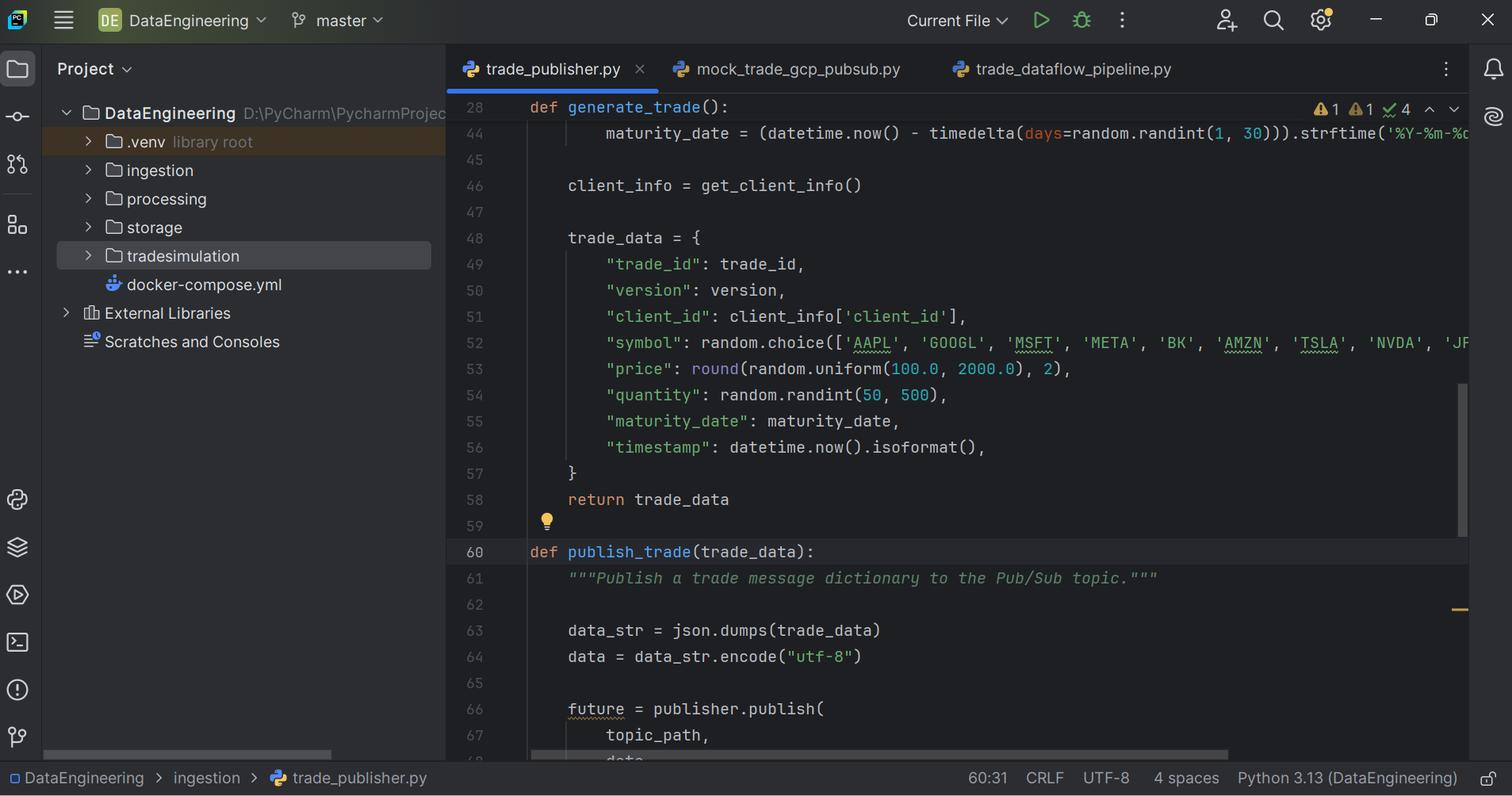
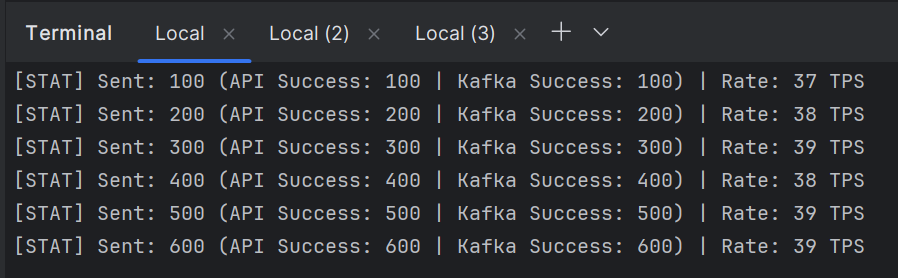
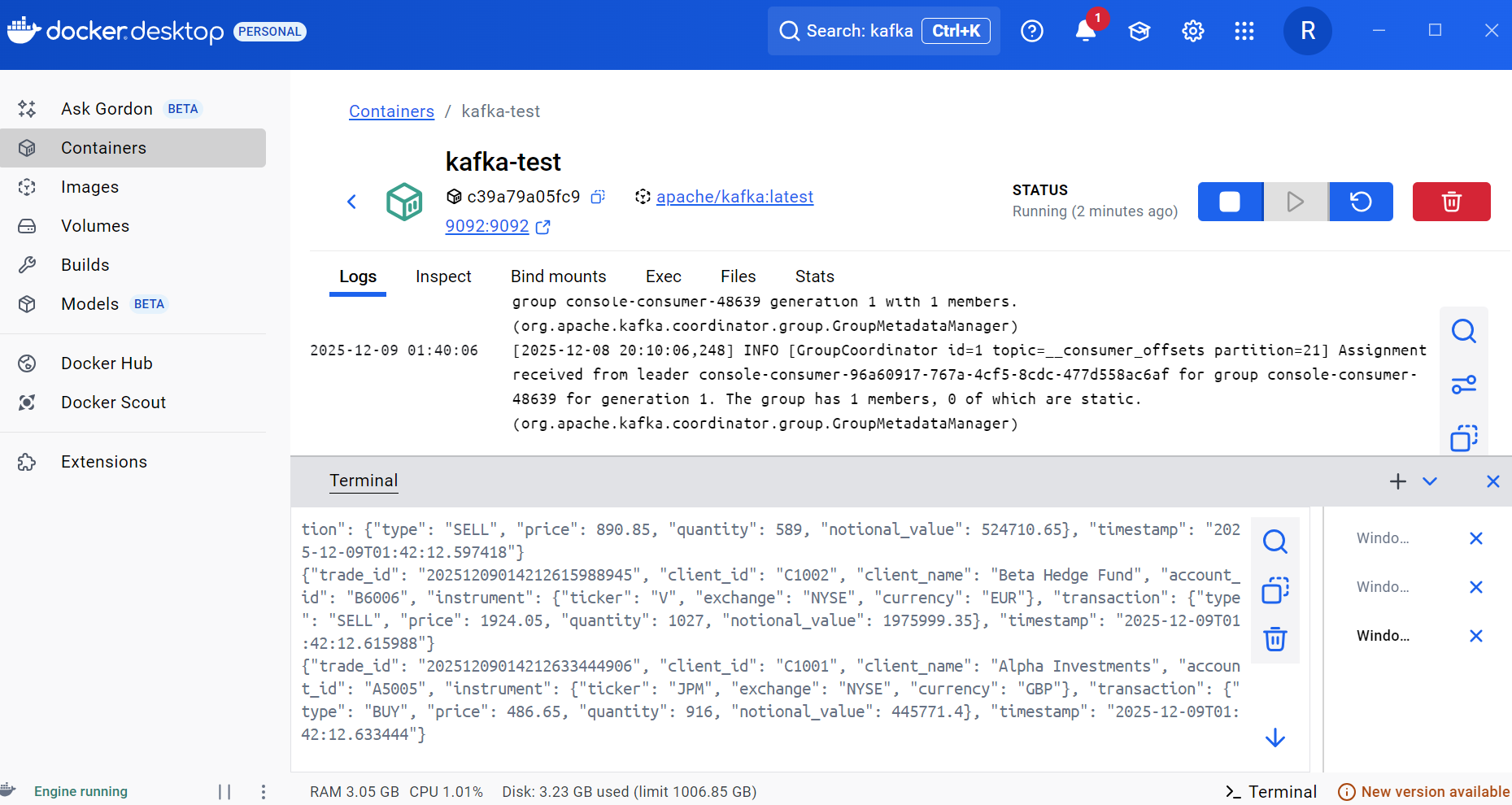
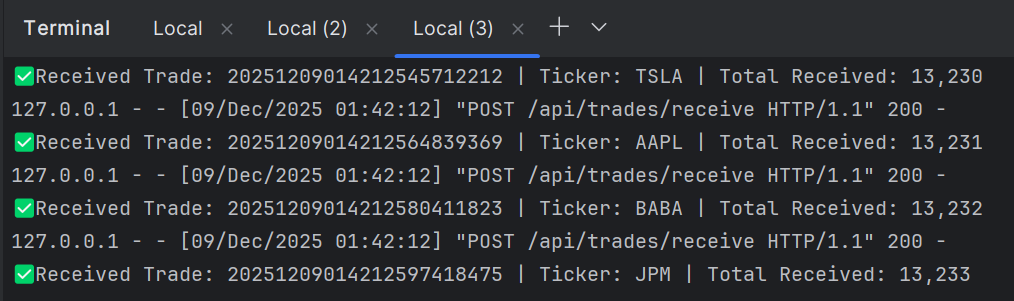
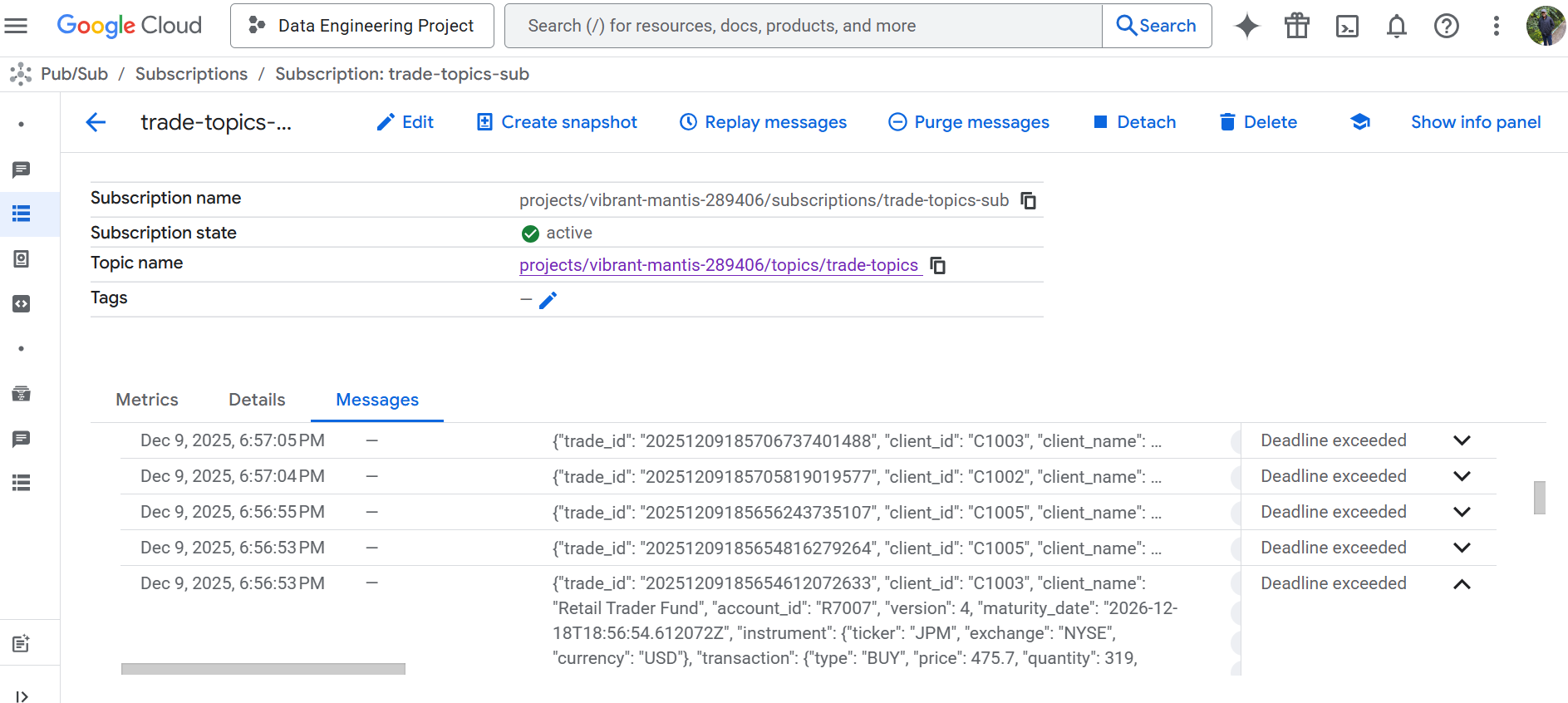
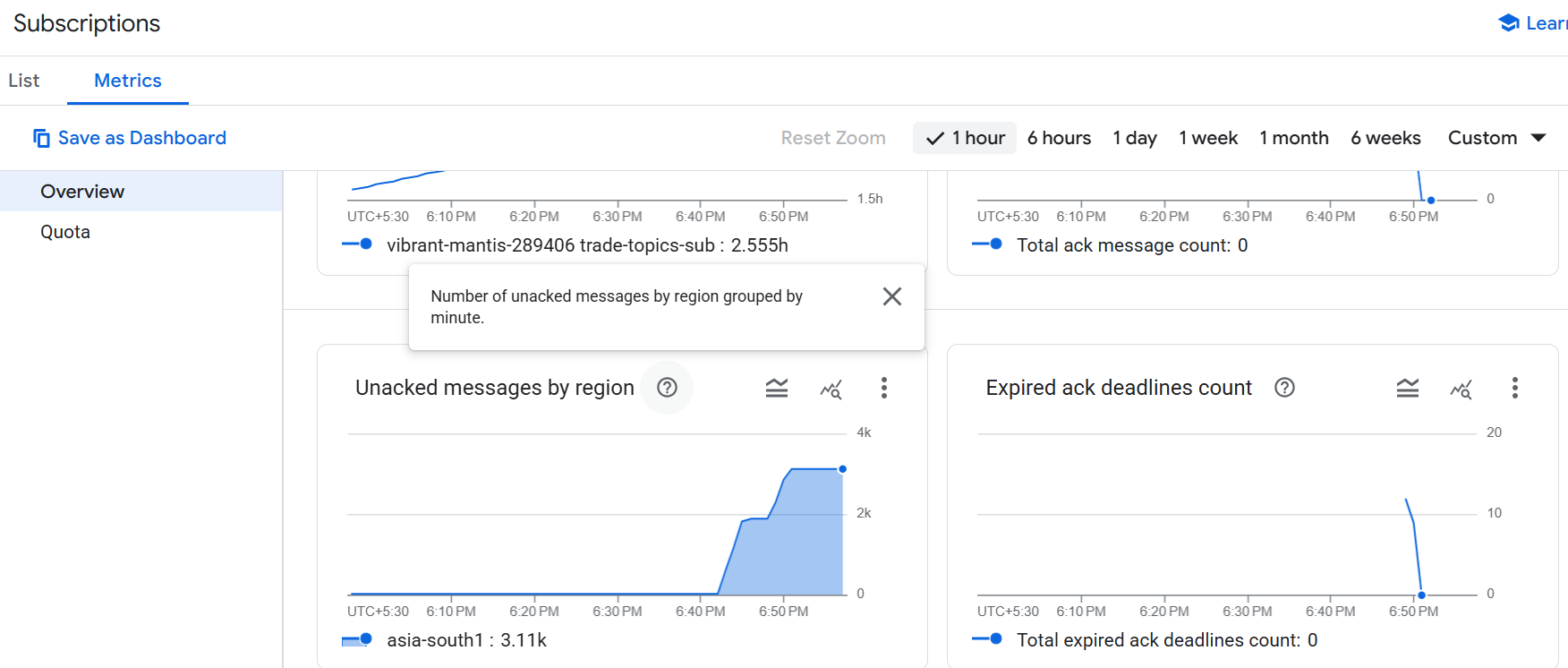
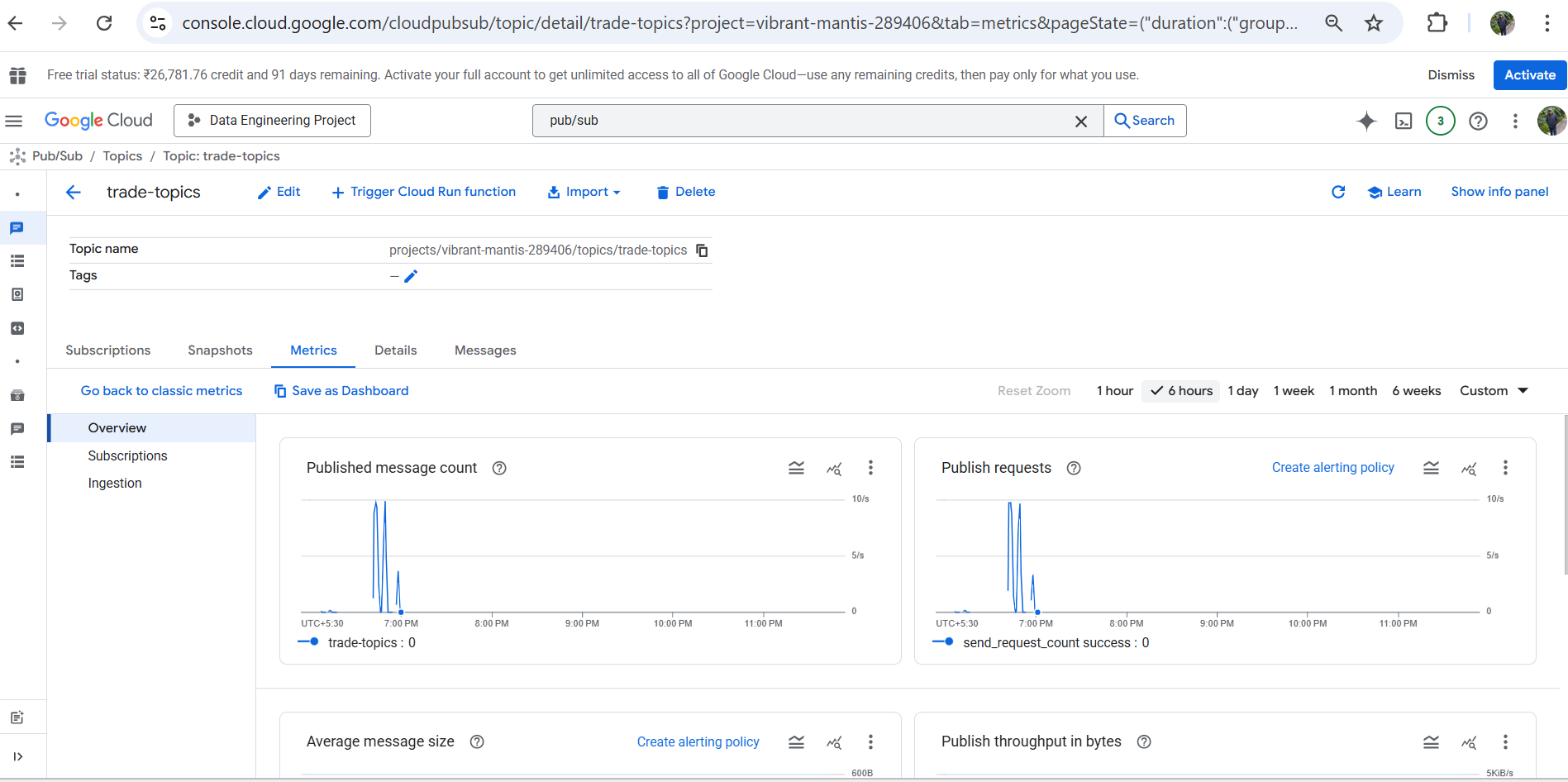
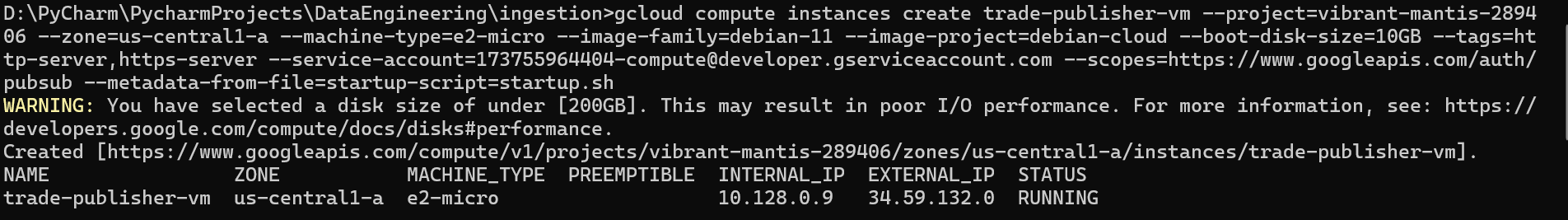
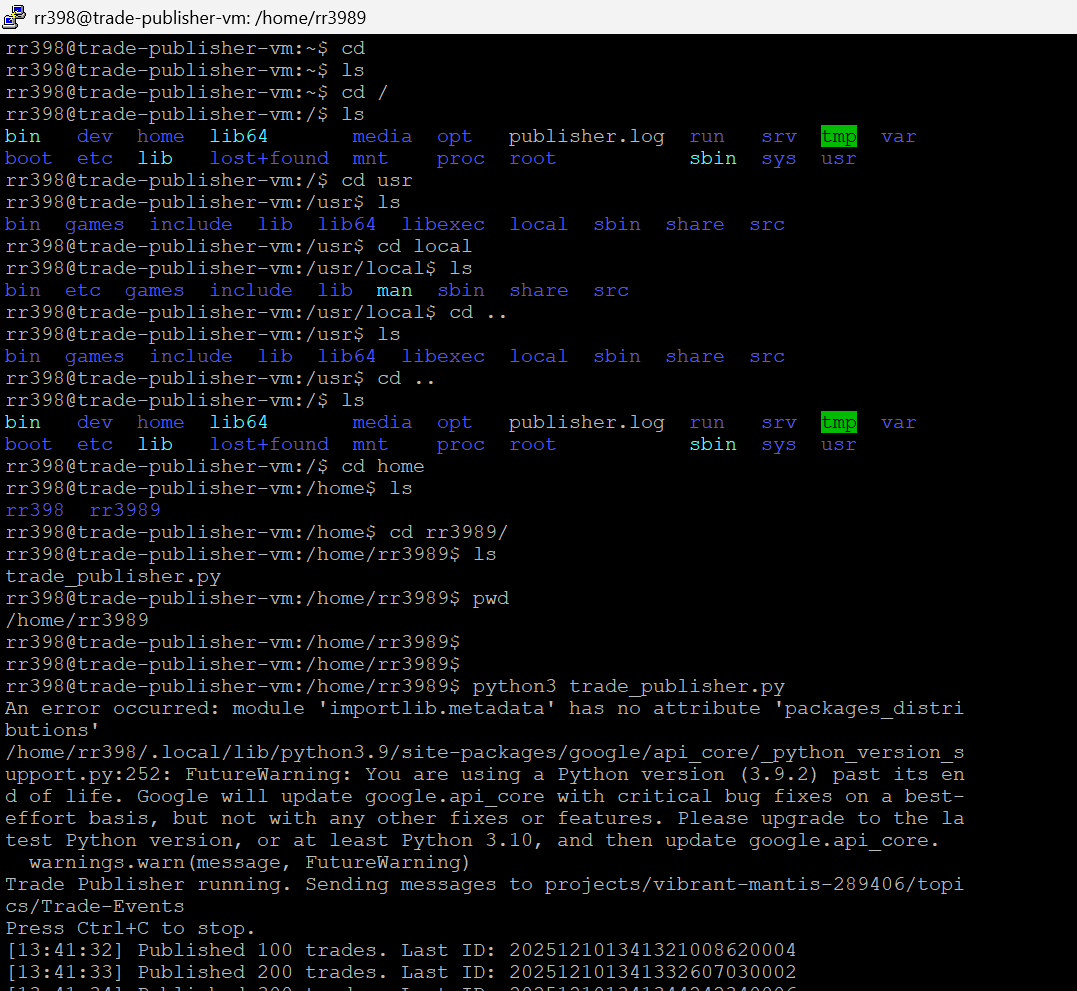
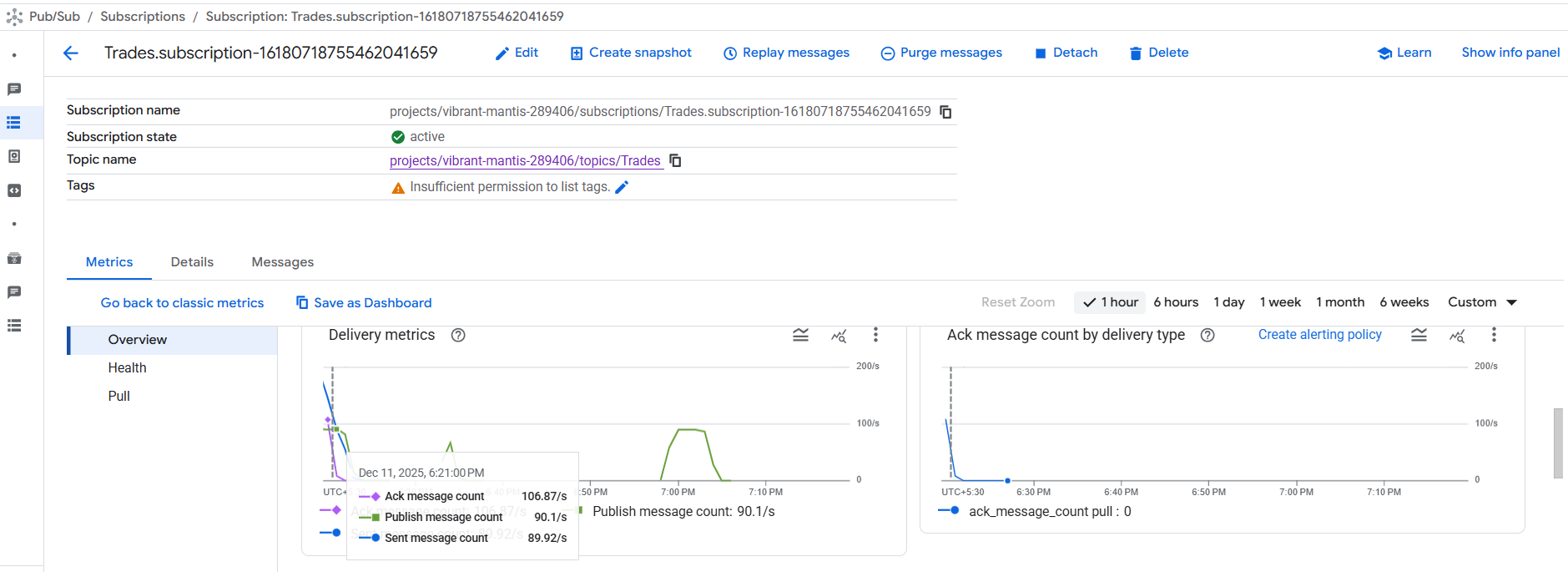
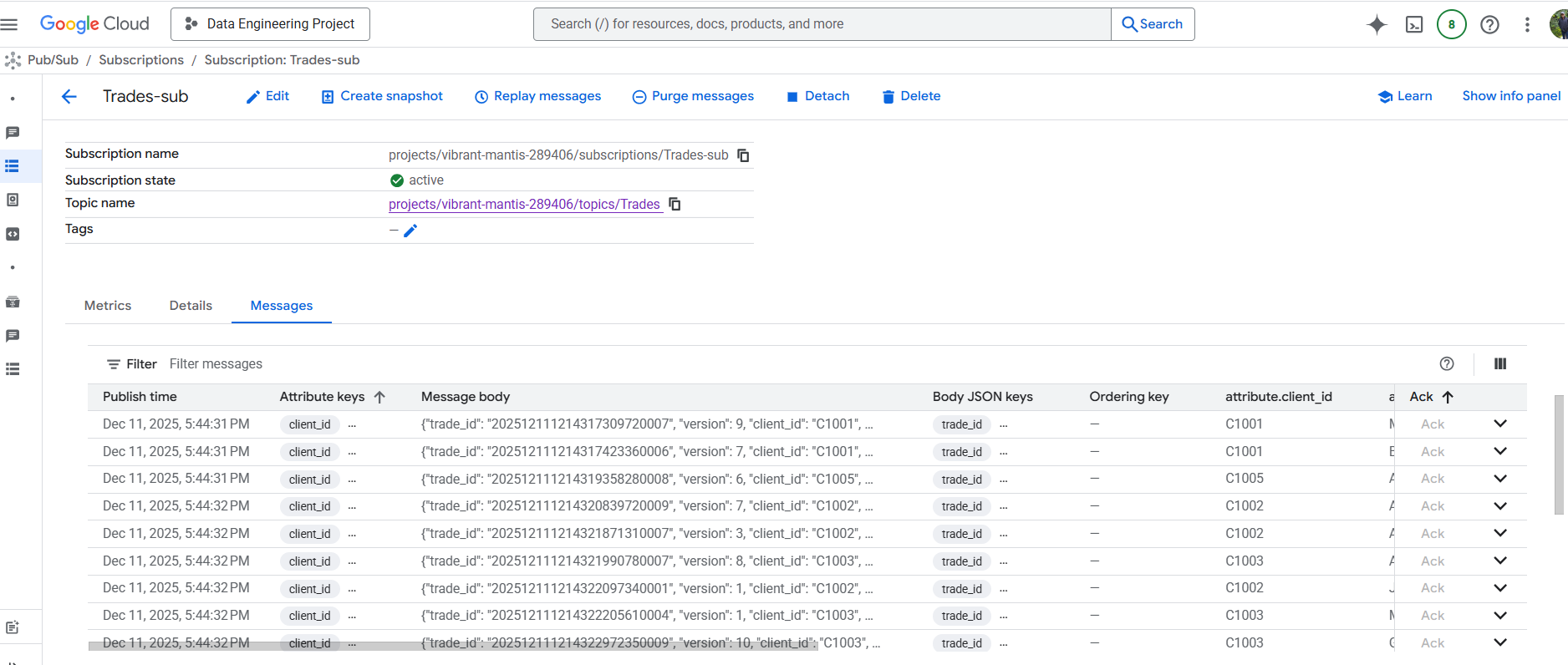
Data Engineering Case Study

# Background

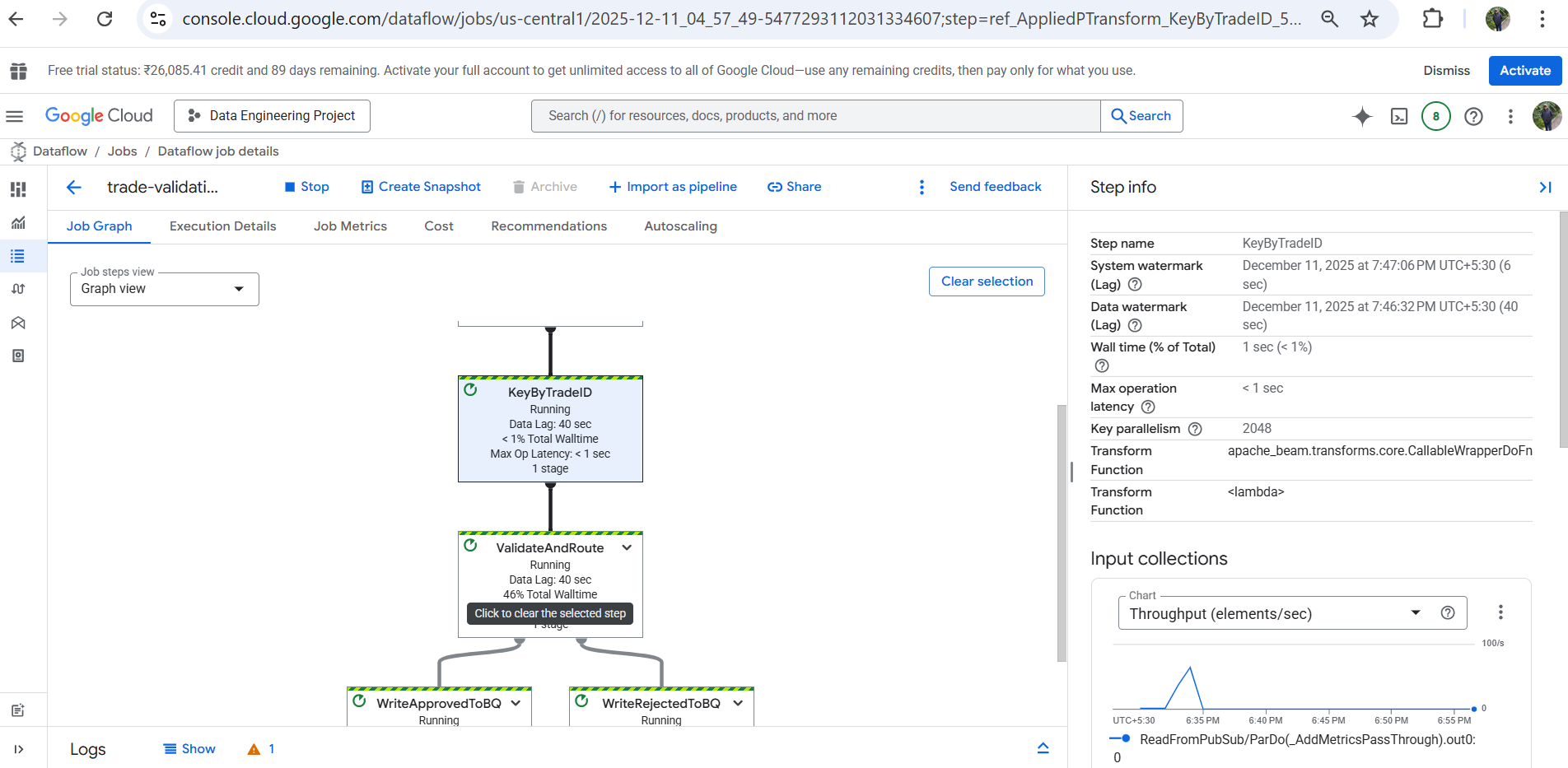
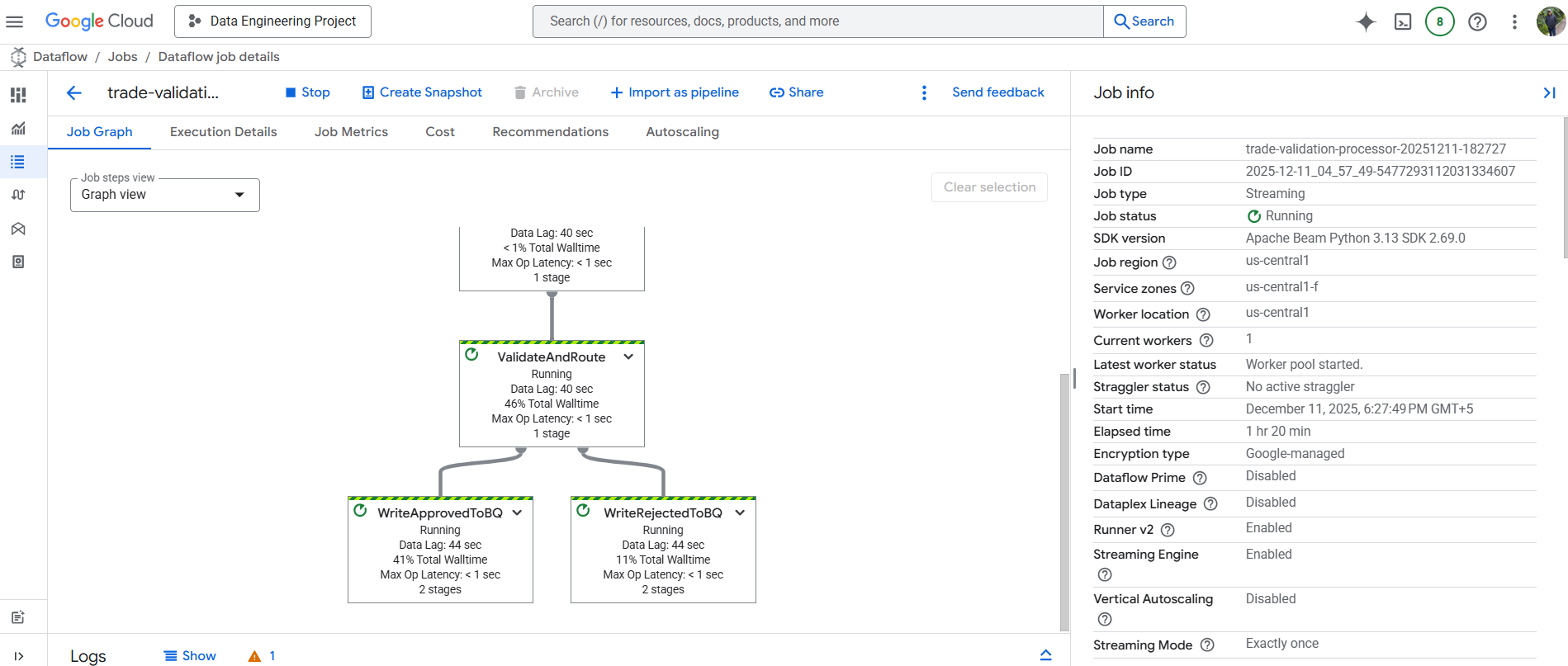
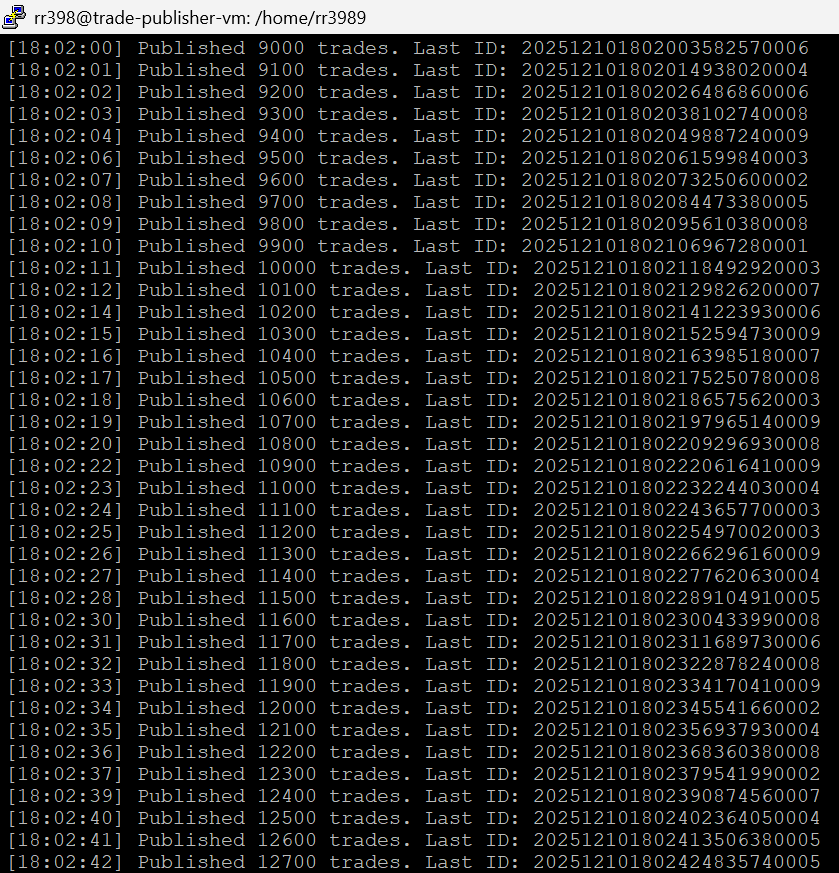
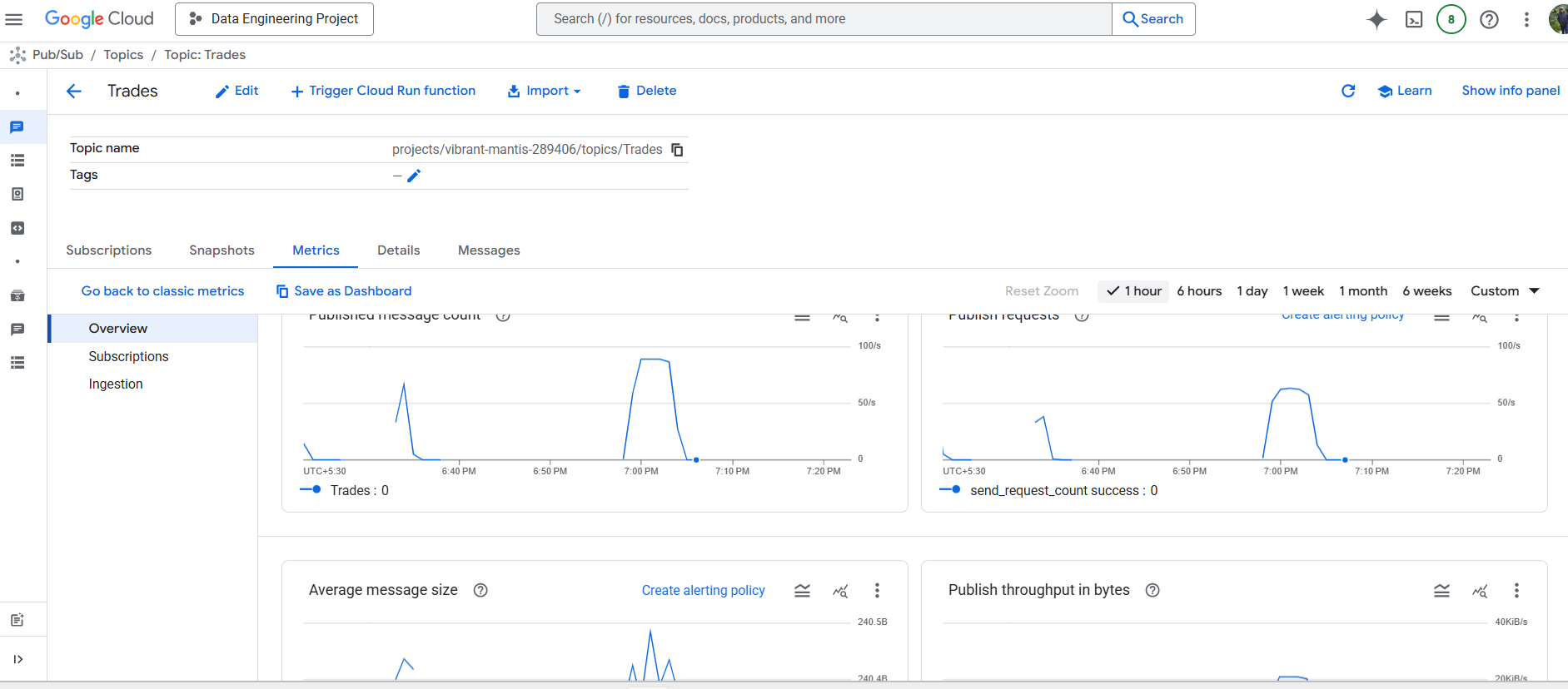
Thousands of trades are generated and transmitted to a central store daily. The organization wants to modernize its data infrastructure to support real-time analytics, compliance, and reporting. You are tasked with designing and implementing a robust, scalable ETL pipeline that ingests, processes, validates, and stores trade data using cloud-native tools.

# Requirements

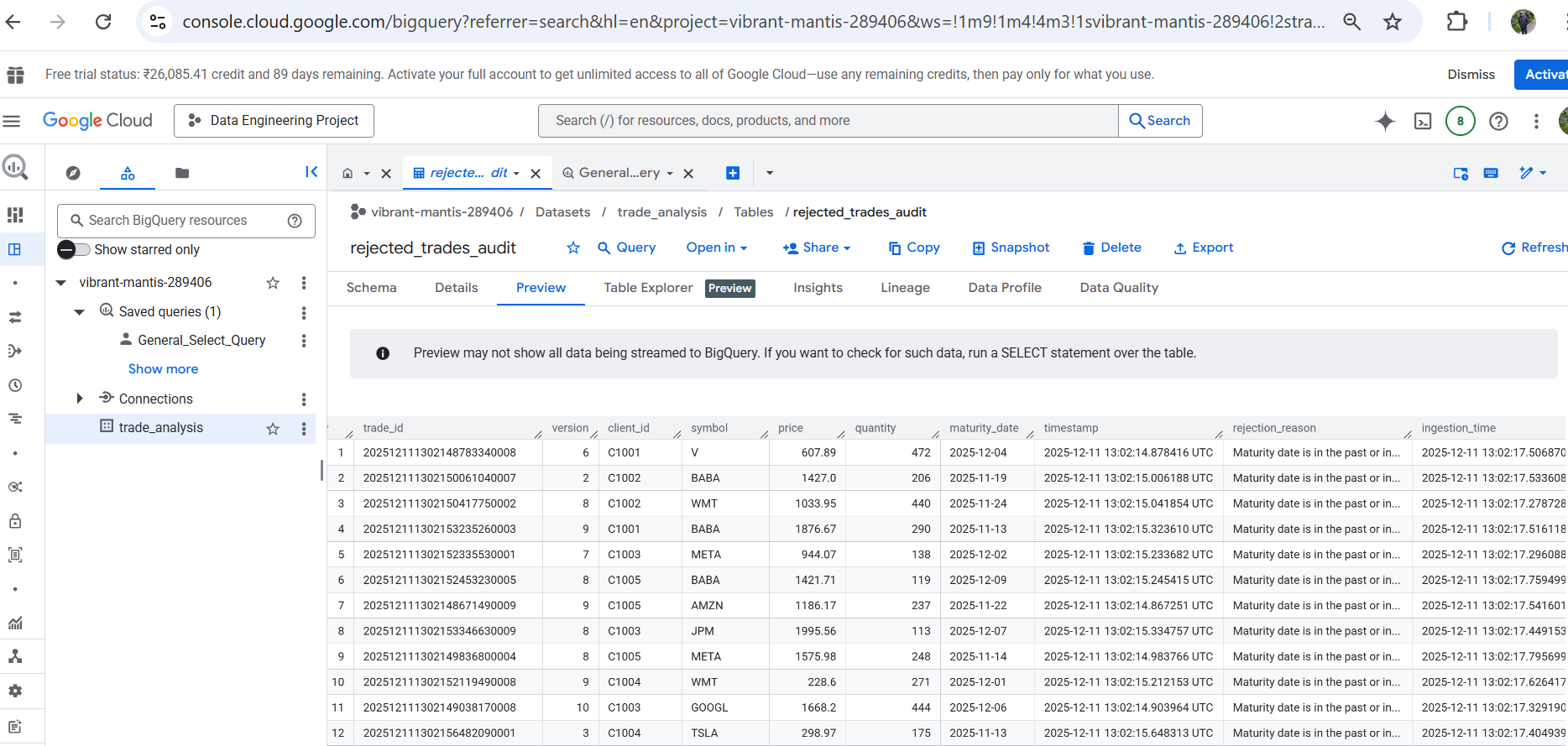
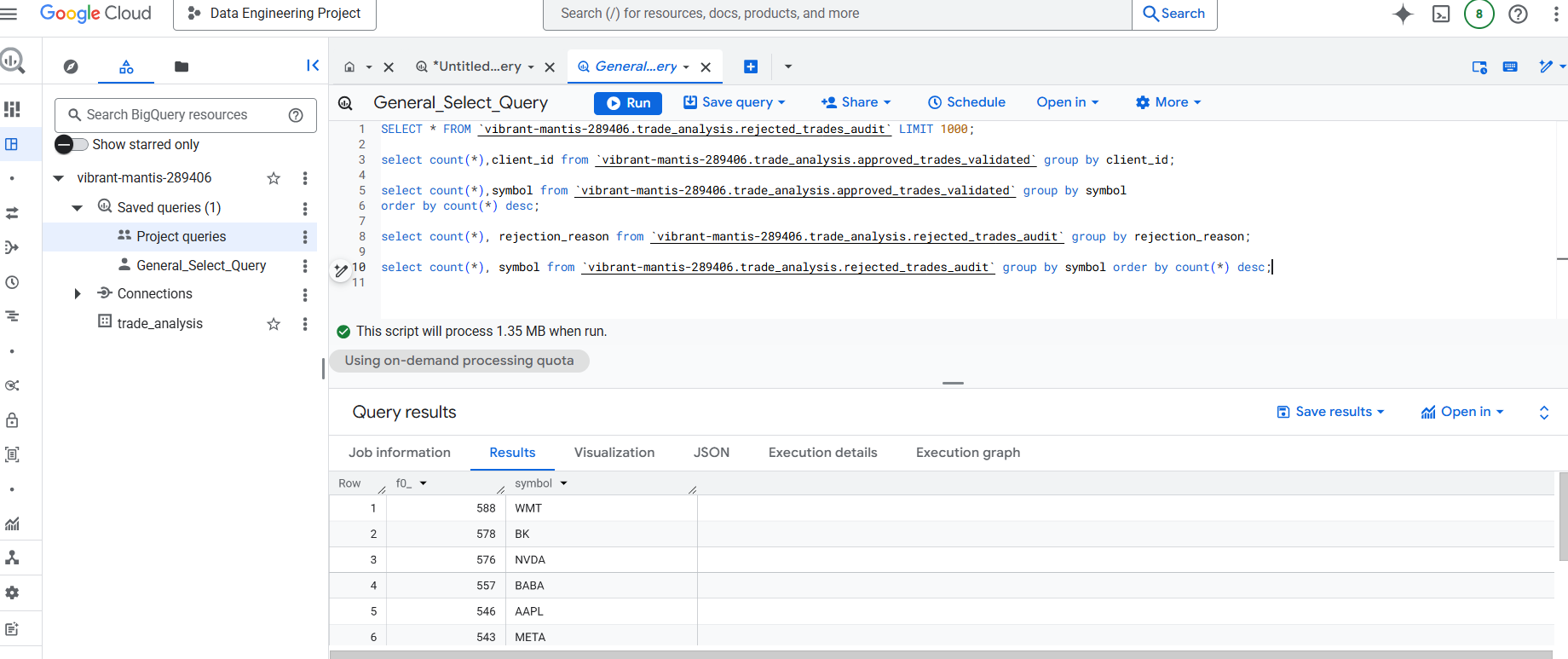
## Data Ingestion – Step 1

* **Simulate trade data generation (using a script or a mock service).**
* Developed mock trade simulation script in Python/Flask using Pycharm
* 
* The Trade structure/payload has client (id and name) and trade detail information
* TICKERS = ['AAPL', 'GOOGL', 'MSFT', 'META', 'BK', 'AMZN', 'TSLA', 'NVDA', 'JPM', 'V', 'BABA', 'WMT']
* TRADE\_TYPES = ['BUY', 'SELL']
* CURRENCIES = ['USD', 'EUR', 'GBP']
* EXCHANGES = ['NASDAQ', 'NYSE', 'LSE']
* MATURITY\_DATE
* VERSION
* MOCK\_CLIENTS = [
* {'client\_id': 'C1001', 'account\_id': 'A5005'},
* {'client\_id': 'C1002', 'account\_id': 'B6006'},
* Script can write to Kafka queue and API endpoint
* Receiver is listening at /api/trades/receive
* **Publish trade events to a messaging system (e.g., Google Pub/Sub or Kafka).**
* Configured kafka queue on docker image and sent trades to queue
* Trades sent to API and Kafka Queues
* 
* Created a Docker Kafka queue to receive Trades – Step 2
* Trades received and tested through local API receiver – Step 3
* As part of next iteration – Loaded Trades into the GCP Pub/Sub Queue – Step 4 – but had a lot of latency from local desktop to GCP Pub/Sub
* 
* 
* Moved the code to a GCP VM/container to be on the same region– Step 5
* 
* 
* Created new Topic – Trades and the latency reduced significantly as the messages originated from the same region. – Step 6
* 
* Pulled and displayed messages in the queue – successful transmission of trades
* 

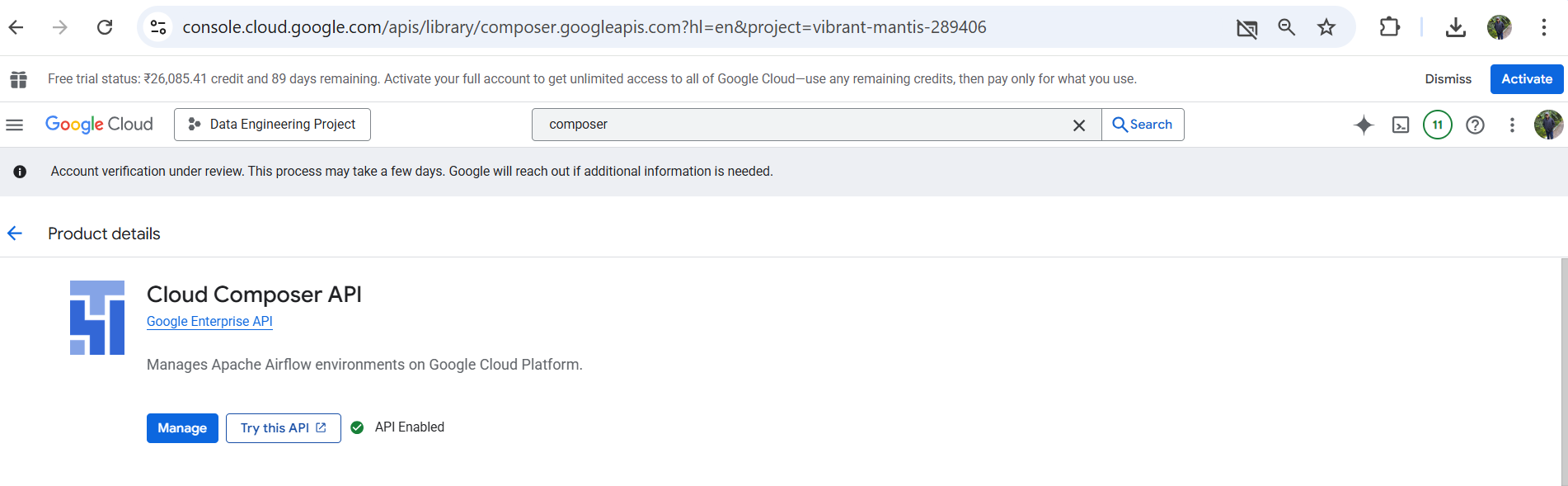
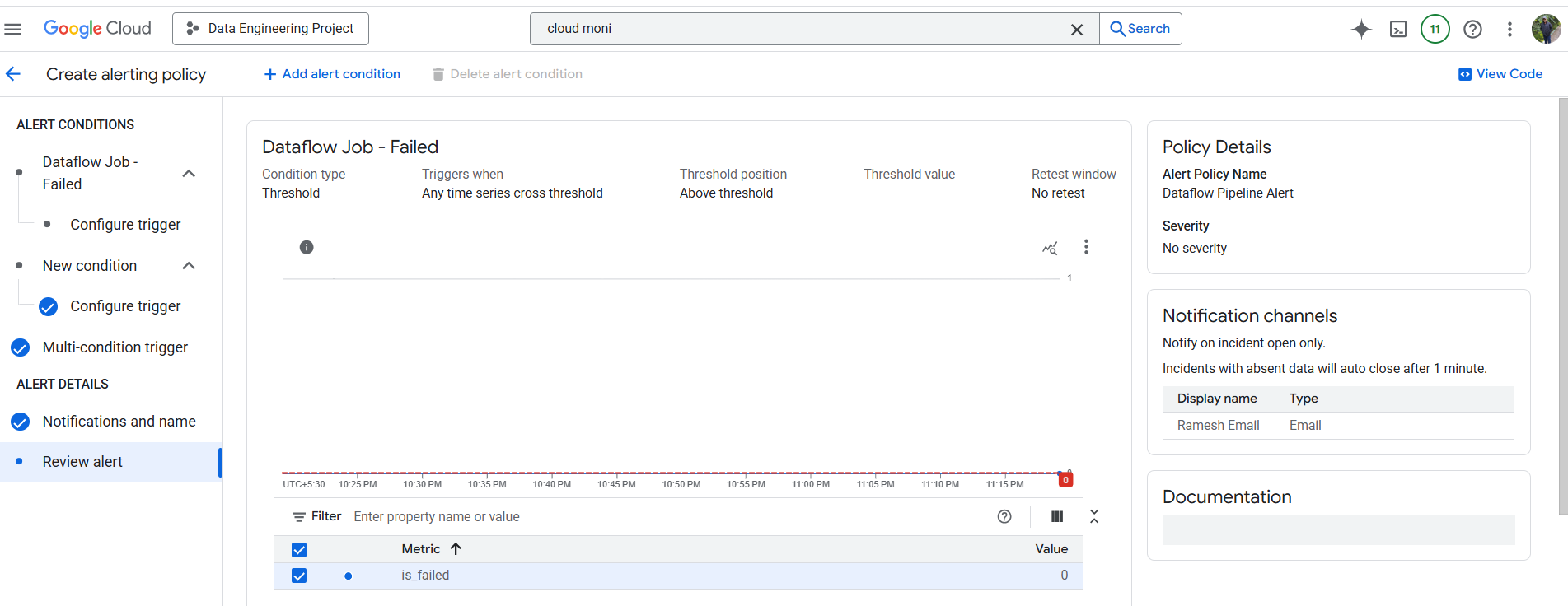
## Data Processing & Validation

* **Use a data processing engine (e.g., Google Dataflow,** Dataproc, or Data Fusion**) to:**
* **Consume trade events from Pub/Sub.**
* Google Dataflow job created and pipeline accepting trades – Step 7
* 
* **Apply business rules:**
* - Reject trades with a lower version than existing.
* - Replace trades with the same version.
* - Reject trades with a maturity date earlier than today.
* - Mark trades as expired if the maturity date has passed.
* Log rejected trades for audit.
* Applied business rules within the pipeline using custom logic for incoming versions and maturity dates – Step 8
* 
* Trades being published from the VM and sent to the Trades PUBSUB on GCP – Step 9
* 
* 

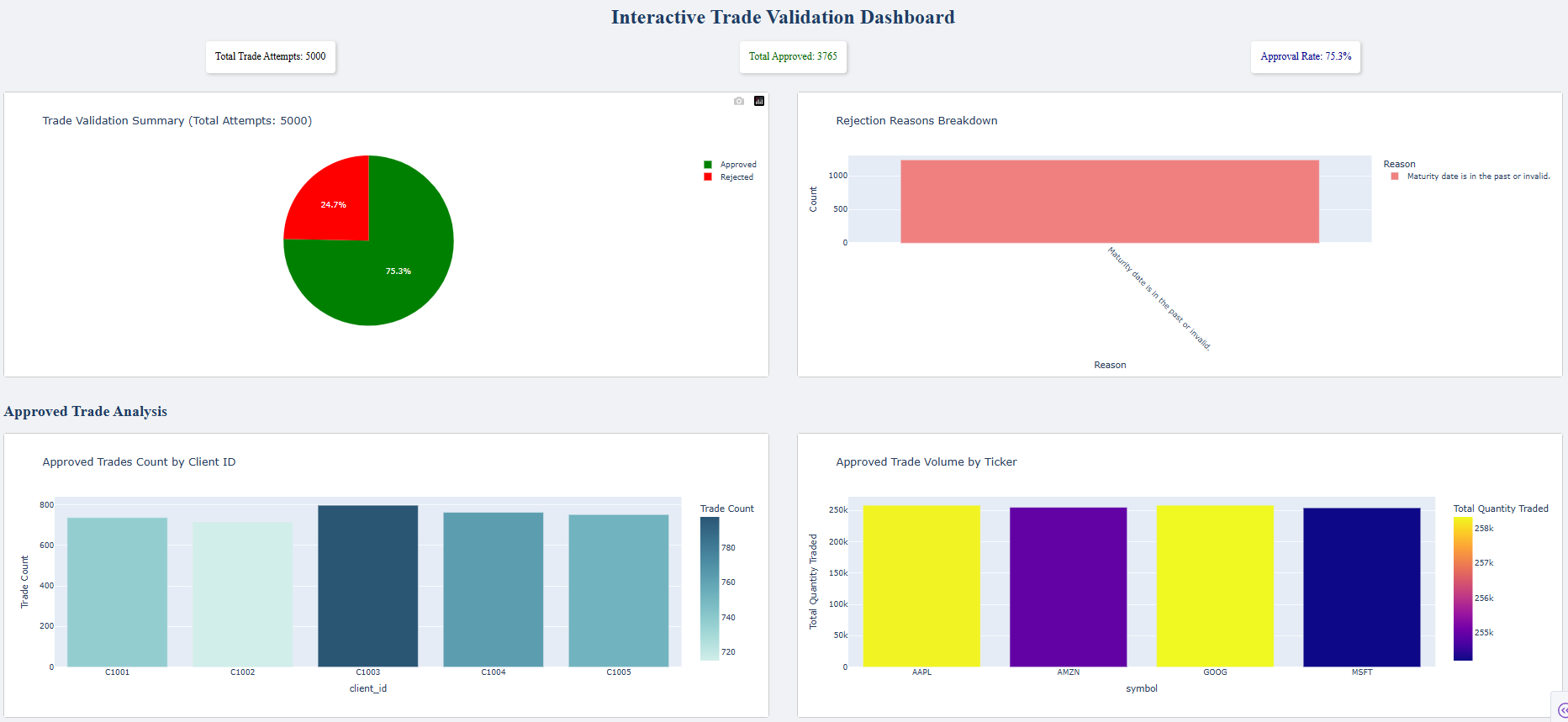
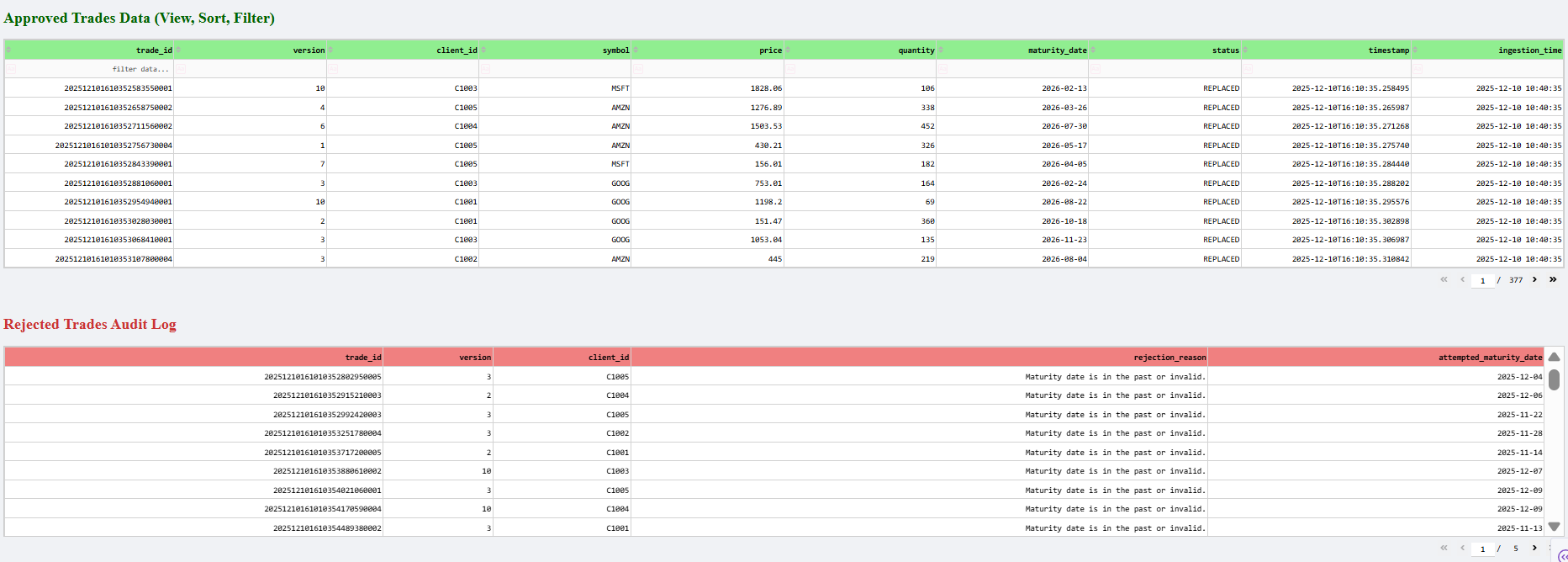
## Data Storage

* **Store valid trades in a cloud data warehouse (e.g.,** Snowflake**, BigQuery).**
* Created Dataset trade\_analysis in BigQuery – Step 10
* Schema for both tables was replicated from the trades JSON to the table structure
* Additional columns like rejection\_reason was added to the audit table
* Valid trades stored in BigQuery Table - approved\_trades\_validated
* 
* Store rejected trades in a separate table for compliance. – Step 11
* Stored rejected trades in BigQuery Table rejected\_trades\_audit
* 
* Created General Queries to check for counts and aggregates – Step 12
* 

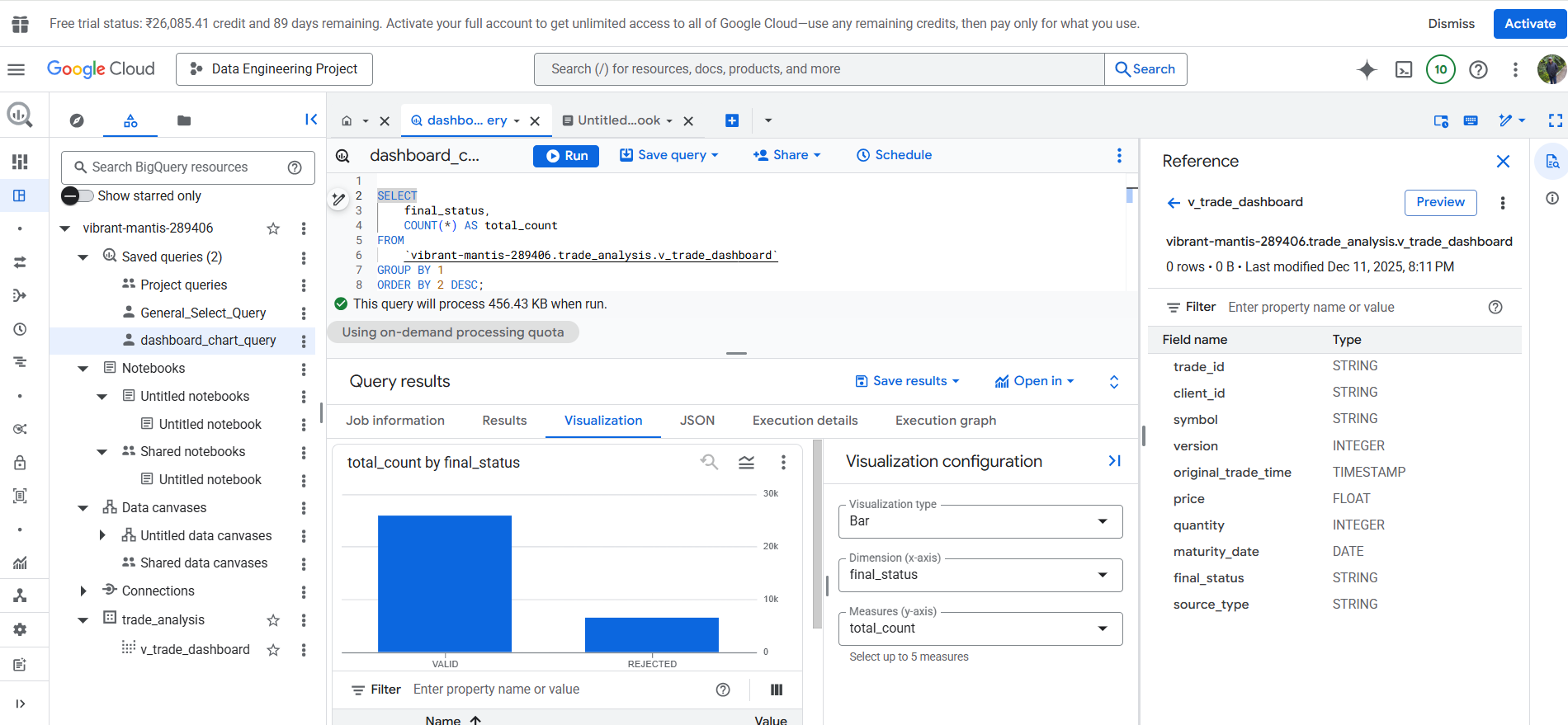
## Orchestration & Monitoring

* **Use Airflow/Composer to orchestrate the ETL workflow:**
* 
* - Schedule batch jobs (if needed).
* - Monitor pipeline health and failures.
* Implement basic alerting (e.g., email on pipeline failure). – Step 13
* Created a basic email alert notification mechanism based on pipeline failures.
* 

## Optional Enhancements

* Create a dashboard/report to visualize trade status (expired, valid, rejected). – Step 14
* Before even storing on GCP, I stored the data locally in sqllite as backup to start and used python (matplotlib) to visualize trades and generate charts 
* 

The below chart was visualized in GCP using charts to avoid looker studio and others.



* Use Data Fusion for low-code pipeline design (optional).
* Document your architecture with diagrams (PlantUML or similar).
* I used Miro for process flow and Architecture diagram

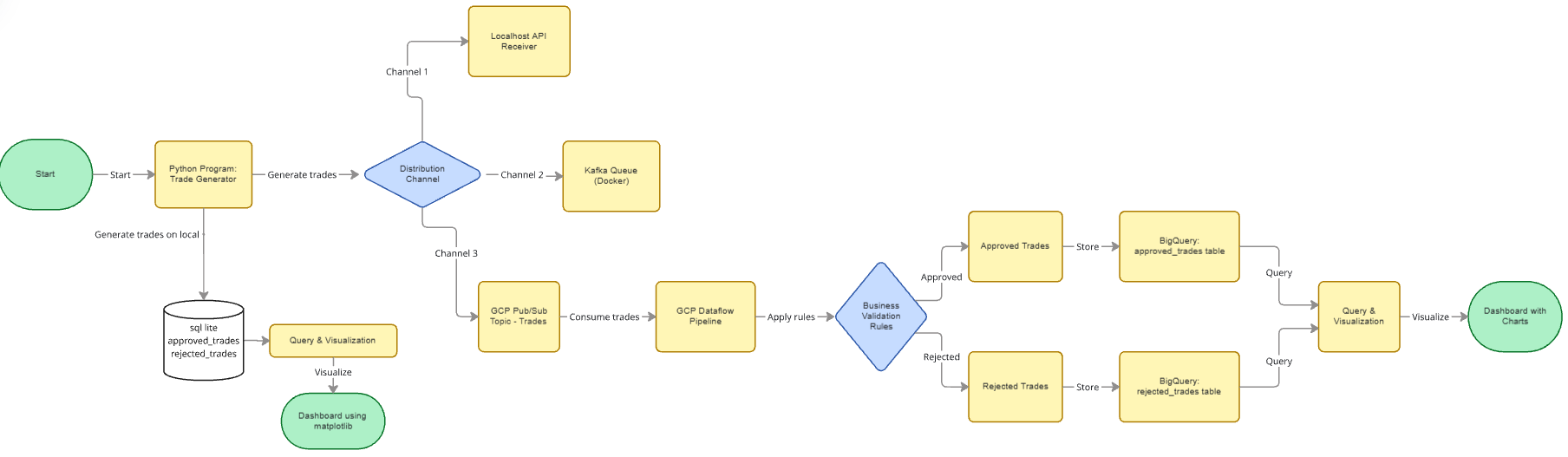
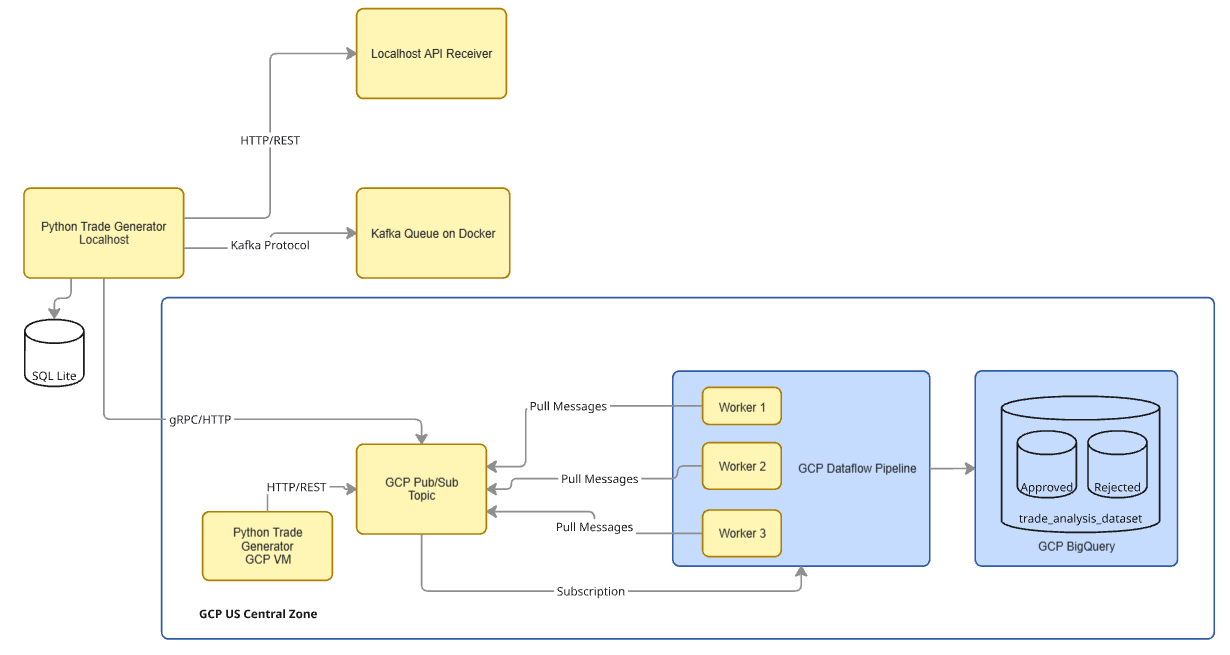
# Deliverables

* Source code/scripts for data generation, ETL pipeline, and orchestration.
* Infrastructure-as-Code (IaC) scripts (Terraform or Deployment Manager) for provisioning resources (optional).
* Documentation:
* - Architecture diagram.
* - Step-by-step setup and execution guide.
* - Description of validation logic and tech stack choices.
* Store all code and documentation in a public GitHub repository.

# Preferred Tech Stack

* Messaging: Google Pub/Sub (or Kafka)
* Processing: Dataflow (Apache Beam), Dataproc (Spark), or Data Fusion
* Orchestration: Airflow/Composer
* Storage: Snowflake (or BigQuery)
* Visualization: Looker Studio, Tableau, or similar (optional)
* IaC: Terraform (optional)

# Evaluation Criteria

* Correctness and completeness of ETL logic.
* Use of appropriate cloud-native tools.
* Code quality, modularity, and documentation.
* Monitoring, error handling, and alerting.
* Clarity of architecture and design decisions.
* Data and Process Flow
* 
* Architecture Diagram
* 

Business Validation

* In trading, a trade (bond or contract) can be changed multiple times over its life. Version validation prevents an older, outdated instruction from accidentally overwriting the current state of the trade, which could lead to incorrect risk exposure calculations, wrong settlements, and compliance failures.
* The Maturity Date is the date on which the financial contract expires and the final exchange or settlement is due. This date determines the term of the contract and is important for risk management
* Version Check: It uses Beam State (a persistent storage mechanism) to remember the existing\_version for a given trade\_id and rejects any trade whose new\_version is not strictly greater than the existing one.
* Maturity Check: It compares the incoming maturity\_date string against the current date to ensure the trade is not expired.

Tech Stack Justification

* The decision to build this trade validation case study on Google Cloud Platform using services like BigQuery and Dataflow, instead of Snowflake and Kafka, was based on the advantages of using a tightly integrated, native cloud stack for streaming data.
* Dataflow is the industry best for stateful stream processing. By using Dataflow directly and tightly with Pub/Sub and BigQuery as the sink, we were able to utilize the end to end tool stack from one single platform and reduced latency and had better integration
* Using snowpipe and kafka will add an additional hop to the data flow
* GCP Dataflow gave features for beam state managed storage to make decisions on version and maturity
* No low level plumbing required for Pub/Sub and no setting up infrastructure like zoo keeper and Kafka queues
* Most of the GCP stack was serverless and managed on cloud to transmit trades from one system to another