1. **Type Casting**
   - Ensures data types are consistent for calculations and joins:

```
order_date::DATE AS order_date
CAST(purchase_price_per_unit AS NUMERIC(10,2)) AS unit_price
CAST(quantity AS INTEGER) AS quantity
```

2. **Whitespace Trimming & Null Normalization**

- Remove leading/trailing spaces on all string fields.
- Convert empty strings ('') into SQL NULL via NULLIF(...,''), so missing values are explicit:

```
NULLIF(TRIM(shipping_address_state), '') AS state

NULLIF(TRIM(title), '')                  AS title

NULLIF(UPPER(category), ''), 'UNKNOWN')  AS category
```

Deduplication

Identify exact duplicate rows—where the combination of
(survey_responseid, order_date, state, unit_price, quantity, product_code)
appears more than once—using ROW_NUMBER() OVER (PARTITION BY …).

Keep only the first occurrence (rn = 1), dropping any genuine duplicates in the raw export.

```
ROW_NUMBER() OVER (PARTITION BY survey_responseid, order_date, state,
unit_price, quantity, product_code)
```

Basic Filtering

- **Keys present**: drop any row missing the survey_responseid user link.
- **Valid dates**: retain only orders between January 1, 2018 and today.
- **Positive measures**: filter out zero or negative unit_price or quantity.
- **Shipping state**: require a non-null two-letter code (physical orders).

Date Parts Extraction
Pre-compute common date dimensions to speed up downstream queries:

```
EXTRACT(YEAR FROM order_date) AS order_year

EXTRACT(MONTH FROM order_date) AS order_month

EXTRACT(QUARTER FROM order_date) AS order_quarter

TO_CHAR(order_date, 'DY') AS order_dow
```

Why in staging?

- **Performance & Consistency**: Type casting and filtering are handled once centrally
- **Safety**: Deduplication and validations prevent bad data from polluting metrics
- **Flexibility**: Structural rules are separated from business rules (e.g. segmentation)

This model acts as a reliable foundation for all joins, aggregations, and metrics used in the refinement and delivery layers.

View full script in GitHub→ stg_amazon_purchases.sql

7.1.2 Staging Layer: Survey Data (stg_survey)

This staging model takes the raw user survey export and prepares it for downstream joins, aggregations, and segmentation.

Source Table

- **Name**: raw_data.raw_survey
- **Rows**: 5 027 (one per user)
- **Key**: survey_responseid

Raw fields include demographics (Q_demos_*), usage buckets (Q_amazon_use_*), lifestyle flags (Q_substance_use_*, Q_personal_*), consent questions (Q_sell_*, Q_small_biz_use, Q_census_use, Q_research_society), and open-text life changes (Q_life_changes).

Transformations Applied

1. **Filter out blank records**

```
WHERE survey_responseid IS NOT NULL
```

Ensures every row can safely join to purchase data.

2. **Trim whitespace & normalize nulls**

- Wrap every text field in NULLIF(TRIM(...), '') to convert empty strings into SQL NULL.
- Guarantees downstream code sees true NULLs, not blank strings.

3. **Uppercase for consistency**

```
UPPER(TRIM(q_field)) AS field
```

Applied to fields like race, education, state, sexual orientation, and gender so that case variations (e.g. “female”, “Female”, “FEMALE”) become uniform.

4. **Preserve categorical buckets as text**

- Age (Q_demos_age), household size (Q_amazon_use_hh_size), account sharing (Q_amazon_use_howmany), and purchase frequency (Q_amazon_use_how_oft) remain string buckets (e.g. “25 – 34 years”, “4+”, “<5 per month”).
- Retains full survey nuance; numeric proxies can be derived later if needed.

5. **Boolean mapping for Hispanic flag only**

```
CASE
  WHEN LOWER(q_demos_hispanic) IN ('yes','true') THEN TRUE
  WHEN LOWER(q_demos_hispanic) IN ('no','false') THEN FALSE
  ELSE NULL
END AS is_hispanic
```

All other multi-choice or yes/no fields stay as text, because they include more than two possible answers.

View full script in GitHub→ stg_survey.sql

7.1.3 Staging Layer: State Demographics (stg_state_demographics)

This staging model ingests the raw **JSON-formatted state-level demographic data** from the U.S. Census API and transforms it into a structured, typed, and joinable format for downstream analytics.

Source Table

- **Table: raw_data.state_demographics_raw**
- **Source: Loaded via Python script from the ACS 5-Year Estimates API (2018–2023)**
- **Key Columns:**
  - survey_year (INTEGER): Year of the estimate
  - json_payload (VARIANT): Raw array of 4 values:
    1. State name (e.g., "Alabama")
    2. Median household income (stringified int, e.g., "54500")
    3. Total population (stringified int, e.g., "4822023")
    4. ANSI/FIPS code (e.g., "01")

Transformations Applied

1. **Load Raw Columns**

```
SELECT survey_year, json_payload
FROM {{ source('raw_data','state_demographics_raw') }}
```

2. **Parse JSON Array into Structured Columns**

Each index of the JSON payload is extracted and cast to the appropriate datatype:

```
json_payload[0]::STRING  AS state_name,
json_payload[1]::INTEGER AS median_household_income,
json_payload[2]::INTEGER AS total_population,
json_payload[3]::STRING  AS state_fips
```

- Ensures `median_household_income` and `total_population` are numeric
- Keeps `state_fips` as a string to preserve leading zeroes (e.g., `"01"` not `1`)

3. **Data Quality Filtering**

```
WHERE state_fips IS NOT NULL
  AND median_household_income >= 0
  AND total_population > 0
```

- Drops any row with missing state code
- Ensures only valid, positive population and income figures are included

View full script in GitHub→ stg_state_demographics.sql

7.1.4 Canonical Dimension: State Codes

To harmonize the three different "state" formats in the pipeline: shipping postal codes in the orders data, full state names in the survey data, and two-digit FIPS codes in the ACS demographics, a **State Codes** table was created. This canonical mapping enables every downstream model to join on a single, well-defined dimension.

1. Raw Layer: RAW_DATA.STATE_CODES

- **Table**: ECOM_ANALYTICS_DB.RAW_DATA.STATE_CODES
- **Source**: Manually loaded from state_codes.csv
- **Columns**:

Column	Type	Description
postal_code	VARCHAR(2)	USPS two-letter code (e.g. CA, NY, TX)
state_name	VARCHAR	Full state name (e.g. California, New York)
state_fips	VARCHAR(2)	Two-digit ANSI/FIPS code with leading zeros (e.g. 06)

2. Staging Layer: stg_state_codes

- **View**: DBT_ACADRI_STAGING.STG_STATE_CODES
- **SQL**:

```
{{ config(materialized='view') }}

select
  postal_code,
  state_name,
  state_fips
from {{ source('raw_data','state_codes') }}
```

3. Refinement Layer: dim_state

- **Table**: DBT_ACADRI_REFINEMENT.DIM_STATE
- **SQL**:

```
{{ config(materialized='table') }}

select
  postal_code,
  state_name,
```

```
  state_fips
from {{ ref('stg_state_codes') }}
```

7.1.5 Test Coverage & Metadata (schema.yml)

Each staging model includes automated data quality tests defined in `schema.yml`. These ensure not-null constraints, unique identifiers, valid formats, and field-level documentation.

Summary of Defined Tests

Model	Test Type	Fields Tested
`stg_amazon_purchases`	Unique combination	`survey_responseid`, `order_date`, `state`, `unit_price`, `quantity`, `product_code`
	Not null	`survey_responseid`, `order_date`
	Value constraint (> 0)	`unit_price`, `quantity`
`stg_survey`	Unique	`survey_responseid`
	Not null	`survey_responseid`
`stg_state_demographics`	Unique combination	`survey_year`, `state_fips`
	Not null + constraints	`median_household_income`, `total_population`, `state_fips`
`stg_state_codes`	Unique	`postal_code`
	Not null	`postal_code`, `state_fips`

Documentation Location: `models/staging/stg_schema.yml`
View full file: View on GitHub

Why It Matters

- **Data Quality Enforcement**: Validates assumptions before transformations run.
- **Modular Testing**: Each model documents its own field-level constraints.
- **Reliable Pipelines**: Helps catch issues like duplicates, nulls, or invalid data early.
- **Audit-Ready**: Schema tests double as a documentation layer for team onboarding and review.

7.2 Refinement Layer Documentation

The refinement layer is where business logic is applied to the clean staging tables and assemble them into a classic star schema, ready for BI, dashboards, and analytics. It consists of:

1. **Enrichment** of raw events into a single, wide fact view
2. **Metric calculations** and default-value handling

3. **Dimension tables** for lookup attributes
4. **Fact table** connecting dimensions by surrogate keys

7.2.1. Overview & Purpose

- **Goal**: Turn scattered staging views (stg_amazon_purchases, stg_survey, stg_state_demographics, stg_state_codes) into a unified analytical model.
- **Key outputs**:
  - A wide table ref_orders_enriched with one row per order, enriched with user attributes, state info, demographics, and computed metrics.
  - Dimension tables (dim_date, dim_user, dim_state, dim_product) capturing standardized lookup attributes.
  - A lean fact table fct_orders that references dimension keys and carries numeric measures.

7.2.2. ref_orders_enriched

Path: models/refinement/ref_orders_enriched.sql
Materialization: table

Transformation Step	SQL Logic	Why?
1. Flag digital orders	CASE WHEN state IS NULL THEN TRUE ELSE FALSE END AS is_digital	Identify non-shippable orders for fallback geography logic.
2. Join survey for user residence state	LEFT JOIN stg_survey u ON o.survey_responseid = u.survey_responseid	Capture user home state when state is NULL.
3. Join postal ↔ state ↔ FIPS via staging	LEFT JOIN stg_state_codes s_ship ON o.state = s_ship.postal_code LEFT JOIN stg_state_codes s_home ON u.state = s_home.state_name	Map both shipping codes and survey full names to a single dimension.
4. Join ACS demographics by FIPS + year	LEFT JOIN stg_state_demographics d ON final_fips = d.state_fips AND order_year = d.survey_year	Enrich each order with median income & population for that year.
5. Compute business metric	order_value = unit_price * quantity	Total revenue per order, core KPI for analysis.
6. Coalesce geography keys	final_postal = COALESCE(shipping_postal, home_postal) final_fips = COALESCE(shipping_fips, home_fips)	Guarantee every order has a valid state key for joins.

Output: Foundation for fact and dimension tables.

Code: See full SQL in ref_orders_enriched.sql

7.2.3. Dimensions

Each dimension is a persistent table, optimized for joins and filtered down to only the needed attributes.

a) Date Dimension: dim_date

- **Model**: models/refinement/dim_date.sql
- **Key**: date_key (order_date)
- **Attributes**: order_year, order_quarter, order_month, day_of_week
- **Why**: Enables time-series analysis by year, quarter, month, and weekday.

Code: See full SQL in dim_date.sql

b) User Dimension: dim_user

- **Model**: models/refinement/dim_user.sql
- **Key**: user_key (survey_responseid)
- **Attributes**: demographics (age_group, race, education, income_bracket, gender, sexual_orientation), household & account characteristics
- **Why**: Profiles each customer once, for segmentation and cohort studies.

Code: See full SQL in dim_user.sql

c) State Dimension: dim_state

- **Model**: models/refinement/dim_state.sql
- **Key**: state_fips (two digits state codes)
- **Attributes**: state_name, state_postal
- **Why**: Canonical lookup for all geography joins—supports choropleth maps and state-level filters.

Code: See full SQL in dim_state.sql

d) Product Dimension: dim_product

- **Model**: models/refinement/dim_product.sql
- **Key**: product_code (ASIN/ISBN)

- **Attributes**: title, category, is_gift_card flag
- **Why**: Creates a clean, deduplicated dimension table of product attributes (title, category), enabling consistent product-level analysis, segmentation, and filtering.

Key Transformation Logic

Transformation	Purpose
`COALESCE(product_code, 'UNKNOWN') AS product_key`	Replace missing product codes with a sentinel value so joins and aggregations work consistently.
Use `ROW_NUMBER()` to deduplicate by most frequent (title, category) pair per product_key	Ensures one canonical row per product, avoiding variations in names or category assignments.
`LOWER(title) LIKE '%gift card%'` → `is_gift_card`	Adds a boolean column to allow grouping and filtering by gift card vs other product types.

Code: See full SQL in dim_product.sql

e) State Demographic Dimension: dim_state_demographics

- **Model**: models/refinement/dim_state_demographics.sql
- **Key**: state_fips
- **Attributes**: state_name, state_postal, median_household_income,total_population,survey_year
- **Why**: Clean separation of geographic vs demographic dimensions.

Code: See full SQL in dim_state_demographics.sql

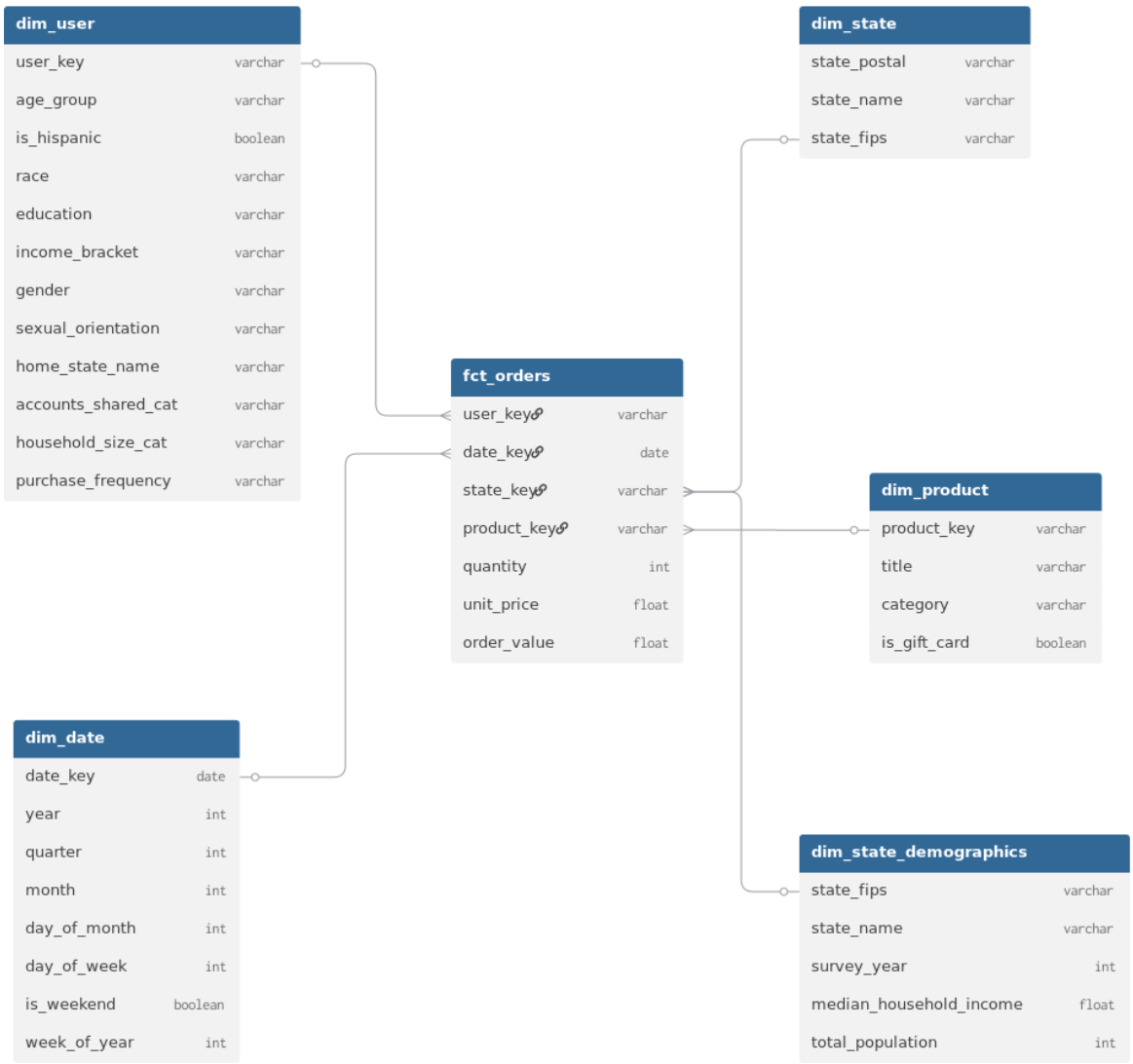
7.2.4. Fact Table

- **Model**: models/refinement/fct_orders.sql
- **Grain**: one row per order
- **Foreign Keys**:
  - user_key → dim_user
  - date_key → dim_date
  - state_key → dim_state
  - product_key → dim_product

- **Measures**: quantity, unit_price, order_value
- **Why**: This narrow fact table drives high-performant joins in BI tools, enabling fast slice-and-dice across customers, time, geography, and products.

Code: See full SQL in [fct_orders.sql]

7.2.5. Star Schema Diagram



Center: fct_orders containing all foreign keys and measures.

- **Surrounding**: five dimension tables for lookups.
- **ref_orders_enriched** sits as a denormalized precursor to the fact, useful for wide-table exports or dashboard tooling that prefers fewer joins.

Why This Matters

- **Separation of Concerns**: staging handles raw cleaning; refinement handles business logic and metric computation.
- **Ease of Use**: BI consumers can point directly at the star schema—no need to reimplement joins or coalesces.
- **Performance**: dimension tables are small and static; fact table is narrow with integer/fkey columns, ideal for OLAP engines.
- **Extensibility**: future models (e.g. customer lifetime value, cohorts, date hierarchies) can easily re-use these dimensions.

This refinement layer showcases a production-grade dbt pipeline: tested, documented, and optimized for downstream analytics.

7.2.6 Test Coverage & Metadata (schema.yml)

Summary of Defined Tests

Model	Test Type	Fields Tested
ref_orders_enriched	Not null	user_key, date_key, state_name, product_key, order_date
	Expression constraint	unit_price > 0, quantity > 0
	Not null	order_value, state_fips, final_postal
	Expression constraint	order_value > 0
	Not null (nullable fields managed)	category, title
	Nullable field documented as fallback	final_fips, home_state_name, shipping_state_name
	Implied test via coalesce	final_fips = COALESCE(shipping_fips, home_fips)
	Business logic sanity	is_digital flag correctness
	Type checks / joins	All key joins mapped to cleaned STG tables
dim_date	Unique, Not null	date_key
	Not null	order_year, order_month, order_quarter, order_dow
	Derived fields	Date parts extracted via EXTRACT()
dim_user	Unique, Not null	user_key
	Not null	All major profile columns: age_group, gender, education, etc.

	Categorical fields retained as strings	e.g., income_bracket, household_size_cat
dim_state	Unique, Not null	state_fips, postal_code
	Not null	state_name
	Dimensional lookup integrity	Ensures mapping from FIPS ↔ name ↔ postal code
dim_product	Unique, Not null	product_key
	Value inferred	is_gift_card based on keyword matching in title
	Deduplication logic	Most frequent (title, category) per product_key using row_number()
	Sentinel handling	UNKNOWN code for null product_keys
dim_state_demographics	Unique combination	(survey_year, state_fips)
	Not null + Expression	median_household_income >= 0, total_population > 0
	Accepted values	state_fips checked against full U.S. code list (e.g., '01' to '56', '72')
fct_orders	Not null	user_key, state_key, product_key, date_key, quantity, order_value
	Expression constraint	unit_price > 0, quantity > 0
	Foreign Key integrity	All keys reference dimensions: user, product, state, date

Documentation Location: `models/refinement/schema.yml`
View full file: View on GitHub

7.3 Delivery-Layer Marts

Overview

The Delivery Layer in the project is the final step in the transformation pipeline. It exposes curated, query-optimized data marts for business users and dashboards. These models are tailored for specific analytical needs, pre-aggregated for performance, and formatted for readability in BI tools like Streamlit or Power BI.

Each mart uses clean, enriched data from the refinement layer and often includes dimensions like year, month, state_name, category, or user segments to support detailed slicing and filtering.

7.3.1 mart_sales_by_state_month

- **What it does**: Aggregates orders by state/month, computing counts, total revenue, and average order value.
- **Why**: Provides geo-temporal trends at a glance. Defaulting to the last 3 years ensures the dashboard loads quickly, while exposing year lets users drill into any year from 2018–2023.

Code: See full SQL in : **mart_sales_by_state_m_y.sql**

7.3.2 mart_sales_by_category_month

- **What it does**: Similar to the state mart, but slices by product category/month.
- **Why**: Tracks how category mix shifts over seasons—key for assortment and marketing decisions.

Code: See full SQL in : **mart_sales_by_category_m_y.sql**

7.3.3 mart_customer_segment_metrics

- **What it does**: Joins fact orders to dim_user and groups by demographic buckets (age, income, Hispanic, household size), calculating total users, revenue, and order frequency.
- **Why**: Identifies high-value or under-served customer segments for targeted campaigns.

Code: See full SQL in :**mart_customer_segment_metrics.sql**

7.3.4 mart_cohort_retention

- **What it does**: Builds user cohorts by first-order month and counts how many return in each following month.
- **Why**: Illuminates customer loyalty and the effectiveness of retention initiatives.

Code: See full SQL in : **mart_cohort_retention.sql**

7.3.5 mart_top_products

- **What it does**: Ranks the top 50 SKUs by revenue and units sold.
- **Why**: Helps merchandising prioritize best-selling items and manage inventory.

Code: See full SQL in : **mart_top_products.sql**

7.3.6 mart_revenue_by_income_state_year

- **What it does**: Builds a state × year panel combining order revenue with ACS demographics (median household income, population).
- **Why**: Directly compares your commercial performance against economic context, perfect for correlation, per-capita metrics, and trend analysis.
- **Enables**: Scatter: revenue (or revenue per capita) vs. median income by year

Code: See full SQL in : **mart_revenue_by_income_state_year.sql**

7.3.7 mart_revenue_by_income_state

- **What it does**: Aggregates revenue by user income bracket × state. Joins facts to dim_user and dim_state, then sums revenue and counts orders per bucket.
- **Why**: Lets you spot where certain income segments over/under-index by geography (great for geo-targeted offers and budgeting).
- **Enables**: Comparing order mix across income tiers within a state

Code: See full SQL in : **mart_revenue_by_income_state.sql**

8. E-Commerce Analytics Dashboard Documentation

8.1 Executive Summary

This dashboard provides a comprehensive analysis of 1.8 million Amazon purchases made by over 5,000 U.S. users. The data, collected through a crowdsourced survey between 2018 and 2023, combines transactional and demographic information to uncover key trends in purchasing behavior across time, states, product categories, and customer segments.

8.2 Business Value

This dashboard answers key business questions:

- What states or segments generate the most revenue?
- Which categories or customer segments are underperforming?
- How does income influence spending behavior?
- What are the long-term retention patterns of users?
- How did revenue and income trends evolve nationally?

By surfacing these insights, businesses can optimize:

- Product category investments
- State-based pricing or advertising
- Targeted campaigns for high-value segments
- Retention strategies

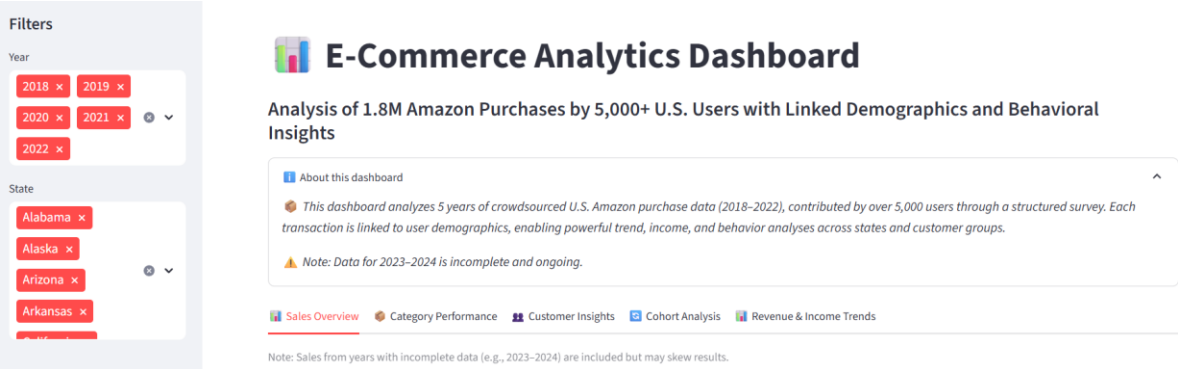
8.3 Data Source & Pipeline Context

- **Source:** Publicly available dataset of 1.8M Amazon transactions with demographics
- **ELT Flow:** Raw → Staging → Refined → Delivery (dbt model)
- **Storage:** Snowflake (Data Warehouse)
- **Transformation**: dbt (SQL), Snowpark (Python where needed)
- **Serving Layer:** Streamlit dashboard with live Snowflake queries

8.4 Data Marts Overview

- **mart_sales_by_state_m_y:** Sales by state and month
- **mart_sales_by_category_m_y:** Revenue and orders by category
- **mart_customer_segment_metrics:** Revenue/Orders grouped by age and income
- **mart_cohort_retention:** Retention rates over months after first purchase
- **mart_top_products:** Top-selling products
- **mart_revenue_vs_income_state_year :** Income-revenue correlation

8.5 Dashboard Structure & Visualizations

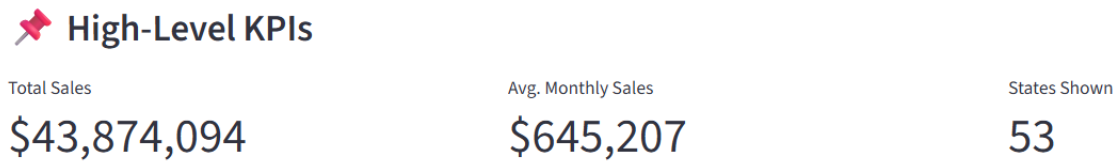


8.5.1 Tab 1: Sales Overview

Goal: Provide a high-level view of overall sales performance across time and geography.

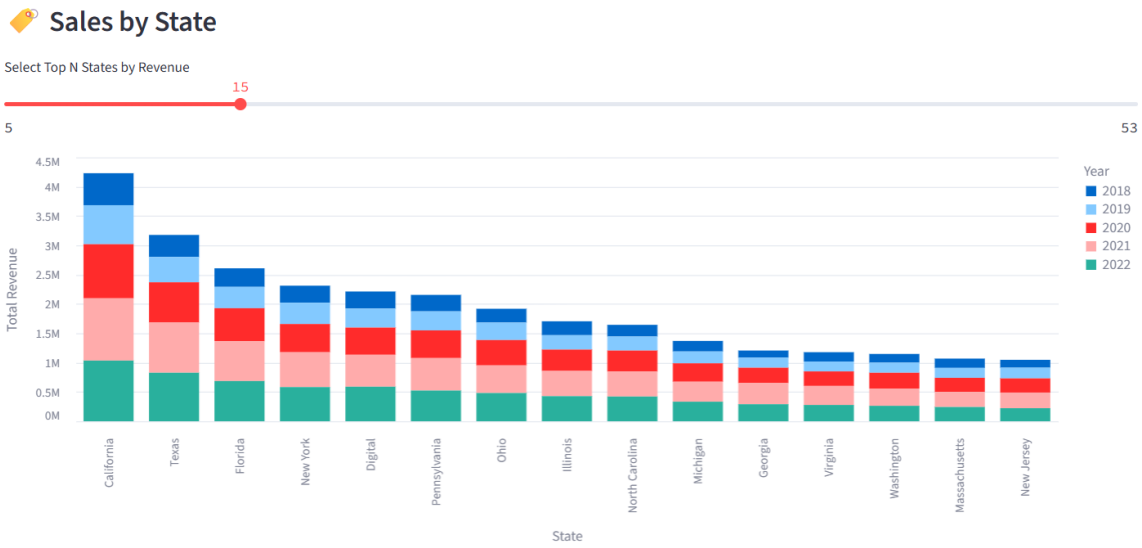
Visuals:

- **KPIs: Total sales, average monthly sales, number of states**

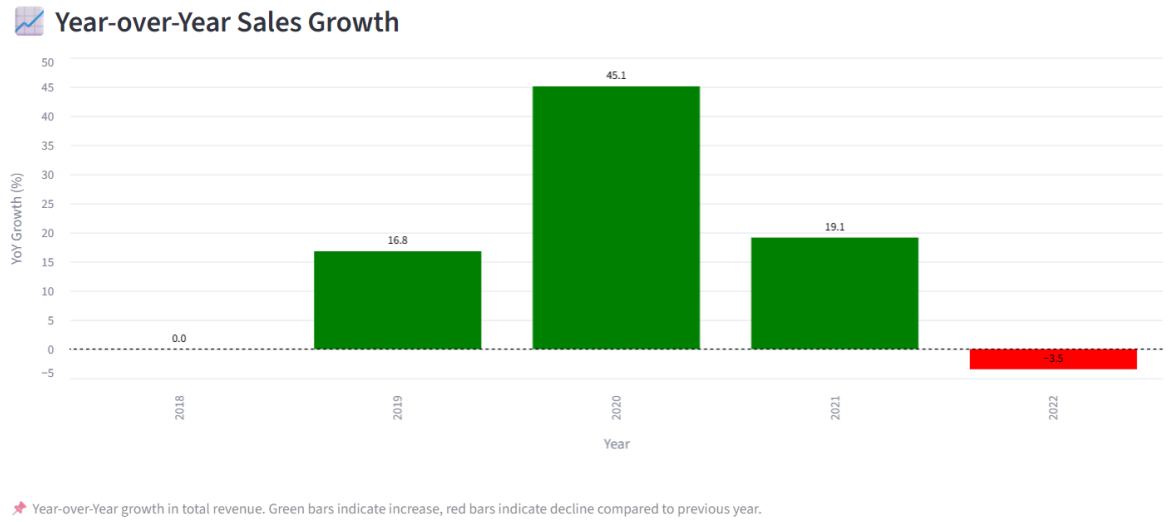


- **Monthly Sales Trend: YoY line + total bars**

- **Sales by State: Stacked bar with Top N selector**



- **YoY Growth: Growth bars with conditional color (red = decline, green = growth)**



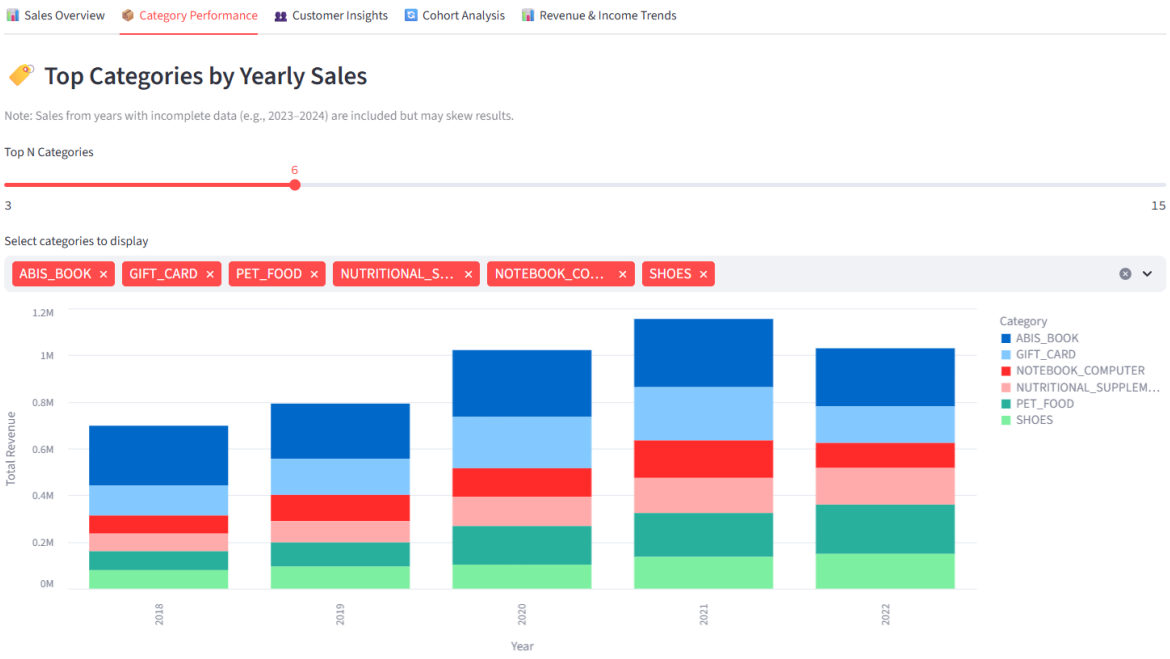
Business Insight: **Spot seasonal patterns, state-wise sales distribution, and market expansion or decline trends**

8.5.2 Tab 2: Category Performance

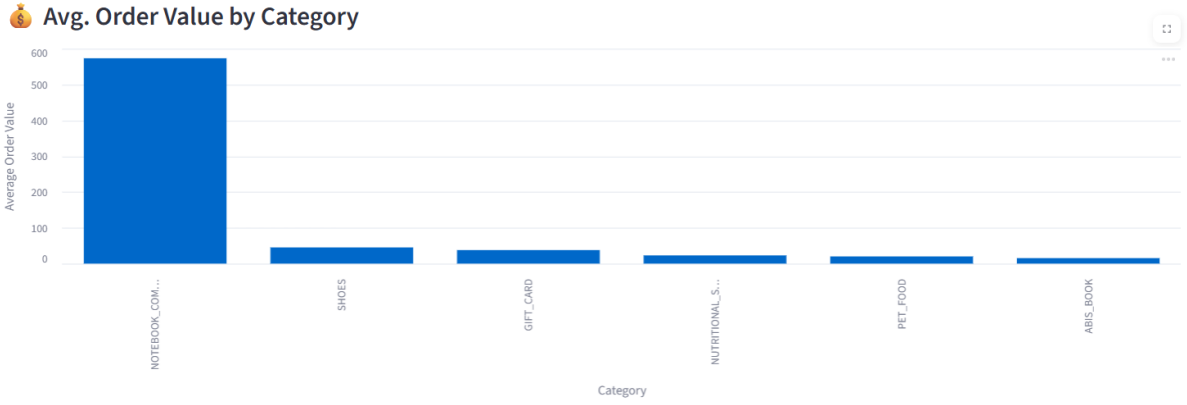
Goal: Understand which product categories drive revenue and identify underperformers.

Visuals:

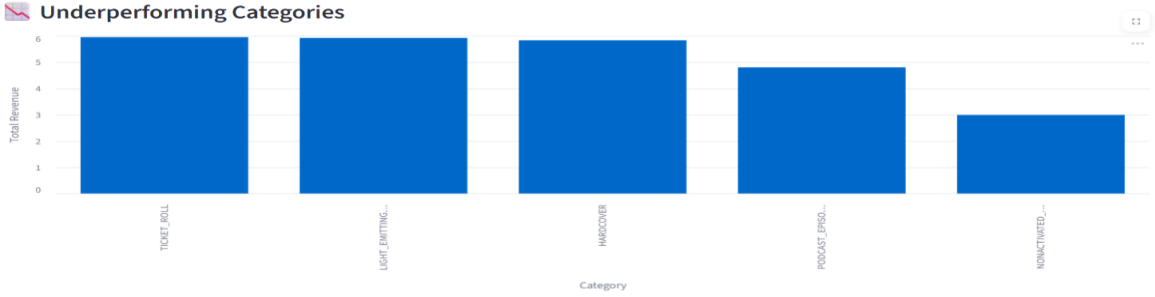
- **Top Categories by Year:** Stacked bars across years



- **Avg. Order Value by Category:** Compare price points by category

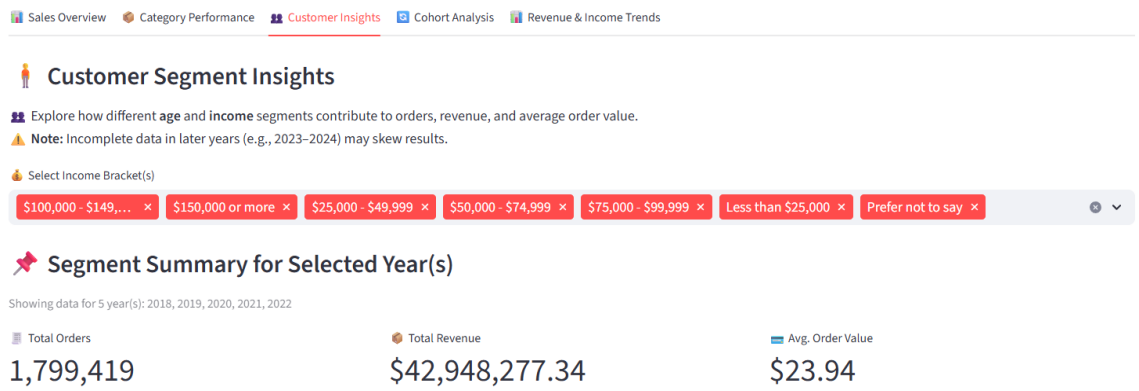


- **Underperforming Categories:** Bar chart of lowest revenue categories



📊 *Business Insight:* Optimize product assortment, pricing, and promotions.

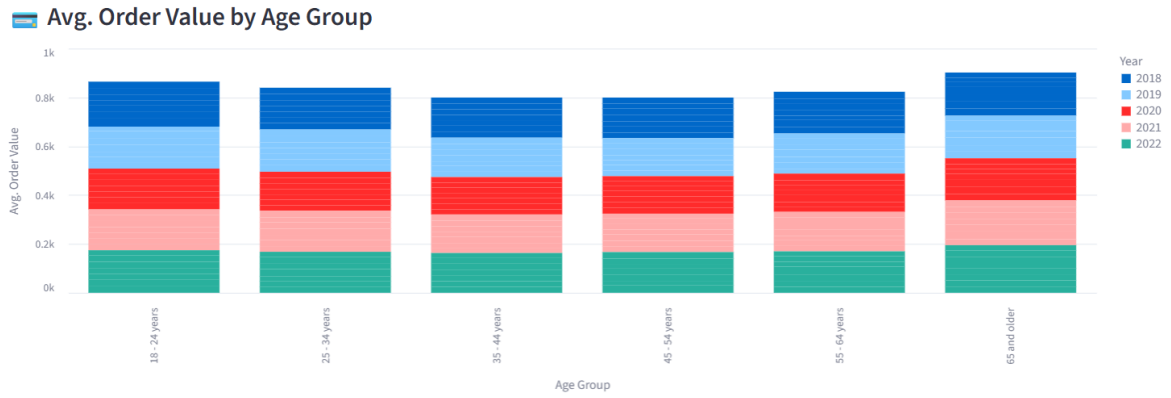
8.5.3 Tab 3: Customer Insights



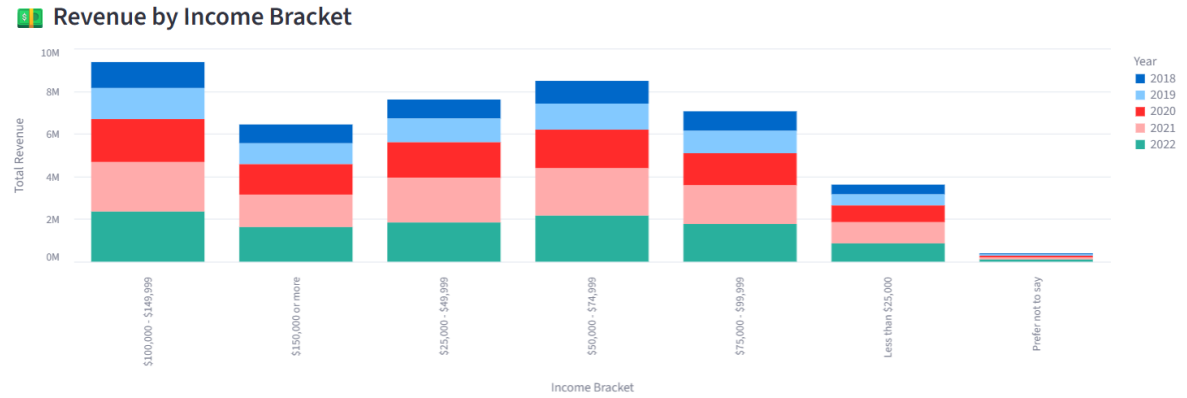
Goal: Explore performance across different demographic segments.

Visuals:

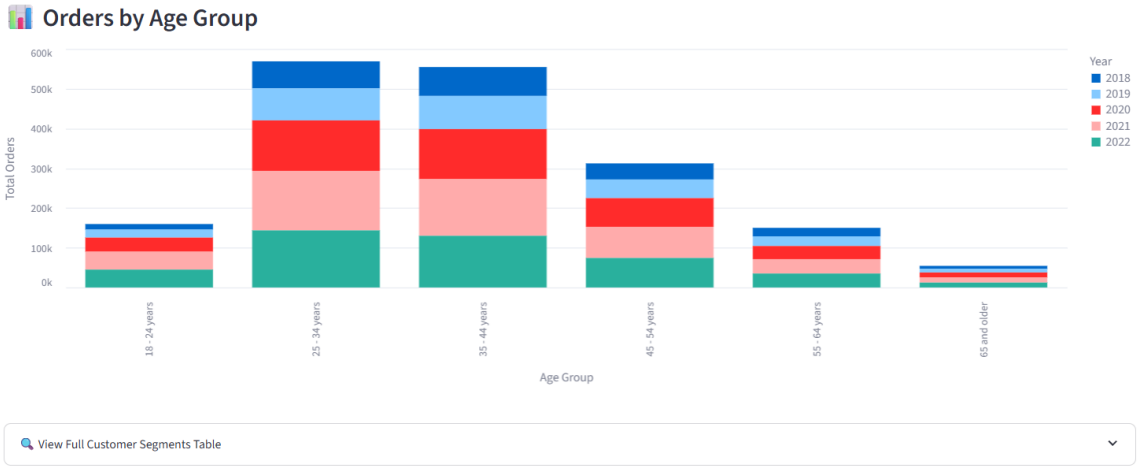
- **Avg. Order Value by Age Group:** Understand spending by age



- **Revenue by Income Bracket:** Gauge profitability by income



- **Orders by Age Group:** Demand distribution by demographic



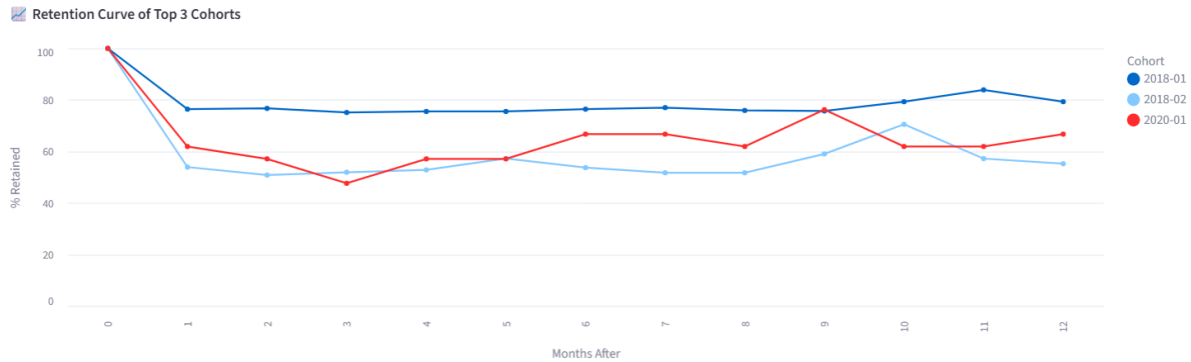
 Business Insight: Tailor marketing by age/income; understand high-value customer segments.

8.5.4 Tab 4: Cohort Retention Analysis

Goal: Analyze loyalty and retention over time.

Visuals:

- **Retention Curve (Top 3 Cohorts):** % users returning by month



📘 How to Interpret the Retention Curve

- Each line shows how a **cohort** (group of users who made their first order in the same month) retained over time.
- **X-axis**: Number of months after the first order.
- **Y-axis**: What % **of original users** from that cohort returned that many months later.
- For example, in cohort **2018-01**, **79.3%** of users were still active **12 months** after their first order.
- Retention typically drops after the first month, but cohorts with better retention show flatter curves.
- Only cohorts with **at least 20 users** are included to ensure meaningful trends.

- **Summary Table:** Retention for Months 1–3

📌 Retention Summary for Top Cohorts (Months 1–3)

	Cohort	Month 1	Month 2	Month 3
0	2018-01	76.4	76.7	75.1
1	2018-02	53.9	50.8	51.9
2	2020-01	61.9	57.1	47.6

- **Pivot Table:** Active users + retention% by cohort/month

📑 Cohort Retention Table (User Count & Retention %)

cohort_label	0	1	2	3	4	5	6	7	8	9
2018-01	2632 users (100.0%)	2011 users (76.4%)	2018 users (76.7%)	1976 users (75.1%)	1988 users (75.5%)	1987 users (75.5%)	2011 users (76.4%)	2026 users (77.0%)	1999 users (75.9%)	1993 u
2018-02	451 users (100.0%)	243 users (53.9%)	229 users (50.8%)	234 users (51.9%)	238 users (52.8%)	258 users (57.2%)	242 users (53.7%)	233 users (51.7%)	233 users (51.7%)	266 us
2020-01	21 users (100.0%)	13 users (61.9%)	12 users (57.1%)	10 users (47.6%)	12 users (57.1%)	12 users (57.1%)	14 users (66.7%)	14 users (66.7%)	13 users (61.9%)	16 use

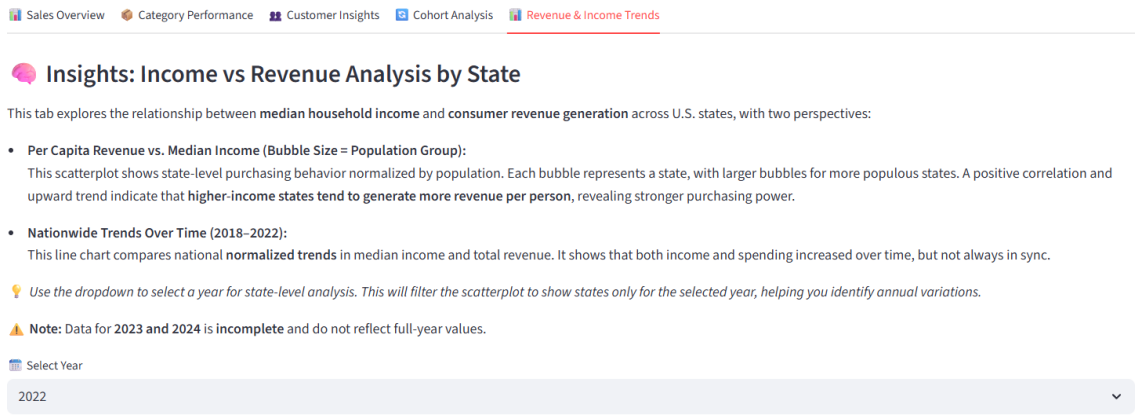
⚠ *Only cohorts with ≥20 users are included.*

🔍 Insights

- 🏆 **Top Cohorts**: 2018-01, 2020-01, 2018-02 had the highest average retention.
- 📉 Most user drop-offs occur after Month 1.
- 🧪 This analysis only includes cohorts with **at least 20 users** to ensure reliability.
- 🧭 Use the filters above to explore trends by cohort year or retention window.
- 📈 Flatter curves signal better long-term loyalty.

🔍 *Business Insight:* Detect churn patterns, evaluate loyalty strategies, identify long-lasting cohorts.

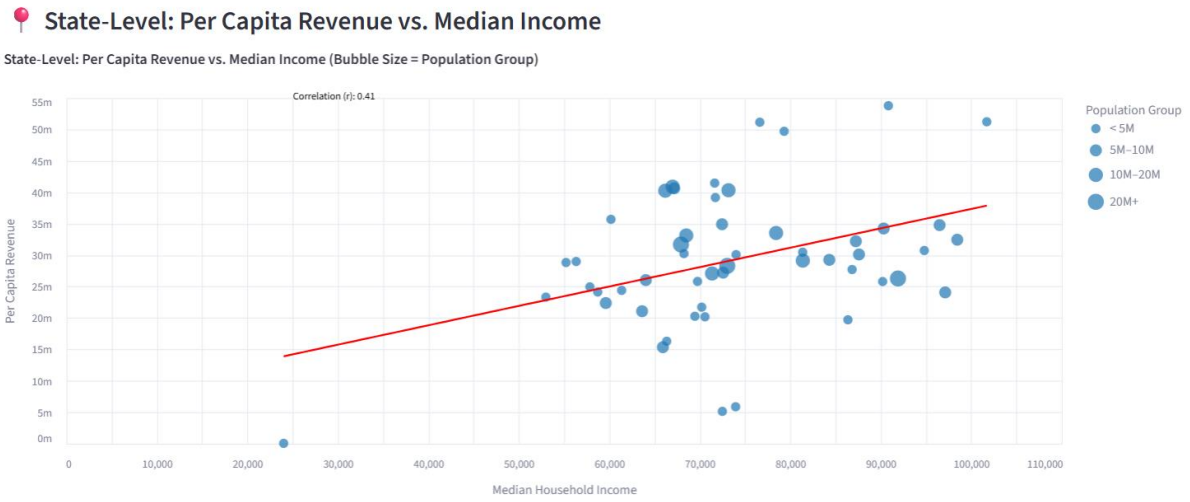
8.5.5 Tab 5: Revenue & Income Trends



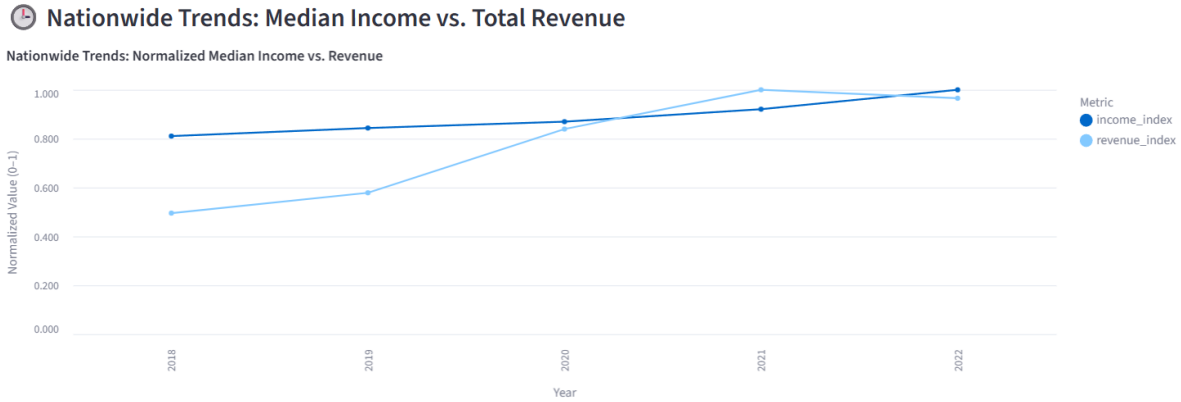
Goal: Examine the influence of state-level income and demographics on purchase behaviour.

Visuals:

- **Scatterplot:** Per Capita Revenue vs. Median Household Income
  - Bubble size = population group
  - Trendline + correlation coefficient included
  - *Dynamic by year*



- **Nationwide Trend:** Normalized median income vs revenue over time
  - *Shows decoupling or alignment between income growth and revenue growth*



☑ *Business Insight:*

- High-income states tend to spend more per person
- Revenue growth is somewhat aligned with income, but not always
- Supports decisions on geo-targeting and regional investment

9. Notes & Recommendations

- **Incomplete Data Warning:** 2023–2024 data is partial and should not be used for trend inference.
- **Customization Potential:** Add filters by category, income, or product type in more tabs.
- **Incremental Processing:** Convert large marts (e.g., cohorts, top products) to incremental models, with logic for late-arriving records.
- **Automated Data Quality Alerts:** Integrate dbt test results with Slack or email notifications to flag schema drift, null spikes, or duplicate keys.
- **Orchestration:** Schedule builds via Snowflake Tasks/Streams or dbt Cloud jobs to avoid relying solely on manual runs.
- **Materialized Views/Dynamic Tables:** For frequently queried marts, consider pre-aggregation to reduce query cost.
- **Query Observability:** Use Snowflake QUERY_HISTORY and object tags to monitor cost drivers and usage patterns.
- **Automated Data Quality Alerts:** Integrate dbt test results with Slack or email notifications to flag schema drift, null spikes, or duplicate keys.

Advanced Analytics

- **Customer Lifetime Value:** Model predicted revenue per customer for targeted marketing.
- **Market Basket Analysis:** Identify frequently co-purchased items to suggest cross-sells.
- **Price Sensitivity:** Measure order frequency vs. AOV changes.
- **Forecasting & Anomaly Detection:** Apply time-series forecasting (Prophet) and highlight unusual trends.

10. Future Work

Looking ahead, the project can be expanded to improve both technical robustness and business insight.

On the data side, full-year snapshots should be maintained for all reporting periods, with more sophisticated mapping to reduce “UNKNOWN” categories and explicit segmentation for digital orders and gift cards.

Pipeline efficiency can be enhanced by adopting incremental dbt models, late-arriving data handling, and automated test alerts integrated into CI/CD workflows. Performance can be

optimized through warehouse separation, clustering keys, and materialized views for high-demand marts.

From an analytics perspective, the dashboard could introduce advanced filtering, drill-through interactivity, and seasonal trend visualizations, alongside new metrics such as customer lifetime value, market basket associations, and churn forecasting.

Finally, integrating external datasets, and adding executive-level insight summaries would make the solution both a technical showcase and a decision-making asset.