Overview

This report outlines the design and implementation of a real-time log processing pipeline aimed at ingesting JSON log data, processing it, and storing results in BigQuery. The pipeline utilizes Google Cloud Pub/Sub for log capture, a Cloud Function for log processing, and BigQuery for data storage. Additionally, FastAPI is used to create a REST API endpoint for log ingestion, which allows logs to be sent to the pipeline in real-time. Automation and resource provisioning are handled using Terraform, with CI/CD integration via GitLab. The final project is hosted on GitHub, and the GitLab repository can be accessed here.

Implementation Details

Log Ingestion

Logs are ingested through a **FastAPI** endpoint that receives JSON data and publishes it to Google Cloud Pub/Sub for processing. Since the specific structure of the JSON was not specified, I defined the log data to include three keys: **timestamp, log_level,** and **message**.

- Example JSON structure

```
{
  "timestamp": "2025-03-11T12:00:00Z",
  "log_level": "INFO",
  "message": "User logged in successfully"
}
```

Processing

Logs are processed by a Google Cloud Function written in Python. The function is triggered by messages from Google Cloud Pub/Sub. Each time a log is published to the Pub/Sub topic, the Cloud Function is invoked to process the log data. It parses the log and then writes the processed data to BigQuery for storage and querying.

BigQuery Integration

Processed logs are stored in a BigQuery table, with a schema optimized for efficient querying. To enhance and optimize performance and manage large volumes of data, time partitioning and clustering are used.

- Time Partitioning: The table is partitioned by day based on the timestamp field. This ensures that queries targeting recent logs are faster and more efficient by limiting the amount of data scanned
- Clustering: The table is also clustered by the log_level field, which groups logs by category (e.g. INFO, WARN, ERROR). This optimizes query performance when filtering logs based on their category.

Terraform

Google Cloud infrastructure is provisioned using Terraform.

CI/CD Pipeline

The GitLab CI/CD pipeline automates the build and deployment of the pipeline components using Terraform and Python scripts. Since no tests were written for this project, the CI/CD pipeline focuses on automating the deployment and ensuring the infrastructure is correctly provisioned.

Outcome Metrics

Since the pipeline was built within a constrained 8-hour timeframe, no production metrics have been collected yet. However, the design incorporates mechanisms to measure and optimize performance. For example, using time_partitioning and clustering in the BigQuery table can improve performance and efficiency.

The Cloud Function responsible for processing log data and inserting it into BigQuery demonstrated an execution time of **157 ms**. This metric was obtained from Cloud Function logs, which track execution start time and completion time.

AI Tool Impact

ChatGPT was used during the development and testing stages to optimize code, enhance solution design, and troubleshoot issues. Specifically, it helped with:

- Code Optimization: ChatGPT provided suggestions for improving the efficiency and robustness of the Cloud Function. This allowed for better error handling and reduced execution time.
- Terraform Assistance: Rather than spending extra time going through Terraform docs, ChatGPT helped generate code, such as the BigQuery schema, quickly. This saved me a lot of time when provisioning cloud infrastructure.
- Learning: Since I had never worked with Google Cloud before, ChatGPT helped me understand its services, best practices, and how to integrate components like Pub/Sub, Cloud Functions, and BigQuery effectively. This accelerated my onboarding and enabled me to build the pipeline more efficiently

Challenges & Resolutions

One of the main challenges encountered was that GitLab's free tier does not support repository mirroring. Therefore, I couldn't automatically sync my GitHub repository with GitLab. This made testing the CI/CD pipeline more difficult, as I had to manually push code to GitLab instead of relying on automatic updates from GitHub.

To work around this, I ensured that every change made in the GitHub repository was also pushed to GitLab manually. While this added extra steps to the workflow, it allowed me to fully test and validate the CI/CD pipeline without upgrading to a paid GitLab tier.