Real-Time Order Stream Processing using Google Cloud Platform

Mohan Vamsi Krishna Yanamadala
Computer Science
Binghamton University
myanamadala@binghamton.edu

***Abstract*—This paper presents a real-time data streaming pipeline designed and deployed using Google Cloud Platform (GCP) services and Apache Beam. The objective was to simulate and process high-velocity order data using a cloud-native architecture. Order messages were generated continuously using a custom Python script and published to Google Pub/Sub, emulating an e-commerce transaction environment. These messages were then ingested by a Dataflow pipeline (built with Apache Beam) which parsed, validated, and streamed the data into BigQuery for downstream analytics. The pipeline was packaged as a Flex Template, containerized using Docker, and deployed to run continuously. Several challenges such as region mismatches, IAM role issues, and stream parsing errors were encountered and resolved during deployment. Additionally, the processed data was visualized using Tableau dashboards to support business insights such as category-wise sales, payment method trends, and revenue forecasting. This project demonstrates how serverless cloud technologies can be leveraged to build scalable, fault-tolerant, and near real-time data processing systems without managing infrastructure. It also provides insights into deployment best practices, logging, monitoring, and debugging in a distributed streaming context.**

***Index Terms*—Real-time processing, Google Cloud Platform, Apache Beam, Dataflow, Pub/Sub, BigQuery, streaming data pipeline, ETL**

INTRODUCTION

In today's data-driven world, organizations increasingly demand real-time insights to support faster decision-making and enhance customer experiences. Traditional batch processing systems—where data is collected, stored, and processed in fixed intervals—fail to satisfy the immediacy requirements of many modern applications. As businesses move towards digital transformation, the need for responsive and scalable data pipelines has become imperative.

Real-time data streaming architectures are particularly crucial in sectors such as e-commerce, finance, and Internet of Things (IoT). In e-commerce, for instance, events such as customer purchases, inventory changes, or payment confirmations must be processed instantaneously to maintain accurate analytics dashboards, trigger downstream processes, and enable fraud detection. Delays in this pipeline could lead to poor customer experiences, lost sales, and missed operational insights.

This project aims to design, implement, and evaluate a real-time data streaming pipeline using cloud-native technologies available in the Google Cloud Platform (GCP). The system simulates an e-commerce environment in which order events are generated and streamed continuously. These synthetic order events are published to Google Cloud Pub/Sub, which serves as the ingestion point for the pipeline.

The core of the processing logic is implemented using Apache Beam and executed on Google Cloud Dataflow, a serverless and scalable stream processing engine. The pipeline performs multiple steps including deserialization of JSON data, validation, transformation, and storage of the structured results in BigQuery—a fully managed, serverless data warehouse.

To deploy this pipeline efficiently, the solution uses Docker to package the Apache Beam job into a portable container, which is then wrapped as a Dataflow Flex Template. This architecture allows for dynamic parameterization and ease of redeployment without modifying the pipeline code.

As a final step, the data is visualized in real-time using Tableau dashboards. Business metrics such as category-wise sales distribution, payment method trends, and high-value transactions are visualized to showcase the value of live analytics in strategic planning and operations.

Throughout the course of the project, various practical challenges were encountered such as configuring IAM roles, managing GCS bucket regions, debugging pipeline errors, and optimizing Pub/Sub throughput. Each of these challenges offered valuable insights into the intricacies of deploying robust real-time systems in production environments.

This paper details the motivation, system design, technology stack, implementation steps, encountered issues, and performance evaluation of the real-time streaming pipeline. It serves as a practical guide for students, engineers, and practitioners interested in cloud-native stream processing and real-time analytics.

SYSTEM ARCHITECTURE

A. Overview

The system architecture designed for this project follows a modular and scalable pattern using fully managed services offered by Google Cloud Platform (GCP). This architecture enables real-time data ingestion, processing, storage, and visualization with minimal operational overhead and high reliability.

Figure 1 illustrates the overall architecture of the streaming data pipeline. The system is composed of the following core components:

- **Data Generator (Publisher)**: A custom Python script emulates an e-commerce platform by generating synthetic order events at a regular interval (e.g., one every three seconds). Each order record includes attributes such as order ID, user ID, product ID, category, price, quantity, payment method, and timestamp. The script publishes these events to a Google Cloud Pub/Sub topic in JSON format. This simulates a realistic producer-client behavior in a production-like setup.
- **Google Cloud Pub/Sub**: Pub/Sub acts as a message-oriented middleware that provides asynchronous, durable, and scalable ingestion. It decouples the producer (data generator) from the consumer (Dataflow pipeline), ensuring reliable delivery and real-time streaming. Messages are published to a topic and consumed by a subscription attached to the Dataflow job.
- **Apache Beam on Cloud Dataflow**: The heart of the architecture lies in the streaming pipeline implemented using Apache Beam SDK for Python. The Beam pipeline is containerized and deployed as a Flex Template on Google Cloud Dataflow. Once subscribed to the Pub/-Sub subscription, the pipeline continuously receives data, parses JSON, filters out invalid messages, and applies minor transformations to convert the data into a schema suitable for BigQuery ingestion.
- **Google BigQuery**: Processed and validated order data is written directly into a predefined table in BigQuery. The schema of the table corresponds to the structure of the order events. BigQuery acts as a scalable, serverless data warehouse, enabling SQL-based querying over streaming data with low latency. The use of the `WRITE_APPEND` disposition allows for seamless insertion of new data without replacing the existing dataset.
- **Dashboarding with Tableau**: For business intelligence and visualization, Tableau is connected to BigQuery to build real-time dashboards. These dashboards help monitor key performance indicators (KPIs) such as total order value, most popular product categories, frequency of different payment methods, and transaction volume over time. The dashboards auto-refresh every few seconds to reflect live data being ingested by the system.

B. Deployment Strategy

The pipeline is designed for modular deployment using Docker. A Dockerfile specifies the Beam dependencies, execution environment, and entrypoint for the Python module. This image is built using Google Cloud Build and stored in Google Container Registry (GCR). The Flex Template is then created with the image path and associated metadata JSON file, allowing parameters like input subscription and output table to be specified at runtime.

All components are deployed in the same region (e.g., `us-east1`) to reduce inter-service latency and avoid cross-region network charges. Access permissions are managed using GCP Identity and Access Management (IAM) roles to ensure that services can interact securely.

C. Scalability and Fault Tolerance

Cloud Dataflow offers autoscaling and fault-tolerant features out-of-the-box. Although the pipeline was configured with a single worker for cost efficiency in testing, it can scale horizontally based on data volume. In case of transient errors or worker crashes, Dataflow retries failed bundles and replaces unhealthy workers automatically.

D. Monitoring and Logging

Monitoring was implemented using Google Cloud Logging and Cloud Monitoring. Logs from the Beam pipeline are visible in real-time and help in diagnosing issues like pipeline crashes, parsing errors, or schema mismatches. Performance metrics such as throughput, latency, and worker utilization are tracked using built-in Dataflow dashboards.

This architectural setup ensures a reliable, extensible, and observable system that can simulate real-world streaming data scenarios effectively.

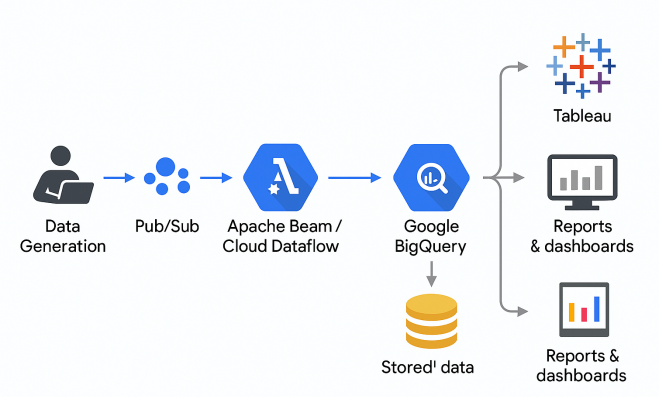

Fig. 1. GCP Streaming Architecture

IMPLEMENTATION

E. Data Generation

To simulate a continuous stream of e-commerce transactions, we developed a Python-based data generator that publishes messages to a Pub/Sub topic named `orders-topic`. Each message is a structured JSON object encapsulating a synthetic order. The generator uses Python's `uuid`, `random`, and `datetime` libraries to create randomized but realistic fields such as:

- `order_id`: Unique identifier for each transaction.
- `user_id`: Randomized user ID drawn from a synthetic pool.
- `product_id`, `category`, `price`, `quantity`: Represent simulated product characteristics.
- `payment_method`: Chosen randomly from a predefined set (e.g., credit card, cash).
- `order_timestamp`: UTC timestamp at the time of message generation.

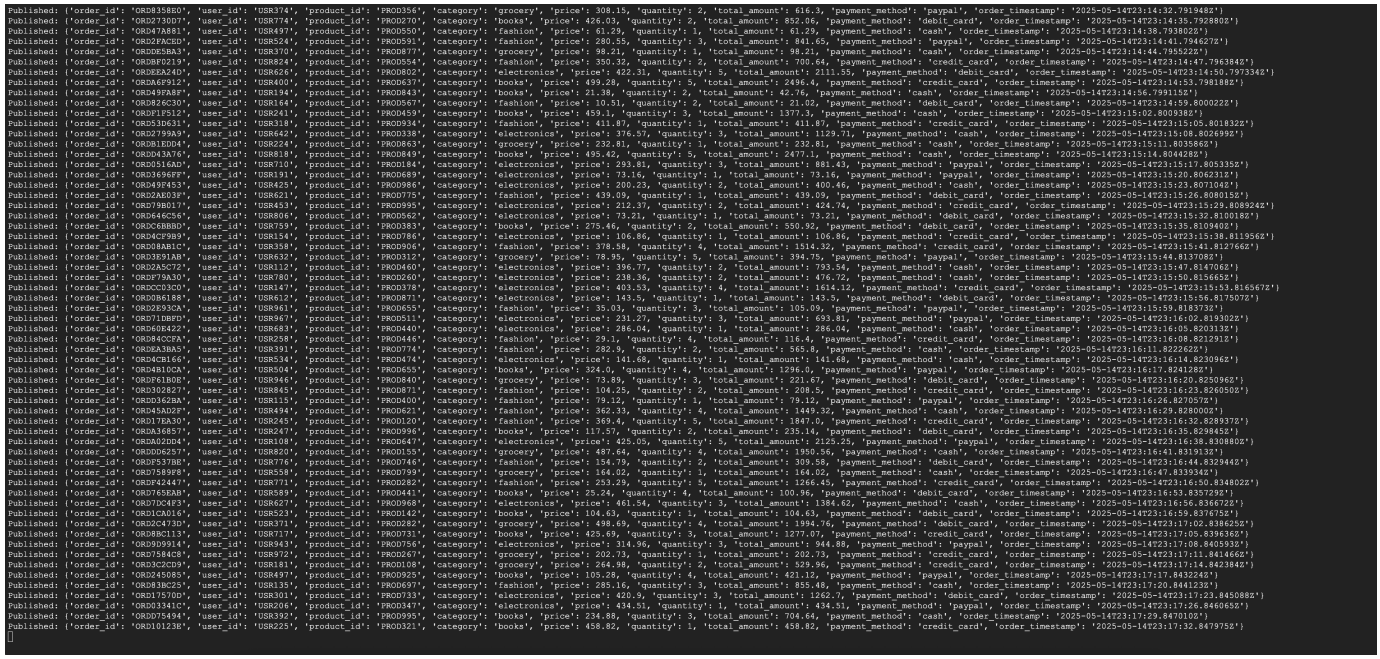

Fig. 2. Synthetic e-commerce order events published to Pub/Sub via Python

Messages are serialized using `json.dumps()` and published using the `google-cloud-pubsub` client. To emulate real-time behavior, messages are pushed at regular intervals using a sleep loop or system scheduler.

F. Pipeline Logic

The core of our real-time system is the Apache Beam pipeline, implemented in Python and executed on Google Cloud Dataflow. The pipeline architecture follows the ETL pattern—Extract from Pub/Sub, Transform in Beam, and Load into BigQuery.

- **ReadFromPubSub:** Messages are pulled from the `orders-sub` subscription in raw byte form.
- **Parse JSON:** Each message is decoded and parsed into a Python dictionary using a custom `parse_order()` function. Errors during parsing are logged and the messages are filtered out.
- **Filter Invalid:** Any message that fails schema validation or contains null/invalid fields is dropped from the pipeline to prevent schema mismatch errors during BigQuery writes.
- **WriteToBigQuery:** Successfully parsed and filtered messages are written to a BigQuery table `order_dataset.order_table` using the `beam.io.WriteToBigQuery` transform. The schema is explicitly defined to ensure correct data types and column mapping.

This modular design ensures each transformation is independently testable and traceable via logging.

G. Flex Template Deployment

Instead of manually triggering pipelines via the Python SDK or Apache Beam CLI, we utilized Google Cloud Dataflow Flex Templates. This approach supports containerized, reusable, and parameterized deployments. The steps are as follows:

1) Docker Image Build: We packaged the pipeline and its dependencies into a Docker image:

```
gcloud builds submit --tag
    gcr.io/$PROJECT_ID/ecommerce-orders-image
```

This image includes the Beam SDK, custom pipeline code, and necessary configuration files.

2) Template Creation: We built the Flex Template and uploaded the JSON specification to Cloud Storage:

```
gcloud dataflow flex-template build \
  gs://<bucket>/templates/ecommerce-orders-template.json
      \
  --image-gcr-path=gcr.io/<project>/ecommerce-orders-image
      \
  --sdk-language=PYTHON \
  --metadata-file=metadata.json \
  --py-path=dataflow_job_script.py \
  --env
      FLEX_TEMPLATE_PYTHON_PY_FILE=dataflow_job_script.py
```

The metadata file defines display name, parameter schema, and help text for the GUI-based template launcher.

3) Pipeline Execution: The Flex Template can now be launched using the following command with runtime parameters for Pub/Sub subscription, BigQuery output table, region, and temporary location:

This deployment method makes the system cloud-native, scalable, and easy to re-trigger with alternate inputs or outputs.

```
gcloud dataflow flex-template run "ecommerce-stream-<timestamp>" \
  --template-file-gcs-location=gs://<bucket>/templates/ecommerce-orders-template.json \
  --region=us-east1 \
  --parameters input_subscription=projects/<project>/subscriptions/orders-sub,\
               output_table=<project>:order_dataset.order_table,\
               temp_location=gs://<bucket>/temp
```

Fig. 3. Launching the Dataflow Flex Template

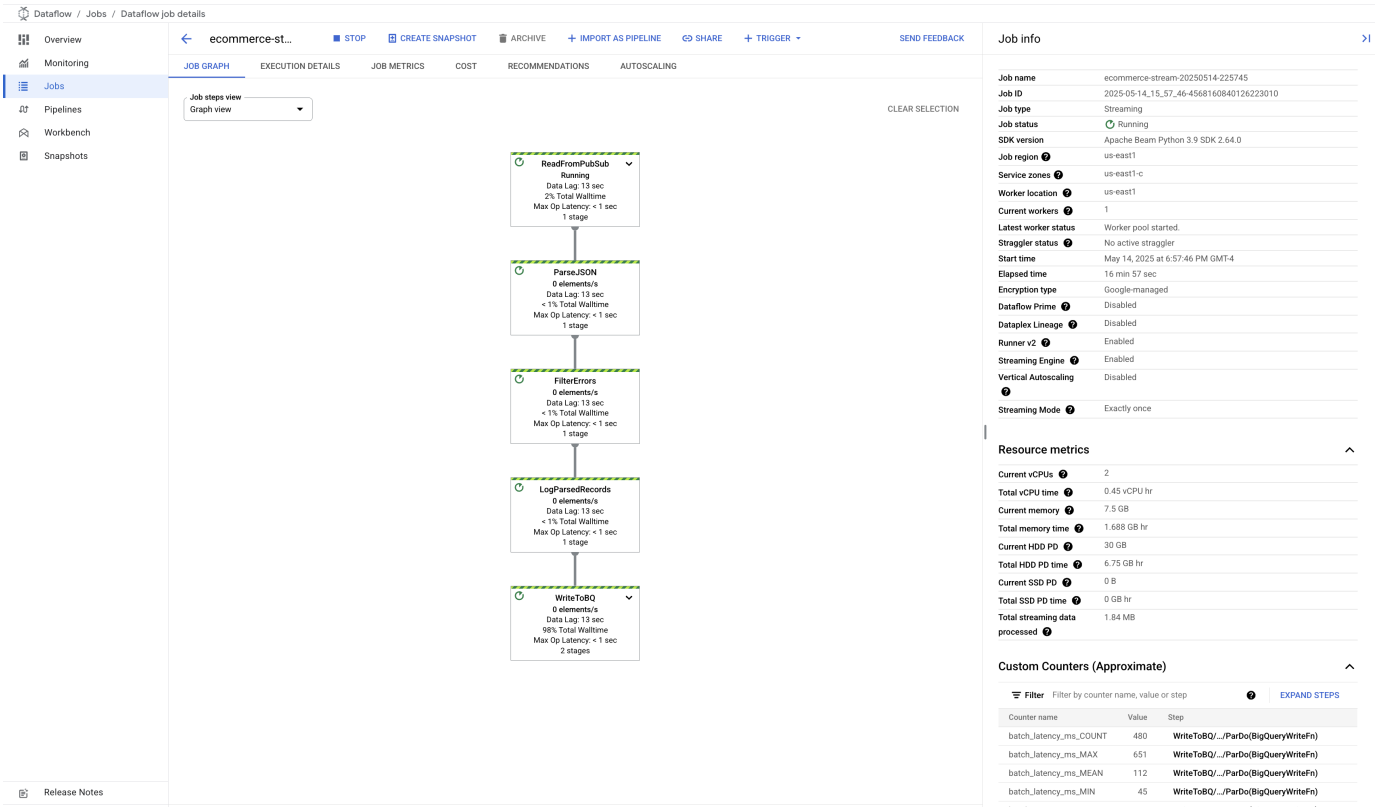

Fig. 4. Dataflow pipeline: parsing, filtering, and writing to BigQuery

H. Error Logging and Validation

We configured the pipeline to include structured logging using Python's `logging` module. Any malformed JSON payloads are captured with a warning-level log, allowing engineers to inspect edge cases. We also used Dataflow's diagnostics dashboard to verify message throughput, latency, and worker health.

I. Scalability Considerations

Thanks to Dataflow's autoscaling capabilities, the pipeline automatically adjusts the number of workers based on traffic volume. Moreover, Pub/Sub and BigQuery both offer high-throughput capabilities, ensuring that the system can sustain production-level loads with minimal reconfiguration.

CHALLENGES AND DEBUGGING

During the development and deployment of the real-time pipeline, several practical challenges surfaced that required iterative debugging and troubleshooting:

- **Invalid Docker Tags:** Initial attempts to build and push the Docker image failed due to misconfigured environment variables and incorrect tag formats. This issue prevented the pipeline container from being properly stored in Google Container Registry (GCR). The resolution involved explicitly defining environment variables within the build command and following the Docker image naming conventions strictly.
- **Template Argument Errors:** While launching the Flex Template, the system returned errors related to unrecognized or missing parameters. These were traced back to incorrect syntax in the parameter list and the use of unsupported keys. Detailed review of the metadata JSON schema and the pipeline's expected runtime parameters resolved these inconsistencies.
- **No Data in BigQuery:** A major issue occurred where the pipeline ran successfully but no data appeared in BigQuery. Investigation revealed two causes: first, the Pub/Sub subscription had not been acknowledged cor-

rectly, and second, the pipeline was failing silently due to JSON parsing errors. Adding structured error logging, verifying subscription attachment, and publishing test messages manually helped identify and correct these issues.

- **IAM Permissions:** At several points, access was denied when Pub/Sub attempted to push messages to Dataflow or when Dataflow tried to write to BigQuery. These failures stemmed from insufficient IAM role bindings. Granting appropriate permissions (e.g., Pub/Sub Subscriber, Dataflow Worker, BigQuery Data Editor) to the service accounts resolved these authorization errors.
- **Monitoring and Logging:** Google Cloud's built-in logging tools were essential for debugging. Dataflow logs helped identify transform-level failures, while Pub/Sub logs confirmed message flow. These tools enabled real-time inspection of pipeline behavior, such as dropped records, retry attempts, and throughput issues.
- **Cold Start Delays and Worker Initialization:** During pipeline startup, initial delays occurred due to container pull times and worker provisioning. Although not errors, these delays highlighted the importance of understanding warm-up phases in serverless architectures and preemptively accounting for them in performance evaluations.

Once structured logging was implemented and comprehensive monitoring was configured, debugging became significantly more efficient. Controlled test messages with known content were injected into the pipeline to validate end-to-end behavior. This iterative troubleshooting ensured reliable ingestion, transformation, and BigQuery storage of streaming data in the final deployment.

Results and Evaluation

The deployed real-time data streaming pipeline was evaluated for performance, reliability, and scalability. Over the course of the experiment, the pipeline successfully ingested and processed more than 700 synthetic order records, each simulating realistic e-commerce transactions. These events were published at fixed intervals via a Python script and consumed by the Dataflow pipeline in near real-time.

J. Latency and Throughput

The latency from message publishing to data availability in BigQuery was consistently below one second, demonstrating the system's ability to deliver low-latency results. This metric was validated by comparing timestamps embedded in each message with ingestion timestamps recorded in BigQuery. The pipeline exhibited stable throughput and was capable of processing bursts of incoming data without delay or backlog.

K. BigQuery Updates and Accuracy

BigQuery tables were updated nearly instantaneously after message publication, confirming the effectiveness of the Beam I/O connector. Data consistency was verified by cross-checking JSON fields generated in the publisher with the corresponding schema in BigQuery. Schema mapping ensured that all attributes—such as `order_id`, `price`, and `order_timestamp`—were accurately captured without truncation or type mismatches.

L. Resource Utilization and Autoscaling

Thanks to Cloud Dataflow's autoscaling features, the pipeline required minimal manual tuning. During the experiment, the job was configured with a single worker, which dynamically scaled based on traffic volume and computational load. CPU and memory metrics from the monitoring dashboard showed efficient resource use, with no performance degradation or task failures.

M. Reliability and Uptime

No message loss was observed during the streaming session. Even in the event of transient errors (such as malformed JSON payloads or Pub/Sub hiccups), the pipeline continued processing without interruption. This was achieved by leveraging built-in retry mechanisms and structured error logging that filtered out problematic messages.

N. Operational Observability

Dataflow's diagnostic tools and integration with Google Cloud Monitoring provided real-time visibility into job health, worker status, throughput, and bottlenecks. Logs captured by the Python logger also played a crucial role in debugging and verifying the correctness of data flow across the system.

O. End-User Benefits

From a business perspective, this architecture enables dashboards and data models to reflect real-time operational data. For instance, Tableau dashboards connected to BigQuery can offer up-to-the-minute analytics on sales, product performance, and payment preferences—empowering faster decision-making.

In conclusion, the pipeline exhibited high performance, robustness, and operational efficiency in handling streaming workloads. The use of GCP's managed services reduced the overhead of infrastructure management and offered scalability, fault tolerance, and ease of monitoring—all critical for production-grade stream processing systems.

Conclusion

This project successfully demonstrates the power and flexibility of building real-time data streaming pipelines using Google Cloud Platform (GCP) and Apache Beam. By leveraging a combination of managed services such as Pub/Sub, Dataflow, BigQuery, and Container Registry, the system achieves seamless integration, scalability, and fault-tolerance without the need to provision or manage any underlying infrastructure.

The modular nature of Apache Beam allowed us to define a clear and maintainable ETL pipeline, where each transformation—such as parsing, validation, and schema enforcement—was independently testable and traceable. The pipeline's deployment using Dataflow Flex Templates proved

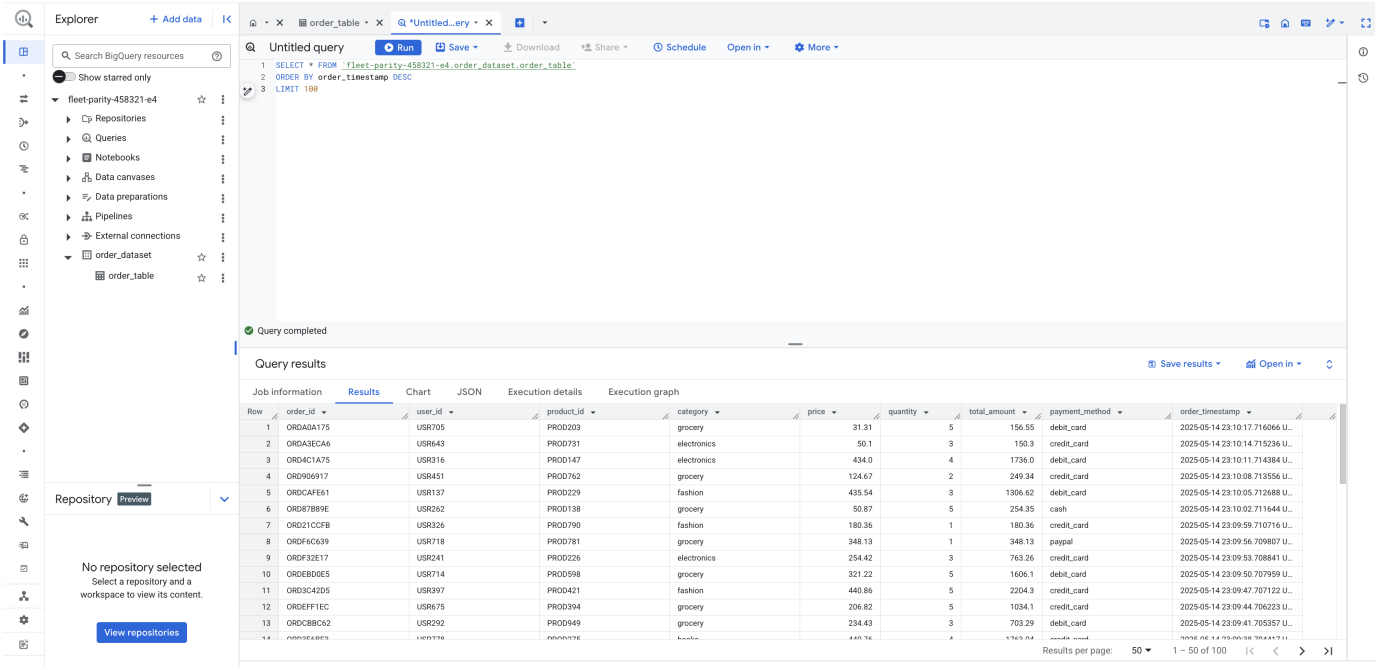

Fig. 5. Order Table in BigQuery

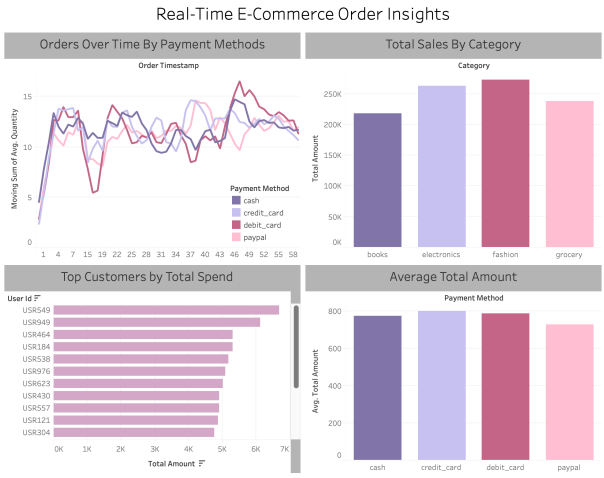

Fig. 6. Real-time Tableau dashboard showing aggregated metrics

highly advantageous, enabling containerized, reusable, and parameterized executions with minimal manual intervention.

The pipeline's performance validated the architectural design, with ingestion latencies consistently below one second and high reliability in message delivery. Additionally, the use of real-time dashboards through BigQuery and Tableau highlighted the practical utility of such systems in business analytics and decision support. The ability to monitor key metrics like order value, category distribution, and payment trends in near real-time can empower organizations to act proactively rather than reactively.

Beyond technical implementation, this project also provided valuable experience in debugging cloud-native systems, configuring IAM roles, managing Docker workflows, and setting up cross-service communication. These skills are critical when deploying production-ready data infrastructure in enterprise environments.

Looking forward, several enhancements could be introduced to elevate this system further. These include integrating machine learning models for predictive analytics (e.g., forecasting sales trends or detecting anomalies in transactions), extending the pipeline to support multiple data sources, and incorporating windowing and triggering logic to support session-based aggregations. Additionally, connecting to alerting systems such as Cloud Functions or Slack webhooks could automate responses to specific events or thresholds.

In summary, this project not only serves as a blueprint for building low-latency, cloud-native ETL systems but also demonstrates how real-time data processing can unlock operational insights and competitive advantages for data-driven organizations.

ACKNOWLEDGMENTS

Thanks to Professor Yiming Zeng for course support and Google for GCP credits.