OULUN AMMATTIKORKEAKOULU

Tan Bui

REAL-TIME DATA ANALYTIC ON GOOGLE CLOUD

A Complete ELT Pipeline from Self-Host Databases to GCP Services

REAL-TIME DATA ANALYTIC ON GOOGLE CLOUD

A Complete ELT Pipeline from Self-Host Databases to GCP Services

Tan Bui
Master Thesis
Autumn 2024
The Degree Programme in Data Analytics and Project Management
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
The Degree Programme in Data Analytics and Project Management

Author: Tan Bui
Title of the thesis: Real-time data analytic on google cloud: A complete ELT pipeline from self-host databases to GCP services
Thesis examiner: Ilpo Virtanen
Term and year of thesis completion: Spring 2025 Pages: 54 + 4 appendices

Data analysis plays an important role in business operations these days. There is a growing demand for integrating a data analytic pipeline into an existing software system. Although cloud services such as AWS and Google Cloud provide many on-the-cloud solutions, many businesses may have self-host infrastructures that are independent from those cloud services, making it challenging to design an architecture for a data analysis system.

Especially in real-time processing, data changes must be streamed continuously from the existing software system to the data analysis pipeline. However, many businesses are still using traditional data storage systems that do not support event streaming.

The thesis introduces a system architecture of a complete real-time ELT pipeline for data analysis. In this pipeline, data changes from multiple sources, such as MySQL and MongoDB databases in an existing infrastructure, are extracted and loaded to Google Cloud services in real time. The data is then transformed to become available for visualization or other services.

To verify the architecture, an implementation of the whole system was conducted in the thesis. The existing software system was simulated with a data generator that populates thousands of sample data records per second to the databases. There are new components to observe and convert the changes to events streamed to Google Cloud services.

The results showed that the architecture can be applied to different data sources, and millions of events can be processed every second, depending on network bandwidth and cloud services quotas.

Keywords: Real-time, ELT, Data Analysis, Pipeline, Big Data, Stream Processing, Scalability, Google Cloud, System Architecture, High Availability

CONTENTS

1 INTRODUCTION

In today's software systems, the ability to provide data insights has become a critical competitive advantage. While batch-oriented data integration architectures are effective for historical analysis where data is accumulated and processed as bunches, stream-oriented architectures are widely adopted in modern businesses that require immediate access to information. In a stream-oriented system, data is captured, processed, and delivered in real time as it is generated. This enables organizations to make decisions swiftly, optimize operations, and identify emerging trends proactively (Mange Ram Tyagi 2024).

Implementing real-time data analytics presents several significant challenges. The immense volume and velocity of data generated in today's digital world demand robust infrastructures and efficient processing capabilities. Data may come from various sources and different formats. Maintaining low latency and high availability while processing and analyzing large datasets is another critical challenge. Although services such as AWS or GCP provide many cloud-based solutions, developing advanced analytics models capable of extracting meaningful insights from real-time data requires specialized skills and expertise.

The research objective is to propose an effective system architecture for real-time data analytic that can be integrated to an existing software system. Performance, scalability and cost estimation of the architecture are also addressed.

The thesis aims to answer these questions:

- How can data be extracted automatically and in real-time for different data sources?
- How to load the extracted data to Google Cloud services?
- How to transform data in Google Cloud services to create reports and visualize the data?
- What is the performance, scalability, and infrastructure cost of the proposed architecture?

To address these questions, the thesis implements the proposed architecture as a complete pipeline from data sources to a visualized report. Because MySQL and MongoDB database engines are commonly used in many software systems (MongoDB, Inc. 2015), this research focuses on extracting data from these sources instead of CSV or unstructured data sources.

2 BACKGROUND

2.1 Real-time data analytic

Real-time data pipelines are instrumental in driving business growth and efficiency. By rapidly processing and analyzing the stream data as it is generated, organizations can make informed decisions, optimize operations, and identify new opportunities. Example use cases that utilize stream processing:

- Real-time analytics and insights
- Fraud detection
- IoT data processing
- Stock trading
- Clickstream analysis

For example, Uber's real-time data system supports demand/supply forecasting, surge pricing, ETA calculation that improve user experiences (Uber Technologies Inc 2021). In Netflix, real-time processing is one of the key factors of its leading position in entertainment industry (Netflix Technology Blog 2022). The real-time data is processed to generate personalized recommendations, optimize content placements, and inform content creation decisions. By understanding viewer preferences in real-time, Netflix can enhance user engagement and retention.

Real-time systems are classified as hard, soft, and near, as in Table 1 (Psaltis 2017, 5). For example, embedded systems are usually *hard real-time*, with strict time requirements that, if missed, may result in total system failure.

TABLE 1. Hard, soft and near real-time system comparison

Type	Examples	Latency measured in	Tolerance for delay
Hard	Pacemaker, anti-lock brakes	Microseconds - milliseconds	None - total system failure, potential loss of life
Soft	Airline reservation system, online stock quotes, VoIP (Skype)	Milliseconds - seconds	Low - no system failure, no life at risk
Near	Skype video, home automation	Seconds - minutes	High - no system failure, no life at risk

Another paradigm of data analytic pipelines is batch processing. Despites of its weaknesses such as high latency, limited real-time insights and potential data staleness, batch-based data pipeline offers cost-effective resources and simpler implementation. For use cases which are not time-critical, batch processing is more suitable. Especially, when the existing system already adopted batch processing paradigm, it is recommended to not stream everything just because stream processing technology is popular (Bryant 2018). Common use cases of batch processing include:

- Daily or monthly financial reporting
- Machine learning model training
- Customer segmentation

It is important to understand that a *real-time* system does not always have to be a *streaming* system. (Psaltis 2017, 8) defines a real-time system makes its data available at the moment a client application needs it. For example, a reporting system needs to show charts to clients with fresh data every hour, even when it uses batch process approach with interval time is one hour, this is considered a real-time system to the clients.

When designing a data analytic pipeline, ETL (Extract – Transform – Load) and ELT (Extract – Load – Transform) are two popular approaches. In an ETL pipeline, the data is extracted from source systems and transformed before being loaded into a target data warehouse. This paradigm is inherently batch-oriented, encountering a delay between every batch of data. This approach is suitable in OLAP (Online analytical processing) systems, as data is pre-processed before loaded into a data warehouse, ensuring consistency and integrity.

In contrast, ELT offers a more flexible and scalable solution. As soon as the data is extracted, it is loaded into the target system and then transformed within the target environment. This approach prioritizes data ingestion, allowing the target environment to leverage cloud-based capabilities such as storage or processing units to handle large volumes of data with minimal latency. ELT is commonly used in OLTP (Online transactional processing) systems where raw data is loaded quickly while transformations are done on demand.

2.2 Common data pipeline architectures

A good data architecture serves business requirements with a standard, widely reusable set of building blocks while maintaining flexibility and making appropriate trade-offs (Reis and Housley 2022, 118). There are four common data pipeline architectures.

2.2.1 Basic data pipeline

In a data analytic system, data is transferred through different components. The chain of components that ingests the data and produces outputs is a data pipeline. Common stages in a basic data pipeline include:

- **Data ingestion**: the data is collected and ingested from source systems. Depending on the characteristics and technologies of the data sources, the incoming data could be batched or streamed. Batched data is ingested each time interval or when data reaches a specific size, while streamed data is processed immediately when it is generated.
- **Data transformation**: the data in a pipeline can be transformed multiple times for different purposes, such as cleaning, storing, or becoming input for other systems. Transformations in large systems might be complex and require an orchestration.
- **Data storage**: the data can be stored across the pipeline. It is fundamental to understand performance requirements, costs and storage technologies before selecting a storage system. Security is especially critical at this stage.
- **Serving data**: as data becomes available in the system as a useful structure, the final stage is to get the most values from it, such as reporting, visualizing, machine learning and decision making.

Figure 1 represents a basic ELT data pipeline, which data is extracted from multiple sources, transformed and loaded into a data warehouse.

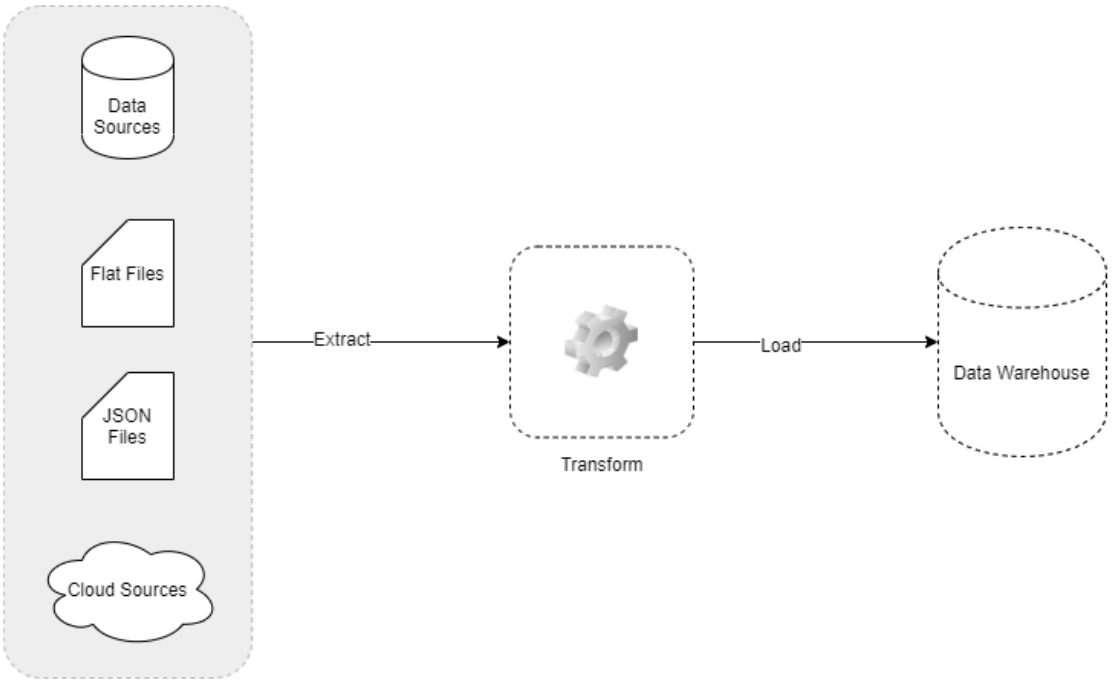

FIGURE 1. Basic ETL data pipeline

2.2.2 Data streaming pipeline

In this pipeline, data is propagated through the system as a continuous stream, although the pipeline stages are similar to the basic data pipeline. For systems that can tolerate some delay but also require relatively fresh data, micro-batching is an alternative with lower complexity. Data streaming is preferred in the following situations:

- Time-sensitive applications: when the system demands immediate action or insights based on incoming data such as fraud detection or infrastructure monitoring.
- Continuous data source streams: the data from sources is generated as a continuous flow of messages.
- High frequencies data: data arrives at high velocity and requires processing on the fly.
- Event-driven architecture: when the system needs to react to specific events or triggers in real-time

Figure 2 illustrates a streaming ETL pipeline:

- Continuously extracts incoming data from sources like application logs or sensors.
- Immediately processes each data record through the transform stage as it arrives.
- Directly loads the processed data into the target database without batching.

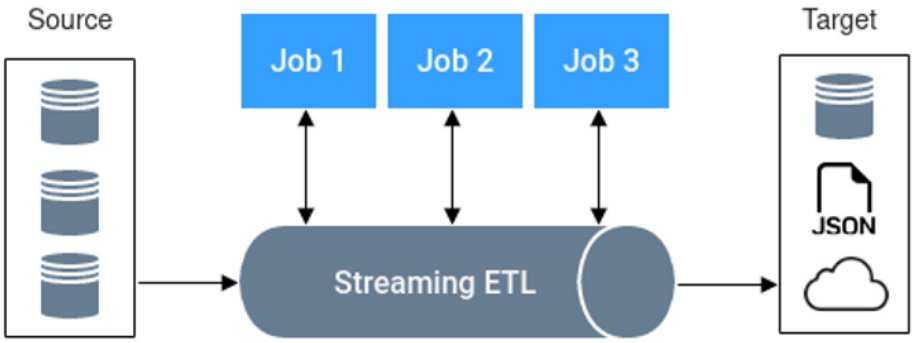

FIGURE 2. Streaming ETL pipeline

A modern streaming data pipelines have 5 principles:

- Streaming: Data streams and streaming platforms are applied to maintain real-time, event-level of reusable data. Data schemas are defined as contracts between system components, ensuring data compatibility.
- Decentralized: Domain-oriented, decentralized data ownership to support shared and reusable data streams within different teams across the entire business.
- Declarative: The logical flow of data can be retrieved by a decelerative language such as SQL without low-level operational details.
- Developer-oriented: Code-based pipeline implementation and agile development practices such as CI/CD pipeline are integrated, allowing teams to develop, test and deploy into different environments with version control.
- Governed: Maintain centralized standards for data observability, security, and compliance while ensuring data visibility, transparency, and compatibility for intuitive search, discovery, and lineage.

2.2.3 Cloud-based data pipeline

To optimize the flexibility and maintainability of a data pipeline, cloud services can be integrated into the system, especially when the existing system is already running in a cloud platform. This approach enhances the pipeline capabilities and operational efficiency, including:

- Scalability and elasticity: Cloud platforms allow seamless scaling of resources such as compute unit or storage on demand, as data volume in the pipeline can be fluctuated.
- Fully managed services: Cloud providers manage their services infrastructure, leaving the clients to focus on developing their main businesses.

- Seamless integration: Services in a specific cloud platform often offer built-in connectors for various data sources.
- High availability and fault tolerance: Cloud platforms provide high availability through multi-zone and multi-region replication, ensuring minimal downtime and data lost. Failover mechanisms support quick recovery from disruptions.
- Global reach and accessibility: Cloud providers' network of data centers across the globe allows deploying the streaming pipeline closer to data sources and users. Cloud-based pipelines can be accessed and managed from anywhere, facilitating collaboration and remove work from distributed teams.

In this architecture, there is a component that acts as a connector to transfer data from an on-premises system to a cloud-based data pipeline, such as AWS DataSync, Azure Data Factory, or Google Cloud Data Fusion.

2.2.4 Data Mesh

As businesses grow, a centralized data pipeline can become complex with a huge data volume. The data mesh attempts to decentralize monolithic data platforms by applying the concepts of domain-driven design to data architecture (Reis and Housley 2022, 161). The benefits of data mesh architecture include:

- Decentralized Data Ownership and Management: Each domain or business unit owns and manages its data products, ensuring data quality and providing self-service access to other domains.
- Agility and Scalability: By decentralizing data management, domains can respond more quickly to changing data requirements and scale their data products independently.
- Improved Data Quality: Domain teams have the expertise and context to ensure data quality at the source, leading to more reliable and trustworthy data.
- Flexibility: Different domains can use the most appropriate technologies and tools for their specific data needs, promoting innovation and experimentation.
- Collaboration and Data Sharing: A data mesh platform facilitates data discovery, access, and governance across domains, encouraging collaboration and data sharing while maintaining control and security.

(Dehghani 2022, 172) proposed a multi-plane architecture to distinguish between different classes of platform services based on their scope of operation:

- Data infrastructure plane: Atomic services to provision and manage physical resources such as storage, compute unit, and pipeline orchestration.
- Data product experience plane: Higher-level abstraction services operate a data product, allow data producers and consumers to create, access and secure a data product.
- Mesh experience plane: Services that operate on a mesh of interconnected data products, such as searching or monitoring the data lineage.

In Netflix, the Data Mesh system is divided into the control plane and the data plane. The controller receives user requests via UI or APIs, then deploys and orchestrates pipelines. Once deployed, the pipeline performs the actual data processing work, as illustrated in Figure 3 (Netflix Technology Blog 2022).

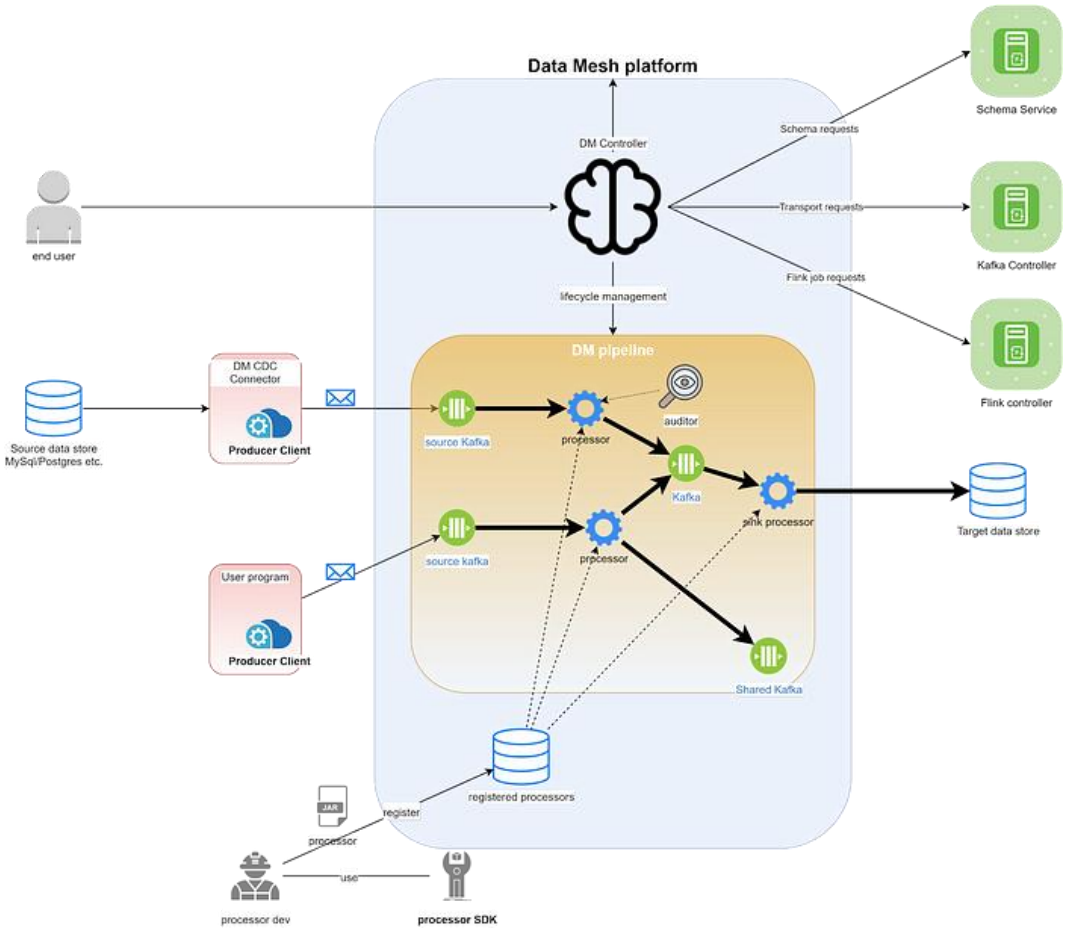

FIGURE 3. Data Mesh system at Netflix

2.3 Google Cloud services for data analytic

Google has been named a Leader in *The Forrester Wave™: Data Lakehouses, Q2 2024* report (Noel Yuhanna 2024). Google Cloud platform offers a comprehensive suite of services tailored for modern data analytics needs. It seamlessly integrates data ingestion, storage, processing, analysis, and visualization, enabling organizations to derive actionable insights from their data, whether it's batch or streaming, structured or unstructured.

The key services include:

- **Data Acquisition**: Cloud Storage, Cloud Pub/Sub. These services facilitate the gathering of data from diverse sources, ensuring its readiness for further processing and analysis.
- **Data Transformation & Processing**: Cloud Dataflow, Dataproc. These services enable the cleaning, structuring, and enrichment of data, making it suitable for analysis and insights extraction even in large-scale systems.
- **Data Storage & Management**: BigQuery, Cloud Spanner. These services provide persistent storage for data, ensuring its accessibility, organization, and integrity. Google's BigQuery delivers not only storage capabilities but also serverless data processing engines with Vertex AI integration, providing a unified access layer for all data, supporting various data formats, and enabling scalable governance across the entire data lakehouse ecosystem.
- **Data Analysis & Visualization**: Looker Studio, Data Studio. These services empower users to explore, analyze, and visually represent data, extracting valuable insights and facilitating data-driven decision-making.
- **Monitoring & Orchestration**: Cloud Monitoring, Cloud Composer. These services enable the monitoring, management, and coordination of data pipelines and workflows, ensuring efficient execution and data integrity.
- **Machine Learning**: Vertex AI, AutoML, BigQuery ML. These services enable the creation, training, and deployment of machine learning models, facilitating predictive analytics, intelligent applications, and automated decision-making.

Figure 4 summarizes the keys Google Cloud analytic services in a landscape, grouped by functionalities.

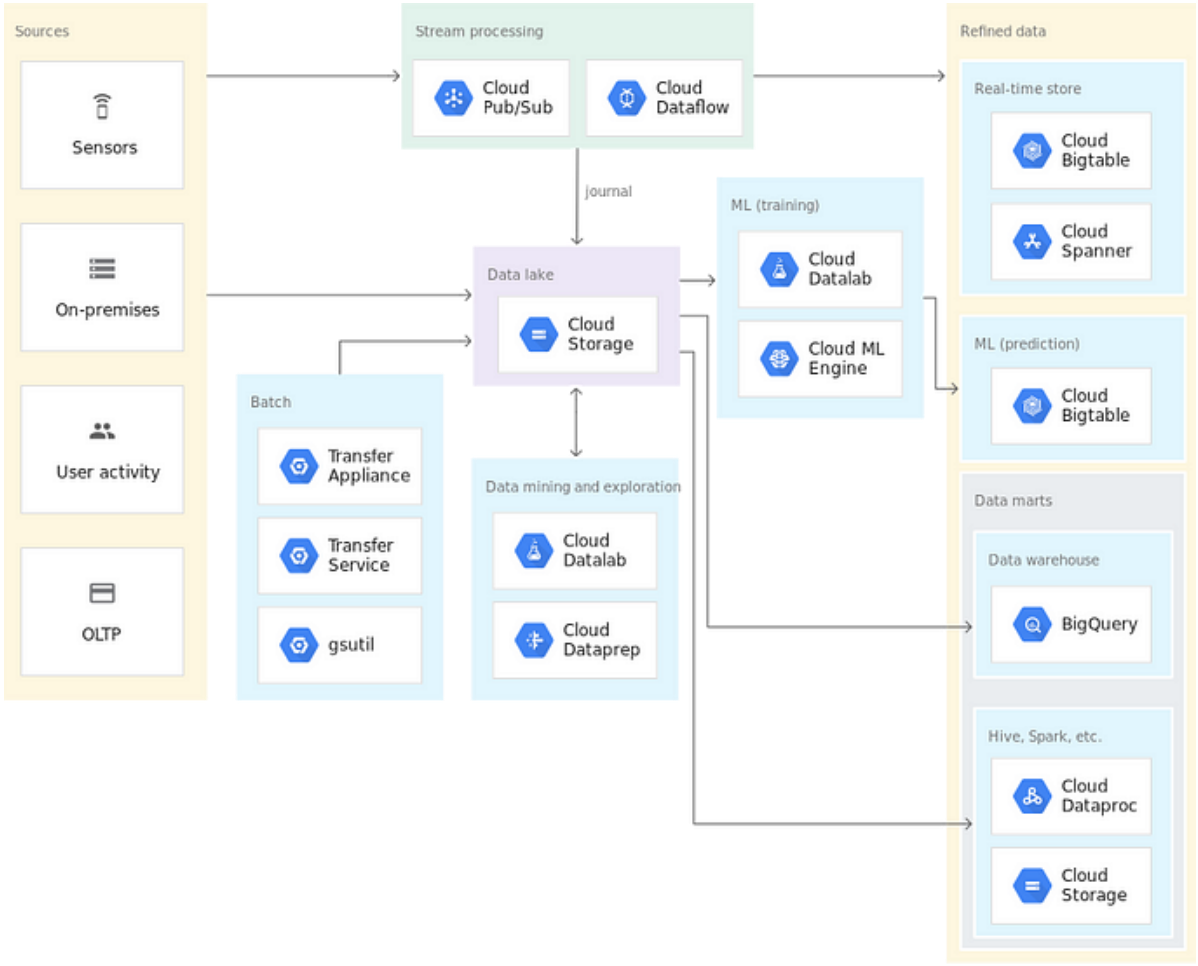

FIGURE 4. Data analytic services on Google Cloud platform (Mahapatra 2020)

3 THEORETICAL FRAMEWORK

The thesis aims to introduce a streaming architecture to integrate a real-time data pipeline into an existing software system. The architecture uses Google Cloud services to load, transform, and present the data, assuming that the existing system runs on an independent platform. Other popular cloud providers, such as AWS or Azure, also support similar methodologies that can be used in exchange.

If the existing system is already cloud-based, it is recommended that the data system be designed based on the same cloud platform, utilizing built-in services for performance improvement and cost optimization.

3.1 Data pipeline using Google Cloud Dataflow

The recommended Google Cloud service for processing data in stream or batch is Dataflow. This service defines pipelines to read data from one or more sources, transform it, and write it to storage. The same programming model can be used for both batch and stream processing, using popular programming languages such as Java, Python, and Go.

Google proposes a typical ELT pipeline using Dataflow service as in Figure 5 below (Google LLC n.d.). The pipeline includes:

- Google Cloud Pub/Sub service: ingests data from an external system and temporarily store the incoming data until it is processed.
- Dataflow: reads data from Pub/Sub, possibly transform the data before writing it to BigQuery storage.
- BigQuery: works as a structured data warehouse
- Looker: connects to BigQuery and provides data insights in real-time

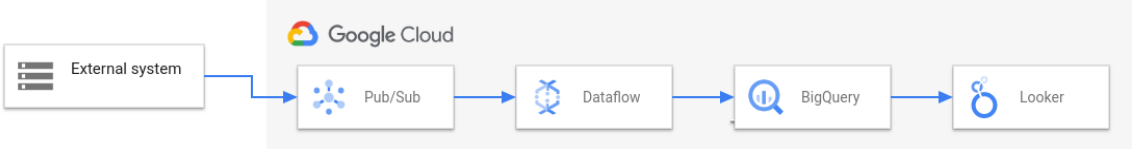

FIGURE 5. A typical ELT pipeline using Dataflow

3.2 ELT architecture for real-time streaming data analytic

Before cloud platforms became popular, a common approach in software architecture was to have their databases self-hosted. In those cases, the databases are set up on a server and managed by in-house administrators, making it challenging to integrate with other cloud services.

Firstly, changes from the existing databases must be observed in real time. This can be done by a Debezium service that supports multiple connectors for different types of database engines. Secondly, the changes must go into an event streaming system as messages and become available for other real-time services. This can be achieved by an Apache Kafka service that can receive messages from Debezium connectors and propagate the messages to any services via Kafka Connect, including the Google Cloud Pub/Sub service. Figure 6 illustrates how Debezium connectors convert existing database changes into event streaming.

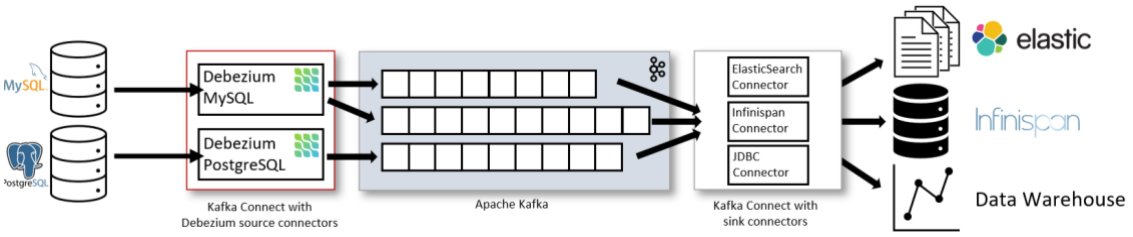

FIGURE 6. Debezium architecture (Debezium Community 2024)

By integrating the Debezium architecture as an intermediate layer between existing self-host databases and Google Cloud Pub/Sub services, the thesis introduces a complete ELT architecture for real-time streaming that works with database engines supported by Debezium, as in Figure 7.

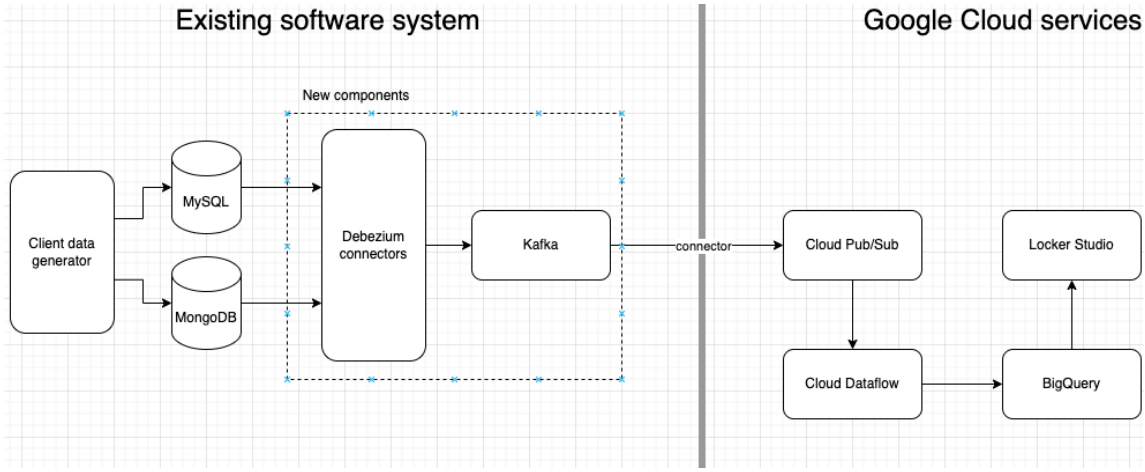

FIGURE 7. A proposed real-time ELT architecture

3.3 Performance metrics

Measuring and monitoring the performance of a pipeline is essential. The metrics should be based on what matters the most to stakeholders, such as uptimes, speed, and cost. It is dangerous to rely on a single metric to observe the entire pipeline story. In contrast, too many metrics make it difficult to focus on the most important ones. Ideally, each metric should have a unique purpose rather than overlapping in what they measure (Densmore 2021, 225–26).

There are numerous metrics for measuring different services in Google Cloud. This thesis discusses some essential metrics that are monitored by Google Cloud by default. However, depending on the real system's characteristics, the engineer can decide which metrics are most important. The performance of the existing system is not considered in this thesis because it varies between different systems.

Google Cloud Pub/Sub

- Throughput: Because Google Cloud Pub/Sub is the entrance of the data system, it is important to measure the amount of incoming data via these metrics:
    - Publish requests: Rate of publish requests sent to this topic.
    - Publish message count: Rate of messages published to the topic.
    - Publish throughput in bytes: Rate of bytes sent to this topic.
    - Average message size: The average message size in this topic is grouped by minute.
- Ingestion: Messages in Pub/Sub are expected to be processed by subscribers.
    - Ingested bytes: Number of bytes of messages ingested by the topic by minute. If this metric is not close to the Publish throughput in bytes metric, the system has unprocessed data.
    - Ingested messages: Number of messages ingested by the topic by minute. If this metric is not close to the Publish message count metric, the system is stacking up by messages.

Dataflow

- Throughput: Number of elements consumed by the Dataflow job every second.
- System latency: The number of seconds an item of data has been processing or waiting inside any pipeline source.
- Resource metrics

- CPU utilization: Fraction of CPU utilization for all cores on a single App Engine flexible instance.
- Memory utilization: Estimated total memory used vs memory limit in bytes across all workers in this Dataflow job.

BigQuery

- Inserted Bytes and Inserted Rows: The number of bytes and rows uploaded by the project
- Bytes Processed: Total Bytes processed within the selected time frame.

In the Google Cloud Monitoring service, there are built-in dashboards for each running service with popular metrics. Creating a dashboard with multiple related metrics in one chart for comparison is also possible.

4 METHODOLOGY

Chapter 3 introduces a theoretical architecture of a real-time data pipeline that reacts to changes from the data sources, illustrated by *Figure* 7. In this chapter, the architecture will be discussed in detail, such as the technologies behind the components, how they communicate with each other and important configurations.

4.1 Data sources

MySQL is a highly popular open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) to manage and query data. Known for its performance, reliability, and ease of use, MySQL is widely deployed in various applications, ranging from small websites to large-scale enterprise systems.

MongoDB is a popular NoSQL database known for its flexibility and scalability. It uses a document-oriented data model, storing data in JSON-like documents with dynamic schemas, making it well-suited for handling unstructured or semi-structured data. MongoDB's horizontal scaling capabilities and high performance make it a preferred choice for modern applications with evolving data requirements and high traffic loads.

In this thesis, MySQL and MongoDB databases are the experimental data sources. They are installed on a dedicated server to simulate a separate existing system. The thesis implements a data generator script to populate sample data into these databases with some controllable parameters, such as the total number of entities, the number of entities in a batch, and the interval of each batch.

4.2 Connector component

The connector component propagates data from the software system to the data analytic system. Different cloud platforms provide different tools, such as AWS Airbyte, Azure Data Factory, Skyvia, etc. This thesis uses the Debezium platform for change data capture (CDC). It monitors and captures row-level changes in databases and converts them into a stream of events.

Debezium is built on top of Apache Kafka with a set of Kafka Connect connectors, which supports many databases such as Cassandra, Db2, MongoDB, MySQL, Oracle Database, PostgreSQL, SQL Server, and Vitess. The connector requires a database connection credential and the Kafka cluster connection information. MongoDB and MySQL connectors are experimented in this thesis.

4.2.1 MySQL Connector

MySQL has a binary log (binlog) that records all operations in the order in which they are committed to the database. This includes changes to table schemas as well as changes to the data in tables. MySQL uses the binlog for replication and recovery. The Debezium MySQL connector reads the binlog, produces change events for row-level INSERT, UPDATE, and DELETE operations, and emits the change events to Kafka topics. (Debezium Community 2018).

This thesis experiments with a MySQL standalone instance, although different topologies are also supported such as standalone, primary – replica, high available clusters, multi-primary and cloud-hosted.

The data change event follows JSON structure with two main fields:

- Schema: There is a schema to describe the table structure such as primary key or unique key. Another schema type is to specify the structure of the row that was changed.
- Payload: This contains the actual data for the table or row that was changed.

The MySQL Connector captures create, update, delete events for row changes, Primary key updates, and truncate events for table changes. This thesis dives deeper into the create event, assuming that new data is constantly being inserted into the system.

The *create* event schema describes the data structure of the payload's values, including the type, field name and nullability. The *name* fields indicate the structure of a specific payload's section. For example, mysql-server-1.inventory.customers.Value defines the structure of the data before changing, where `mysql-server1` is the connector name, `inventory` is the database name and `customer` is the table name.

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "mysql-server-1.inventory.customers.Value",
        "field": "before"
      }
    ],
    "optional": false,
    "name": "mysql-server-1.inventory.customers.Envelope"
  }
}
```

Since the table and row structures are not usually changed, it is recommended to not include the schema in every events, this can be configured in the connector with the `value.converter.schemas.enable` option.

The *create* event payload contains the actual data changes. There is an optional *before* field to specify the state of the row before the event occurred. This field is null in a *create* event. The *after* field specifies the state of the row after the changes, this field can also be null in a *delete* event. The *source* field is mandatory as it describes the source metadata for the event. Below is an example of *create* event payload.


```
{
 "schema": { ... },
 "payload": {
  "before": null,
  "after": {
   "id": 1004,
   "first_name": "Anne Marie",
   "last_name": "Kretchmar",
   "email": "annek@noanswer.org"
  },
  "source": {
   "version": "2.7.2.Final",
   "name": "mysql-server-1",
   "connector": "mysql",
   "name": "mysql-server-1",
   "ts_ms": 1465581029100,
   "ts_ms": 1465581029100000,
   "ts_ms": 1465581029100000000,
   "snapshot": false,
   "db": "inventory",
   "table": "customers",
   {...}
  },
  "op": "u",
  "ts_ms": 1465581029523,
  "ts_ms": 1465581029523758,
  "ts_ms": 1465581029523758914
 }
}
```

4.2.2 MongoDB Connector

The Debezium's MongoDB connector tracks a MongoDB replica set or a MongoDB sharded cluster for document changes in databases and collections, recording those changes as events in Kafka topics. MongoDB has the change streams feature which exposes the changes that occur in a collection as an event stream. The Debezium connector monitors the stream and then delivers the changes downstream. The MongoDB *replica set* topology is implemented in this project, but *shared cluster* topology is also supported.

The data change event has a similar structure as the MySQL Connector. A key different is that the before and after fields, which specify the state of the MongoDB document are in a string-format, compared to JSON format of the MySQL Connector. This might require an extra conversion process to deserialize the string to a programmable object. Below is an example of create event payload from the MongoDB Connector.


```
{
   "schema": { ... },
   "payload": {
     "op": "u",
     "ts_ms": 1465491461815,
     "ts_us": 1465491461815698,
     "ts_ns": 1465491461815698142,
     "before":"{\"_id\": {\"$numberLong\": \"1004\"},\"first_name\": \"unknown\",\"last_name\":
\"Kretchmar\",\"email\": \"annek@noanswer.org\"}",
     "after":"{\"_id\": {\"$numberLong\": \"1004\"},\"first_name\": \"Anne Marie\",\"last_name\":
\"Kretchmar\",\"email\": \"annek@noanswer.org\"}",
     "updateDescription": {
       "removedFields": null,
       "updatedFields": "{\"first_name\": \"Anne Marie\"}",
       "truncatedArrays": null
     },
     "source": {
       "version": "2.7.2.Final",
       "connector": "mongodb",
       "name": "fulfillment",
       "ts_ms": 1558965508000,
```

```
    "ts_us": 1558965508000000,
    "ts_ns": 1558965508000000000,
    {...}
   }
  }
}
```

4.2.3 Apache Kafka and Pub/Sub connection

Since the Debezium connectors capture changes from data sources and emit the change events to Kafka topics, data from Apache Kafka can be streamed to any other systems that support the Kafka Connect tool. This tool can run in two modes of execution: standalone (single process) and distributed (Apache Software Foundation 2020). It also accepts third-party plugins to work with any other systems.

The plugin *Pub/Sub Group Kafka Connector,* developed by Google Cloud (googleapis 2023), is an extension to Kafka Connect tool. It has two separate connectors in a single library:

- *The sink connector* reads records from Kafka topics and publishes them to Pub/Sub.
- *The source connector* reads messages from a Pub/Sub topic and publishes them to Kafka.

Kafka Connect also uses converters to serialize keys and values to and from Kafka. The converters specify the format of data in Kafka and how to translate it into Kafka Connect data. Since Debezium connectors capture data in JSON format, the built-in library `org.apache.kafka.connect.json.JsonConverter` of Kafka can be used.

4.3 Google Cloud services

Google Cloud Platform supports multiple services working together to implement a comprehensive data analytic system. In this thesis, the main services, including Pub/Sub, Dataflow, BigQuery, and Looker Studio, are combined into a complete real-time data analytic pipeline that ingests, processes, stores, and visualizes data.

4.3.1 Pub/Sub

Messaging technologies are essential in modern software systems, enabling event-driven architecture with asynchronous communication between different components for high throughput, scalability, and fault tolerance. While popular tools such as Apache Kafka and RabbitMQ are open-source and can be self-hosted, Google introduced Pub/Sub messaging services, which are fully managed by the Google Cloud platform with many advantages, including:

- Administration simplification: The processes for infrastructure provisioning, scaling, maintaining, and monitoring are much simpler and more flexible.
- Global network: Messages can be propagated across regions with low latency.
- Cost-effectiveness: only pay for the consumed resources
- Google Cloud platform integration: Many other Google Cloud services can be integrated into Pub/Sub.

Using the publish/subscribe model, the Pub/Sub service decouples the sender of messages (the publisher) from the receivers (the subscriber). Figure 8 highlights the main Pub/Sub components:

- Publisher: the data source where messages are created
- Topic: Messages from the publishers are categorized into topics and stored in a storage until they are acknowledged
- Schema: defines formats for messages, supporting data validation and data integrity.
- Subscription: an observer on a specific topic that supports controlling the message order and delivery status for multiple subscribers
- Subscriber: consume messages from a subscription

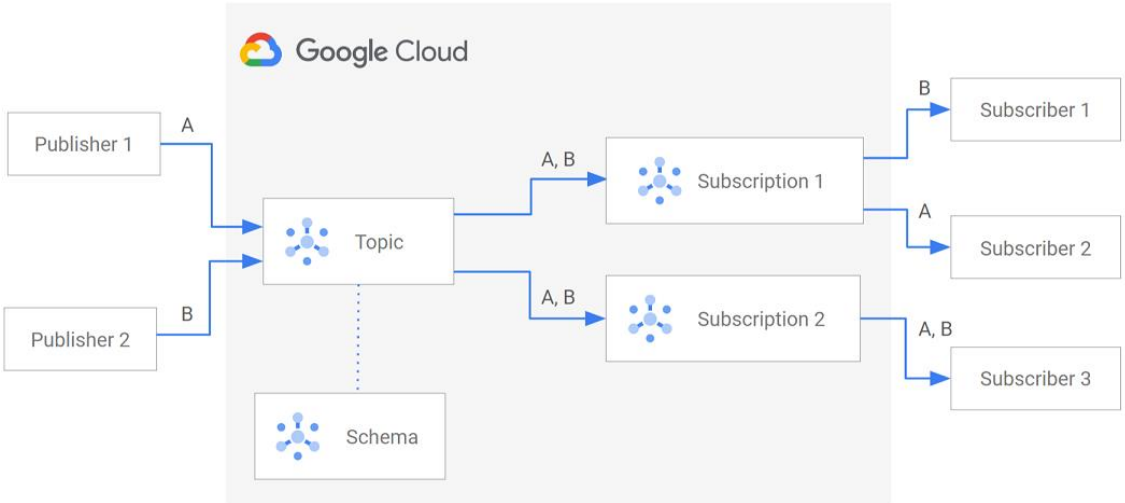

FIGURE 8. Google Cloud Pub/Sub service (Google LLC 2024)

The Pub/Sub service pricing is mainly based on Throughput cost, calculated as $40 per TiB after the first free 10 GiB. If messages are retained, it costs $0.27 per GiB-month. This storage fee also applies for messages that are unacknowledged for more than 1 day. In case the data is transferred across regions, the Virtual Private Cloud (VPC) network pricing is applied, for example, $0.14 per GiB between Europe and North America.

4.3.2 Dataflow

The Dataflow service supports unifying stream and batch data processing at scale. The pipeline definition is programmed in Java, Go, or Python. When deployed, it runs as virtual machine instances that can read data from multiple sources, transform it, and write it to a destination. Google Cloud Dataproc is a similar service, but it focuses on complex batch processing pipelines with more control over the infrastructure.

The main advantages of Dataflow include:

- Administration simplification: Similar to Pub/Sub, Dataflow is fully managed by the Google Cloud platform, reducing the complexity of infrastructure work.
- Scalability with cost efficiency: More resources can be automatically allocated on demand to support parallel processing and cost efficiency.
- Portability: The Apache Beam project supports defining the pipeline in different programming languages that can run on Dataflow and other platforms, such as Apache Flink or Apache Spark.
- Observability: The Dataflow monitoring interface provides jobs management with graphical presentation of the pipeline, including execution details of each stage, error logs or bottleneck detection.

Dataflow is based on the Apache Beam, an open-source programming model for data pipelines, an example pipeline is visualized in Figure 9.

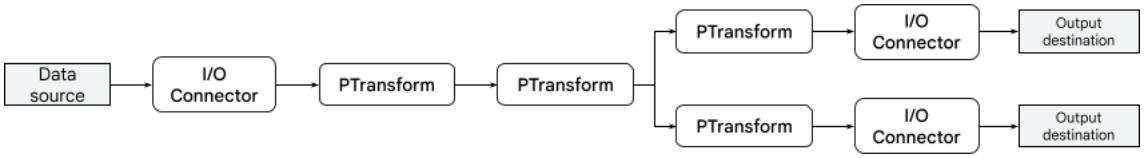

FIGURE 9. An Apache Beam pipeline (Google LLC 2025)

Basic concepts in Apache Beam includes:

- Pipeline: an object that encapsulates the complete series of data read, transformation and output. Each pipeline runs as a single job in Dataflow.
- PCollection: a dataset object which acts as an input for a step in the pipeline or an output after every step. PCollection can be a fixed size dataset in a batch processing pipeline, or unbounded data in a stream processing pipeline.
- Transformation: a processing process that takes one or more PCollections, performs the defined operations and produces one or more PCollections.
    - ParDo: the core parallel processing operation that invokes a user-specified function on each element of the input dataset.
- Pipeline I/O: the connectors for reading data into the pipeline and writing output data.

The code below defines an example pipeline that streams data messages from Pub/Sub and group them into fixed-sized minute intervals before writing to Google Cloud Storage.

```
public static void main(String[] args) throws IOException {
  int numShards = 1; // The maximum number of shards when writing output.

  PubSubToGcsOptions options =
      PipelineOptionsFactory.fromArgs(args).withValidation().as(PubSubToGcsOptions.class);
  options.setStreaming(true);

  Pipeline pipeline = Pipeline.create(options);
  pipeline
      .apply("Read PubSub Messages", PubsubIO.readStrings().fromTopic(options.getInputTopic()))
      .apply(Window.into(FixedWindows.of(Duration.standardMinutes(options.getWindowSize()))))
      .apply("Write Files to GCS", new WriteOneFilePerWindow(options.getOutput(), numShards));

  pipeline.run().waitUntilFinish();
}
```

The pricing model of Dataflow depends mostly on the compute resources, including the Streaming Engine data processing, worker CPU and worker memory. For example, in *europe-west1* region, on monthly basis, 1 vCPU cost $51.84 and 1 GB memory cost $3 and 1GB SSD persistent disk costs $0.22. The streaming worker by default has 4 vCPU, 15 GB memory and 30 GB SSD persistent disk if using Streaming Engine.

4.3.3 BigQuery

BigQuery is a data warehouse which is fully managed by Google Cloud with rich built-in features for data analytics. Data can be streamed continuously and analyzed in real-time in a scalable-distributed fashion to query terabytes of data in seconds. BigQuery includes a replicated and distributed storage layer and a high-scalable compute layer.

BigQuery shares similar concepts to RDBMS databases:

- **Dataset**: a top-level logical container for BigQuery resources including tables, views, functions and procedures. This concept is similar to a database in RDBMS databases. Dataset is location-specific, it can base on a single region or multi-region.
- **Table**: a dataset can have many tables. Each table contains structured data which is stored in a columnar format, every column must have a data type defined.
- **View**: a logical table that is defined by a SQL query. This query is run every time the view is access. There is also Materialized view that periodically run the query and catch the result for faster access.
- **Routine**: a block of SQL code which can be a stored procedure, user-defined function or table function, which performs complex calculations and business logic. This improves code reusability, maintainability and modularity.
- **Indexing**: unstructured text and semi-structured JSON data can be indexed to improve full-text searching performance. There is also vector search in BigQuery which supports searching entities by semantic similarity.

Data analytics in BigQuery can be done via the built-in GoogleSQL dialect. It supports geospatial analytics with a set of geography functions, business intelligence analytics with many functions in BigQuery BI Engine, or machine learning with BigQuery ML to train and use a model to predict outcomes. With data exploration features such as Table Explorer, Data Insights, Data Profile Scan, and Data Canvas, BigQuery helps understand the data even before running a SQL query.

A query can be run on demand via the Google Cloud interactive console or scheduled to run as batch or continuous query jobs. The process of running a query includes execution tree generation, preparing data in shuffle tier, query plan generation and execution, query monitoring, and writing query results to persistent storage.

To improve query performance, the execution graph provides a visual representation of each execution stage, such as the number of records processed and a breakdown of the processing time in a stage, as shown in Figure 10. Various metrics, logs, and metadata views can be monitored to ensure the result's reliability.

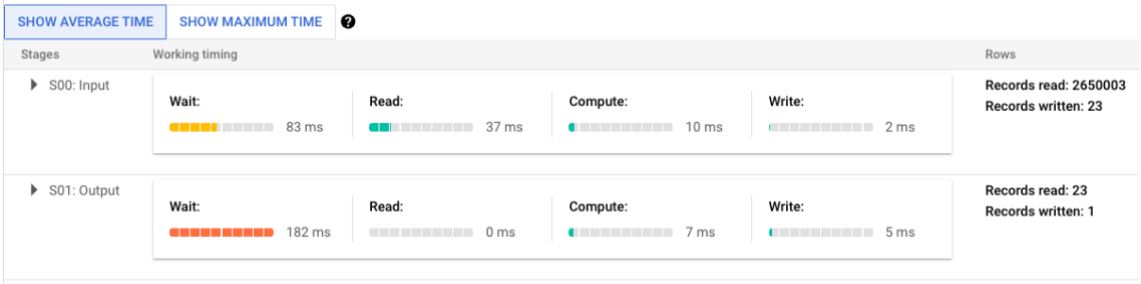

FIGURE 10. BigQuery execution details

Pricing in BigQuery is calculated separately for the compute layer and storage layer. The computation price depends on the data scanned by queries, which is $6.25 per TiB, with the first 1 TiB per month being free. It is possible to switch to a capacity compute pricing model, which is based on processing units (virtual CPUs) over time, a vCPU costs $31,68 per month. The storage prices include active, long-term, and metadata storage. Data in a table or a table partition is considered as long-term when not modified for 90 days. Data can be stored logically or physically. For example, in Europe region, it costs $0.02 GiB for logical storage per month compared to $0.044 for physical storage. Long-term storage usually costs 50% cheaper. Therefore, a recommended practice is to split unchanged data into a separate shared table.

Machine learning in BigQuery

By integration machine learning features as functions in GoogleSQL, machine learning models can be run like a simple SQL query without programming knowledge. BigQuery supports multiple training models:

- Built-in models: linear regression, logistic regression, k-mean clustering, matrix factorization, principal component analysis (PCA) and time series.
- External models via Vertex AI: deep neural network (DNN), Wide & Deep learning, Autoencoder, Boosted Tree, Random Forest and AutoML

In the example GoogleSQL query below, a logistic regression model is created to predict whether a website visitor will make a transaction.

```
CREATE MODEL `bigquery-public-
data.google_analytics_sample.ga_sessions_20170801`
OPTIONS(model_type='logistic_reg') AS
SELECT
IF(totals.transactions IS NULL, 0, 1) AS label,
totals.pageviews AS pageviews,device.operatingSystem AS os,
device.isMobile AS is_mobile, geoNetwork.country AS country
FROM `bigquery-public-data.google_analytics_sample.ga_sessions_*`
WHERE _TABLE_SUFFIX BETWEEN '20160801' AND '20170630'
```

The logistic regression model splits the training data into two classes – for example, `0` for *no transaction* and `1` for *transaction made*. Base on the training data, it builds a linear combination with multiple independent variables such as *pageviews*, *os*, is_*mobile* and *country* in this case. The linear combination is used to estimate the probability that a new given data is one of the classes. The training data ranges from 01/08/2016 to 30/06/2017 in this example.

To evaluate the performance of the trainned model, GoogleSQL uses the `ML.EVALUATE` function to compare predicted values against the actual data from 01/07/2017 to 01/08/2017.

```
SELECT * FROM
ML.EVALUATE(MODEL `bqml_tutorial.sample_model`, (
SELECT
IF(totals.transactions IS NULL, 0, 1) AS label,
totals.pageviews AS pageviews,device.operatingSystem AS os,
device.isMobile AS is_mobile, geoNetwork.country AS country
FROM `bigquery-public-data.google_analytics_sample.ga_sessions_*`
WHERE _TABLE_SUFFIX BETWEEN '20170701' AND '20170801'))
```

For a logistic regression model, the evaluation result includes *precision, recall, accuracy, f1_score, log_loss and roc_auc* metrics.

Finally, the model can be used as below to predict the number of transactions made for each country with `ML.PREDICT` function.

```
SELECT country,SUM(predicted_label) as total_predicted_purchases
FROM ML.PREDICT(MODEL `bqml_tutorial.sample_model`, (
SELECT
totals.pageviews AS pageviews,device.operatingSystem AS os,
device.isMobile AS is_mobile, geoNetwork.country AS country
FROM `bigquery-public-data.google_analytics_sample.ga_sessions_*`
WHERE _TABLE_SUFFIX BETWEEN '20170701' AND '20170801')
) GROUP BY country
```

BigQuery ML pricing depends on the data used in different operations. The creation operation costs $312.5 per TiB for regression, clustering, PCA or time series models. Evaluation, inspection and prediction costs $6.25 per TiB.

4.3.4 Looker Studio

Looker Studio is a free business intelligence platform that helps creating data visualizations. It has built-in and partner connectors to access various data storages, including BigQuery. The Looker Studio Pro version allows creating team workspace and shareable reports. A similar service is Looker which inspires in-depth data analytics and modeling, offering more control and insights through LookML but requires more technical expertise.

The process of creating a Looker Studio report is:

- Connect the new report to one or more data sources: there are more than a thousand of built-in and partner connectors. The data can be blended or filtered, new data fields can be created logically via calculations or grouping from original fields.
- Create a new report from scratch or from a template.
- Add charts or other controls such as label, input box, button, drop-down list, date range selector, data control.
- Publish, share, embed or download the report.

4.4 System monitoring

This section explains performance monitor of the Google Cloud services in the proposed ELT architecture against different experimental scenarios in the thesis.

4.4.1 Performance testing

In the thesis, the proposed architecture in Figure 7 is set up in Google Cloud within the same region to avoid network congestion. The existing systems run on a separate Compute Engine instance. There are two testing scenarios were conducted:

1. One existing system produces 2000 messages per second to a Pub/Sub topic for 60 minutes. This is the default throughput quota of the service (Google LLC 2025a) which can be increased via customer support. The expectation is all data would be available in BigQuery as soon as the data generation is completed.
2. Similar to the first scenario with two existing systems, connects to two Pub/Sub topics and two Dataflow jobs. The expectation is all data would be available in BigQuery after 60 minutes, means that the pipeline can be scaled horizontally for parallel processing.
3. Test a heavy query performance against millions of records in BigQuery, comparing to when it grows to a billion of records.

4.4.2 Metrics monitoring

Google Cloud services usually have their own metrics that are monitored automatically. It is possible to select some metrics and put them in a customized dashboard. The thesis selects the metrics below for each service:

- Pub/Sub topic: number of messages, requests, average message size. Publish throughput is the main metric that affects the service cost. For Pub/Sub subscription, the amounts of acknowledge and unacknowledged are important to ensure no congestion when consuming messages from Pub/Sub.
- Dataflow service: compute resources such as CPU and memory utilization
- BigQuery: scanned bytes and stored bytes are metrics that affect the cost, while Query count and Query execution times reflect the performance.

The completed dashboard is illustrated as in Figure 11.

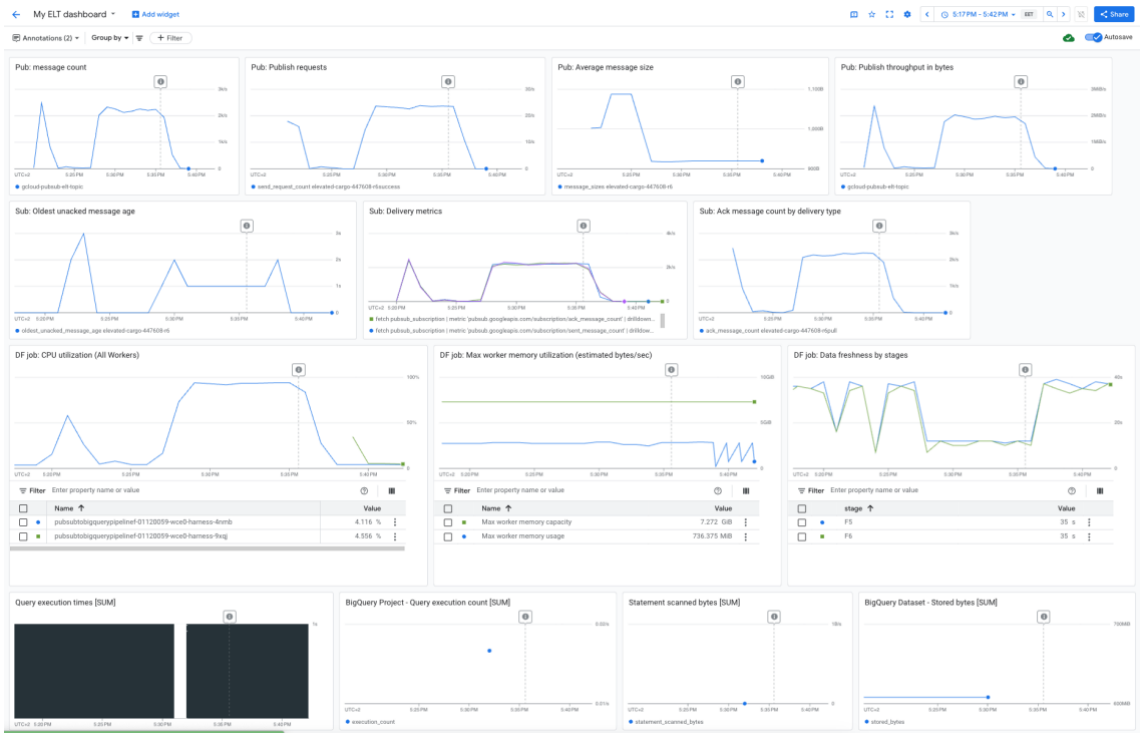

FIGURE 11. A customized dashboard for system monitoring

It is possible to combine multiple metrics into one chart for a clearer comparison.

5 IMPLEMENTATION

This chapter explains detailed instructions for setting up the proposal real-time pipeline system as in Figure 7, including an example self-host software system and Google Cloud data analytic services. Technical highlights of each component in the system are already discussed in Chapter 4.

The whole existing system is set up using Docker virtualization technology (Docker 2024). Each component runs as a Docker container and can connect to each other via a Docker internal network. This setup makes it more convenient to deploy all components in any environment that supports Docker. In the real world, Kubernetes is more suitable since each component may have multiple instances when scaling horizontally.

5.1 Sample data source

There are a MySQL database and a MongoDB database, representing different database technologies in the existing system. The `user_registration` MySQL table is declared as the code snippet below, while it is not necessary to define any data structure in MongoDB database.

```
CREATE TABLE user_registration (
    id INT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    gender VARCHAR(50),
    birth_date TIMESTAMP,
    image_url VARCHAR(255),
    phone VARCHAR(50),
    address TEXT,
    country VARCHAR(100),
    job_type VARCHAR(100),
    job_title VARCHAR(100),
    iban VARCHAR(34),
    bic VARCHAR(11),
    currency_code VARCHAR(3),
```

```
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

The data generator component is implemented by Typescript, generating sample data of new users registered into the system, using a library `@faker-js/faker` which supports generating different kinds of mock data. For example, the data generator script bellow generates 1000 new users with all details every one second into both databases until each has 60000 new users.

```
server yarn generate -db=all -total=60000 -batchSize=1000 -interval=1000 -full
```

5.2 Connector components

Debezium connectors and local Kafka are two new components that observe database changes and emit related events. Setting up a local Kafka cluster with Docker can be done based on `docker.io/bitnami/kafka` community Docker image. The Google Cloud Pub/Sub Group Kafka Connector library is included in `Dockerfile.kafka` file for building the local Docker image of Kafka.

```
FROM docker.io/bitnami/kafka:3.6-debian-11
ENV KAFKA_HOME /opt/bitnami/kafka
WORKDIR /home/PubSubKafkaConnector
COPY dockerfiles/gcloud-pubsub/pubsub-group-kafka-connector-1.2.0.jar .
EXPOSE 9092
```

The Google Cloud credentials file path also has to be defined as `GOOGLE_APPLICATION_CREDENTIALS` environmental variable when running the Kafka container.

Running the Debezium Connect service in docker only requires the Kafka cluster URL in `BOOTSTRAP_SERVERS` environmental variable. However, it requires extra configurations to create connectors for each database it wants to listen. For example, the two databases in the thesis requires two connectors to be configured as described in the table below:

TABLE 2. Debezium connectors to MySQL and MongoDB databases

MySQL connector	
`connector.class`	`io.debezium.connector.mysql.MySqlConnector` The Java library that supports MySQL connection

`database`	Including hostname, port, user and password for the connector to have access to the listening table
`key.converter`	`org.apache.kafka.connect.json.JsonConverter` How the message key is converted while transporting. It is recommended to have `key.converter.schemas.enable=false` to avoid unnecessary metadata.
`value.converter`	Similar to `key.converter` but for the value of the message, `value.converter.schemas.enable=false` is also recommended.
`topic.prefix`	The Kafka topic to receive events from this connector
`database.include.list`	The databases to be observed
`table.include.list`	The tables to be observed
MongoDB connector	
`connector.class`	`io.debezium.connector.mongodb.MongoDbConnector` The Java library that supports MongoDB connection
`mongodb.connection.string`	The connection string to the MongoDB instance, it usually includes hostname, port, username and password
`key.converter`	Similar to in MySQL connector
`value.converter`	Similar to in MySQL connector
`topic.prefix`	In the thesis, this is the same Kafka topic with MySQL connector. In case there are special data from MongoDB to be handled, it should be in a separate topic with a separate Dataflow job.
`database.include.list`	The databases to be observed
`collection.include.list`	The collections to be observed

The list of connectors can be verified in the Debezium Connector service via the command:

```
curl -H "Accept:application/json" localhost:8083/connectors/
```

The last step is to setup a Kafka Connect process together with the add-on library Google Cloud Pub/Sub Group Kafka Connector via executing this command inside the Kafka container:

```
kafka /opt/bitnami/kafka/bin/connect-standalone.sh \
  /hostdata/config/connect-standalone.properties \
  /hostdata/config/cps-sink-connector.properties
```

There is a `connect-standalone.properties` file to specify the add-on library and a `cps-sink-connector.properties` file to specify the configurations of both sides: the local Kafka topics and the Google Cloud Pub/Sub details such as the project ID, Pub/Sub topic.

With these services running locally, Debezium connectors can observe changes from databases and send related events to Kakfa, where they will be deliver to Pub/Sub service over the network.

5.3 Pub/Sub

Google Cloud services can be managed via the web interface at https://console.cloud.google.com/, there is a Cloud Shell tool that works like a terminal running on a server that connects to all services. Any computer which has the *gcloud CLI* tool with valid authentication can act as a Cloud Shell terminal, meaning that all the Cloud Shell commands mentioned in the thesis can be prepared as Linux scripts and executed in an automated CI/CD pipeline (GitHub 2024).

Many Google Cloud services are disabled by default to avoid unexpected charges. Enabling a service such as Pub/Sub can be done in *Enable APIs & services* menu, as in the figure below.

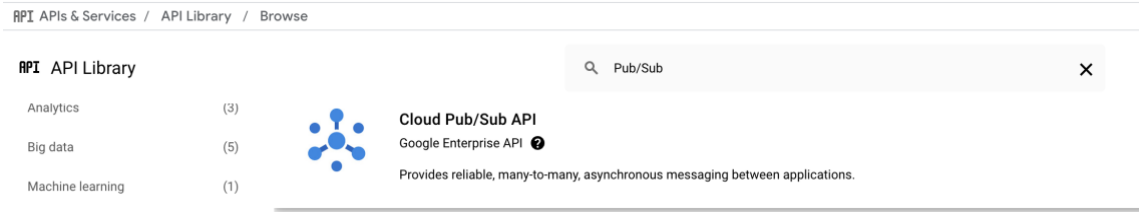

FIGURE 12. Enabling a Google Cloud service

The Cloud Shell commands below create a topic and a subscription, where `$TOPIC_ID` or `$SUBSCRIPTION_ID` are customizable names.

```
gcloud pubsub topics create $TOPIC_ID
gcloud pubsub subscriptions create $SUBSCRIPTION_ID --topic=$TOPIC_ID
```

They will be visible on the web intrface when ready.

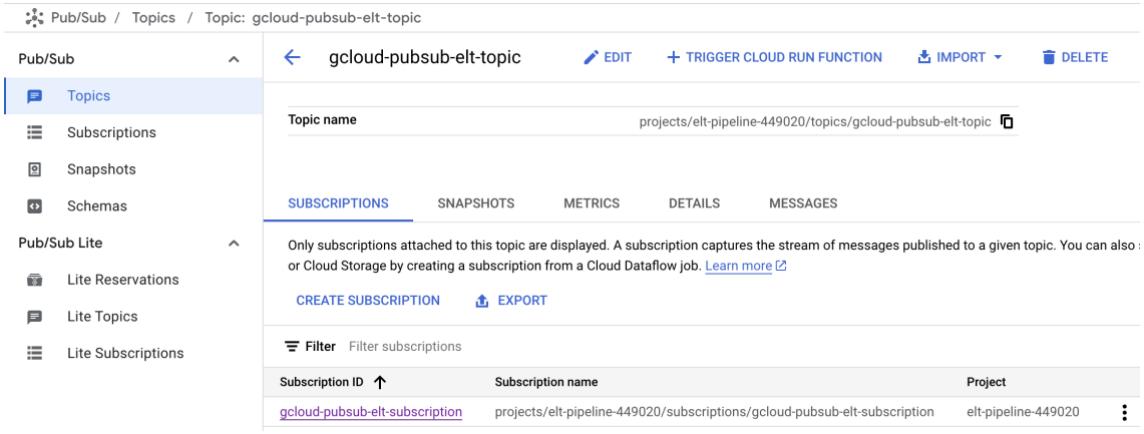

FIGURE 13. Google Cloud Pub/Sub interface

5.4 BigQuery

BigQuery storage should be ready before Dataflow Job can start. With the Cloud Shell commands below, a dataset *bq_elt_dataset* is created together with a table *bq_user_registration* inside.

```
bq mk --location $REGION --dataset bq_elt_dataset
bq mk \
--schema
"data_connector:STRING,id:STRING,email:STRING,password:STRING,first_name:STRING,last_name:ST
RING,
gender:STRING,
birth_date:TIMESTAMP,image_url:STRING,phone:STRING,address:STRING,country:STRING,job_type:S
TRING,job_title:STRING,iban:STRING,bic:STRING,currency_code:STRING,created_at:TIMESTAMP" \
-t bq_elt_dataset.bq_user_registration
```

The storage can be verified by navigating to BigQuery service in the web interface.

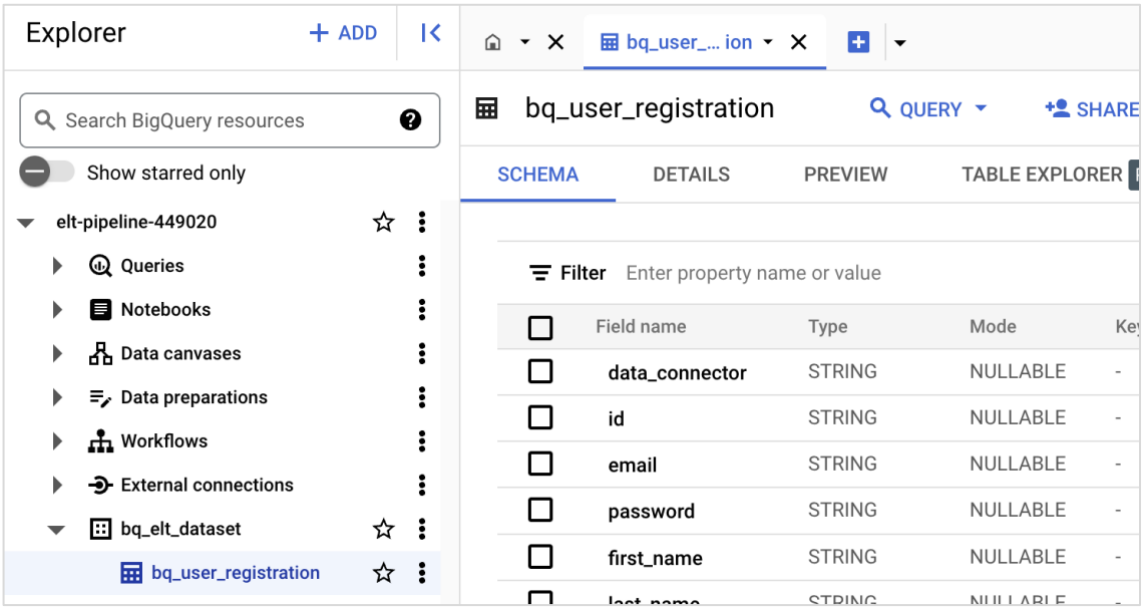

FIGURE 14. BigQuery interface

5.5 Dataflow and Apache Beam

Dataflow manages the jobs which do the heavy lifting of receiving messages from Pub/Sub subscribers, processing them and writing to BigQuery storage. The Apache Beam open-source project provides libraries in Java, Python and Go languages to define a job. In the thesis, Java was used together with the Apache Beam Java SDK.

The main Java class of the job is defined in a similar way as in section 4.3.2, but there is a customized ParDo operation to process the message before calling the BigQueryIO library to write it as a table row in BigQuery.

```
PubSubToGcsOptions options =
PipelineOptionsFactory.fromArgs(args).withValidation().as(PubSubToGcsOptions.class);
options.setStreaming(true);
Pipeline pipeline = Pipeline.create(options);
pipeline.apply("Read PubSub Topic", PubsubIO.readMessagesWithAttributes().fromTopic(options.getInputTopic()))
  .apply(ParDo.of(new UserRegistrationToTableRowDoFn()))
  .apply(BigQueryIO.writeTableRows()
  .withSchema(UserRegistrationTableSchema.defineSchema())
  .withWriteDisposition(BigQueryIO.Write.WriteDisposition.WRITE_APPEND)
                .to(options.getOutput()).withoutValidation());

pipeline.run().waitUntilFinish();
```

The ParDo implementation is as below, which simply reads the message payload to a string. The string is then converted to Java objects such as *KafkaConnectorMessage* or *UserRegistration* using the Google Gson library. Although these objects are optional, the thesis chooses this implementation to follow object-oriented programming principles in Java, making it more convenient for data validations or complex logics. The final purpose is to have a valid *TableRow* object as an output for the next step in the pipeline.

```
static class UserRegistrationToTableRowDoFn extends DoFn<PubsubMessage, TableRow> {
  @ProcessElement
  public void processElement(@Element PubsubMessage jsonPubSubMessage, OutputReceiver<TableRow> out) {
    byte[] payloadBytes = jsonPubSubMessage.getPayload();
    String jsonString = new String(payloadBytes, StandardCharsets.UTF_8);

    KafkaConnectorMessage kMessage = CommonUtil.getKafkaMessageObject(jsonString);
    UserRegistration element = CommonUtil.getUserRegistration(jsonString);
    TableRow row = new TableRow();
    row.set("data_connector", kMessage.source.connector);
    row.set("id", element.getId());
    row.set("email", element.email);
    // other properties are similar
    row.set("created_at", element.getCreatedAt());
    out.output(row);
  }
}
```

A complete implementation of the Dataflow job is included in Appendix 1. The job requires a Google Cloud Storage bucket to store temporary file, which can be created via:

```
gcloud storage buckets create gs://$BUCKET_NAME --location $REGION
```

After uploading the completed code base to Cloud Shell and enable Dataflow API, the job can be started via the command:

```
mvn compile exec:java \
-Dexec.mainClass=org.example.PubSubToBigQueryPipelineForUserRegistration \
-Dexec.cleanupDaemonThreads=false \
-Dexec.args=" \
    --dataflowServiceOptions=enable_preflight_validation=false \
    --enableStreamingEngine=true \
    --project=$PROJECT_ID \
    --region=$REGION \
    --inputTopic=projects/$PROJECT_ID/topics/$TOPIC_ID \
```

```
--output=$PROJECT_ID:bq_elt_dataset.bq_user_registration \
--gcpTempLocation=gs://$BUCKET_NAME/temp/ \
--runner=DataflowRunner"
```

In the web interface, the running job is visualized as a graph of steps. The execution details, job metrics and cost can be easily monitored. There is a Job Logs panel which is very helpful for troubleshooting.

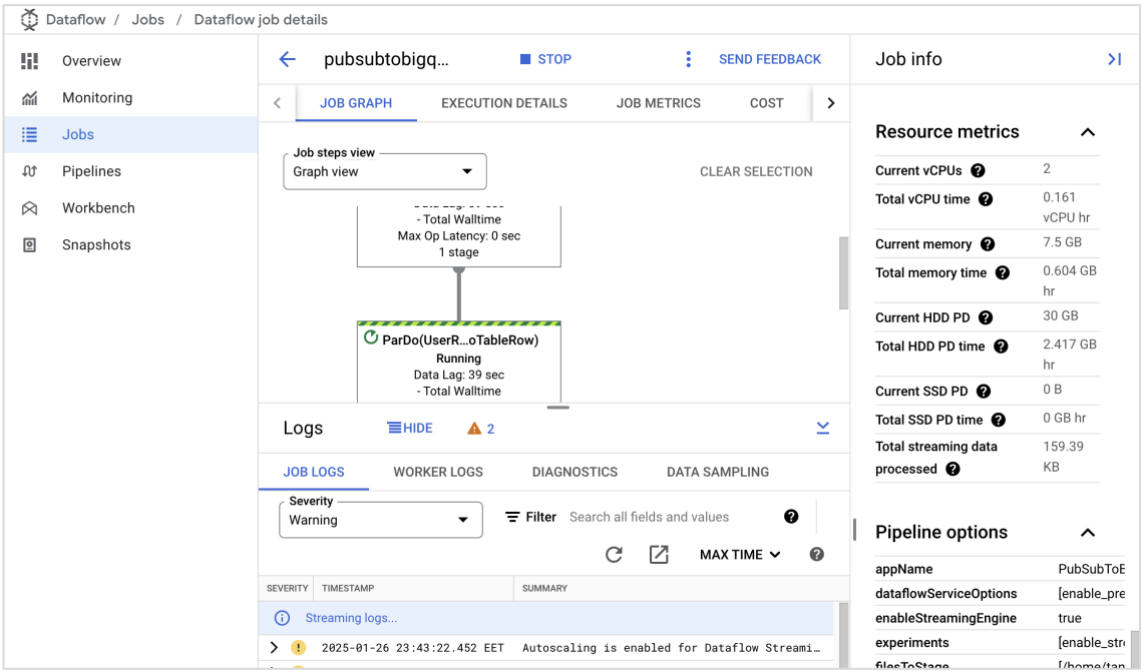

FIGURE 15. Dataflow job interface

5.6 Looker Studio

When creating a new report in Looker Studio, BigQuery can be selected as a data source via a built-in connector.

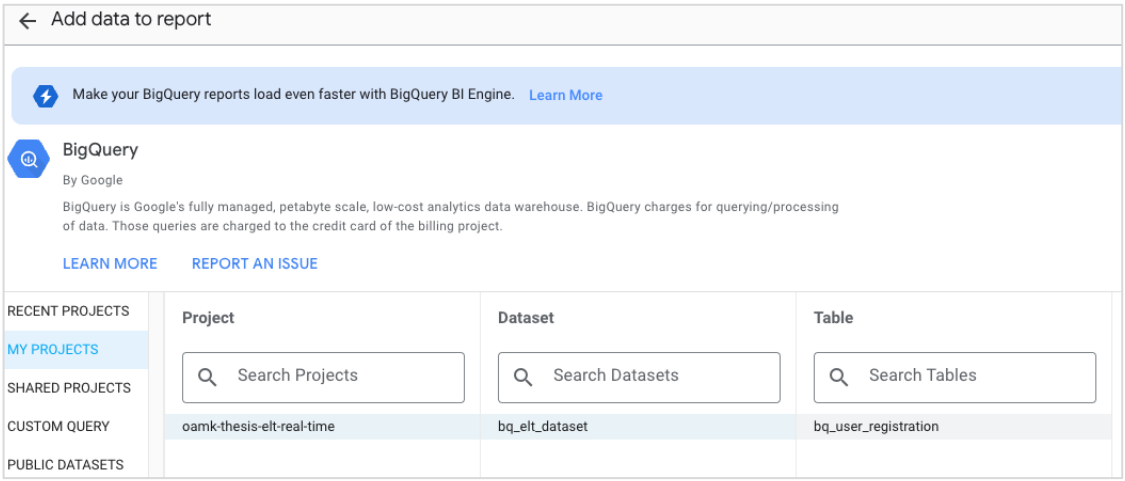

FIGURE 16. Adding BigQuery data source to Looker Studio

Managing data sources can be done via *Resource* menu. In this thesis, there are examples of creating extra logical fields. For example, to group data into age groups, based on the *birth_date* field, there is a new *birth_date_age_calc* to calculated age of users via the formula:

```
DATETIME_DIFF(CURRENT_DATE(), birth_date, YEAR)
```

Based on that field, another group field *birth_date_age_group_calc* is defined.

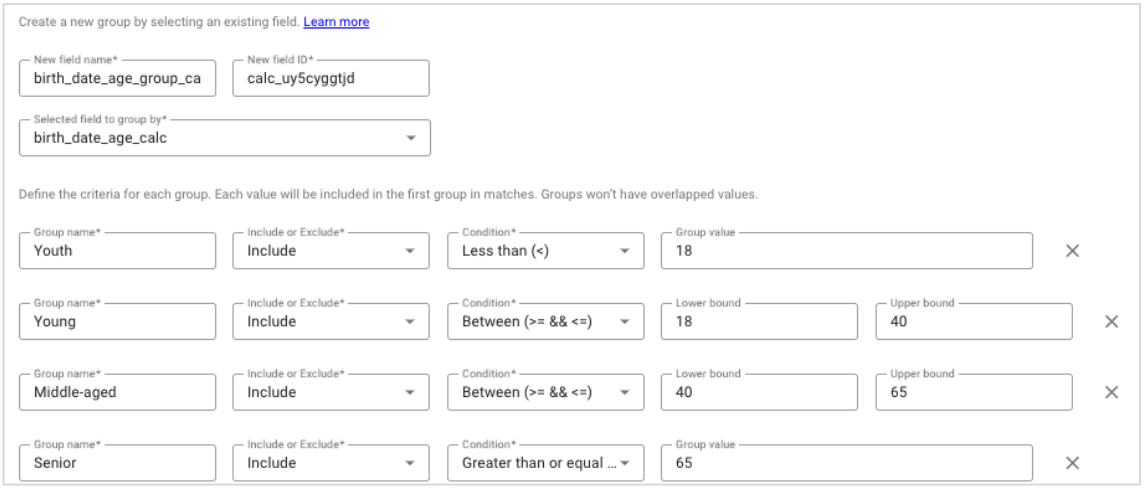

FIGURE 17. Looker Studio group field by ages

Looker Studio has most of the common chart templates, which are easy to use. The final result is an example dashboard with visualizations of the number of new users, the number of users with contact details, user proportions by genders and age groups, and the number of registrations over time and geography.

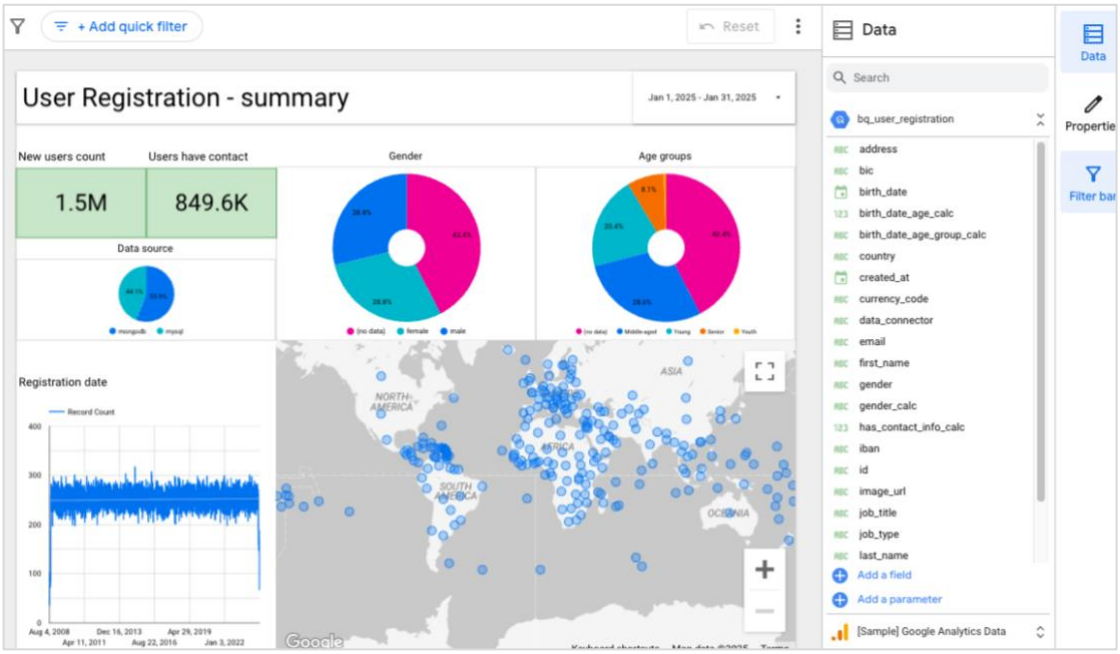

FIGURE 18. A sample Looker Studio report

6 RESULTS

This chapter presents the experiment results from different scenarios in section 4.4.1, including performance metrics measurement and explanations. An estimation on service costs is also discussed.

6.1 Performance monitoring

Performance scenarios run for 60 minutes to ensure the services have enough time to auto-scale their infrastructures for higher performance. Because the generated data is also stored in the generators' virtual machines, which have limited disk storage, it is not feasible to run the generator continuously for a longer time.

6.1.1 Scenario 1

The pipeline catches up quickly with the ingress at the rate of 2000 elements per second or 7.2 million items an hour. The Dataflow job is already running with 2 vCPUs and 8GB memory, enough to handle the throughput immediately. Data is inserted into BigQuery after a one-minute delay.

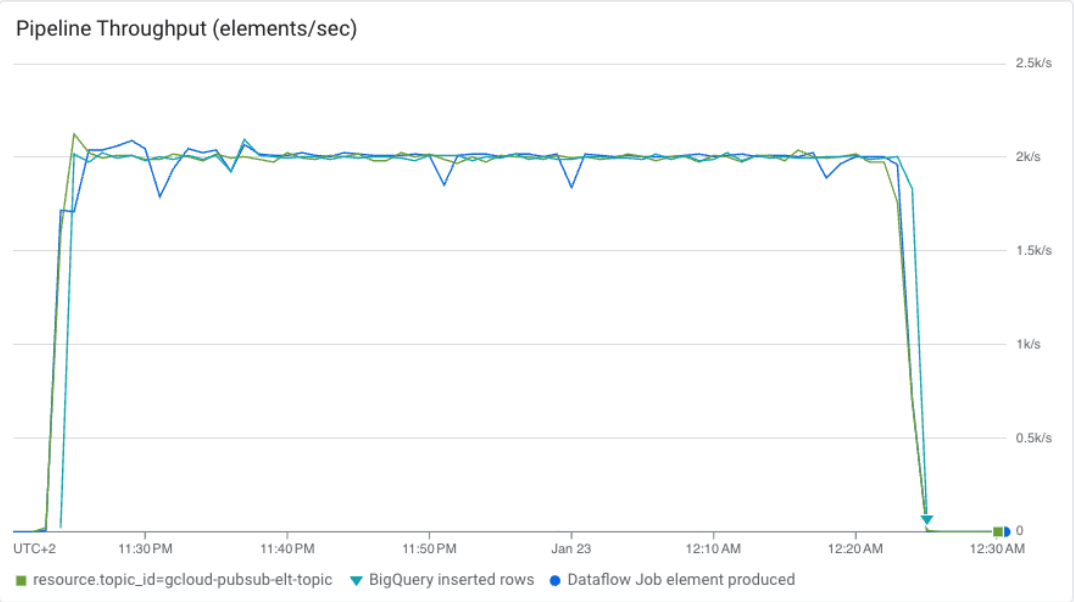

FIGURE 19. Pipeline throughput by elements/sec in Scenario 1

The measured average message size is about 1Kb.

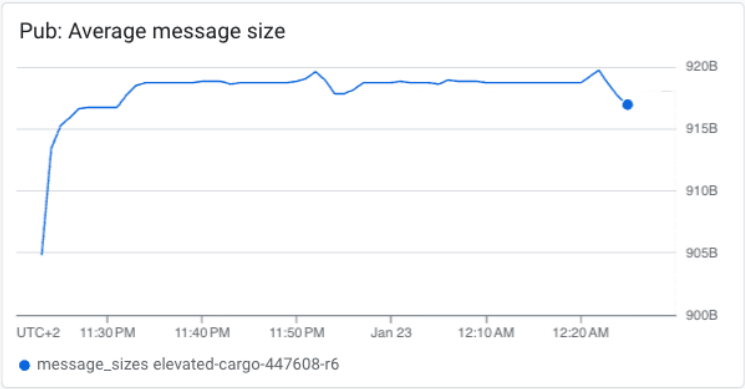

FIGURE 20. Average message size

Therefore, the calculated ingress on one topic is about 1,9Mib/s. This is close to the measured throughput at 1.7MiB/s. The job throughput varies around 1.5Mib/s, and BigQuery data persistent speed is a bit less.

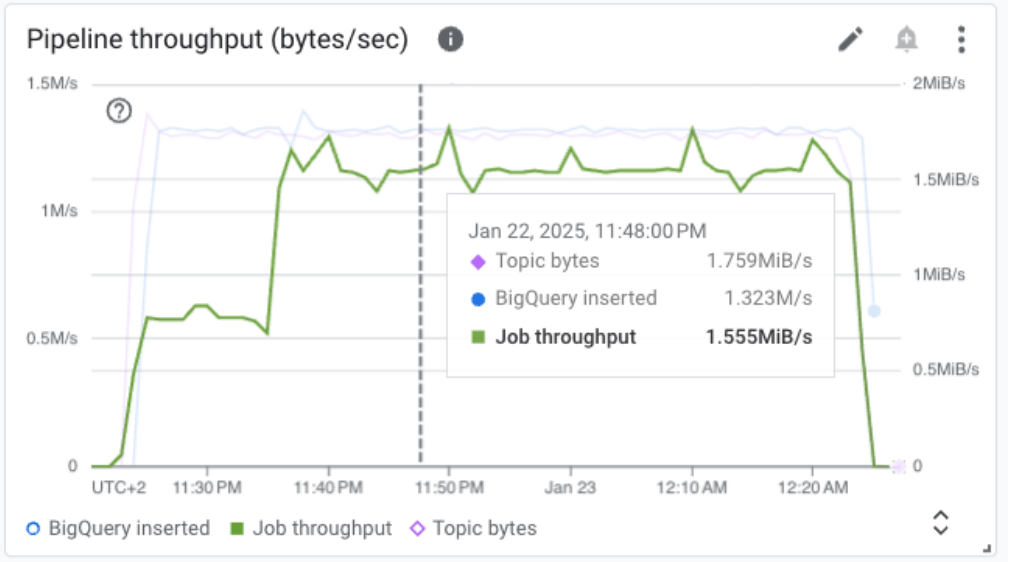

FIGURE 21. Pipeline throughput by bytes/sec in Scenario 1

Across the pipeline, there are no noticeable congestions in terms of element processing and throughput, except the Dataflow job takes about 5 minutes to warm up.

6.1.2 Scenario 2

At the rate of 2000 elements per sec on 2 topics, there are 14,4 million items to be inserted during an hour. From observations, there is a delay of about 10 minutes for Dataflow jobs to start streaming

data from two Pub/Sub topics and another 3 minutes for data to start arriving at BigQuery. Because of this delay, Dataflow jobs scaled up from 2 vCPUs to 4 vCPUs to process the queueing messages. This reveals that there are still a lot of capabilities in the pipeline to process and write data.

After the job processing catches up with the ingress, the pipeline is stable at 4000 elements per second in all three stages.

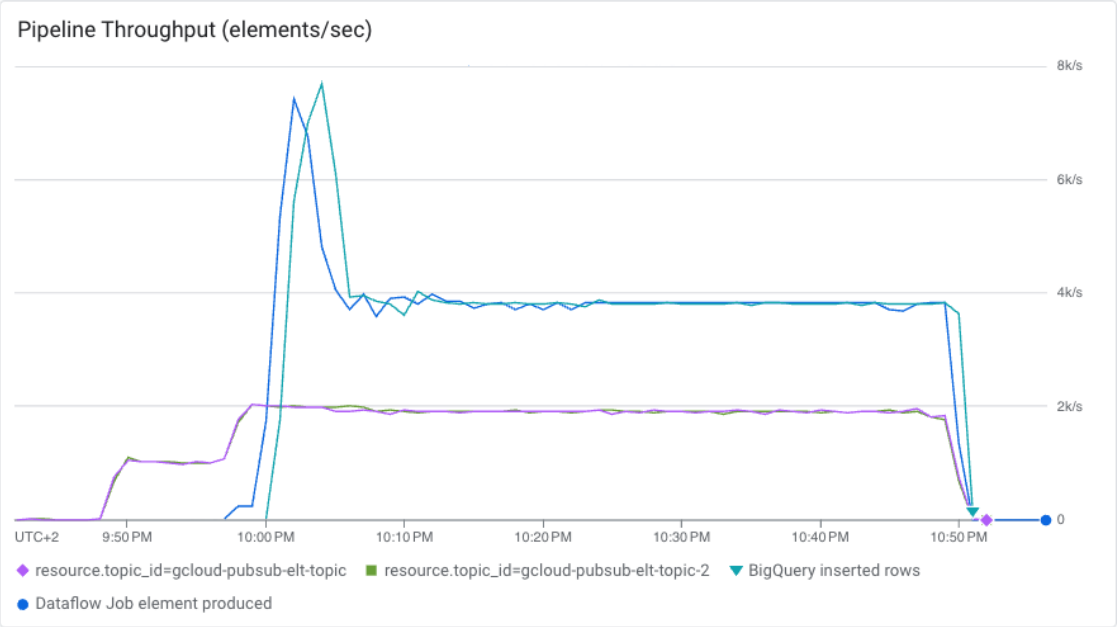

FIGURE 22. Pipeline throughput by elements/sec in Scenario 2

When data stops coming to Pub/Sub topics, all jobs finish processing at the same time and data persistent is completed shortly. The pipeline is adjusted automatically with the exact necessary resources.

As message size is about 1Kb, the calculated ingress is about 3,9Mb/s in total. This matches with the measured throughput. Comparing to scenario 1, data inserted into BigQuery is much less than the original size from the topics. This can be explained as job processing already trims off unnecessary data before sending to storage.

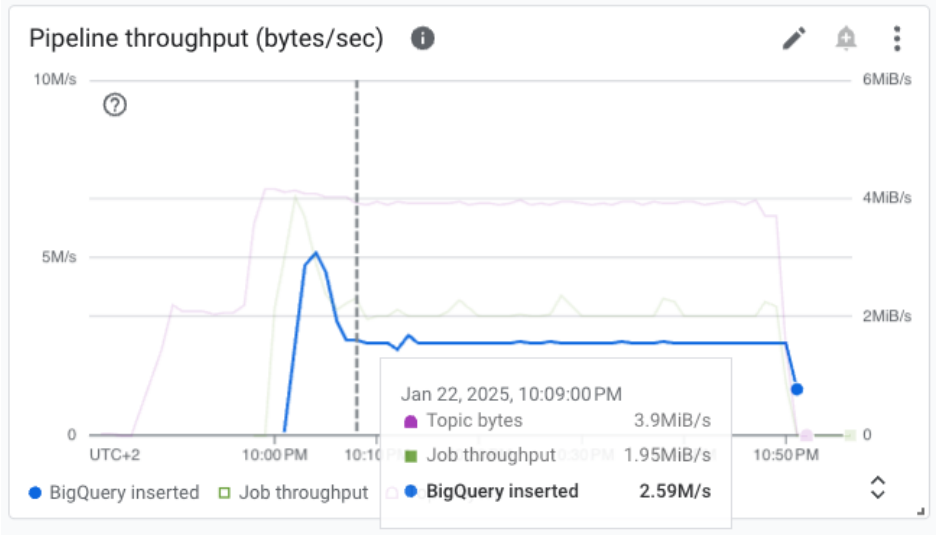

FIGURE 23. Pipeline throughput by bytes/sec in Scenario 2

6.1.3 Scenario 3

This scenario experiments the BigQuery warehouse performance by stress testing. A billion of rows were inserted into one topic by an ingress of 2000 elements per second. There are 5 cloned jobs that process and stream the data to BigQuery at the total rate of 10000 elements/second. The pipeline handles this throughput smoothly without significant congestions or fluctuations, as in the figure below. This proves that BigQuery warehouse is capable for high data rate.

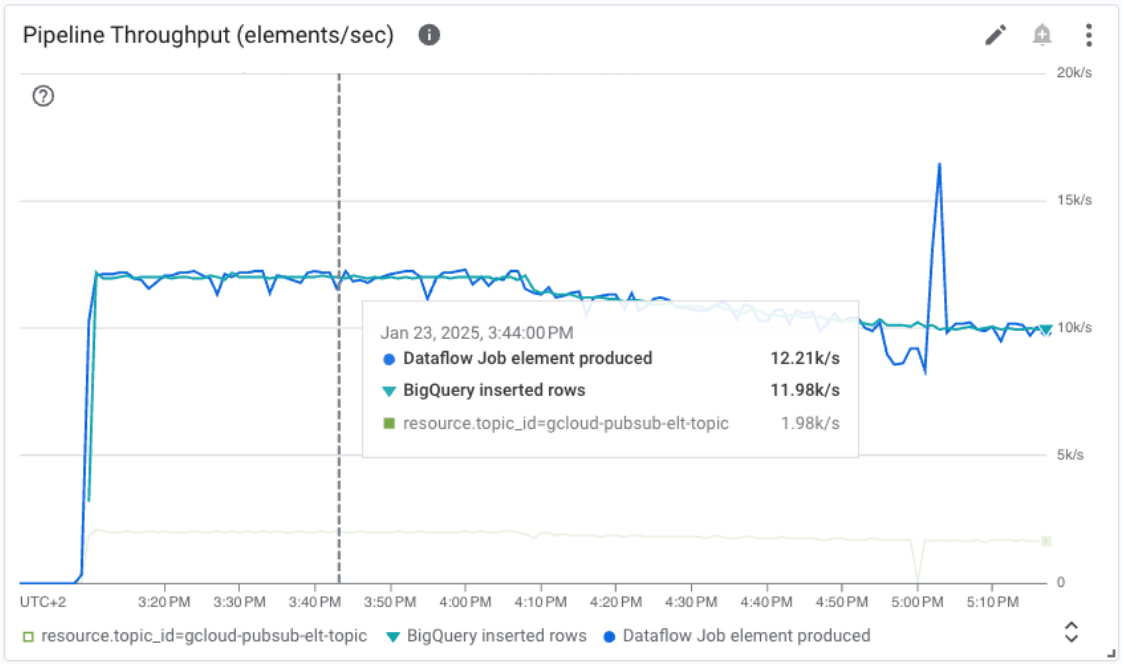

FIGURE 24. Pipeline throughput by elements/sec in Scenario 3

The complex query to be tested against the big amount of data is as below. It stresses both aggregation and grouping capabilities.

```
SELECT
    DATE(created_at) AS registration_date,
    COUNT(*) AS daily_registrations,
    SUM(CASE WHEN email IS NULL THEN 0 ELSE 0 END) AS conditional_count
FROM
    `elevated-cargo-447608-r6.bq_elt_dataset.bq_user_registration`
GROUP BY
    registration_date
ORDER BY
    registration_date;
```

At about 21,6 million of records, the execution time is 1 seconds but there is a 29 seconds to prepare computational capacity. At 375 million of records, the execution time remains the same while preparation time increases to 54 seconds.

Even though the experiment could not reach 1 billion of records because of budget limitation, it manages to generate 828 million of records, which takes 3 seconds to see the result but nearly 24 minutes for slot preparation. This number is the key factor to calculate cost in capacity pricing models.

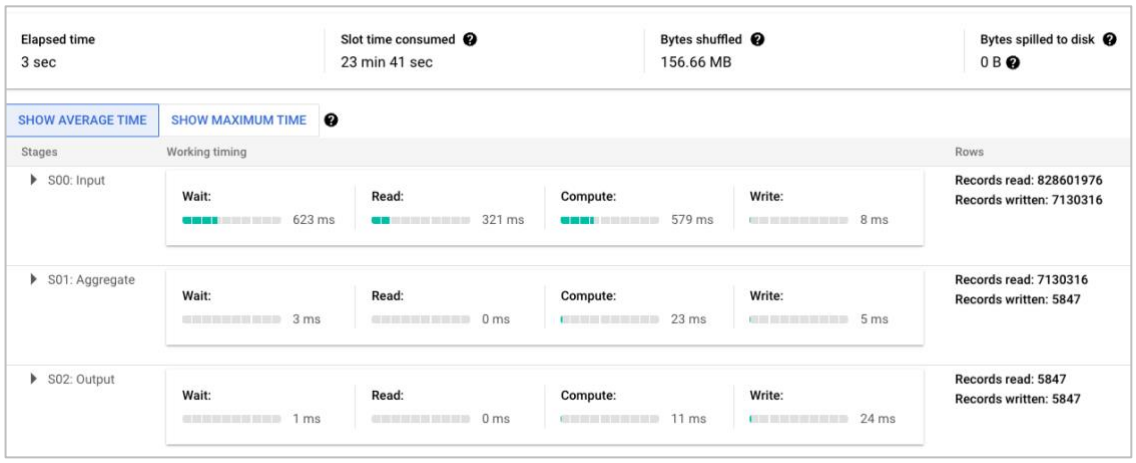

FIGURE 25. Execution time of a complex query on 800 million of records

Otherwise, in on-demand compute pricing models, cost is calculated based on the scanned bytes, which can be seen in the job information.

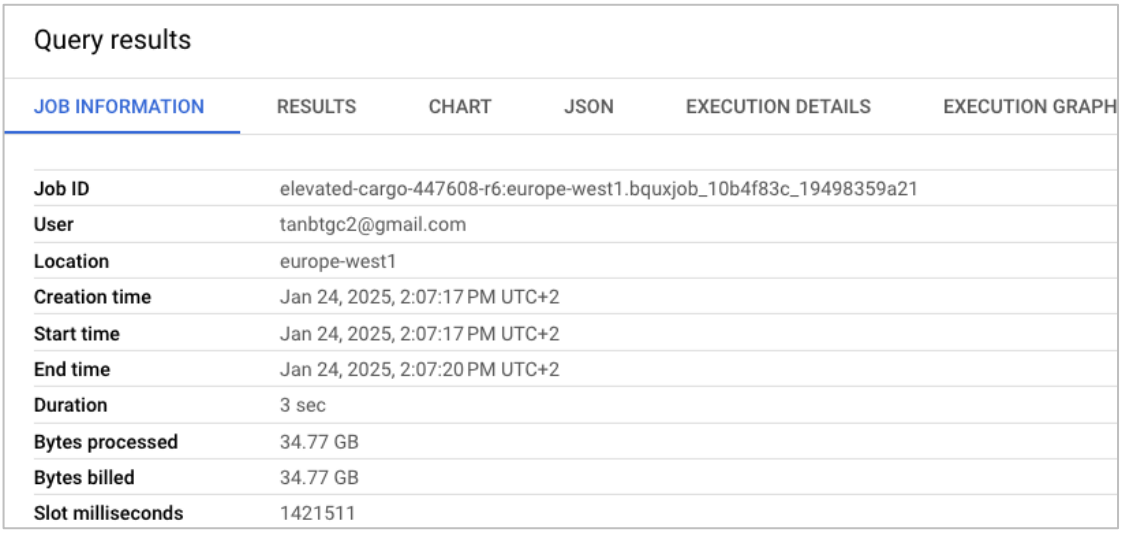

FIGURE 26. BigQuery job information

The execution graph shows detailed steps and amount of records have been processed in each step. This is helpful to optimize the query with a better logic, indexing or partition.

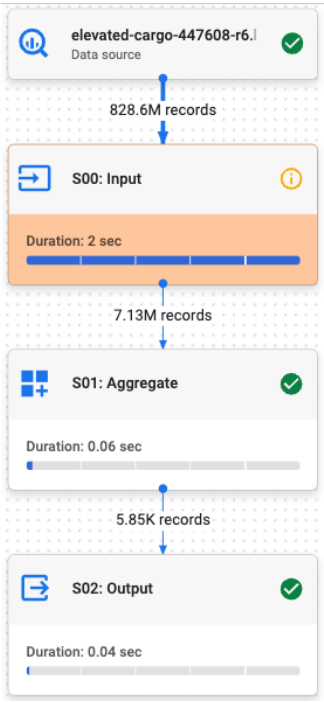

FIGURE 27. BigQuery job execution graph

6.2 Services Cost

About 500 million Twitter posts are sent daily (Shepherd 2022), equal to nearly 6000 posts per second. Assuming each post is 1Kb, the throughput is nearly 6Mb/s. The performance measurement from scenario 2 is close to this rate, with available capacities to increase. This means that the proposed architecture is efficient enough to handle high throughputs. In this Twitter case, the monthly cost for cloud services is approximately $1499, as in the breakdown below:

- Pub/Sub: assuming that all messages are acknowledged and no storage needed, the total throughput is 6Mb/sec or 15,2TiB/month which costs about $608 as price of $40/TiB.
- Dataflow: for a simple computation as in this thesis, one Dataflow job with a n1-standard-2 Compute Engine instance can handle 2000 messages per second. Therefore, the cost for 3 instances is $230/month, as $76/instance/month. The total 90Gb standard persistent disk cost $3.6 for 3 instances.
- BigQuery storage: the cost for 15,2TiB in an active logical storage is $0.02/GiB, or $311 per month. For streaming data with BigQuery API, there is a cost of $0.025/GiB and the first 2TiB per month is free. In this case it costs $338 per month.
- BigQuery computation: It is hard to estimate the data scanned by a query because it is optimized automatically. At the price of $6.25/TiB, a query that scans the whole data costs $95. However, BigQuery has a caching mechanism that reserves the previous result for incremental scanning when running the query again, thus reducing costs. For example, in scenario 3, querying 800 million records was expected to scan hundreds of gigabytes, but it actually was about 35 gigabytes. There is also a capacity pricing plan to pay by slot workloads instead of per query. An enterprise slot tier costs $0.066 per slot/hour, or $47,5/month. Depending on how complex the schema structure and the query are, it could take 300-500 slots to process a billion rows within a few minutes.

6.3 Cost optimization

By focusing on the main pricing factors, as explained in section 4.3, there are good practices to reduce the costs of Google Cloud services.

In Pub/Sub service, the throughput can be optimized by sending only the necessary data to be processed. For example, in scenario 2, bytes inserted to BigQuery are 2,59M/s, much less than

the Pub/Sub topic ingress of 3,9M/s, indicating that the incoming message size can be trimmed. It is also recommended that small messages be aggregated into larger ones to reduce the number of messages. To reduce storage costs, ensure messages in the topics are processed as soon as possible. The topics and subscribers should be located in the same region to optimize speed and network cost.

In the Dataflow service, it is mainly about optimizing computational resources. With autoscaling capabilities, the service can dynamically adjust the number of workers based on the incoming data volume. It is essential to select a suitable machine type based on the workload. For example, when the message size is large, using `n1-highmem` virtual machine is more suitable. Google Cloud has Spot VMs for non-critical processing tasks which is significantly cheaper. Besides, it is crucial to optimize the implementation of Dataflow job with efficient logics and algorithms.

To optimize BigQuery, some common practices with other database engines can be applied. For example, partitioning tables by date or relevant attributes helps reduce data scanned, where old data can be automatically deleted or moved to an appropriate storage tier such as Coldline or Archive. Query optimization can reduce computation cost significantly, such writing efficient queries, indexing and cluster on specific columns.

Thesis experiment discovers that Cloud Logging is the extra service which takes the highest proportion in the cost structure as in Figure 28. At the beginning, it is hidden because there is a free 50GiB log storage space per month, but it peaks up quickly because the Dataflow job implementation writes many logs when processing a single message.

Service	Cost	Discounts	Promotions & others	↓ Subtotal	% Change
Cloud Logging	€142.97	–	–	€142.97	↑ 1160%
Cloud Dataflow	€41.77	–	–	€41.77	↑ 210%
BigQuery	€23.81	–	–	€23.81	↑ 433%
Cloud Pub/Sub	€22.03	–	–	€22.03	↑ 471%
Compute Engine	€4.20	–	–	€4.20	↓ 12%
Networking	€0.40	–	–	€0.40	↑ 60%

FIGURE 28. Service cost structure in the experiments

In general, monitoring resource usage and cost continuously is important to identify potential fees. Setting up budget alerts is a safe guard to avoid over spending.

7 CONCLUSION

The main purpose of the thesis is to propose an architectural solution for real-time streaming data analytics, especially for software systems that need data pipeline integration on top of the existing components. Google Cloud is discussed and experimented in the thesis, as it is an enterprise platform with a variety of high-quality services, intuitive user interfaces, and flexible pricing plans. Nowadays, Google Cloud has become more competitive with Amazon AWS and Microsoft Azure due to the integration of Gemini AI for simple code generation, troubleshooting, infrastructure optimization, or data exploration.

Three services of Google Cloud Platform, including Pub/Sub, Dataflow, and BigQuery, were explained, configured, programmed, and experimented with different scenarios and a large amount of sample data. Performance and pricing metrics for each service were discussed and monitored throughout the experiments. As a result, the proposed architecture shows its capabilities of handling high throughput efficiently and reliably.

The thesis experiments are limited by a simple data structure of the generated data, which could, in fact, be complex with long computation times. The networking factor was also removed, as the existing systems are assumed to be in the same network as the cloud services infrastructures. In reality, system components could be anywhere and significantly affected by the networking situation.

Although unexpected quotas and limits were encountered while stressing the cloud services, they can be solved quickly via customer support. Also, a significant extra service charge on Cloud Logging was discovered via stress testing. Another drawback is that pricing estimation could be challenging on services such as BigQuery with on-demand pricing models; thus, it requires careful budget monitoring and alerting.

REFERENCES

Apache Software Foundation. 2020. “Apache Kafka.” Apache Kafka. 2020. https://kafka.apache.org/documentation.html#connect_overview .

Bryant, Daniel. 2018. “Migrating Batch ETL to Stream Processing: A Netflix Case Study with Kafka and Flink.” InfoQ. February 8, 2018. https://www.infoq.com/articles/netflix-migrating-stream-processing/.

Debezium Community. 2018. “Debezium Connector for MySQL: Debezium Documentation.” Debezium.io. 2018. https://debezium.io/documentation/reference/2.7/connectors/mysql.html.

Debezium Community. 2024. “Debezium Architecture.” Debezium.io. 2024. https://debezium.io/documentation/reference/stable/architecture.html.

Dehghani, Zhamak. 2022. Data Mesh. O’Reilly Media, Inc.

Densmore, James. 2021. Data Pipelines Pocket Reference. O’Reilly Media.

Docker. 2024. “Enterprise Application Container Platform | Docker.” Docker. 2024. https://www.docker.com/.

GitHub. 2024. “CI/CD: The What, Why, and How.” GitHub. GitHub. 2024. https://github.com/resources/articles/devops/ci-cd .

Google LLC. 2024. “Overview of the Pub/Sub Service.” Google Cloud. 2024. https://cloud.google.com/pubsub/docs/pubsub-basics.

Google LLC. 2025a. “Pub/Sub Quotas and Limits.” Google Cloud. 2025. https://cloud.google.com/pubsub/quotas#throughput_quota_units.

Google LLC. 2025b. “Programming Model for Apache Beam.” Google Cloud. January 17, 2025. https://cloud.google.com/dataflow/docs/concepts/beam-programming-model.

Google LLC. n.d. “Dataflow Overview: Real-Time Data Intelligence.” Google Cloud. Accessed January 13, 2025. https://cloud.google.com/dataflow/docs/overview.

googleapis. 2023. “GitHub - Googleapis/Java-Pubsub-Group-Kafka-Connector.” GitHub. May 10, 2023. https://github.com/googleapis/java-pubsub-group-kafka-connector.

Mahapatra, Anish. 2020. “The Cloud: Google Cloud Platform Made Easy.” Medium. Towards Data Science. August 25, 2020. https://towardsdatascience.com/the-cloud-google-cloud-platform-gcp-made-easy-anish-mahapatra-3d0aed3fe7fa.

Mange Ram Tyagi. 2024. “Comprehensive Guide to Real-Time Data Integration.” Adeptia. February 20, 2024. https://www.adeptia.com/blog/real-time-data-integration.

MongoDB, Inc. 2015. “Most Popular NoSQL Database.” MongoDB. 2015. https://www.mongodb.com/resources/basics/databases/nosql-explained/most-popular-nosql-database.

Netflix Technology Blog. 2022. “Data Mesh — a Data Movement and Processing Platform @ Netflix.” Medium. August 1, 2022. https://netflixtechblog.com/data-mesh-a-data-movement-and-processing-platform-netflix-1288bcab2873.

Noel Yuhanna. 2024. “The Forrester WaveTM: Data Lakehouses, Q2 2024.” Forrester.com. Forrester Research, Inc. April 29, 2024. https://reprints2.forrester.com/#/assets/2/157/RES180732/report.

Psaltis, Andrew. 2017. Streaming Data. Simon and Schuster.

Reis, Joe, and Matt Housley. 2022. Fundamentals of Data Engineering. “O’Reilly Media, Inc.”

Shepherd, Jack. 2022. “22 Essential Twitter Statistics You Need to Know in 2022.” The Social Shepherd. February 16, 2022. https://thesocialshepherd.com/blog/twitter-statistics.

Uber Technologies Inc. 2021. “Uber’s Real-Time Data Intelligence Platform at Scale: Improving Gairos Scalability/Reliability.” Uber Blog. January 19, 2021. https://www.uber.com/en-IN/blog/gairos-scalability/.

The main class `PubSubToBigQueryPipelineForUserRegistration.java`:

```
package org.example;

import com.google.api.services.bigquery.model.TableFieldSchema;
import com.google.api.services.bigquery.model.TableRow;
import com.google.api.services.bigquery.model.TableSchema;
import org.apache.beam.sdk.Pipeline;
import org.apache.beam.sdk.io.gcp.bigquery.BigQueryIO;
import org.apache.beam.sdk.io.gcp.pubsub.PubsubIO;
import org.apache.beam.sdk.options.PipelineOptionsFactory;
import org.apache.beam.sdk.transforms.DoFn;
import org.apache.beam.sdk.transforms.ParDo;

public class PubSubToBigQueryPipelineForUserRegistration {
 public static void main(String[] args) {
  PipelineOptionsFactory.register(PubSubToBigQueryPipelineOptions.class);
  PubSubToBigQueryPipelineOptions options = PipelineOptionsFactory.fromArgs(args)
    .withValidation()
    .as(PubSubToBigQueryPipelineOptions.class);

  Pipeline pipeline = Pipeline.create(options);

  pipeline
   .apply("ReadFromPubSub", PubsubIO.readStrings().fromSubscription(options.getInputSubscription()))
   .apply("TransformToBigQueryRow", ParDo.of(new DoFn<String, TableRow>() {
     @ProcessElement
     public void processElement(@Element String message, OutputReceiver<TableRow> out) {
       String jsonString = message;
       UserRegistration element = CommonUtil.getUserRegistration(jsonString);
       TableRow row = new TableRow();
       row.set("data_connector", kMessage.source.connector);
       row.set("id", element.getId());
       row.set("email", element.email);
       row.set("password", element.password);
       row.set("first_name", element.first_name);
       row.set("last_name", element.last_name);
       row.set("gender", element.gender);
       row.set("birth_date", element.getBirthDate());
       row.set("image_url", element.image_url);
       row.set("phone", element.phone);
       row.set("address", element.address);
       row.set("country", element.country);
       row.set("job_type", element.job_type);
```

```
        row.set("job_title", element.job_title);
        row.set("iban", element.iban);
        row.set("bic", element.bic);
        row.set("currency_code", element.currency_code);
        row.set("created_at", element.getCreatedAt());
        out.output(row);
      }
    }))
    .apply("WriteToBigQuery", BigQueryIO.writeTableRows()
      .to(options.getOutputTable())
      .withSchema(UserRegistrationTableSchema.defineSchema())
      .withWriteDisposition(BigQueryIO.Write.WriteDisposition.WRITE_APPEND)
      .withCreateDisposition(BigQueryIO.Write.CreateDisposition.CREATE_IF_NEEDED));

  pipeline.run().waitUntilFinish();
}

static class UserRegistrationTableSchema {
  public static TableSchema defineSchema() {
    TableSchema schema = new TableSchema();
    schema.setFields(java.util.Arrays.asList(
      createField("data_connector", "STRING"),
      createField("id", "STRING"),
      createField("email", "STRING"),
      createField("password", "STRING"),
      createField("firstName", "STRING"),
      createField("lastName", "STRING"),
      createField("gender", "STRING"),
      createField("birthDate", "TIMESTAMP"),
      createField("imageUrl", "STRING"),
      createField("phone", "STRING"),
      createField("address", "STRING"),
      createField("country", "STRING"),
      createField("jobType", "STRING"),
      createField("jobTitle", "STRING"),
      createField("iban", "STRING"),
      createField("bic", "STRING"),
      createField("currencyCode", "STRING"),
      createField("createdAt", "TIMESTAMP")
    ));
    return schema;
  }

  private static TableFieldSchema createField(String fieldName, String fieldType) {
    TableFieldSchema field = new TableFieldSchema();
```

```
    field.setName(fieldName);
    field.setType(fieldType);
    field.setMode("NULLABLE");
    return field;
   }
  }
}
```

The utility class `CommonUtil.java`:

```
package org.example;

import com.google.gson.Gson;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;

import java.time.Instant;
import java.util.Date;

public class CommonUtil {
  public static UserRegistration getUserRegistration(String kafkaMessageBody) {
    Gson gson = new Gson();
    KafkaConnectorMessage kMsg = gson.fromJson(kafkaMessageBody, KafkaConnectorMessage.class);
    String afterJsonString = getStringFromJsonElement(kMsg.after).replace("\\\\", "").replace("\\\"", "\"");
    return gson.fromJson(afterJsonString, UserRegistration.class);
  }

  private static String getStringFromJsonElement(JsonElement element) {
    if (element.isJsonPrimitive()) {
      return element.getAsString();
    } else if (element.isJsonObject()) {
      JsonObject jsonObject = element.getAsJsonObject();
      return jsonObject.toString();
    }
    return element.toString();
  }

  private static Date fromISOTimestamp(String timestamp) {
    // ISO 8601  "2024-07-28T20:18:24Z"
    Instant instant = Instant.parse(timestamp);
    return Date.from(instant);
  }

  public static class UserRegistration {
    private Long id;
```

```
  private JsonElement _id;
  public String email;
  public String password;
  public String first_name;
  public String last_name;

  public String gender;
  private JsonElement birth_date;
  public String image_url;

  public String phone;
  public String address;
  public String country;

  public String job_type;
  public String job_title;

  public String iban;
  public String bic;
  public String currency_code;

  private JsonElement created_at;

  public String getId() {
    if (_id == null) {
      return String.valueOf(id);
    }
    JsonObject jsonObject = _id.getAsJsonObject();
    return jsonObject.get("$oid").getAsString();
  }

  public Date getBirthDate() {
    return birth_date.isJsonNull() ? null : fromJsonElementUnixTimeStamp(birth_date);
  }

  public Date getCreatedAt() {
    return fromJsonElementUnixTimeStamp(created_at);
  }

  private Date fromJsonElementUnixTimeStamp(JsonElement e) {
    if (e.isJsonPrimitive()) {
      return fromISOTimestamp(e.getAsString());
    } else if (e.isJsonObject()) {
      JsonObject jsonObject = e.getAsJsonObject();
      return fromISOTimestamp(jsonObject.get("$date").getAsString());
```

```
    }
    return null;
  }
 }

 public static class KafkaConnectorMessage {
  public JsonElement after;
 }
}
```