Abstract

ABSTRACT

In today's digital economy, individuals receive a high volume of transaction alerts via SMS and email. However, this data is unstructured, inconsistent, and difficult to analyze, making manual financial tracking tedious and error-prone. This project presents the design and implementation of a **Bank Transaction Visualizer**, an interactive web application built in Python.

The application solves this problem by creating a complete data pipeline. It leverages **Streamlit** for the user interface, allowing a user to upload a simple '.txt' file containing raw transaction alerts. The core of the project is a robust parsing engine built with Python's **Regular Expression ('re') module**, which extracts key details like amount, date, vendor, and transaction type from each line.

This extracted data is then processed and structured using the **Pandas** library. A custom categorization logic maps vendors to user-friendly categories (e.g., 'Food & Coffee', 'Groceries', 'Bills & Utilities'). A key feature is a custom-built function to format currency into the **Indian numbering system** (Lakhs, Crores) for clear and region-specific financial reporting.

The final output is an interactive dashboard featuring key metrics (Total Spent, Total Income, Net Cash Flow), time-series line charts, and categorical bar charts, providing the user with immediate, actionable insights into their spending habits. This project demonstrates the power of Python, Streamlit, and Pandas in transforming messy, unstructured text into a clean, interactive data application.

Contents

Chapter 1

Introduction

1.1 Project Background

In the modern financial landscape, almost every bank transaction, from a major salary credit to a minor purchase, generates a notification. These alerts are sent as SMS messages or emails, creating a real-time log of a user's financial activity. While this data is abundant, it is inherently "unstructured." Each bank uses a different format, alerts are mixed with promotional messages, and the data itself is just plain text, not rows and columns in a spreadsheet.

This lack of structure makes it incredibly difficult for an individual to perform meaningful analysis. To understand their spending habits, a user would have to manually read hundreds of messages, copy details into a tool like Excel, categorize each expense, and then build charts. This process is time-consuming, prone to human error, and must be repeated every month.

1.2 Problem Statement

The core problem is the gap between the **availability** of raw transaction data (in text files) and the **usability** of that data for financial analysis. The project aims to automate this entire process by building a tool that can:

- Ingest unstructured text data from a single file.
- Parse and extract meaningful, structured information (amount, date, vendor).
- Clean and categorize this information automatically.
- Present the results in a simple, visual, and interactive dashboard.

1.3 Project Objectives

The primary objective of this project is to design and implement an interactive web application that serves as a personal data pipeline for bank transactions.

The specific objectives are:

1. **Implement a User-Friendly Interface:** Use Streamlit to create a simple web app where a user can upload a '.txt' file.

2. **Develop a Robust Parsing Engine:** Use Python's Regular Expression ('re') module to accurately parse varying transaction strings and extract the amount, date, and vendor.
3. **Automate Data Cleaning & Categorization:** Use the Pandas library to structure the extracted data, convert data types (e.g., to datetime), and automatically assign a spending category to each transaction based on the vendor.
4. **Create an Insightful Visual Dashboard:** Display high-level metrics (Total Spent, Income), charts (Spending by Category, Spending Over Time), and data tables.
5. **Add Custom Formatting:** Implement a specific function to display all currency values in the Indian numbering system (Lakhs, Crores) for regional relevance.

Chapter 2

Technologies Used

This project leverages a small but powerful stack of modern Python libraries to create a complete web application without the need for complex web frameworks or databases.

2.1 Python 3

The core programming language used for all logic, from file handling to data manipulation and server-side rendering of the web app.

2.2 Streamlit

Streamlit is an open-source Python library that makes it easy to create and share beautiful, custom web apps for machine learning and data science. In this project, it is used for:

- Building the entire front-end user interface.
- Providing the ‘st.file_uploader‘ widget for data ingestion.
- Rendering all interactive elements, charts (‘st.bar_chart‘, ‘st.line_chart‘), metrics (‘st.metric‘), and tables (‘st.dataframe‘).
- Running the web server to host the application.

2.3 Pandas

Pandas is the industry-standard library for data analysis and manipulation in Python. It is the backbone of the data processing phase. Its key roles are:

- Converting the list of parsed dictionaries into a structured **DataFrame**.
- Cleaning and transforming data (e.g., using ‘pd.to_datetime‘ to convert date strings).
- Performing powerful data aggregation (e.g., ‘groupby(’category’).sum()‘) to prepare data for visualization.
- Handling and filtering the data (e.g., separating ’Debit’ from ’Credit’ transactions).

2.4 Regular Expressions (re module)

This built-in Python module is the project’s parsing engine. Regex provides a powerful, declarative syntax for finding and extracting patterns from text. It is used to ”read” the unstructured alert strings and pull out the specific pieces of data (amount, date, vendor) that are needed, while ignoring the rest of the text.

Chapter 3

Process Diagram

The application functions as a linear data pipeline, transforming raw text into interactive insights. The workflow is as follows:

1. **User Upload:** The user visits the Streamlit web app and uploads their ‘transactions.txt‘ file.
2. **Read & Split:** The application reads the file, decodes it, and splits the content into a list of individual transaction lines.
3. **Parse & Extract (Loop):** The script iterates through each line. The ‘parse_transaction‘ function uses Regular Expressions to find and extract the date, amount, vendor, and transaction type.
4. **Structure Data:** The extracted data (a list of dictionaries) is loaded into a Pandas DataFrame.
5. **Clean & Categorize:** The DataFrame is processed. The ‘date‘ column is converted to datetime objects, and the ‘category‘ column is created by mapping vendors using ‘CATEGORY_MAP‘.
6. **Render Dashboard:** Streamlit uses the final, clean DataFrame to render the Key Metrics, ”Spending by Category” bar chart, and ”Spending Over Time” line chart.

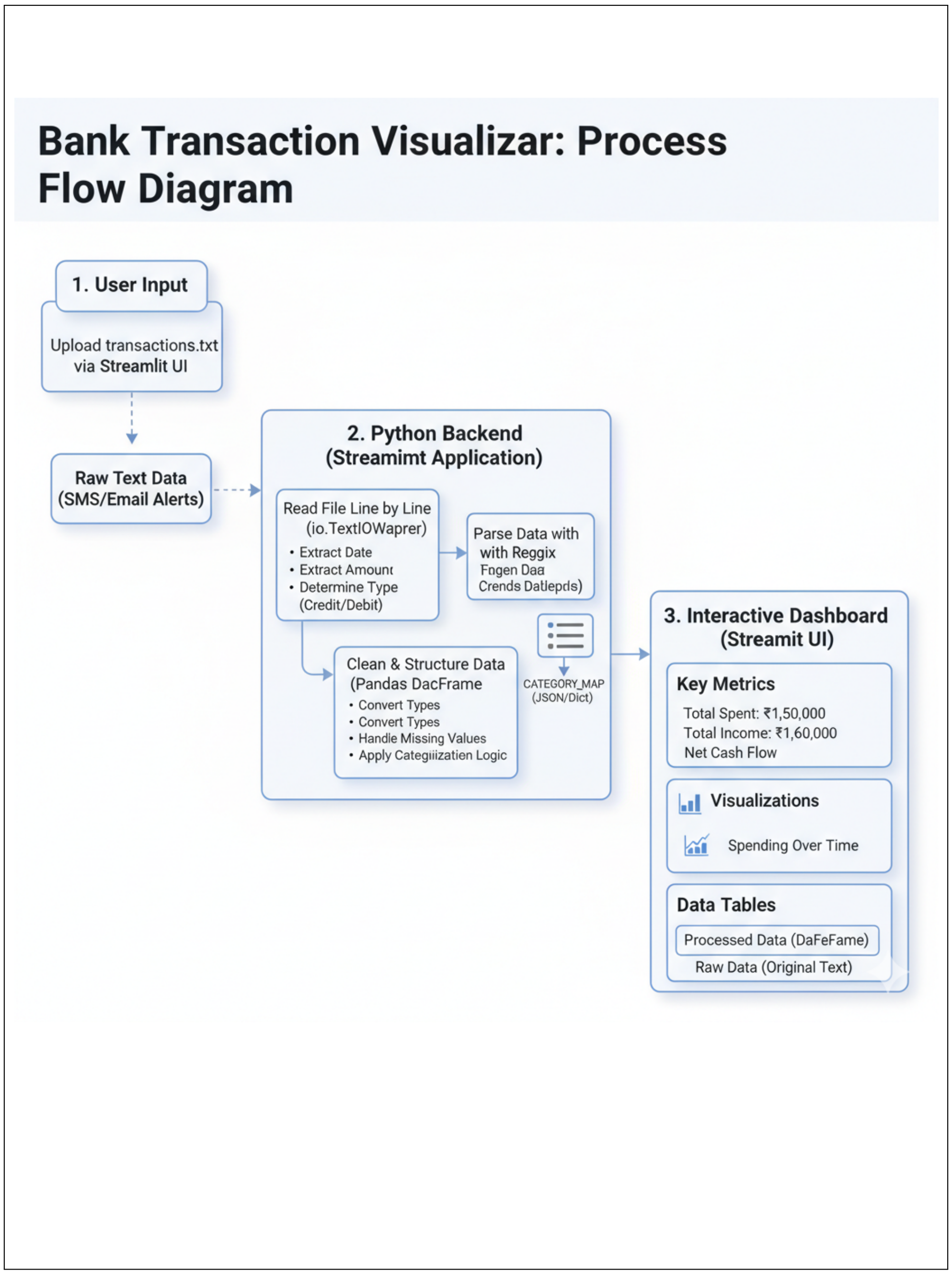

Bank Transaction Visualizar: Process Flow Diagram
1. User Input
Upload transactions.txt via Streamlit UI
Raw Text Data (SMS/Email Alerts)
2. Python Backend (Streamimt Application)
Read File Line by Line (io.TextIOWaprer)
• Extract Date
• Extract Amounι
• Determine Type (Credit/Debit)
Parse Data with with Reggix Fngen Daa Crends Datlepds)
Clean & Structure Data (Pandas DacFrame
• Convert Types
• Convert Types
• Handle Missing Values
• Apply Categiiization Logic
CATEGORY_MAP (JSON/Dict)
3. Interactive Dashboard (Streamit UI)
Key Metrics
Total Spent: ₹1,50,000
Total Income: ₹1,60,000
Net Cash Flow
Visualizations
Spending Over Time
Data Tables
Processed Data (DaFeFame)
Raw Data (Original Text)

Figure 3.1: Process Flow Diagram of the Application

Chapter 4

Implementation

The project is implemented as a single Python script (‘dashboard.py‘) that leverages several key functions to achieve its goal.

4.1 Data Ingestion

Data ingestion is handled by Streamlit’s file uploader, which is configured to accept only ‘.txt‘ files.

```python
# --- NEW FILE UPLOADER WIDGET ---
uploaded_file = st.file_uploader("Upload your transactions.txt file", type=["txt"])

if uploaded_file is not None:
    # To read file as string:
    string_io = io.StringIO(uploaded_file.getvalue().decode("utf-8"))

    # Read the string data and split it into a list by newlines
    raw_data_list = string_io.read().splitlines()
```

Listing 4.1: File Uploader and Data Reading

4.2 Categorization Logic

The categorization ”brain” is a simple Python dictionary (‘CATEGORY_MAP‘) that maps known vendor strings to clean categories. This makes the logic easy to update and expand.

```python
CATEGORY_MAP = {
    # Food & Coffee
    'STARBUCKS': 'Food & Coffee',
    'ZOMATO': 'Food & Coffee',

    # Groceries
    'SWIGGY INSTAMART': 'Groceries',
    'BIGBASKET': 'Groceries',

    # Bills & Utilities
    'PAYTM BILL PAY': 'Bills & Utilities',
    'BSES YAMUNA POWER': 'Bills & Utilities',
    ...
}
```

Listing 4.2: Snippet of the CATEGORY_MAP dictionary

4.3 Parsing Engine

The ‘parse_transaction‘ function uses ‘re.search()‘ to find patterns. Using named groups or simple ‘group(n)‘ calls extracts the relevant data. The ‘re.IGNORECASE‘ flag is used for dates to match ’Oct’ and ’OCT’ interchangeably.

```
def parse_transaction(email_body):
    # 1. Find Amount
    amount_re = re.search(r"(Rs\.|INR|\$)\s*([\d,]+\.?\d{2})", email_body)

    # 2. Find Date
    date_re = re.search(r"on\s+(\d{1,2}-[A-Za-z]{3}-\d{4})", email_body, re.
   IGNORECASE)

    # 3. Find Type (Debit/Credit)
    if "credit" in email_body.lower() or "received" in email_body.lower():
        type_ = "Credit"
    ...
    # 4. Find Vendor
    for key in CATEGORY_MAP.keys():
        if key in email_upper:
            vendor = key
            break
    ...
```

Listing 4.3: Core Regex patterns from parse_transaction

4.4 Custom Currency Formatting

A key feature is the custom ‘format_indian_currency‘ function. It formats numbers into the ”lakh, crore” system (e.g., ‘2,15,305.96‘) instead of the Western ”million” system (e.g., ‘215,305.96‘), which is critical for a user-facing financial app in India.

```
def format_indian_currency(number):
    s = f"{number:.2f}"
    parts = s.split('.')
    integer_part = parts[0]
    decimal_part = parts[1]
    ...
    l = len(integer_part)
    if l <= 3:
        formatted_integer = integer_part
    else:
        last_three = integer_part[-3:]
        other_digits = integer_part[:-3]

        # Add a comma every 2 digits to 'other_digits'
        other_digits_with_commas = ','.join([other_digits[max(i-2, 0):i] for i in
   range(len(other_digits), 0, -2)][::-1])

        formatted_integer = other_digits_with_commas + ',' + last_three

    return f"₹{sign}{formatted_integer}.{decimal_part}"
```

Listing 4.4: Indian Currency Formatting Logic

4.5 Data Visualization

The final phase uses the cleaned DataFrame to populate Streamlit's components. The 'st.metric' calls use the custom formatter, and the charts are generated directly from Pandas 'groupby' operations.

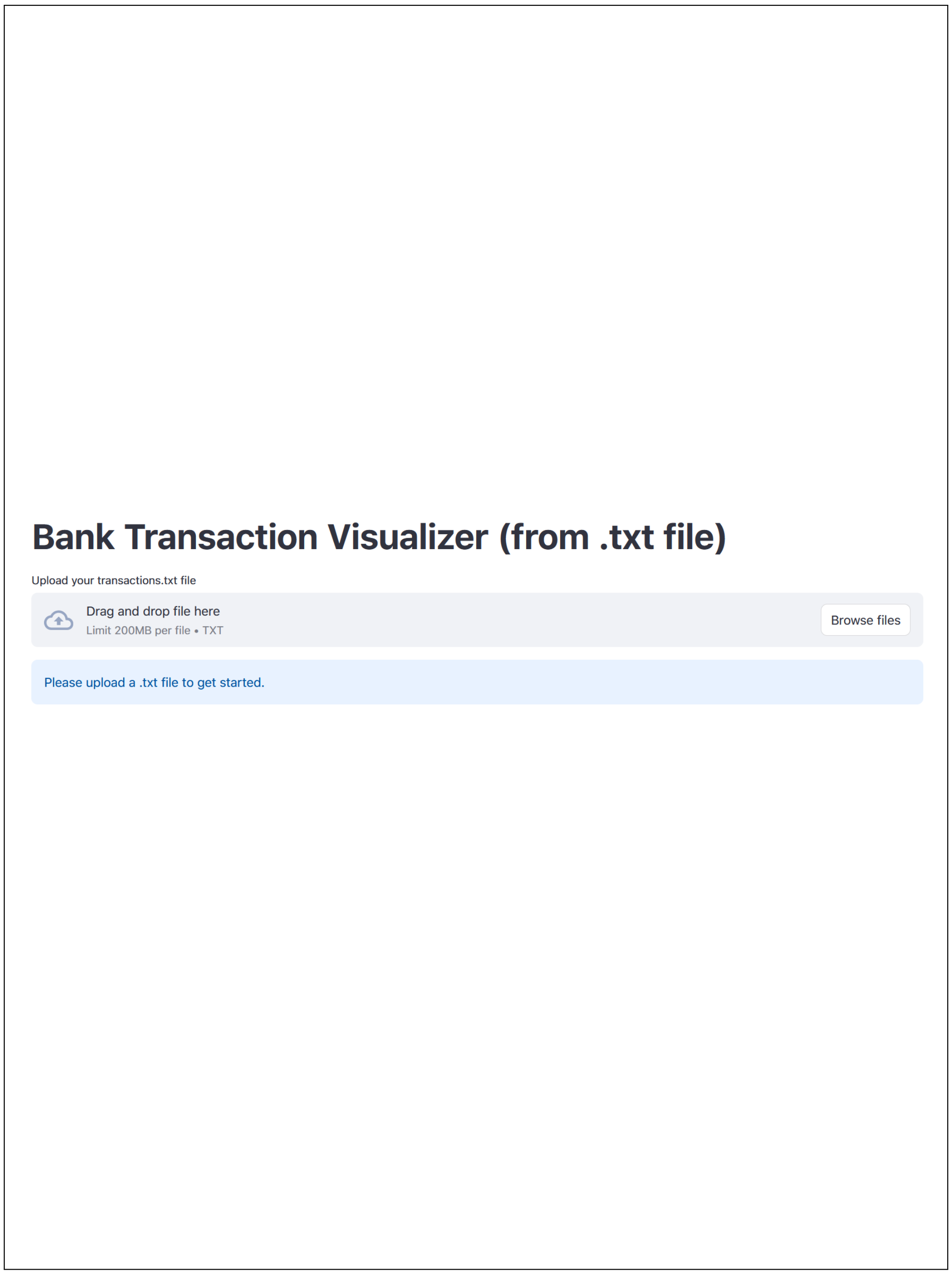
```python
    # 4. Key Metrics
    total_spent = expenses_df['amount'].sum()
    ...
    m1.metric("Total Spent", format_indian_currency(total_spent))
    ...

    # 5. Charts
    with col1:
        st.subheader("Spending by Category")
        category_spending = expenses_df.groupby('category')['amount'].sum()
        st.bar_chart(category_spending)

    with col2:
        st.subheader("Spending Over Time")
        daily_spending = expenses_df.set_index('date').resample('D')['amount'].sum()
        st.line_chart(daily_spending)
```

Listing 4.5: Dashboard Visualization Code

Chapter 5

Screenshots

This chapter displays the final output of the Bank Transaction Visualizer application, from the file upload interface to the generated dashboard.

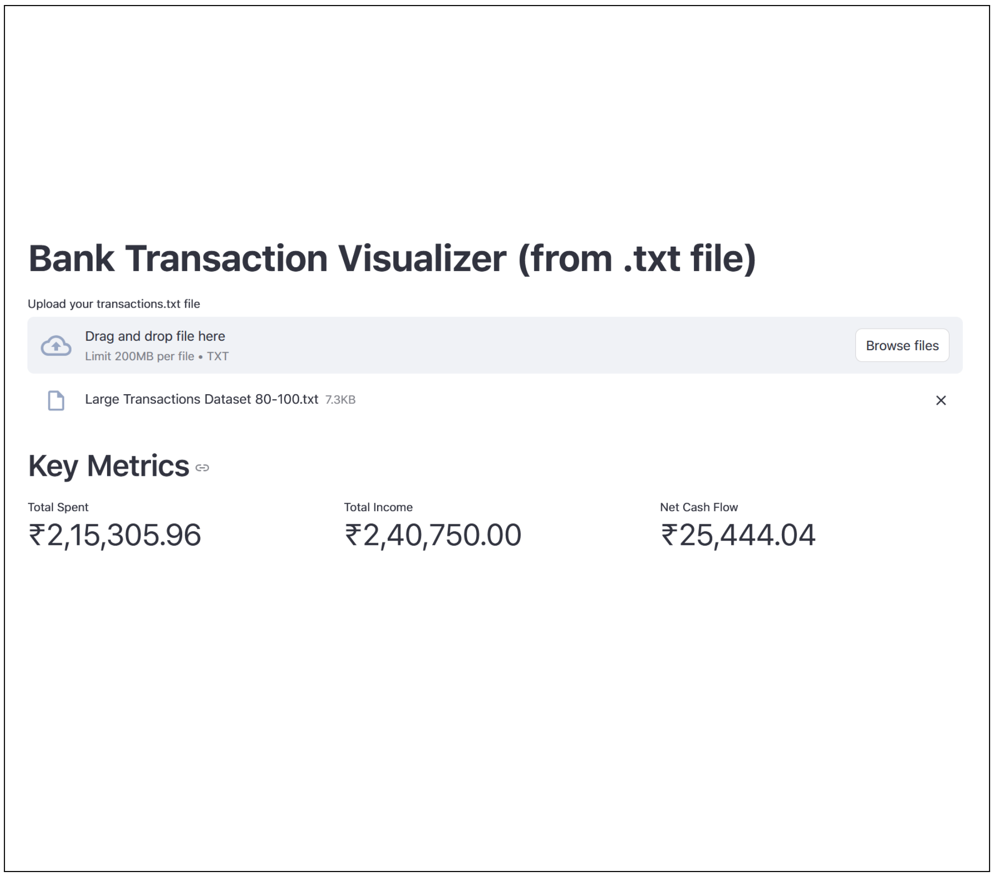


Bank Transaction Visualizer (from .txt file)

Upload your transactions.txt file

Please upload a .txt file to get started.

Figure 5.1: The main application interface showing the file uploader.


Bank Transaction Visualizer (from .txt file)

Upload your transactions.txt file

Drag and drop file here
Limit 200MB per file • TXT

Browse files

Large Transactions Dataset 80-100.txt 7.3KB

Key Metrics

Total Spent	Total Income	Net Cash Flow
₹2,15,305.96	₹2,40,750.00	₹25,444.04

Figure 5.2: Dashboard Key Metrics with Indian Currency Formatting.

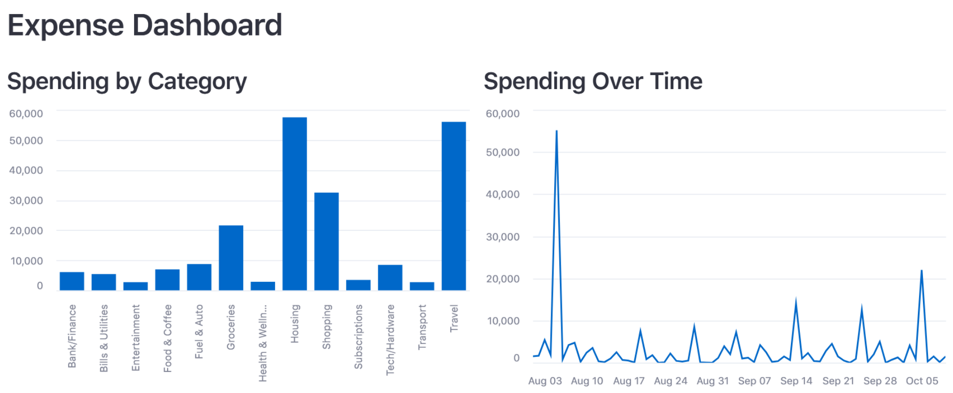
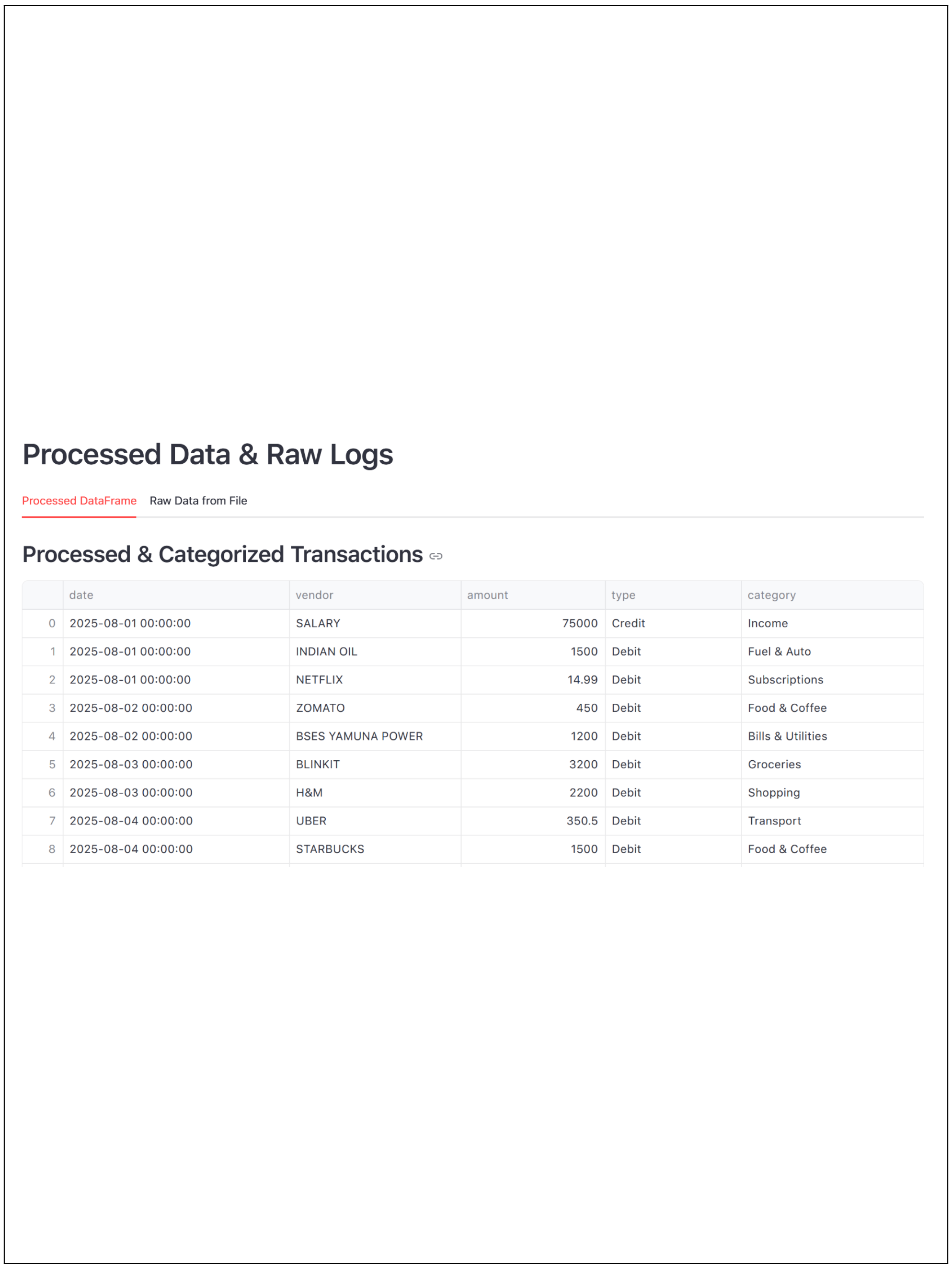


Figure 5.3: Expense Dashboard Visualizations: By Category and Over Time.



Processed Data & Raw Logs

Processed DataFrame | Raw Data from File

Processed & Categorized Transactions

	date	vendor	amount	type	category
0	2025-08-01 00:00:00	SALARY	75000	Credit	Income
1	2025-08-01 00:00:00	INDIAN OIL	1500	Debit	Fuel & Auto
2	2025-08-01 00:00:00	NETFLIX	14.99	Debit	Subscriptions
3	2025-08-02 00:00:00	ZOMATO	450	Debit	Food & Coffee
4	2025-08-02 00:00:00	BSES YAMUNA POWER	1200	Debit	Bills & Utilities
5	2025-08-03 00:00:00	BLINKIT	3200	Debit	Groceries
6	2025-08-03 00:00:00	H&M	2200	Debit	Shopping
7	2025-08-04 00:00:00	UBER	350.5	Debit	Transport
8	2025-08-04 00:00:00	STARBUCKS	1500	Debit	Food & Coffee

Figure 5.4: Tabs showing the final processed and categorized DataFrame.

Chapter 6

Conclusion

This project successfully demonstrates the creation of a complete, end-to-end data pipeline in a single Python script. The **Bank Transaction Visualizer** effectively solves the problem of unstructured financial alerts by parsing, cleaning, and visualizing them in an interactive dashboard.

The use of Streamlit for the UI, Regular Expressions for parsing, and Pandas for data manipulation proves to be a highly effective and modern technology stack. The application successfully provides immediate, actionable insights from a simple text file, empowering users to understand their financial habits without tedious manual labor.

6.1 Future Work

The project has a strong foundation and can be expanded in several ways:

- **Direct Email Integration:** The next logical step is to remove the file upload requirement. The script could be enhanced with Python's 'imaplib' to securely log in to a user's email (using an App Password), search for transaction alerts automatically, and parse them in real-time.
- **Database Persistence:** Instead of processing in-memory, the transactions could be saved to a 'SQLite' database. This would allow the app to build a long-term financial history and prevent re-processing of old emails.
- **Advanced Categorization:** A simple machine learning model (e.g., Naive Bayes) could be trained to categorize new or "Other" vendors automatically.
- **More Visualizations:** Add more charts, such as a pie chart for category distribution or filtering by date range.

References

- Streamlit Documentation. (n.d.). Retrieved from https://docs.streamlit.io
- Pandas Documentation. (n.d.). Retrieved from https://pandas.pydata.org/pandas-docs/stable/
- Python ‘re‘ — Regular expression operations. (n.d.). Python Software Foundation. Retrieved from https://docs.python.org/3/library/re.html