Politecnico di Torino

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Anno Accademico 2020/2021

Design and implementation of a real time data lake in cloud

Relatore:
Prof.ssa Tania CERQUITELLI

Candidato:
Vincenzo SICILIANI

Contents

List of Figures

1 Introduction

1.1 Structure of the thesis

The following thesis will be based on my work experience in developing a new architecture to *build a Data lake with Big Data technologies*. It is a project carried out by **NTT Data Italia** for one of its clients in the media sector. The goal of this project was to guide the client in his evolving technology for the management of their data by moving from an architecture based on different data warehouses to a single data lake that centralized all data.

In this thesis we will analyze all the phases of the project, from the collection of requirements to the production and management of the business as usual. The thesis will be structured in the following chapters:

1. **Data model of the Data lake**: in this chapter we will analyze how the data model at the base of the entire data lake was designed, indicating for each layer of it which are its characteristics, the type of data saved and for which use cases is addressed. To highlight in a simple way how the data is modeled in the passage from one layer to another, samples of the tables belonging to each layer will be used.

2. **Platform architecture**: in this chapter, after understanding the design of the data model, we will analyze the software architecture that will allow its realization. We will see what are the requirements to which each component must comply and how they will communicate with each other. In addition, we will investigate how this new architecture will communicate with all the other systems already in use at the client (therefore source and target systems for the data lake) and how the client's business users will be able to access the data for their analysis and reporting activities.

3. **Implementation on Google Cloud Platform**: in this chapter we will discuss how the architecture presented above will be implemented on the Google Cloud Platform. We will analyze the tools made available by GCP and how they can be functional in our project.

4. **Software components**: after studying the design of the components to be deployed on the cloud, we will analyze the software components

developed, focusing on the development methodologies adopted to meet the requirements imposed by the client and the architecture.

5. **Operation and monitoring**: in this last chapter we will study all the aspects related to the deployment of the components and their configuration in the various development, test and production environments with continuous improvement and continuous delivery methods based on Git. In addition to the deployment, in this chapter we will see how the platform is monitored in all its components to promptly detect any problems in data acquisition or processing.

1.2 AS-IS customer architecture

The AS-IS client's data management architecture is distributed in a series of **data warehouses** belonging to the different departments of the company. Each of these DWHs uses different technology and data standardizations which makes communication between them very complicated. In addition to the databases of the various departments, there are also two CRM[1] systems: one legacy based on Oracle technology and the new one based on the Salesforce product[2] which do not communicate with each other.

At the moment, the client in order to create reports with cross data from the various databases has created a system for exporting the data from one system to another one based on Control-m[3]. This solution has several critical issues because the system that exports the data must generate it in a certain time limit, i.e. before the Control-M workflow starts, so that the copy of the data to the target system is successfully completed. Plus it's an extra component to manage and control.

A final problem of the current data organization is the *total lack of support for a self analytics system.* Data scientist or analyst does not have a database in which they have all the data to be analyzed and immediately available, but they should navigate the different databases to make them analysis, and then

[1]CRM: Customer Relationship Management is a strategy for managing all the relationships and interactions of a company that take place with potential and existing customers.

[2]Salesforce: it is an entirely cloud-based CRM solution (www.salesforce.com)

[3]Control-M: orchestrator of a workflow for on-premis or cloud systems (https://www.bmc.com/it-solutions/control-m.html)

to create them reports they must request the creation of one new Control-M export flow. This obviously slows down the analysis and development of new reports considerably and makes it impossible to create Machine Learning models that require the availability of data in real time.

1.3 Client requirements

The customer's request for this project is to create a **data lake system** that collects the data present in the various DWH and CRM of the company in order to better analyze the data available. Data Lake is a type of data repository that can store large and varied raw data sets in their **native format**. "Raw data" means data that has not yet been processed for a specific purpose. *Data scientists* can access raw data while using advanced analytics or predictive modeling tools.
With data lakes, no data is removed or filtered before it is stored. Unlike when data is processed according to its specific purpose, in the case of data lakes there are no time constraints or restrictions on data analysis, which can be used multiple times. Data Lakes allow users to access and analyze data where and as it is, without having to move it to another system.
Data lakes must have governance and require ongoing maintenance to make data usable and accessible. Without all this, the data would risk becoming inaccessible, cumbersome, expensive and therefore useless.

Using this new data lake, business and data science users must be able to continue to carry out the analyzes they performed on the old architecture but at the same time must be enabled for new types of developments that require data replicated from the various source systems in **real time**.
It must also be possible to create a self analytics system in which business users can navigate, join and analyze the data in a totally autonomous way without having to worry about the underlying system that generated that data.

To make all this achievable, the data lake must be able to *replicate data from source systems* with very low latency and it must standardize the received data. In fact, one of the biggest problems with the old architecture is the total lack of standardization to represent data of the same type or with the same value. The most striking example is the management of dates: each system uses its own format and a different timezone (between UTC and

Europe/Rome) which makes the time-based join of the data complex.

Another client requirement for the data lake is the system **scalability** and **configurability**. In fact, it must be possible to add the replica of a new data structure or a new source system by simply updating or inserting a new configuration on the data lake components without having to modify anything in terms of code or hardware infrastructure. To implement this requirement, the simplest and most efficient way is to use **cloud computing** solutions in which it is possible to have maximum flexibility on the available computing and storage resources depending on the load to which the system is subjected.

A final request is to use they partners platforms and tool or software on which they have skilled employee. The tools/platforms to be used indicated by the client are the following:

- **Google Cloud Platform** as a cloud computing platform;
- **Collibra** as a tool for managing Data Governance and Data Lineage;
- **Tableau** as a visualization and reporting tool.

1.4 NTT Data Italia

NTT Data

Global IT Innovator

Figure 1: NTT Data Italia's logo

Ntt Data Italia is a company of the NTT (Nippon Telegraph and Telephone) group, one of the largest companies in the ICT field worldwide with offices in 88 countries located in 5 geographic regions for a total of over 310,000 employees and an annual revenue of 109 billion dollars. Ntt Data Italia is part of the EMEA region of the NTT Data division which was ranked eighth in the Gartner ranking for the most popular service providers in the world.

There are 8 offices in Italy with over 3800 employees. The reference sectors are the following:

- **Banking**: solutions for the digital change of customers towards the Open banking business;
- **Media**: shares the challenges of digital transformation and product innovation with the main operators in the sector;
- **Manufacturing**: thanks to skills in the field of Smart Product, Smart Factory Smart Client Relationship we accompany companies in the digital transformation;
- **Insurance**: innovative solutions to support customers in the insurance market;
- **Public Sector**: supports public sector customers with suitable solutions to ride the wave of innovation;
- **Energy & Utilities**: creates innovative solutions for the growth of the sector;
- **Telecommunication**: the distinctive experience in European Telco and the Group's best practices make us a unique player in Italy;
- **Retail**: thanks to dedicated solutions, we help retailers to overcome current market challenges and prepare for future ones.

The company's expertise help customers to implement and maintain projects in all its phases from the analysis of the requirements and requests of the user, design of the architecture according to the most innovative standards of the moment, technical realization, definition of business processes on the product and operation for its maintenance and finally research and definition of new requirements to extract further added value.

This thesis was carried out on a project for a client in the Media sector of the **Data Intelligence** business line and in particular by the **Big Data team** to which I belong which aims to help its customers in improving the processes for the definition of business decisions and strategies. To make all this possible, we place data at the center, with their peculiarities of *Volume*, *Velocity* and *Variety*, around the new technological frontiers in continuous evolution capable of optimizing the transformation cycle, from "Data" to Information, Forecasting, for become Decision and Action necessary to generate value. Distributed, in-memory and streaming processing techniques allow you to create solutions, even in real-time, to expand the business offers of companies.

2 Data model of the Data lake

In this chapter we will analyze the data model used in the implementation of the data lake, what are the different layers created and what are the purposes and constraints of each layer.

The layers that have been designed for this platform are 4: **raw**, **timeseries**, **snapshot** and **enriched**. Each layer differs from the others in the latency with which it is updated, the amount of information contained and the type of model and transformations applied. This is because the platform wants to be able to respond in an efficient and performing way to different types of use cases, from those that require data in **real time/near real time** to others that require an **historical data**. In the first category of use case cases we have, for example, the analysis of the customer's product sales in real time or machine learning models for recommendation systems, instead, in the second category we have reports created using classic BI tools or machine learning models for the prevention of churns.

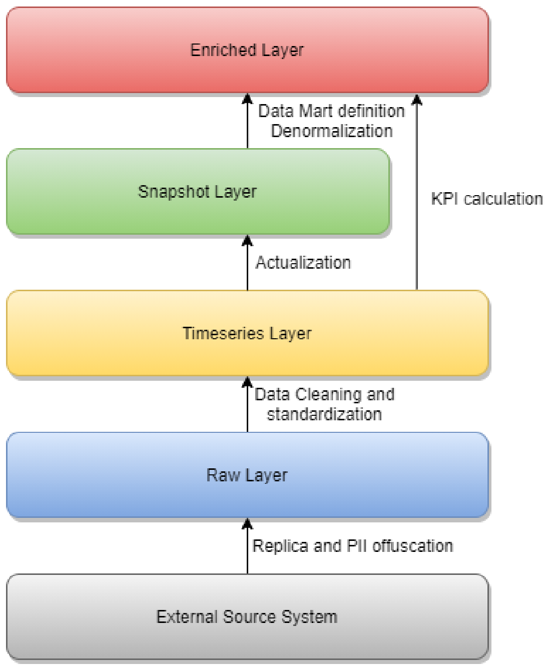


Figure 2: Data model layers of Data Lake

Each of these layers is used as the calculation basis for the next layer. The *chain* provides that the data generated by the source systems land inside the raw layer, on these data normalizations or filters are applied and are saved in the structures of the timeseries layer. Starting from these normalized data, the hourly/daily photos of the structures present in the snapshot layer are generated and finally these data are joined to create the denormalized structures of the enriched layer or the business report.

In the next paragraphs we will analyze the four layers in details. As an example case we will consider the tables replicated by the source systems represented in the following E-R diagram:

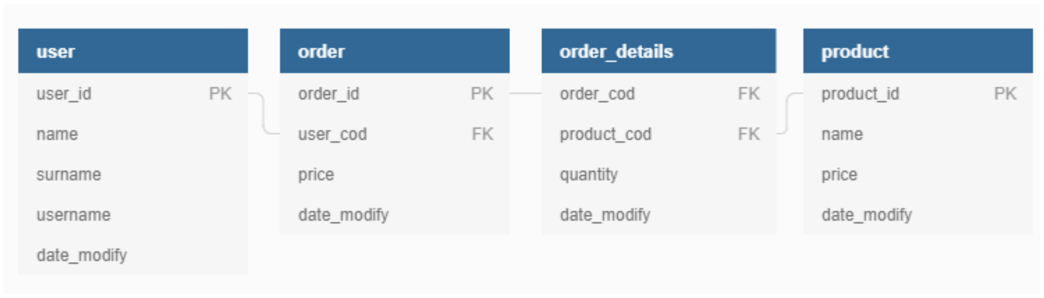


Figure 3: ER diagram of replicated structures for the example use case

2.1 Raw Layer

The first layer that we found in the data lake platform is the one that takes the name of **raw layer**. The name derives from the fact that the data stored at this level is exactly that we receive from the source systems, without any normalization applied.
The purpose of the data structures present in this layer is to make the data available with the *minimum latency* possible from the instant when the upstream systems insert or update a data. Furthermore, using these structures we are always able to have all the data produced by the sources available, which makes possible any recalculations necessary for some remediation or bug fixes in subsequent layers.

The most common use cases in which to use these tables are two: as input data for machine learning models or simply to explore data imported from source systems.
In the first case we are talking, for example, of machine learning models that require processing data in *real time* or in any case with the minimum possible latency because the output of these models generates a result for the end user of the products of the our client.
In the second case, instead, the analyst or data governance teams can analyze

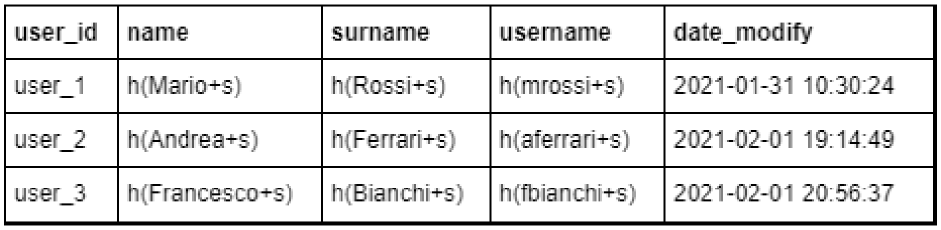
the data imported from the various source systems to identify the normalizations to be applied to the data or identify any pre-processing operations necessary to clean the data.

Each record saved in the structures at this level represents an insert or update of data in the source system and these records have two timestamps. The first is called *date_modify* and represents the moment in which the data was created/updated by the source system, instead, the second is identified with the name of *ingestion_time* which represents the instant of time in which the data is ingested from the data lake.

As previously said, the records are exactly like those are generated by the source system, with the exception of **sensitive data** (PII, Personal Identificable Information) that are all the data that allow to trace the user who performed the operation that generated that record (i.e. name, surname, date of birth, ...). In fact, a hash function (with salt) is applied to these data which allows to mask the real value contained in the fields. The plain value/hashed value pair is saved in a separate structure (with very few access permission) to allow the unmasking of the encrypted value requested when the data is sent to external systems.

Below an example of the data contained in the raw layer for the use case presented in the introduction of this chapter:

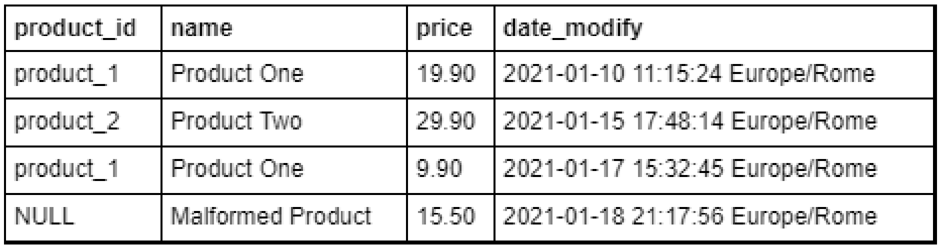
User Raw table:



user_id	name	surname	username	date_modify
user_1	h(Mario+s)	h(Rossi+s)	h(mrossi+s)	2021-01-31 10:30:24
user_2	h(Andrea+s)	h(Ferrari+s)	h(aferrari+s)	2021-02-01 19:14:49
user_3	h(Francesco+s)	h(Bianchi+s)	h(fbianchi+s)	2021-02-01 20:56:37

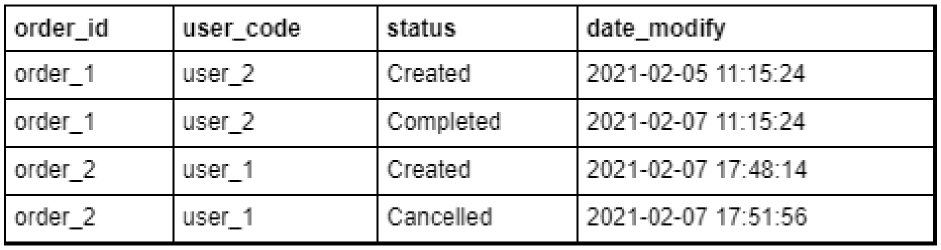
h: funzione di hash, s:sale aggiunto al valore

Product Raw table:



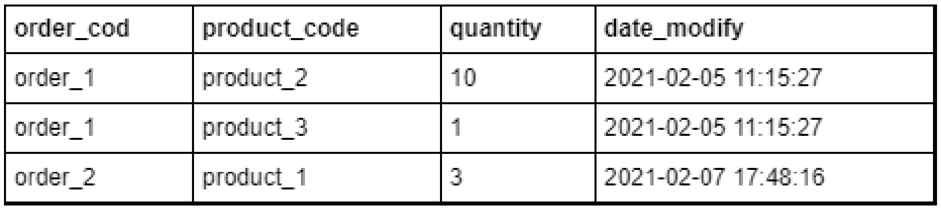
product_id	name	price	date_modify
product_1	Product One	19.90	2021-01-10 11:15:24 Europe/Rome
product_2	Product Two	29.90	2021-01-15 17:48:14 Europe/Rome
product_1	Product One	9.90	2021-01-17 15:32:45 Europe/Rome
NULL	Malformed Product	15.50	2021-01-18 21:17:56 Europe/Rome

Order Raw table:



order_id	user_code	status	date_modify
order_1	user_2	Created	2021-02-05 11:15:24
order_1	user_2	Completed	2021-02-07 11:15:24
order_2	user_1	Created	2021-02-07 17:48:14
order_2	user_1	Cancelled	2021-02-07 17:51:56

Order Details Raw table:



order_cod	product_code	quantity	date_modify
order_1	product_2	10	2021-02-05 11:15:27
order_1	product_3	1	2021-02-05 11:15:27
order_2	product_1	3	2021-02-07 17:48:16

2.2 Timeseries layer

The second layer of the data model is what is called **timeseries layer**. This layer shows all the data read from the source systems but unlike the previous one the data are **filtered** and **normalized**.
The filters applied are used to *clean the data* by excluding all those records that have key fields at NULL or values not included in the expected domain (for example a string in a number field or a date with a format is not expected). All these discarded records are collected in a Dead Letter Queue (DLQ) table so they can be analyzed and possibly recovered later.

The normalizations applied to the data have the purpose to **standardize** all the possible values received in different formats from each of the source systems. An example of normalizations are those applied on fields that contain timestamps, in fact, each source system sends data according to its own format and its own timezone (generally UTC or Europe/Rome). These fields before being saved in the timeseries structures are all standardized in the ISO8601[4] format with UTC timezone. This standardization is essential in order to correctly analyze and join data from different sources.

Not all fields present in the RAW structures are replicated on this layer, but only those that are already been analyzed and **approved** by the data governance team. In fact, if for example an information (not key of the table) is ingested by several sources, only one of these will arrive on the timeseries layer in order to avoid possible duplication of data. In this layer the names of the fields are also changed, in order to apply a standardization to the various structures coming from different sources. For example, if on different structures we have the field representing the unique key of a product called in different ways (id, id_product, cod_product) this field on all the structures of the timeseries layer will be named id_product.

A final difference with the RAW layer is the data refresh rate. In fact, in order to apply these normalizations (and respect some quotas imposed by the cloud provider used) the data are not saved on the timeseries structures in real time but with **microbatches** started every 10 minutes.

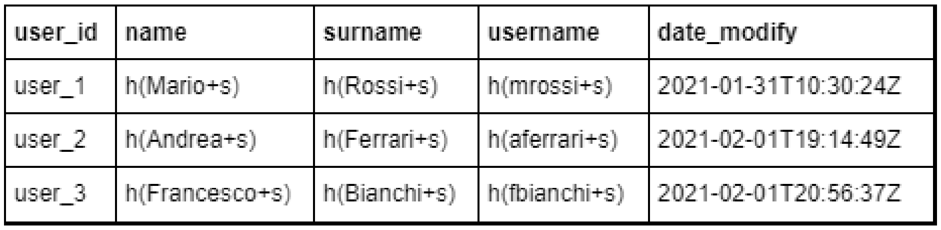
The goal of these structures is to be used by the business users to make trend analyzes on their products or customers that require all the movements and changes of status of an entity. For example, analyze how much time elapses between the creation of an order by a customer, the acceptance of this order by an operator and the closing of the order.
On this layer the business is also enabled to perform self analytics operations on the data because these data are already cleaned and normalized and therefore no further data processing operations are needed.

Below an example of the data contained in the timeseries layer for the use case presented in the introduction of this chapter (for simplicity, only the

[4]Standard ISO 8601: https://www.iso.org/standard/40469.html

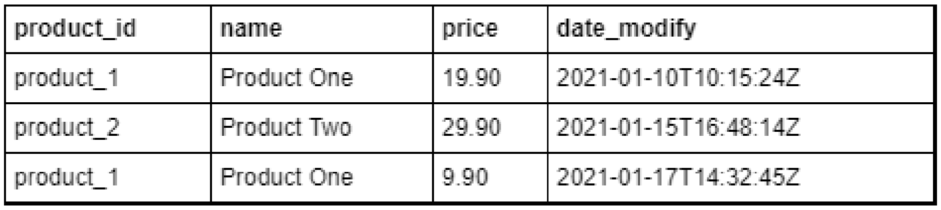
User and Product tables will be shown as the other structures would be a repetition):

User Timeseries:



user_id	name	surname	username	date_modify
user_1	h(Mario+s)	h(Rossi+s)	h(mrossi+s)	2021-01-31T10:30:24Z
user_2	h(Andrea+s)	h(Ferrari+s)	h(aferrari+s)	2021-02-01T19:14:49Z
user_3	h(Francesco+s)	h(Bianchi+s)	h(fbianchi+s)	2021-02-01T20:56:37Z

Product Timeseries:



product_id	name	price	date_modify
product_1	Product One	19.90	2021-01-10T10:15:24Z
product_2	Product Two	29.90	2021-01-15T16:48:14Z
product_1	Product One	9.90	2021-01-17T14:32:45Z

From the example we can see the change of the name of the fields, for example user_id is mapped in id_user, the normalization of the dates in the ISO 8601 format by transforming the received value into the corresponding UTC value and the filter on malformed data with the rejection of the last record of the Product RAW structure because the key is set to NULL.

2.3 Snapshot Layer

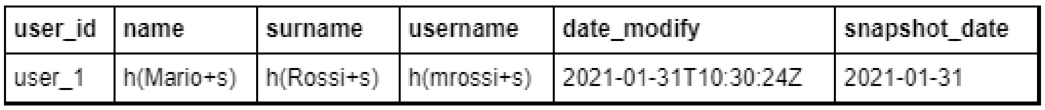
Using the timeseries layer like a base, the structures contained in the **snapshot** layer are calculated. The goal of the structures in this layer is to historicize the last state of a given entity *day by day* (or hour by hour in some cases). These structures are useful for the analyzes that do not require all the movements intra-day of a given entity but only the status associated with the entity for a given day by pre-calculating this information with a single run of the query.

With these structures it is therefore possible to answer in a simple and very fast way to some classic questions of the business users, such as the number of active customer base on a given day, or the number of orders placed in a day. Having also the entire history of these informations, it is also possible to calculate **daily trends** of these metrics (for example how many customers we have today compared to 15 or 30 days ago). Obviously, for how these tables are constructed, they will not be useful when we need data updated in real time or with low latency or when we need all the variations done by an entity.

In this layer we have no changes undergone by the data but simply a **crushing** of them based on a unique key of the table. Even at the level of name of the fields we have no variations. The only addition in this layer is a date field called *snapshot_date* (or *snapshot_hour*) which is used to indicate on which day/hour that record is valid. So to have a complete photo of the table for a given day just apply a filter on this time field. Instead, if we want to analyze the trend of a specific instance of the entity being analyzed, we just need to filter for the key field of the structure, obtaining as a result the status of that instance day by day (example filter for the id_order field on the snapshot of orders for get the status of that particular order day by day).

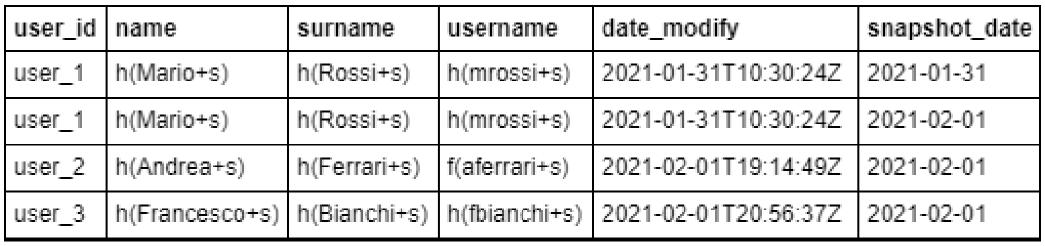
Below an example of the data contained in the snapshot layer for the use case presented in the introduction of this chapter. In this case, the temporal evolution of the two structures seen previously will be shown, the *User* table in which there are no changes made to the information of the various users and the *Order* in which instead we have for the same order of the state changes.

User Snapshot at 2021-01-31:



user_id	name	surname	username	date_modify	snapshot_date
user_1	h(Mario+s)	h(Rossi+s)	h(mrossi+s)	2021-01-31T10:30:24Z	2021-01-31

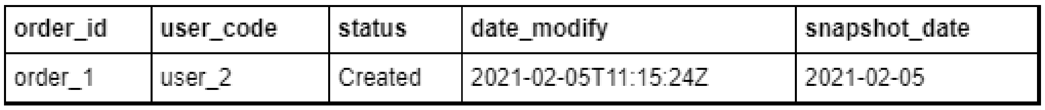
User Snapshot at 2021-02-01:



user_id	name	surname	username	date_modify	snapshot_date
user_1	h(Mario+s)	h(Rossi+s)	h(mrossi+s)	2021-01-31T10:30:24Z	2021-01-31
user_1	h(Mario+s)	h(Rossi+s)	h(mrossi+s)	2021-01-31T10:30:24Z	2021-02-01
user_2	h(Andrea+s)	h(Ferrari+s)	f(aferrari+s)	2021-02-01T19:14:49Z	2021-02-01
user_3	h(Francesco+s)	h(Bianchi+s)	h(fbianchi+s)	2021-02-01T20:56:37Z	2021-02-01

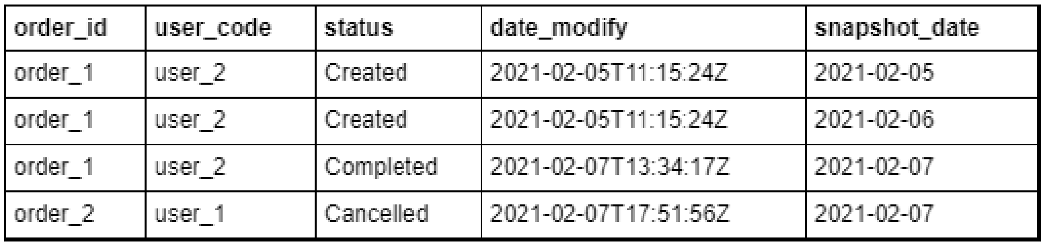
From this example we can see how the records in the snapshot table for the User entity changes day by day. In fact on the day *2021-01-31* there is only the record concerning the status of the user with the id *user_1* because on that date it is the only record received from the source systems. The following day, instead, we also have information on two new users and this leads to adding 3 records to the structure for the day *2021-02-01* reporting the last status of the 3 customers present in the system at that time.

Order Snapshot at 2021-02-05:



order_id	user_code	status	date_modify	snapshot_date
order_1	user_2	Created	2021-02-05T11:15:24Z	2021-02-05

Order Snapshot at 2021-02-07:



order_id	user_code	status	date_modify	snapshot_date
order_1	user_2	Created	2021-02-05T11:15:24Z	2021-02-05
order_1	user_2	Created	2021-02-05T11:15:24Z	2021-02-06
order_1	user_2	Completed	2021-02-07T13:34:17Z	2021-02-07
order_2	user_1	Cancelled	2021-02-07T17:51:56Z	2021-02-07

In the example, the photo of the structure at day *2021-02-06* has been omitted because on that day there are no new data or variations and therefore the record already present is simply carried forward (as can be seen from the photograph to the next day). During the day *2021-02-07* 3 records arrived from the source system composed as follows:

- a first record that changes the status of the order *order_1* bringing it to the completed status;
- a second record for the creation of a new order with id *order_2*;
- a last record that changes the status of the order just created bringing it to the completed status.

The final result is the addition for the day *2021-02-07* of two new records, one for each order present at that moment, with the latest status of each of them. From here we can see how the two movements of the order *order_2* within the same day is lost in this layer.

2.4 Enriched Layer

The last layer of the data model of the Data lake is called **enriched** layer. The idea of this layer is to give added value to the data by joining them together to make them usable in a simple way by area of analysis without having to worry about how to join together the information coming from different sources. In addition we have an advantage in terms of performance of the queries because the query engine does not have to join the information coming from the different tables in each run of the query but already has all the data necessary for a calculation on the same row. These data structures, unlike the previous ones, are structured as a **denormalized data mart**. Also in this case, the analyzes are historicized and a date field called *analysis_date* is added to validate the data for a given day on these structures.

In this layer there are also other types of structures which already contain pre-calculated KPIs according to rules defined by the business of our client. The calculation of these KPIs is materialized on physical structures instead of being calculated through the front-end tools in cases the calculation rules are too complex to be calculated each time the report is reloaded on the visualization tool. The most common case of pre-calculated data is on KPIs that exploit the movimentation of multiple entities that require the complete scan of several timeseries at each request. In this scenario the calculation times is too much longer than the few seconds with which the user wait for an answer. Going to materialize the pre-calculated data instead, the response of the report becomes almost immediate.

The sources tables of this layer can be *timeseries* or *snapshot* structures depending on the type of analysis is required. Normally the data marts exploit the updated data present in the snapshot layer, however, as mentioned above, the tables prepared for the calculation of the KPIs can read the data both from timeseries and from snapshots.

3 Data lake architecture

In the previous chapters we have analyzed what are the client's requests and what type of data model will be implemented in the data lake. In this chapter we will analyze the architecture of the components that we will have to implement to achieve that result and what requirements they will have to meet.

The data processing within the platform can be divided into four macro phases:

- **Ingestion**: in this first step the goal is to read/receive data from the different source systems, mask sensitive data and permanently save these data in the *Raw layer* for future processing;
- **Processing**: in this step the data written in the Raw layer is processed in order to be standardized, as nomenclature and as values, and cleaned of any dirty values to be saved inside the *Timeseries layer*. The processing for the calculation of the tables present in the *Snapshot* and *Enriched* layers that we have seen previously is based on these data;
- **Export**: this (optional) step has the purpose of exporting the data processed in the previous step to target systems by applying the unmasking of sensitive data;
- **Visualization and reporting**: this last step aims to make the data present in the data lake available to our client's business users through visualization and reporting tools such as *Tableau* to carry out self-analysis on the data or generate recurring reports already developed and saved on the tool in order to help them in their decision-making processes.

All components of this architecture must be:

- *Event-driven*: data processing by a component must be triggered when an event occurs (e.g. it receives data from a source system). To guarantee this requirement, each component is started by a message received on a messaging system and at the end of the processing it must also generate a message to be able to trigger subsequent processing (if any);

- *Configurable*: the software configurations must be divided from the code which must read and apply them. This requirement is fundamental in order to be able to add, modify and remove processing flows quickly and without impact on the whole system (for example, activate or deactivate the ingestion of a table from a source system or modify some logic applied to the data in the processing);
- *Scalable*: the components must make the most of the advantages of cloud processing and should be able to scale up or down automatically depending on the current workload;
- *Resilient*: in case of processing error on the data of a flow, the components must discard this data but continue to correctly manage all the others. Furthermore, the discarded data must in any case be saved to allow them to be reprocessed after an analysis and resolution of any problem;
- *Backward compatible*: when a new version of the components is released, it must maintain backwards compatibility with previous versions of the component in order to guarantee always a valid interface between the various components that communicate with each other (an update on a component must not break compatibility with the others);
- *Versionable*: both the code and the configurations must be versioned through a versioning system (for example Git) in order to always know which configurations are valid and the version of the code used for a component at a given moment;

Compliance with all these requirements allows dynamic management of components both at the level of code base and configurations, which is essential to use **CI/CD** techniques necessary to maintain a high level of system availability and at the same time improve the performance of the platform or add new features.

In the next sections we will analyze in more detail what are the functions and requirements that must comply the different steps of the pipeline and what was the chosen architectural solution.

3.1 Ingestion

The goal of the ingestion part is to receive/read data from different source systems, mask sensitive data and save the data on the raw layer permanently. During the design of the architecture of the ingestion phase, the following aspects were considered:

- the type of sources from which the data is received can be of different, in fact, it may happen that we have to retrieve data via JDBC connections on DB, receive CSV or JSON files from which to extract information or make requests to API to retrieve information that must be imported;
- the information about the data to be received/read must be configurable in order to be able to change this information in a flexible way. The information contained in these configurations concerns, for example, the connection info for the source system, what type of data we expect to receive and if this data must be masked or can be saved in plain;
- at the end of the operations and the writing of the data on the *raw layer*, the subsequent components must be informed of the presence of new data in order to be able to process them.

The resulting architecture to meet all these requirements is the following:

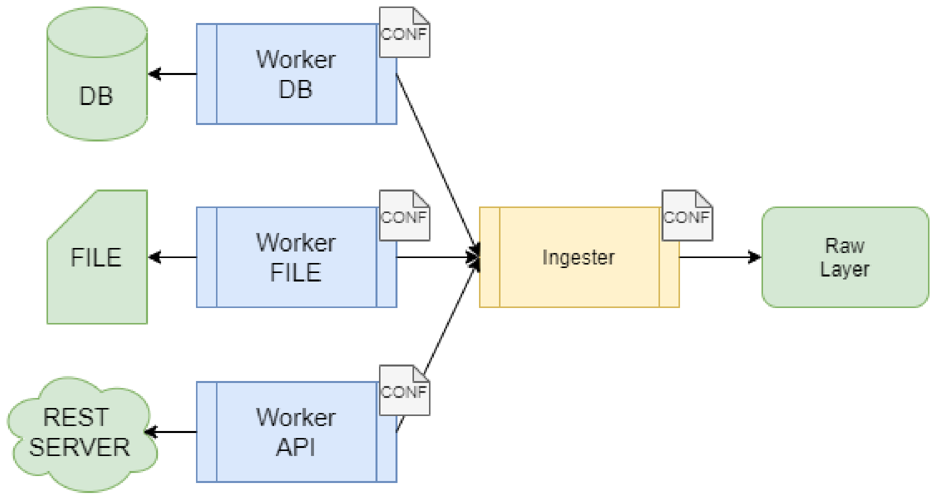


Figure 4: Ingestion architecture

There are components called **ingester-worker** that have the task of reading the new data and each type of worker reads from a different type of source system. After the reading the new data the workers forward them to the next component, the **ingester**, via serialized message. This component has the task of masking the data according to the configurations it receives and subsequently writes the data on the *raw layer.*
In the next paragraphs we will analyze these two components in more detail.

3.1.1 Ingester Worker: DB, File, API

The goal of the workers is to read the data from the different source systems and make them available to the next component which must be agnostic respect to the type of component that is providing the data. In order to do that is necessary to define an interface for data exchange and serialization. The data, as mentioned above, is exchanged through messages that contain a series of metadata relating to the structure being replicated followed by the data payload in JSON format.
An example of a message sent by a worker to the ingester is the following:

```
{
  "source": "name_of_source",
  "ingestion_timestamp": 1614528073568,
  "transcation_id": "49f09c02-79de-11eb-9439-0242ac130002",
  "payload": {
    "field_1": "value1",
    "field_2": "value2"
  }
}
```

where the values are the following:

- *source*: unique identifier of the data source (example for the worker DB we have the name of the source DB followed by the name of the replicated table);
- *ingestion_timestamp*: represent the timestamp in which the worker read the data from the source system and therefore the moment in which the data became available to the data lake;
- *transaction_id*: unique identifier of the replication job that read that

data. It is a UUID[5] which is generated randomly by the worker at each run in order to identify all the data replicated in the same job;

- *payload*: JSON object that contains replicated data from the source system.

With the definition of this interface we are able to develop different workers depending on the type of source system from which we have to replicate without worrying about how this data will be used later.

As mentioned during the presentation of the project, the majority of structures to be replicated derive from DWH systems and therefore the first type of worker that was designed was the **worker DB** but subsequently the we need also to import CSV, JSON or AVRO and retrieve information via REST requests so we design other two types of workers: the **worker file** and the **worker API**. With this architecture, however, it remains simply to implement a new type of worker in case of need.

The worker files and APIs do not have to submit to particular requirements other than to read the information made available to the source system and generate the messages for the ingester component that contains this data. As far as the DB worker is concerned, we have two types of ingestion: a first opportunity is to read all the data in a given table every time a new job is started, and in this case we speak of **full refresh replica**, or the second choice is to read only the **delta** of the new data published on the table after the previous run of the job on the same structure.

In order to realize this last feature it is obviously necessary to keep a history of the runs and know how far in the table it has been read. For this reason, a table on a relationship DB called **statefulness** has been introduced in the architecture of the worker DB to store through a *delta* field how far in the table it has been read. This delta field can be a timestamp field or a numeric field with a progressive sequence.

[5]A UUID is composed by 16 bytes (128 bits). In its canonical form, the UUID is represented by 32 hexadecimal characters, displayed in five groups separated by dashes, in the form 8-4-4-4-12 for a total of 36 characters (32 hexadecimal and four dashes)

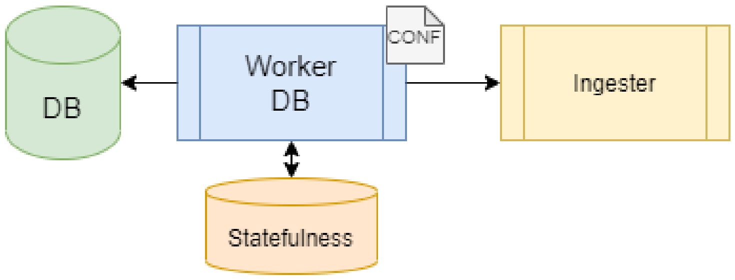


Figure 5: Worker DB architecture

The configurations necessary for this component for each flow are therefore:

- *source type*: the type of worker to use;
- *source system*: name of the source system to replicate from;
- *authentication*: information on how to authenticate on the source system;
- *source table*: name of the table to replicate (only for the worker DB);
- *delta field*: field to be used to apply the delta logic (only for the worker DB and if not specified, replication means full refresh)
- *path file*: URI of the location of the file to be replicated (only for worker files);
- *endpoint*: URL of the endpoint to contact to read the data (for worker API only);

One last point that remains to be analyzed is the trigger of workers. This component is also started by a received message. This message can be sent either by a scheduler at predetermined times (for example I start a replication job for a table every x minutes), it can be generated when an event occurs (for example we send a message when a file is detected on a bucket) or it can be sent directly from the source system when the data is available.

3.1.2 Ingester: Raw layer

The second step of the data ingestion phase is the mask and write of data in the first layer of the data lake, the *raw layer*. These two are the tasks of the next component that we are going to analyze, the **ingester**.

The operations of the ingester are triggered upon receiving the message sent by the ingestion-worker that generated the data and by reading the *source* metadata contained in the message, it retrieves its configurations for that flow. The configurations of the ingester must contain the list of fields considered sensitive and to be masked and the type of hashing to be applied on the field, which can be total (the entire value is encrypted) or partial (only a part of the value is encrypted). This informations are retrieved during the deployment phase of the configurations for a given flow by the Data governane tool used by the customer, named Collibra, in which the dedicated team maps every single field managed by the data lake.

After the component applies the required hashing, it saves in a key-value DB, in a structure dedicated to the table being processed, the pair hash value/plain value. These tables are called **rainbow tables** and we have one for each replicated table/file. This information is essential to be able to retrieve the relative plaintext value of the exported field during the export phase. A key-value DB has been chosen to make the unmasking phase very fast: in fact, we just need to enter in the rainbow table with the hashed value to recover the plain one.

The second task of the ingester is, as mentioned, to save the replicated data permanently in the raw layer. This storage is done via AVRO files saved on a bucket. An AVRO file will contain all messages belonging to the same worker replication job. This reconstruction is possible through the information contained in the metadata *transaction_id* contained in the message received by the worker. However, this first layer must be queried using SQL-like code and therefore *tables* will be built on these files which will allow the data contained in the AVRO files to be queried through SQL queries. In the next chapter we will see how this will be done in a very simple way by exploiting the potential of the *BigQuery* tool made available by GCP. So in the Ingestion layer we don't need other components to complete the tasks.

The last task of the ingester is send a message to communicate to the next component, the *Data Model Mapper*, that new data are ready in the raw layer. To do that, for each saved AVRO file, it sends a notification message to the next component containing the information of the replicated source, the transaction_id relating to the file and the path of the file itself to which the Mapper will find the data to be processed.

In the next paragraphs we will analyze the components necessary for the Processing step and the realization of the subsequent layers.

3.2 Processing

The goal of the Processing phase is to process the data received from the sources to standardize them and make them available to end users for self-analysis activities. This standardization phase is the one that brings the data from the *Raw layer* to the *Timeseries layer*. The normalizations that are applied are both on the data values and on the nomenclature.
After completing this first activity starting from the Timeseries layer, the subsequent layers of the datamodel are calculated, i.e. the *Snapshot layer* to historicize the data and the *Enriched layer* in which the various data marts necessary for users for their complex analyzes are created or some KPIs are calculated based on the client's business rules.

Also for this part of the datalake one of the main requirements is the configurability of the components in order to be able to add, modify or remove subsequent timeseries or structures according to the needs of the moment.

3.2.1 Data Model Mapper: Timeseries layer

In this phase of data processing we want to process the data written by the previous component, the Ingester, in order to standardize them according to the rules of the datalake in order to materialize the result on a table that can be queried via SQL. The component that will carry out these processing steps is called **Data Model Mapper**. The name derives from the fact that through these transformations we are *mapping* the data coming from the source system into a field of a datalake table.

To guarantee the event-driven methodology, the Mapper processing is triggered by a JSON message sent by the Ingester that contains within it the path of the file to be processed, the transaction_id that uniquely identifies the replication job from the source system, the reference to the source table that was replicated and the instant in time in which the ingestion phase started. An example of a message sent by the Ingester to the Mapper is the following:

```
{
  "source": "name_of_source",
  "ingestion_timestamp": 1614528073568,
  "transcation_id": "49f09c02-79de-11eb-9439-0242ac130002",
  "file_path": "gs://bucket/file"
}
```

From the received message, the Mapper retrieves its configurations for that source structure which contains all the information necessary for the transformation of the data. The configurations required by the Mapper for each source are the following:

- the target table in the timeseries layer;
- the filters to be applied on the data through SQL syntax (useful for cleaning up the data);
- mapping information of each single raw field into the related field in timeseries. For each raw field the possible configurations are the following:
  - field name on the timeseries layer;
  - normalization to be applied (optional);
  - normalization parameters (optional);
  - flag to indicate if the field is the partition field (optional);

The standardization of the data is done by applying normalization rules on the received data that add one or more fields (according to the normalization required) to the timeseries containing the standardized value of the field. The field replicated by the source is however maintained at the timeseries level. The normalizations that can be applied to a given field are the following:

- **date**: allows you to create a DATE type field according to the standardized format YYYY-MM-DD. Among the normalization parameters it is possible to specify the format to parse the data received from the source;
- **datetime**: allows you to create one or more fields of type DATETIME according to the standardized format YYYY-MM-DDTHH:mm:ssZ where the Z represents the reference time zone. In this case, among the normalization parameters, in addition to the format to parse the date, there is an indication of which must be the time zones with which to represent the data in the additional fields (the most used are UTC and Europe/Rome);
- **number**: it allows you to create a numeric type field (INTEGER, FLOAT, DOUBLE) starting from the data received from the source. The normalization parameters that can be set are the type of data we want as output, with how many decimal digits we are going to represent the data and which is the separator character present in the source field;
- **boolean**: it allows to create a field of the BOOLEAN type starting from the received data. As a normalization parameter we can indicate which values must be considered as True (example 1, S, Y, ...) and which ones as False (0, N, ...).

Another configuration that can be indicated on a field is the partition indication. In fact, it is possible to indicate a field of type date or timestamp as the partition value of the table. Only one field is settable as a partition and its value will be replicated in the *partition_date* field. This field will be set at the DDL when creating the table as a partition field allowing to make the queries that use that field as a filter more efficient. If no field is indicated as partition, the partition_date field is set with the processing date.

An example of a Mapper configuration file for a source is the following:

```
{
  "destination_table": "timeseries_name",
  "filter": "SQL statement",
  "mapping": [
    {
      "source": "source_field_name",
      "destination": "destination_field_name",
      "partition": True|False,
      "normalization": "date|datetime|number|boolean",
      "normalization_param": {normalization parameters}
    }
  ...
  ]
}
```

Mapper configurations are created automatically from the mapping information created by the Data Governance team for each individual timeseries on the **Collibra**[6] tool. Collibra is an enterprise-oriented data governance platform for data management and stewardship. It empowers businesses to find meaning in their data and improve business decisions. Using the REST API made available by Collibra, the configurations of all fields marked in the "Ready for Prod" status are retrieved. The information that are read are those necessary to create the mapping JSON file: system and source table, dataset and destination table on BigQuery, name of the source and destination field and finally any normalization to be applied on the field.
At the end of the release of the new field in Production, the state of the field on Collibra is automatically set to the "Implemented" state.

This organization allows to automate the updating of Mapper configurations, in fact, once a day a crawler is run to check the presence of new fields in the "Ready for Prod" state and in this case the entire CI/CD process necessary for release is triggered, as we will see later in the dedicated chapter.

In the event that the processing of a record is not successful, it is discarded and written as a string in a Death Letter Queue table in which, in addition to the data, the source table from which the data derives and the reason for the record was discarded are written.

Once all the above activities have been completed and after having written the records inside the destination table, the Mapper publishes a JSON

[6]Collibra: https://www.collibra.com/

message to indicate the end of its activity so that any subsequent component is informed of the presence of new data to be processed.

The final architecture of the Mapper is therefore the following:

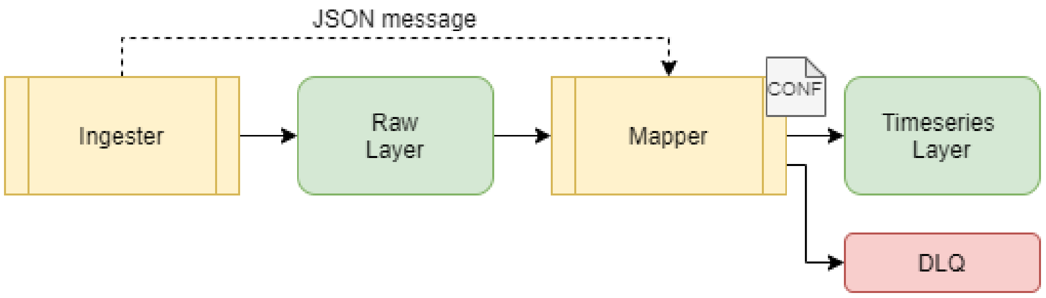


Figure 6: Mapper architecture

3.2.2 Snapper: Snapshot and Enriched Layer

After preparing the data in the timeseries, the second activity of the Processing phase is to materialize the data necessary for the subsequent layers, the *Snapshot* and *Enriched layers*. These two layers are written by the same component, the **Snapper**. The name of this component derives from the fact that it was initially developed to solve the use case of writing data to the Snapshot layer but was later used for the Enriched layer as well.

The goal of this component is to materialize the result of a generic SQL query in a table. In this way, starting from one or more tables, the data can be materialized in a target table by applying selection, filtering, aggregation and join statements, developing the entire logic in SQL language. The input tables from which to read the data to be processed can be in the *Timeseries*, *Snapshot* and *Enriched* layers. So the Snapper uses these last two layers both in the reading and writing phase.

The execution of this component is also started via a JSON message on a messaging system. This message can be the one sent by the Mapper at the end of its execution or it can be sent by an orchestrator when a condition occurs. This allow you to perform executions on a time basis, for example every day at midnight as happens for Snapshots, or executions dependent on the end of other runs as happens in the case of Enriched structures.

The necessary configurations for this component are therefore the following:

- the target table in which to materialize the query result;
- the query to run.

The first is contained in a JSON file associated with the stream, while the second is contained in an SQL file.

The final architecture of the Snapper component is therefore the following:

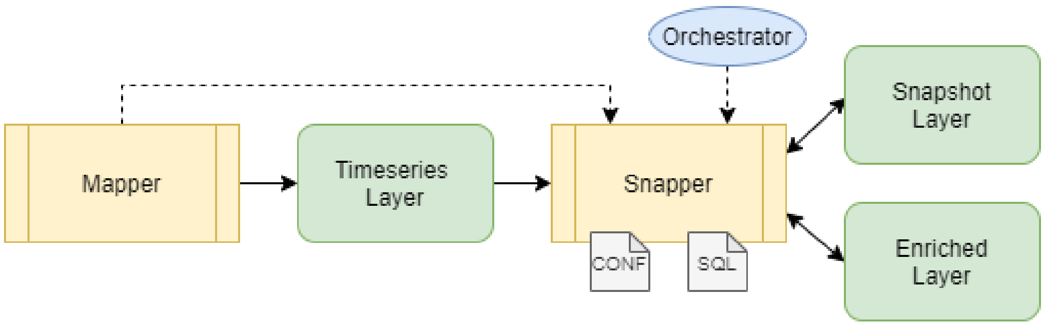


Figure 7: Snapper architecture

3.3 Exporting

For some use cases it was required to export some data present on the data lake to external systems. These data are exploited by processing the data present in the various layers to prepare them for the target system according to the agreed interface specifications.

As in the case of data ingestion, also in this case the downstream systems can be of different types and therefore also the method of transferring the files can differ. Also in this case, the exporter requires connectors, which are called *exporter-workers*, which allow the sending of files, writing to DB or writing data to a remote system via REST API.

In all the previous layers the sensitive data as mentioned are hashed to make it impossible for anyone to read them. However, when we export these data to target systems, they must be unmasked in order to be understandable and usable by the users of the target systems. To do this it is necessary to apply the inverse transformation used in the ingestion phase. Since performing the inverse function of a hashing function is not feasible, the plain text value is retrieved through the previously created rainbow tables.

To ensure data security during the transfer phase, the file to be sent, the connection used on JDBC or HTTP protocol are encrypted in such a way that anyone cannot read the clear content of the export.

In the next paragraphs we will analyze in more detail how these three phases of the export process were designed.

3.3.1 Exporter: data selection

The first operation to be able to export the data to an external system is to select them from the various tables present in the data lake. In this case, the extraction process can take place from any of the structures belonging to the *Timeseries*, *Snapshot* and *Enriched* layers. The component responsible for this activity is called **Exporter**.

This selection is done via SQL queries. This allow you to process the data to produce the desired output. In fact, it is possible to insert complex SQL statements such as joins or aggregations on the data or more simply to format the data according to the interface specifications decided together with the target system (e.g. date formatting).

This first part is actually very similar to a *Snapper* with the difference that the output is not materialized on a BigQuery table but, in this case on a temporary file in CSV. In this phase the data are those extracted from the data lake structures and therefore the fields that represent sensitive information are still hashed. In the next phase these data will be unmasked and made clear.

Also as regards the execution trigger, this component behaves like the *Snapper* and is therefore triggered through a Pub/Sub message that can be sent by an orchestrator or by one of the previous components of the architecture. At the end of its activity and the creation of the temporary CSV file, a trigger message is sent to the next component indicating the location of the produced file and to which export flow it belongs (through a unique code called export-id).

The architecture of the Exporter component is therefore the following:

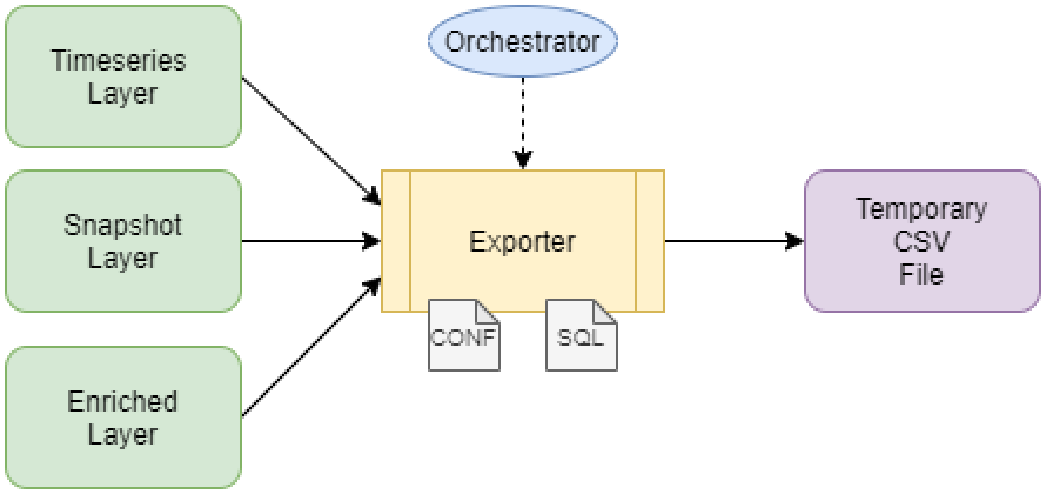


Figure 8: Exporter architecture

3.3.2 Unmasker: unmasking of PII fields

After extracting the data these must be unmasked if necessary. This is the task of the component called **Unmasker**. The execution is triggered by the message received from the *Exporter* from which the path of the Exporter temporary file is stored and the indication of the export flow are retrieved.

Through the *export-id* read, the component retrieves its configurations relating to this flow which contain the list of fields on which the unmasking activity is required. This list consists of key-value pairs where the key is the position of the field in the CSV file and the value is the indication of the source-level field from which that value is derived. Using this information, the Unmasker accesses the *rainbow table* of competence of the indicated source table and searches the plain value corresponding to the hash value read from the file. Recall that these rainbow tables are written on DB key-value indexed by the hashed value of the field and therefore access to the data is practically immediate.

After unmasking all the PIIs present in the file, the component writes a new temporary file with the values in clear text but **completely encrypted** so that no user (who does not know the public key corresponding to the private one used in the encryption phase) can read the clear information. In addition to this, it also deletes the temporary file produced by the previous component as it is no longer necessary.

As a last step, it sends a message to one of the following components to send data to the target system. The next component that is triggered depends on the type of worker needed to send the data. To do this, the Unmasker publishes this trigger message on a **different topic**. Also in this case the message contains the path of the file that he created and the **export-id** useful for the *worker* to understand which flow the file he has to process refers to.

To respect the previous requirements, the Unmasker architecture is therefore the following:

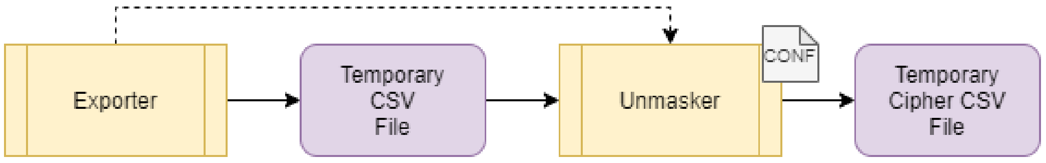


Figure 9: Unmasker architecture

3.3.3 Exporter-Worker: send data to target systems

The last step to complete the export operations is to send the data to the target system. This is the job of the **exporter-workers**, each of them is specialized in writing data to a target system of different type.

The workers that have been developed so far are:

- JDBC: to write the extracted data to database tables outside the data lake;

- file: to export data in *CSV* format to a file system via SFTP protocol or to a bucket in the cloud (Google Cloud Storage or AWS S3);
- REST: allows you to make POST or PUT requests on endpoints to save the extracted data on a REST server.

Given the architecture implemented, a new type of worker can easily be implemented to export data to different types of systems.

Each of the workers will listen for a trigger message on a different *topic* and therefore it will be the task of the previous component (the *Unmasker*) to send the message on the correct topic to start the processing of the worker necessary for the flow in progress. Inside the trigger message there are the *export-id* which is the unique identifier of the export stream and the path of the temporary CSV file generated by the Unmasker.

Using the export-id the activated exporter-worker loads its own configurations which change according to the type of activated worker. In the case of the JDBC worker, the information needed is the endpoint to the target database, the authentication credentials, and the table to write. In the case of the worker file, instead, you simply need to know the server and the path on which to save the file. For the REST worker it is necessary to know the endpoint on which to connect, the type of request to send (POST / PUT) and the authentication information, if needed.

The worker workflow involves the following operations:

- reading and decryption of the file created by the Unmasker;
- send the data to the target system (the implementation of this operation varies by worker);
- deletion of the temporary file created by the Unmasker as it is no longer useful for processing.

To implement these operations and comply with the requirements indicated above, the architecture of the workers is the following:

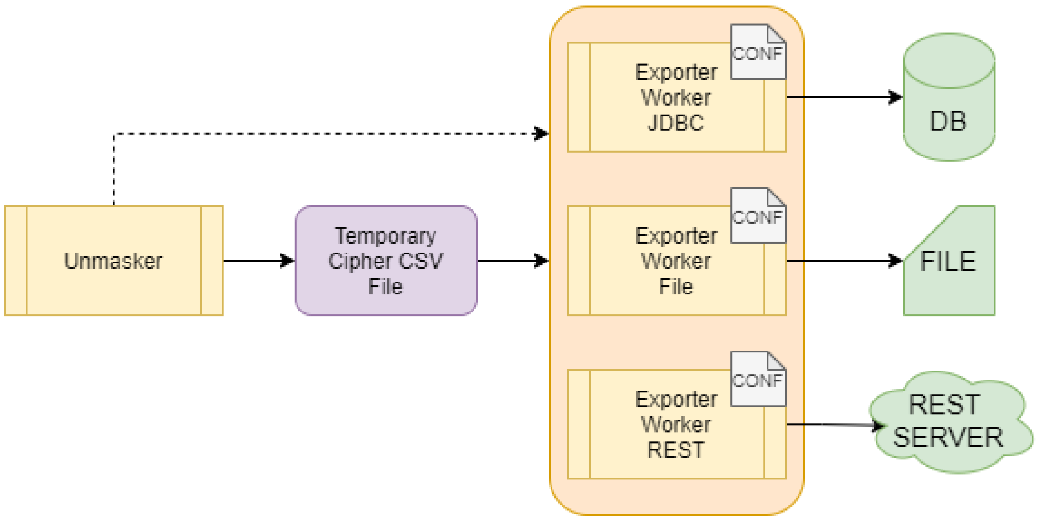


Figure 10: Exporter-worker architecture

4 Implementation on Google Cloud Platform

In this chapter we will analyze the tools made available by the cloud provider **Google Cloud Platform** which, upon customer request, will be used to implement the entire data lake. In addition to studying the various components, we will understand how their main characteristics can help us in creating the architecture described in the previous chapter and what are the limits to pay attention.

4.1 PubSub

Pub/Sub is an **asynchronous messaging service** that decouples services that produce events from services that process events. It can be used as messaging-oriented middleware or event ingestion and delivery for streaming analytics pipelines. Pub/Sub offers durable message storage and real-time message delivery with high availability and consistent performance at scale. Its servers run in all Google Cloud regions around the world.[pub/sub]

The core concepts of Pub/Sub are:

- **Topic**: a named resource to which messages are sent by publishers;
- **Subscription**: a named resource representing the stream of messages from a single, specific topic, to be delivered to the subscribing application;
- **Message**: the combination of data and (optional) attributes that a publisher sends to a topic and is eventually delivered to subscribers;
- **Message attribute**: A key-value pair that a publisher can define for a message to add some metadata to the message.

A publisher application creates and sends messages to a topic. Subscriber applications create a subscription to a topic to receive messages from it. Pub/Sub delivers each published message at least once for every subscription. Communication can be one-to-many (fan-out), many-to-one (fan-in), and many-to-many, as the following diagram shows:

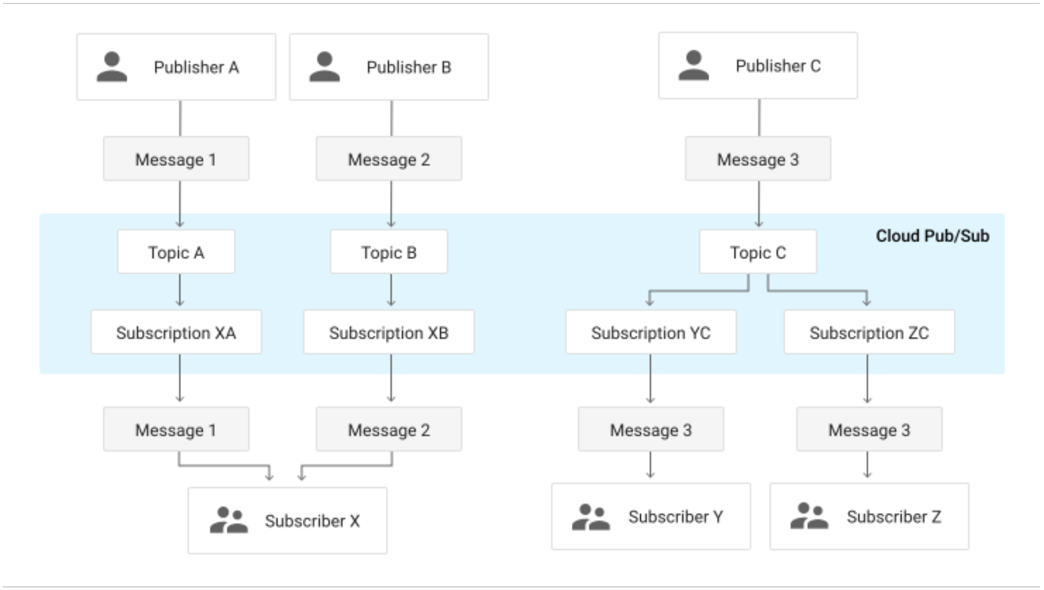


Figure 11: Pub/Sub publisher-subscriber relationships

Pub/Sub is the tool used in the platform for exchanging messages between the various components and therefore realizing the data-driven processing requirement.

4.2 Google Cloud Storage

Cloud Storage is a service for storing objects in Google Cloud. An object is an immutable piece of data consisting of a file of any format. We can store objects in containers called buckets. All buckets are associated with a project, and you can group your projects under an organization.[gcs]

Here are some basic ways you can interact with Cloud Storage:

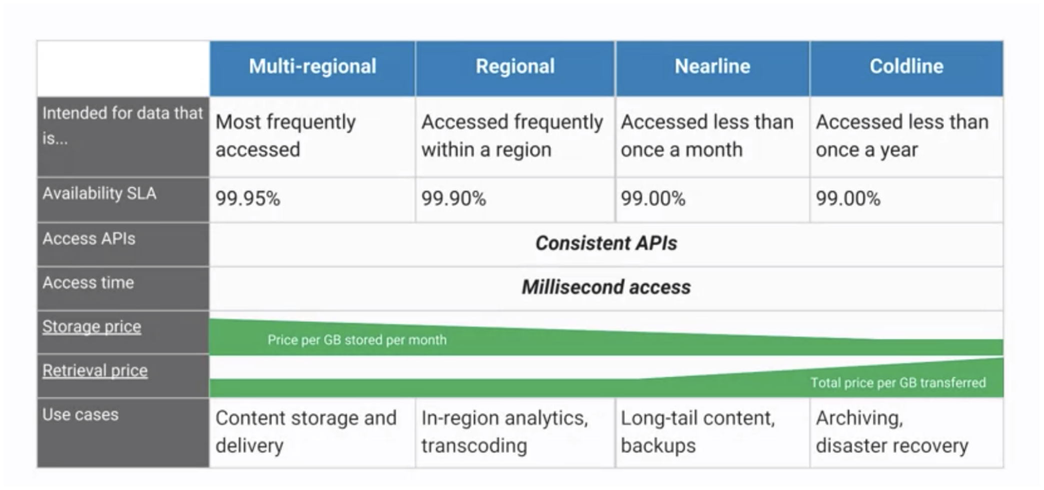
- **Console**: The Google Cloud Console provides a visual interface for you to manage your data in a browser;
- **gsutil**: gsutil is a command-line tool that allows you to interact with

Cloud Storage through a terminal. If you use other Google Cloud services, you can download the Cloud SDK, which includes gsutil along with the gcloud tool for other services;

- **Client libraries**: the Cloud Storage client libraries allow you to manage your data using one of your preferred languages, including C++, C#, Go, Java, Node.js, PHP, Python, and Ruby;

- **REST APIs**: manage data using the JSON or XML API.

Google Cloud Storage is used in the datalake for two purposes: to permanently save data in the RAW layer and to store all configuration files needed by the components. The pricing of GCS is given by the amount of data saved and based on the class to which this data is associated (higher cost classes provide lower access times). So to decrease the cost of the raw storage layer, the files will be subjected to a lifecycle that moves the files from the Multi-Regional class (the fastest but most expensive) to the Coldline class (the slowest but cheapest class) after 90 days from their creation. This too is a feature already managed by GCS.



	Multi-regional	Regional	Nearline	Coldline
Intended for data that is...	Most frequently accessed	Accessed frequently within a region	Accessed less than once a month	Accessed less than once a year
Availability SLA	99.95%	99.90%	99.00%	99.00%
Access APIs	***Consistent APIs***			
Access time	***Millisecond access***			
Storage price	Price per GB stored per month			
Retrieval price	Total price per GB transferred			
Use cases	Content storage and delivery	In-region analytics, transcoding	Long-tail content, backups	Archiving, disaster recovery

Figure 12: Google Cloud Storage bucket class comparison

4.3 BigQuery

BigQuery is an enterprise data warehouse that solves this problem by enabling super-fast SQL queries using the processing power of Google's infrastructure. Simply move your data into BigQuery and let us handle the hard work. You can control access to both the project and your data based on your business needs, such as giving others the ability to view or query your data. You can access BigQuery by using the Cloud Console, by using the bq command-line tool, or by making calls to the BigQuery REST API using a variety of client libraries such as Java, .NET, or Python. There are also a variety of third-party tools that you can use to interact with BigQuery, such as visualizing the data or loading the data.
BigQuery is a service fully-managed. To get started, you don't need to deploy any resources, such as disks and virtual machines.[bigQuery]

A dataset is contained within a specific project. Datasets are top-level containers that are used to organize and control access to your tables and views. A table or view must belong to a dataset, so you need to create at least one dataset before loading data into BigQuery.

A BigQuery table contains individual records organized in rows. Each record is composed of columns (also called fields). Every table is defined by a schema that describes the column names, data types, and other information. You can specify the schema of a table when it is created, or you can create a table without a schema and declare the schema in the query job or load job that first populates it with data. BigQuery supports the following table types:

- **Native tables**: tables backed by native BigQuery storage;
- **External tables**: tables backed by storage external to BigQuery (Cloud BigTable, Cloude Storage, Google Drive or Cloud SQL);
- **Views**: Virtual tables defined by a SQL query.

In the data model of the platform, all the types of tables listed above were used. The RAW layer is composed of external tables that read from the bucket on Google Cloud Storage in which the ingester writes the data, making it possible to query AVRO files in SQL languages. The timeseries, snapshot and enriched layers is instead created through native BigQuery tables in order to store the data in the form preferred by BigQuery on which it is possible to apply all the optimizations of the tool. Finally, for some reporting activities in which it is necessary to carry out very simple transformations, we have preferred to use views instead of native tables in order to save on storage costs.

A partitioned table is a special table that is divided into segments, called **partitions**, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance, and you can control costs by reducing the number of bytes read by a query.

All the native tables present in the datalake are partitioned tables to obtain the maximum performance and reduce execution costs. The timeseries are partitioned by date of competence of the data, instead, the other structures (Snapshot and Enriched) are partitioned by calculation date.

Query pricing refers to the cost of running your SQL commands and user-defined functions. BigQuery charges for queries by using one metric: the number of bytes processed. You are charged for the number of bytes processed whether the data is stored in BigQuery or in an external data source such as Cloud Storage, Google Drive, or Cloud Bigtable. The first 1 TB of data processed per month is free of charge (per billing account). Beyond your first 1 TB of data processed in a month, you are charged (around 5$ for each terabyte processed).

There are several ways to ingest data into BigQuery:

- Batch load a set of data records: with batch loading, you load the source data into a BigQuery table in a single batch operation. For example, the data source could be a CSV file, an external database, or a set of log files. Traditional extract, transform, and load (ETL) jobs fall into this category. This type of data upload is free. This is the case of the uploads performed by the Mapper in the timeseries layer where the data received from the source systems is sinked every 10 minutes in batch mode.
- Stream individual records or batches of records. This type of upload

involves costs depending on the write rate. None of the data lake components use this writing mode.

- Use queries to generate new data and append or overwrite the results to a table: you can use data manipulation language (DML) statements to perform bulk inserts into an existing table or store query results in a new table. This is the case of the uploads carried out by the Snapper component which reads data from one or more tables, processes them according to a supplied query and materializes the result in a target table.

BigQuery writes all query results to a table. The table is either explicitly identified by the user (a destination table), or it is a temporary, cached results table. Temporary, cached results tables are maintained per-user, per-project. There are no storage costs for temporary tables, but if you write query results to a permanent table, you are charged for storing the data. All query results are cached in temporary tables for approximately 24 hours. In this way if the same user rerun the same query multiple times only the first execution process the stored data and is billed.

4.4 Google Cloud Dataflow

Dataflow is a managed service for executing a wide variety of data processing patterns. A Dataflow job is the execution of an Apache Beam framework pipeline. Apache Beam is an open source, unified model for defining both batch- and streaming-data parallel-processing pipelines. The Apache Beam programming model simplifies the mechanics of large-scale data processing. Using one of the Apache Beam SDKs, you build a program that defines the pipeline. Then, one of Apache Beam's supported distributed processing backends, such as Dataflow, executes the pipeline. This model lets you concentrate on the logical composition of your data processing job, rather than the physical orchestration of parallel processing. You can focus on what you need your job to do instead of exactly how that job gets executed.

The Apache Beam model provides useful abstractions that insulate you from low-level details of distributed processing, such as coordinating individual workers, sharding datasets, and other such tasks. Dataflow fully manages these low-level details.

The basic concepts of the Apache Beam model are:

- **Pipeline**: a pipeline is a graph of transformations that a user constructs that defines the data processing they want to do;
- **PCollection**: data being processed in a pipeline is part of a PCollection;
- **PTransforms**: the operations executed within a pipeline.
- **Runner**: you are going to write a piece of software called a runner that takes a Beam pipeline and executes it using the capabilities of your data processing engine.

In Beam, a PTransform can be one of the five primitives or it can be a composite transform encapsulating a subgraph. The primitives are:

- **Read**: parallel connectors to external systems;
- **ParDo**: per element processing;
- **GroupByKey**: aggregating elements per key and window;
- **Flatten**: union of PCollections;
- **Window**: set the windowing strategy for a PCollection.

A PCollection is an unordered bag of elements. Your runner will be responsible for storing these elements. A PCollection may be bounded where it is finite and you know it as in batch use cases or unbounded where it may be never end as in streaming use cases. These derive from the intuitions of batch and stream processing, but the two are unified in Beam and bounded and unbounded PCollections can coexist in the same pipeline. If your runner can only support bounded PCollections, you'll need to reject pipelines that contain unbounded PCollections. If your runner is only really targeting streams, there are adapters in our support code to convert everything to APIs targeting unbounded data.

Every element in a PCollection has a timestamp associated with it. When you execute a primitive connector to some storage system, that connector is responsible for providing initial timestamps. Your runner will need to propagate and aggregate timestamps. If the timestamp is not important, as with certain batch processing jobs where elements do not denote events, they will be the minimum representable timestamp, often referred to colloquially as “negative infinity”.

Every PCollection has to have a watermark that estimates how complete the PCollection is. The watermark is a guess that “we’ll never see an element with an earlier timestamp”. Sources of data are responsible for producing a watermark. Your runner needs to implement watermark propagation as PCollections are processed, merged, and partitioned.

Every element in a PCollection resides in a window. No element resides in multiple windows (two elements can be equal except for their window, but they are not the same). When elements are read from the outside world they arrive in the global window. When they are written to the outside world, they are effectively placed back into the global window (any writing transform that doesn’t take this perspective probably risks data loss). A window has a maximum timestamp, and when the watermark exceeds this plus user-specified allowed lateness the window is expired. All data related to an expired window may be discarded at any time.

The term “runner” is used for a couple of things. It generally refers to the software that takes a Beam pipeline and executes it somehow. Often, this is the translation code that you write. It usually also includes some customized operators for your data processing engine, and is sometimes used to refer to the full stack. Google Dataflow provide a runner to run an Apache Beam pipeline in the Google cloud environment.

The Dataflow service automatically performs and optimizes many aspects of distributed parallel processing. These include:

- **Parallelization and Distribution**: Dataflow automatically partitions your data and distributes your worker code to Compute Engine instances for parallel processing.
- **Optimization**: Dataflow uses your pipeline code to create an execution graph that represents your pipeline’s PCollections and transforms,

and optimizes the graph for the most efficient performance and resource usage. Dataflow also automatically optimizes potentially costly operations, such as data aggregations.

- **Automatic Tuning features**: the Dataflow service includes several features that provide on-the-fly adjustment of resource allocation and data partitioning, such as Autoscaling and Dynamic Work Rebalancing. These features help the Dataflow service execute your job as quickly and efficiently as possible.

The two main components of the platform, the Ingester and the Mapper, is deployed with Dataflow. Both are Dataflow streaming jobs. In the first case, the job receives messages on a Pub/Sub subscription, processes it and produces the output file on the RAW layer, in the second case, instead, to guarantee microbatch writes on BigQuery every 10 minutes, the streaming data is grouped into windows to 10 minutes where the window key is the target table.

Dataflow service usage is billed in per second increments, on a per job basis. Usage is stated in hours (30 minutes is 0.5 hours, for example) in order to apply hourly pricing to second-by-second use. Workers and jobs may consume resources, instead, logs are not billed. Dataflow workers consume the following resources, each billed on a per second basis: virtual CPU, memory, storage (persistent disk), GPU (optional).

4.5 Cloud Function

Google Cloud Functions is a serverless execution environment for building and connecting cloud services. With Cloud Functions you write simple, single-purpose functions that are attached to events emitted from your cloud infrastructure and services. Your function is triggered when an event being watched is fired. Your code executes in a fully managed environment. There is no need to provision any infrastructure or worry about managing any servers.[cloud function]

Cloud Functions can be written using JavaScript, Python 3, Go, or Java runtimes on Google Cloud Platform. You can take your function and run it in any standard Node.js (Node.js 10 or 12), Python 3 (Python 3.7 or 3.8), Go (Go 1.11 or 1.13) or Java (Java 11) environment, which makes both portable and local testable.

Cloud Functions removes the work of managing servers, configuring software, updating frameworks, and patching operating systems. The software and infrastructure are fully managed by Google so that you just add code. Furthermore, provisioning of resources happens automatically in response to events. This means that a function can scale from a few invocations a day to many millions of invocations without any work from you.

Events are things that happen within your cloud environment that you might want to take action on. These might be changes to data in a database, files added to a storage system, or a new virtual machine instance being created. Currently, Cloud Functions supports events from the following providers:[cloud events]

- receive a request to an HTTP endpoint;
- create/delete/modify a file in Google Cloud Storage;
- receive a message in a Pub/Sub subscription;
- an application create a specify log.

In our case we use the Cloud Functions to implement batch ETL executions such as those required to create snapshots, materialize the query result for the enriched layer or produce export files to other target systems.

One of the limits imposed by Google on the execution of the Cloud Functions is the maximum duration of the same. In fact, after 540 seconds the execution is terminated, so in the case there are longer executions it will be necessary to concatenate more Cloud Functions. This is the case of the Snapper and Exporter components that execute potentially complex queries on Big Query and the CF can take more than 540 seconds to complete them. In the next chapter we will analyze how this problem has been overcome.

4.6 Datastore

Datastore is a highly scalable NoSQL database for your applications. Datastore automatically handles sharding and replication, providing you with a highly available and durable database that scales automatically to handle your applications' load. Datastore provides a myriad of capabilities such as ACID transactions, SQL-like queries and indexes.

Datastore supports a variety of data types, including integers, floating-point numbers, strings, dates, and binary data. Ensure the integrity of your data by executing multiple datastore operations in a single transaction with ACID characteristics, so all the grouped operations succeed or all fail.

Datastore is used to build the **rainbow tables**. These structures are created using the masked value as a key and the plain value as a value of the row. In this way, in a linear time, we can retrieve the plain value associated with the hashed value.

4.7 Cloud SQL

Clous SQL is the fully managed product of the Google CLoud Platform that allows to manage transactional relationship databases such as *MySQL, PostegrSQL and SQL Server*. It automatically guarantees the reliability, security and scalability of the databases, so that production activities can continue without any kind of interruption. Cloud SQL automates all backups and replicas, encryption patches and capacity increases while ensuring greater than 99.95 % availability. Obviously, being a product of the GCP catalog, it is natively integrates with all the other tools of the platform such as AppEngine and BigQuery (from which it is possible to simultaneously query native tables and external tables stored on Cloud SQL).

The limits in using Cloud SQL are in terms of storage space and geographic consistency. In fact, the scalability of Cloud SQL instances is *vertical* by increasing the resources of the machine on which the DB runs, to obtain *horizontal* scalability (and therefore manage connections and transactions globally) you need to use another tool of the GCP platform, Cloud Spanner.

In our datalake architecture we needed to use a relationship database with ACID properties to store some metadata used by components to save the computation state. In particular, the *Worker Ingester* component save the delta information of the data read from the source for each run. Given the low amount of data that we need to store and since it is not necessary to manage read/write transactions globally, the choice between the two Google tools fell on Cloud SQL which has a lower management cost than Cloud Spanner.

4.8 Cloud Composer

Cloud Composer is a fully managed workflow orchestration service, enabling you to create workflows that span across clouds and on-premises data centers. Built on the popular Apache Airflow open source project and operated using the Python programming language, Cloud Composer is free from lock-in and easy to use. By using Cloud Composer instead of a local instance of Apache Airflow, users can benefit from the best of Airflow with no installation or management overhead.[composer]

In data analytics, a workflow represents a series of tasks for ingesting, transforming, analyzing, or utilizing data. In Airflow, workflows are created using DAGs, or "Directed Acyclic Graphs". A DAG is a collection of tasks that you want to schedule and run, organized in a way that reflects their relationships and dependencies. DAGs are created in Python scripts, which define the DAG structure (tasks and their dependencies) using code. Each task in a DAG can represent almost anything, for example trigger an application, an ETL flow, run a pipeline or send an email.

In the platform the Google Composer platform is used to trigger scheduled pipelines at certain times or to coordinate the execution of various components in which the execution of a step is dependent on the successful completion of the previous steps. To implement these logics, the Airflow **sensors** are used. In particular the *DateTime Sensor* are used to guarantee the trigger of a subsequent step at a certain time, or the *BigQuery Partition Sensor* to verify the presence of the data necessary to calculate the query before executing it. For some use cases a task is triggered when both types of sensors pass their checks.

An example is the computation of an Enriched structure that reads for its computation data from two Snapshot structures. In this case the execution of the snapper for the calculation of the final structure must be triggered only after the successful completion of the calculation of the two previous structures. This behaviour can be easily achieved through a Composer DAG by creating the appropriate dependencies between the various nodes of the graph (each node represents one execution of a snapper).

4.9 Cloud Logging

Cloud Logging is part of the Google Cloud's operations suite of products. It includes storage for logs, a user interface called the Logs Explorer, and an API to manage logs programmatically. Logging lets you read and write log entries, query your logs, and control how you route and use your logs.

A log entry records status or describes specific events or transactions that take place in computer systems. Log entries are written by your own code, Google Cloud services the code is running on, third-party applications, and the infrastructure that Google Cloud depends on.

Some log entries describe specific events that take place within the system. You can use these log entries to output messages that assure users that things are working well or to provide information when things fail. Other log entries might describe the details of transactions processed by a system or component. For example, a load balancer logs every request that it receives. A load balancer also records information like the requested URL and the HTTP response code, and it might record which backend served the request.

All the software components of the data lake write their logs on Cloud Logging, in this way it is easy to monitor the status of each component and the final status of execution of them. Using the query engine made available by Cloud Logging it was also possible to create automatic platform monitoring and alerting systems.

4.10 Architecture of data lake with GCP tools

In the previous paragraphs we have analyzed the tools made available by Google Cloud Platform. Now let's summarize for each component of the data lake which of these tools are used:

- **Worker Ingester and Exporter**: Cloud Dataflow, GCS (Google Cloud Storage) and Cloud SQL;
- **Ingester**: Cloud Dataflow, Datastore;
- **Data Model Mapper**: Cloud Dataflow, Big/Query;
- **Snapper and Exporter**: Cloud Function, Big/Query;
- **Unmasker**: Cloud Function, Datastore;
- **Data and configuration storage**: Google Cloud Storage;
- **Messagging system**: Pub/Sub;
- **Orchestrator**: Cloud Composer.

On the next page the complete architectural diagram of the data lake.

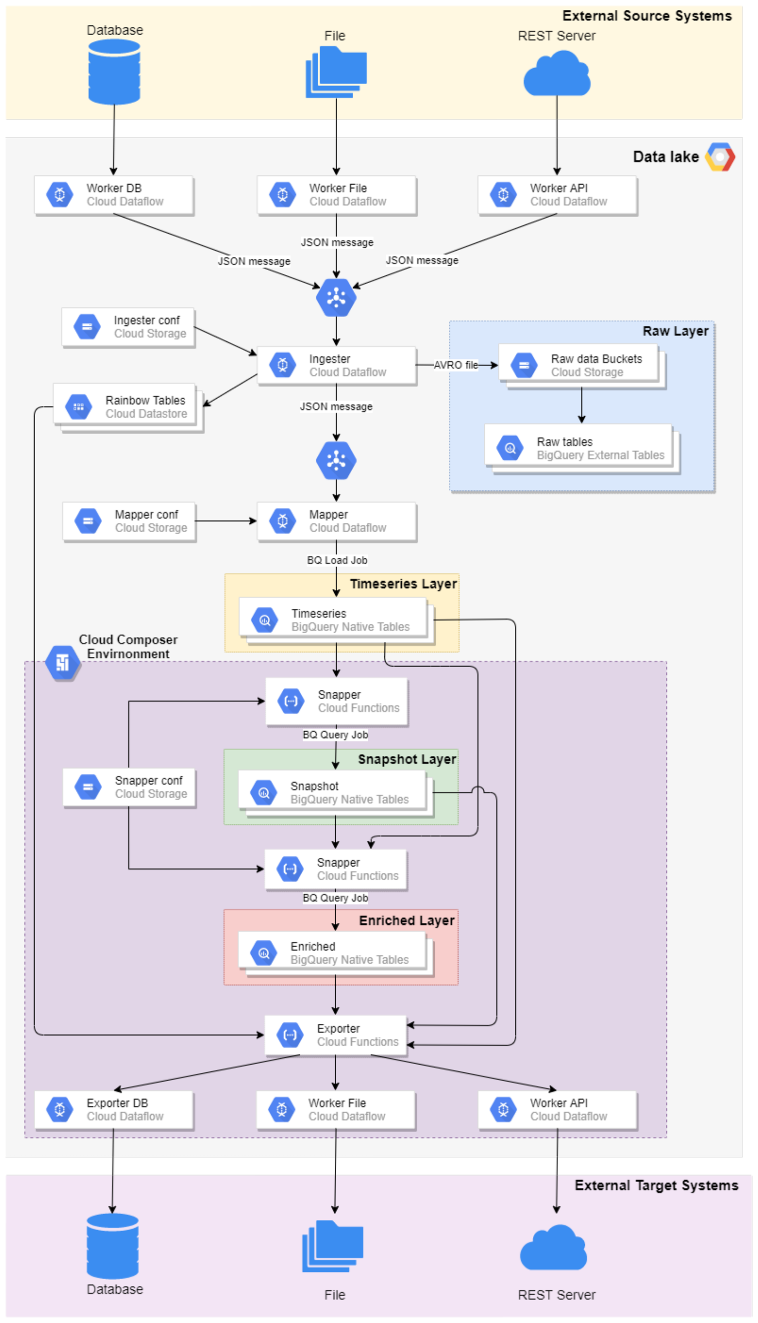


Figure 13: Architecture of data lake with GCP tools

5 Software components

In this chapter we will analyze how some of the components belonging to the data lake architecture have been implemented. In particular we will see the implementations of the components belonging to the **Processing** part: the Data Model Mapper, the Snapper and the Exporter.
We will see these components of the Processing phase as they are the ones I worked on personally within the project.

These components were developed with two different languages. The *Data Model Mapper* is a Dataflow job developed in **Scala**, while the *Snapper* and *Exporter* are Cloud Functions developed in Python language.

To implement **Apache Beam** pipeline (framework used by *Dataflow*) in Scala language, the **Scio**[7] library was used. Scio is an open source Scala API for Apache Beam and Google Cloud Dataflow. It's created by *Spotify* to process petabytes of data in both batch and streaming mode and is adopted by dozens of other companies as well.
With Scio it is possible to develop fully managed services (using Dataflow) and allows to create pipelines integrated natively with other GCP products such as Cloud Storage, BigQuery and Pub/Sub or with external and third-party products such as JDBC Connector, TensorFlow, Cassandra, Elasticsearch and Parquet I/O.

5.1 Implementation of Data Model Mapper

5.1.1 Collibra Crawler

Before analyzing the implementation of the Dataflow job let's focus on how the configurations on which the Mapper processing is based are generated. These configurations are created automatically by a **crawler**, developed in *Python*, which analyzes the fields mapped by the Data Governance team on Collibra and generates the configuration JSON file to be deposited on Google

[7]Scio Library: https://github.com/spotify/scio

Cloud Storage.

This crawler, through the **REST API** made available by the Collibra framework, obtains the list of fields (assets according to Collibra terminology) that are in the ”Ready for Prod” state. Status is one of the attributes of the assets, which is a metadata associated with the asset. Other attributes are the information related to the source system (table name and field name), the timeseries in which to save that field in the data lake and the normalization to apply to the field (optionally).
To filter only the assets that are in the ”Ready for Prod” state, it is possible to add a *query parameter* in the URI endpoint of the *GET* request to Collibra. This request returns a response in JSON format that contains the entire list of assets that meet the condition setted in the request parameters. Since this crawler works entirely in memory, requests are made paginated to avoid any *out of memory* issue during execution.

Once the list of elements to be added in the configuration is obtained, the response is parsed and the configuration JSON file necessary for the Mapper to perform its task is generated. After the deployment of these configurations in the Production environment, the crawler makes a *PUT* request to change the state of the asset and bring it to the ”Implemented” state. To carry out this last phase, however, it is necessary to know the id of the asset we are working on and for this reason the *collibra-id* attribute is added to the JSON configuration file which is not used by the Mapper but by the crawler itself.

As mentioned, this component was developed in **Python 3.7** using the standard language libraries: the **request** library was used for REST requests, the **json** library for generating the JSON file and finally for logging the **logging** library.

5.1.2 Dataflow Job

In this section we will analyze in detail how the Data Model Mapper component or more simply Mapper has been implemented. We remind you that the goal of this component is to process the AVRO file written at Raw level by *Ingester* on Google Cloud Storage with the aim of cleaning and normalizing the data which are then saved on native BigQuery tables.

The Data Model Mapper, as previously mentioned, is a **Dataflow streaming** job and to help us in the analysis of the implementation we study the DAG (directed acyclic graph) of the job automatically generated by Dataflow.

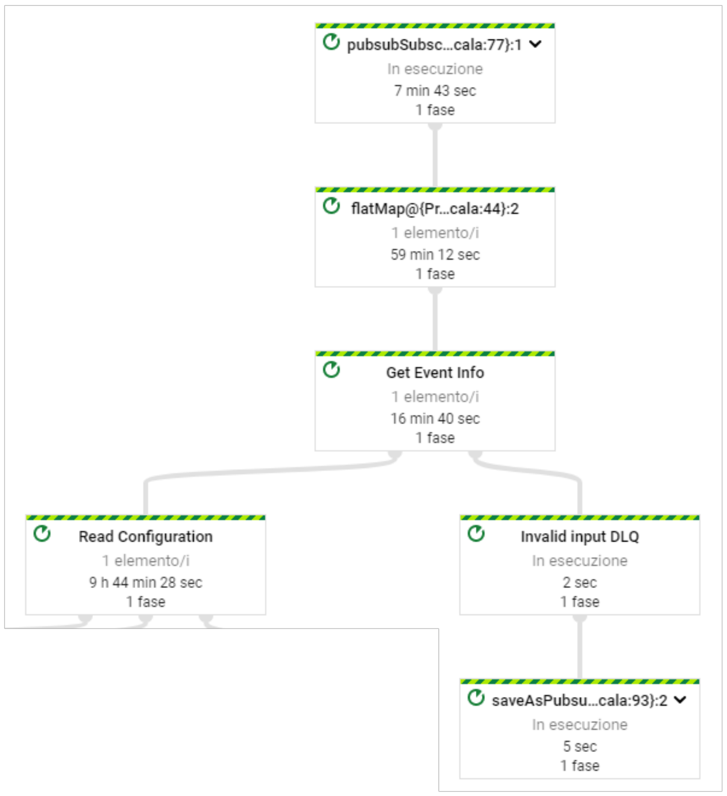


Figure 14: Data Model Mapper DAG - Part 1

Execution in the Mapper workflow is triggered when a message in JSON format is received on a Pub/Sub topic. The reading from the subscription associated with the input topic is the first node of the DAG. This reading operation takes place through the APIs made available by Apache Beam contained in the PubSubIO[8].
Below there is an example of the message received with only the mandatory fields necessary for the Mapper for its functionality (in the next paragraphs we will see other attributes that can be set to change the behavior of the

[8]PubSubIO Java documentation: beam.apache.org/releases/javadoc/2.4.0/

Mapper):

```
{
  {
  "schema-id": "source_system.source_table",
  "transaction-id": "example-message",
  "destinationName": "gs://bucket/file.avro"
  }
}
```

The information contained are the following:

- **schema-id**: unique identifier of the configuration and is composed of the concatenation of the name of the source system and the replicated table;

- **transaction-id**: uniquely identifies the replication job of the worker-ingesters (useful for monitoring the flow of data between the various components);

- **destinationNames**: path of the raw file on the GCS to be processed.

This message is then parsed to reconstruct the information received in a Java object and to store the metadata necessary for processing the file. This step is performed by the node named **Get Event Info** of the DAG.
The parsing functionality was implemented through the *Circe*[9] library and the coders and encoders are automatically generated by Scala for an object whose attributes are objects of the Java Standard Libraries (String, Int, ...).

In case the parsing is not successful, for example the JSON message is malformed, this is discarded by the normal processing flow generating a *Side-Output* in the Beam pipeline. This secondary output has the task of sending the message just received on a Pub/Sub topic dedicated to the DLQ (Dead Letter Queue) with the addition of the message contained in the exception raised by Circe. This message will then be sinked to GCS so these discarded messages can be easily analyzed and possibly re-launched after the fix of the error. The addition of the error message facilitate troubleshooting operations.

[9]Circe: is a JSON library for Scala and Scala.js (https://circe.github.io/circe/)

If, on the other hand, the parsing is successful, the object created is processed by the next node of the pipeline, that is the **Read Configuration**. The task of this node is to retrieve from *Google Cloud Storage* the configurations for the schema-id received in the triggering Pub/Sub message.
This information is contained in a JSON file within a configuration folder on a GCS bucket. The name of the JSON file to be loaded will be equal to the schema-id value associated with the file to be processed. In this way the complete path to be read is easily calculated as the concatenation of the configuration files folder and the schema-id received.
To interface with the GCS buckets and read the files to be processed, both for the component configurations and for the raw layer data to be process, the Java APIs distributed by *Google* itself are used within the Java client for GCS .

Below is a sample configuration JSON file:

```
{
  "sourceTable": "source_table_name",
  "destinations": [
    {
      "dataset": "dataset_name",
      "table": "table_name",
      "mapping": [
        {
          "from": "original_field_name",
          "to": "field_business_name",
          "normalization": "normalization_type",
          "partition_date": true
        },
        {
          "from": "original_field_name_2",
          "to": "field_business_name_2"
        }
      ],
      "filter": "filter_condition"
    }
  ],
  "schemaVersion": 1,
  "configVersion": 1
}
```

In these configurations for each source structure we have the list of destination tables in the *Timeseries* layer, we can map the same raw side table on one or more timeseries. For each destination table there is the mapping of the fields between the two layers, the normalization and partition date configurations that we have already analyzed in chapter 3. In addition, the filters to be applied to the data before writing them on the destination tables are also specified.

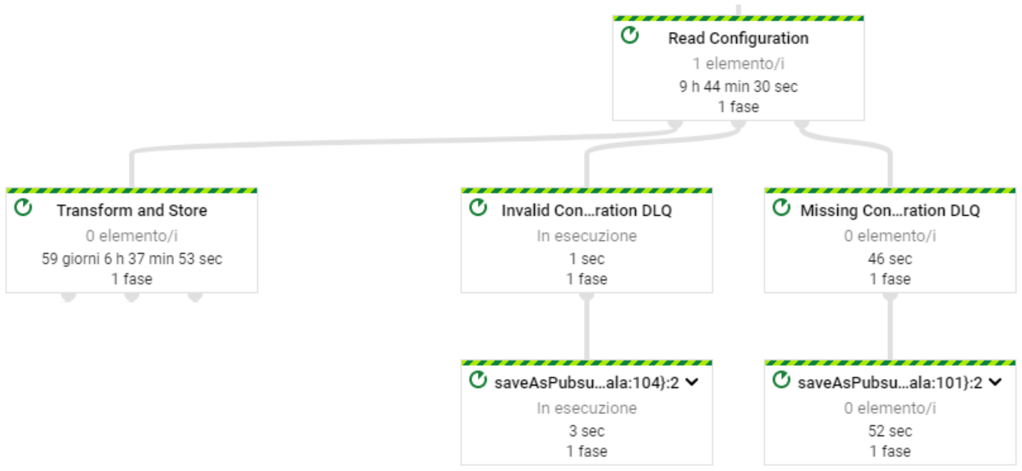


Figure 15: Data Model Mapper DAG - Part 2

After reading the JSON file, also in this case the parsing is performed using the JSON decoder to generate a Scala object that contains all these metadata associated with the stream. If the parsing is successful we can move on to transform the raw file, in case of failure we have the creation of another two *SideOutput* depending on the error found.
If the configurations are read by the Mapper but there are errors in the deserialization phase (example malformed JSON file) the input message received is sent to the topic of the Dead Letter Queue of the component, if instead the Mapper has not found the configurations associated with the schema-id the SideOutput will publish the message on a second DLQ topic in which

there will be only messages referring to schema-ids not yet mapped between raw and timeseries.

This division of the DLQ makes it easier to retrieve data already present in raw when a new structure is mapped at the Timeseries level since these messages are in a dedicated DLQ. In both cases, however, the error message raised by the Mapper is added to the Pub/Sub message received by the Ingester in order to analyze, if necessary, the reason that led to discard the message.

If, instead, the reading of the configuration is successful, you can start with the processing of the AVRO file present in raw.
The first step is reading the AVRO file from GCS which is done through the AVRO Java library[10] which allows us to serialize and deserialize easily received data from the source. The records read from the file are then deserialized into an object of type **GenericRecord**. This type of object allows you to associate a schema indicating the name and type of a field to the record data so that you can access it easily.

The first step is to clean up the records within the file according to the filter rule specified according to an SQL rule within the configuration. If the record complies with this filter condition, it continues processing otherwise it is discarded. This check is done using the *prestosql*[11] library.
Then we move on to remap the data contained in the original GenericRecord, therefore with the names of the source fields, in the new GenerciRecord which will then have the new schema with the names of the modified fields and with the addition of all the normalization fields required for the flow. After creating this new GenericRecord it is filled with the corresponding values present in the original record and with the new fields calculated through the normalizations.
After the creation of the new record this is written to a temporary AVRO file which will then be needed to load the data on the timeseries layer with the BigQuery API.

[10]Avro: is a data serialization system (https://github.com/apache/avro)

[11]prestosql: is a fast distributed SQL query engine for big data analytics (https://github.com/trinodb/trino)

All the processing on the GenericRecord list of the AVRO file is managed through Java Iterator in order to never materialize the entire sequence of data in memory but going to process the single record from time to time, managing the occupation of resources in terms of memory.

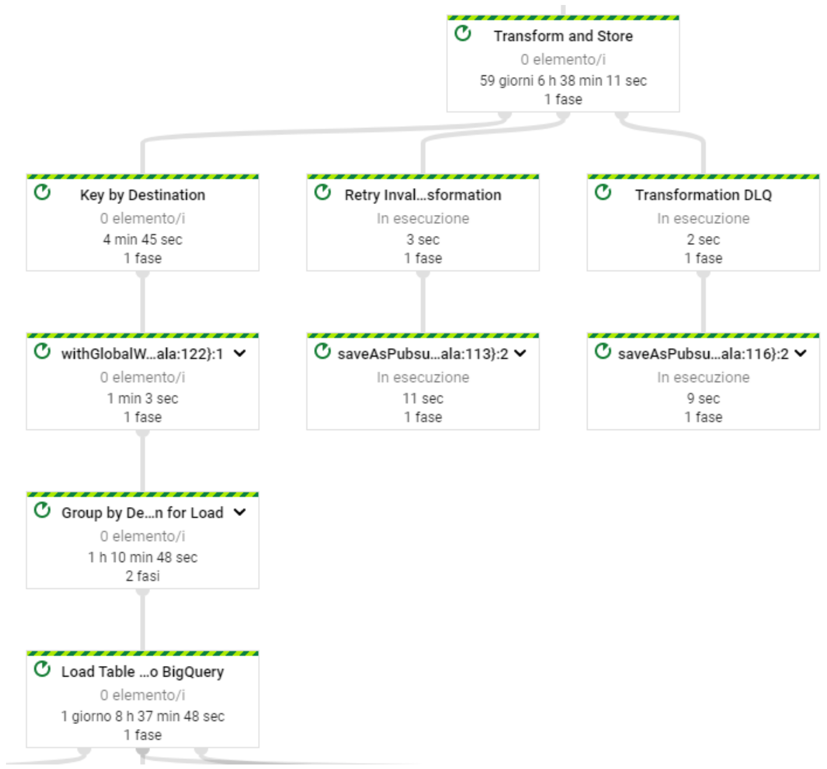


Figure 16: Data Model Mapper DAG - Part 3

At the end of the data processing operations the output, if successful, is an object of type **LoadJob** which contains all the information necessary to load data on *BigQuery*, i.e. the destination table and the path of the temporary AVRO file to load. In addition to this data there are also some metadata such as the number of lines contained in the file, the number of partition_date present in the file and the minimum and maximum value of these partitions. This information will then be used to monitor the platform and the correct reception of data from the source systems. Monitoring will be an important section of the next chapter.

If there is an error in this phase, the Pub/Sub message is sent back to the Mapper in order to retry the execution. A new attribute called *retry* is added to the original message and initialized to value 1. In the next run, if the error occurs again, the value of this attribute is increased by 1.
On this step it was decided to apply a retry policy as there are numerous interactions with external tools, such as *Google Cloud Storage* which interacting via the network could often respond with temporary errors.
The retries are repeated a limited number of times whose value can be specified parametrically during the component deployment phase. After exceeding this number of attempts, the Pub/Sub message is not sent back to the Mapper input topic but rather to the DLQ topic so it can then be analyzed for a possible recovery. Also in this case the error message raised by the component is inserted in the message sent in DLQ.

Furthermore, in this phase, in case of problems only on limited number of records (i.e. malformed dates in the record) the entire file is not discarded but only the records affected by the problem continuing to process all the other data. To ensure that during the data recovery phase only these discarded data are processed in the Pub/Sub message a list of unique record identifiers is added. So during the reprocessing phase the Mapper will process only these records avoiding the creation of duplicates on the layer *timeseries*.

As mentioned before, if the processing takes place correctly, for each message received by the *Ingester* an object is created which is useful for creating a **LoadJob** on BigQuery. The Loadjob is the API provided by Google to batch upload data via CSV or AVRO files to BigQuery. In our architecture we have chosen to load the data in batch mode instead of streaming because this loading mode has no cost and is free unlike streaming insert. In addition, I remember that for all those use cases that require real-time data, the correct layer to use is *raw* and not *timeseries* and therefore having latency times in data propagation does not create problems.
Against the free cost of uploading, however, Google imposes limits on these uploads called **quotas**. In particular in our case the quota to be taken into consideration are the following:

- **Load jobs per table per day**: 1500 (including failures);
- **Maximum number of partition modifications per column partitioned table**: 30000;

- **Maximum number of source URIs in job configuration**: 10000 URIs;

To respect the first two quotas, the load jobs are grouped into a single load job on the basis of the data destination table and executed every 10 minutes (parameterizable value during the component deployment phase). In this way, in a single load job, multiple files are loaded, avoiding to exceed the number of loads available daily for a table and in addition it is possible to group multiple changes on the same partition in a single load by increasing the counter of changes of a single unit.
However, by grouping multiple loads into a single LoadJob, there is a risk of exceeding the limit of 10,000 maximum files for a single loading job (especially in the case of historical load of a table in which the amount of data to be loaded is very high) and for this reason when the uploading job to BQ is created, all the files to be uploaded are split into groups of a maximum of 10000 files.

The windows concept of *Apache Beam* was used to create the 10 minute micro-batches on upload to *BiqQuery*. In particular, **fixed window** have been introduced, i.e. windows of fixed time duration (in our case default 10 minutes) built using the destination table on which to load the data as the key of the window. This means that the processing of messages to different destination tables will fall into two different windows. The time start of the 10 minutes is given by the reception of the first element in the window and the trigger to start the LoadJob is the end of the 10 minute wait. This means that the start of the LoadJob towards the BigQuery structures is not synchronized but is random. This also avoids overloading problems on the BigQuery component.
The splitting by destination table is made in step **Key by Destination** of the DAG, while the time window is built in step **withGlobalWindow**.

At the end of the time window the API request is then sent to *Google BigQuery* to load the data and this call is made through the Java client created by Google[12]. The request to BigQuery is asynchronous and made through a **Future**[13] which executes the request in a parallel thread and

[12]BigQuery API Client Library: https://cloud.google.com/bigquery/docs/reference/libraries

[13]Futures provide a way to performing many operations in parallel in an efficient and non-blocking way. A Future is a placeholder object for a value that may not yet exist.

then is reactivated through a Callback function when the execution on the BigQuery side is over.

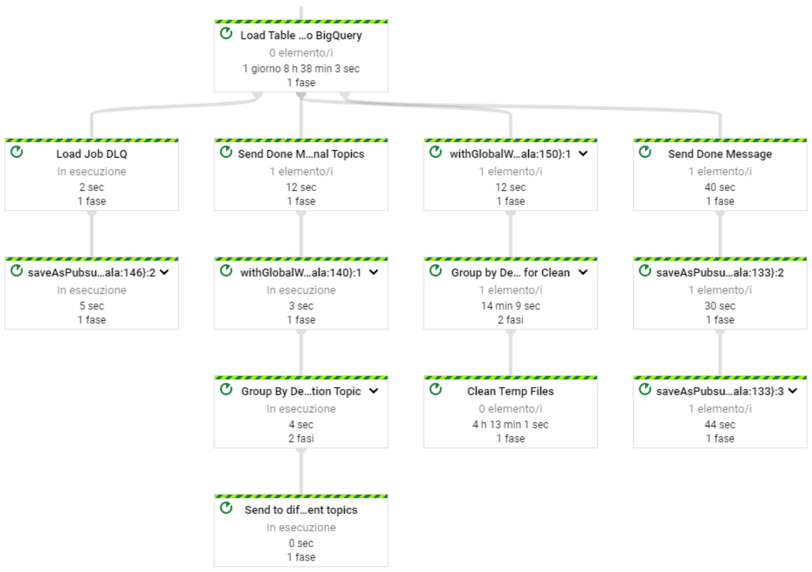


Figure 17: Data Model Mapper DAG - Part 4

If the processing of the LoadJob by *BigQuery* is not successful, also in this case, the pipeline ends with a message in DLQ that contains the same information as the original message received by the *Ingester* plus the information of the error message returned by BigQuery, useful for debugging the problem and solving it. The creation of this Pub/Sub message to be sent is performed in the step named **Load Job DLQ**.
If, on the other hand, the processing ends successfully, the Mapper will have to perform two more tasks before completing its pipeline: delete the temporary AVRO file it created and send the done messages for the subsequent components of the pipeline.

For the deletion of temporary files, to limit the number of requests made on the GCS these are performed in batches every 10 minutes (this time is also configurable) and in this case the references to the files to be deleted

Generally, the value of the Future is supplied concurrently and can subsequently be used. Composing concurrent tasks in this way tends to result in faster, asynchronous, non-blocking parallel code.

are grouped through the use of a fixed Window. The request to delete files is performed in the **Clean Temp Files** step.

The last task of the Mapper is therefore now to send the done messages. By default this message is sent on a dedicated topic and contains as information the source table from which the loaded data derive, the reference to the raw file and the transaction-id of the operation. A done message is then sent for each message received by the *Ingester*. The branch of the DAG that deals with sending this message is the one called **Send Done Message**.
In addition to the message on the standard topic, however, it is possible to send the done message also on other topics on which dedicated components are listening for processing certain flows. These topics are configurable in the conf file of the Mapper and for each destination table in *timeseries* layer it is possible to set one or more destination topics. The sending of these messages is performed in the step **Send to different topics** which, as we can see, is preceded by a time window whose aggregation key is the arrival topic. This window allows us to accumulate multiple messages to be sent on the same topic (even for different destination tables) in order to open a single sender to Pub/Sub and optimize performance by avoiding the overhead introduced by the creation and closing of the sender for each message.

The whole component, in every step, logs the operations performed and a series of metrics useful for monitoring the component. The log library used is **Log4j**[14] and the output of the Log messages is captured by the component *Logging* of the GCP Platform so that the logs of all components are centralized in one place.
An example of logged metrics are the number of lines written in a timeseries, how many partitions have been modified after a loadJob to BigQuery or how long the Mapper took to process a file received by the Ingester. Through this information we are able to check the performance of the component at any time and verify the exceeding of any BigQuery quotas. All these aspects will then be explored in the next chapter.

[14]Log4j: library to handle logs in Java environment: https://logging.apache.org/log4j/

5.2 Implementation of Snapper and Exporter

In this paragraph we will analyze how the *Cloud Function* of the **Snapper** and **Exporter** components have been implemented. We can analyze them at the same time because the flow of components is the same, with the only difference that the Snapper writes the result of the calculations into a BigQuery table while the Exporter generates a CSV file. For simplicity, in the rest of the paragraph we will analyze the case of the Snapper and at the end we will highlight the few differences present in the Exporter.

The goal of these components is therefore to execute an SQL construct to generate output data. To obtain the best possible performance, it was decided to use the BigQuery *Query Job*, avoid the phase of exporting data from BigQuery necessary in the case of processing outside it.
To carry out the processing, we chose to use the **Google Cloud Function** tool because the processing is very simple and it consists simply of three phases:

- reading of the configurations and calculation of the parameters inside the query;
- creation of the query job on BigQuery;
- check the result of the query job.

One of the limits of the Cloud Function is the maximum execution time of the same, in fact, Google automatically ends the execution if it exceeds 9 minutes in duration. This obviously is a problem if the materialization job on BigQuery is particularly complex and long in execution. To overcome this limit, a chain of Cloud Function has been created in cascade in which each one performs one of the steps listed above. The three Cloud Functions are therefore **Snapper-Starter**, **Snapper** and **Snapper-Checker**.

The first Cloud Function, the **Snapper-Starter**, is triggered by sending a Pub/Sub message on a topic on which the Cloud Function itself is listening. This message is generated by an orchestrator (*Cloud Composer*) that can send it either on a time basis or when all the start conditions of the calculation are verified (example wait the end of the calculation of the source tables). This message contains a reference to the table to be calculated which is simply formed by the concatenation of dataset and table in which to materialize the

data.
The goal of this cloud function is to generate some parameters useful for the calculation, and in particular two timestamps:

- *operation_timestamp*: timestamp in which the cloud function was triggered;
- *start_timestamp*: timestamp in which the previous run of the function started.

These two timestamps are essential for all queries that work in delta mode, that is for all those calculations that at each run must analyze only the new data arrived after the previous run. This information is read from a tracking table of the executions stored in BigQuery in which the last component of the chain (the *Snapper-Checker*) in case of successful calculation processing traces the activity indicating all the parameters used for the run.

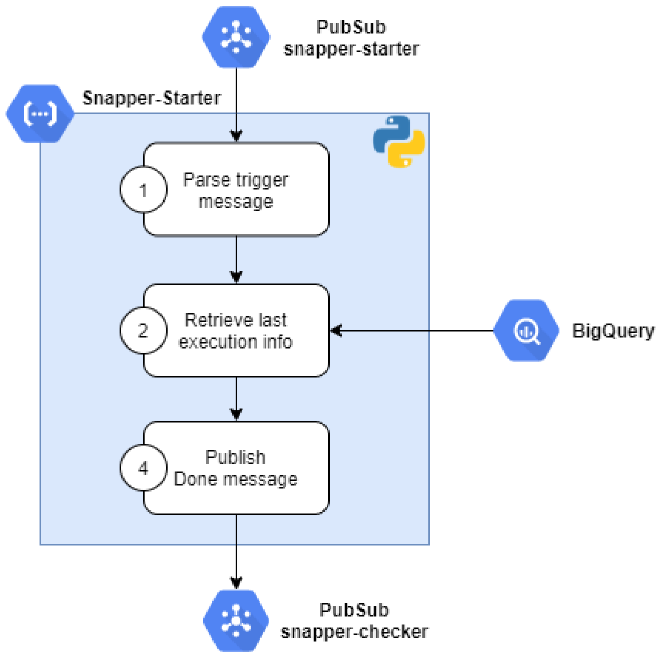


Figure 18: Snapper Starter workflow

Once this parameter definition activity has been completed, the Cloud Function ends by sending a new Pub/Sub message on a second topic which, in addition to the reference to the calculation flow, contains the values of these **query_parameters**. Listening on this topic there is the second Cloud Function of the chain, that is the **Snapper**. Below the workflow of this second step of the chain:

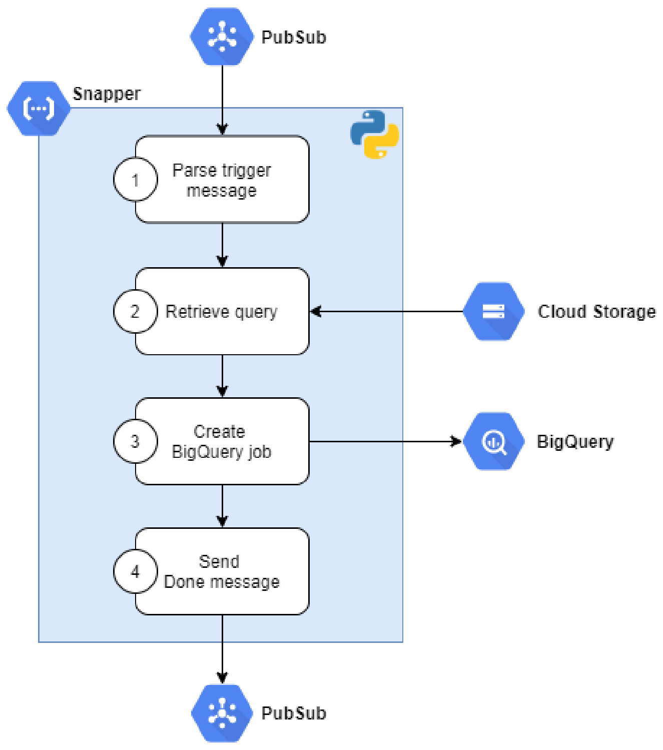


Figure 19: Snapper workflow

The target of this second Cloud Function is to launch the calculation necessary for the materialization of the data. To do that, the information received in the triggering Pub/Sub message are parsed and on the basis of these it retrieves the query to be executed (from Google Cloud Storage) and it replaces the placeholders *@parameter_name* with those received in the Pub/Sub message.
The run of the job and the substitution of the parameters are performed through API calls towards BigQuery generated with the Python client made available by Google. In fact, when we create the **Query job** we can define the destination table, indicate whether the new data should be written in

append or in truncate/insert mode in the table and which parameters to replace.
The job generated on the BigQuery side is executed *asynchronous* and the response to the request to the creation of the job is the **id** assigned to the job by BigQuery. Through this id it will be possible to subsequently check the status of the job to determine if it is still in progress or if it is finished and also check the outcome.

The last task of the Snapper is therefore to generate a new Pub/Sub message towards a third topic in which there is the same information as the message it obtained in input plus a new attribute that contains the job-id returned by BigQuery. This message will trigger the last Cloud Function of the chain, that is the **Snapper-Checker** which has the task of verifying the outcome of the data materialization query job performed by BigQuery.
Below there is the workflow of this Cloud Function:

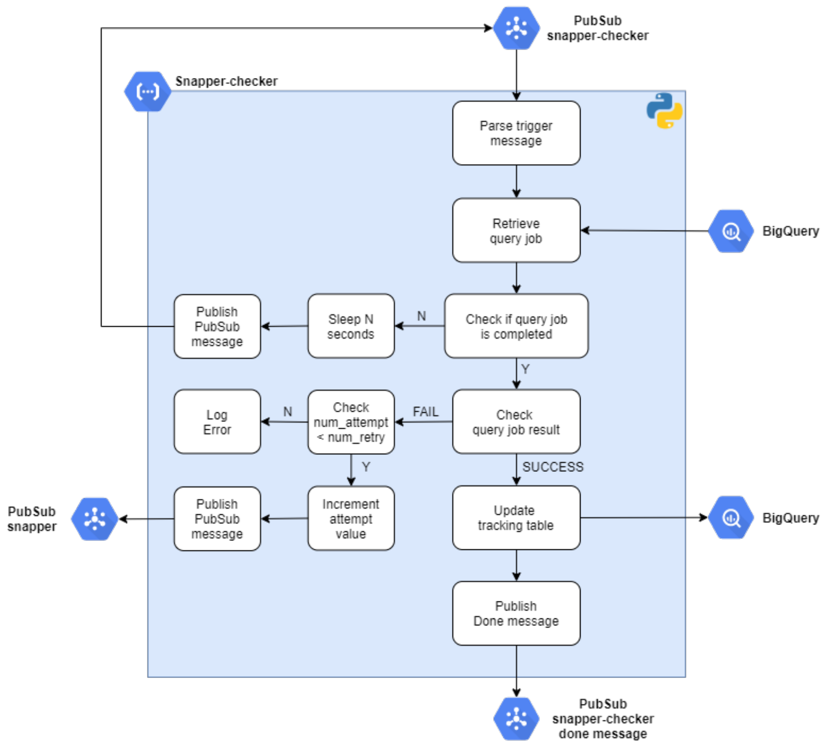


Figure 20: Snapper-Checker workflow

The goal of this last Cloud Function is to verify the outcome of the materialization job performed by BigQuery. To do that, it retrieves the **job-id** from the Pub/Sub message and requests the job status via the API made available by the BigQuery Python client. The first check verifies if the job has finished or is still in progress.
In the case that the execution is not yet finished, the Cloud Function goes into sleep for 30 seconds (value configurable during the deployment phase of this component) and when it wakes up it sends the same Pub/Sub message received on its trigger topic and then ends the execution. In this way, in the case of very long jobs, there is no risk of exceeding the maximum execution time of 9 minutes imposed by Google.

If, on the other hand, the execution is finished, the result of the job is checked. If the job ends successfully the *Snapper-Checker* will insert a new row in the tracking table on BigQuery, marking the instant in time in which it started and the value of the *start_timestamp* parameter useful for *Snapper-Starter* for the next run of the calculation and finally publishes a message on a Pub/Sub topic in order to trigger, if necessary, a subsequent component.
If the execution, however, ends with a failure status, the Snapper will be re-triggered to perform some retries. The retry number is traced by adding an attribute within the Pub/Sub message sent to the Snapper which is incremented by one unit at each execution. The maximum number of retries is by default equal to 3 but it is possible to configure this value when deploying the Cloud Function *Snapper-Checker*. Once the retry limit has been reached, the Cloud Function will track the error in the log, reporting the error message returned by BigQuery in order to allow troubleshooting of the error.

All these Cloud Functions use the standard Python library **Logging** for logging, using 3 different severity levels (INFO, WARNING and ERROR). Through these logs it is possible to control the execution of a run of the calculation or monitor the metrics of the various executions, such as the time required for the execution, the number of bytes moved (useful for calculating the cost of the executions) and the number of written rows on a table at the end of a calculation run.

In the case of the **Exporter** the differences compared to the *Snapper* are the topics, the tracking table used and the type of job performed on BigQuery. In fact, the API request will aim to generate a job of type *Export* and not *Query*. With this type of job it is possible to indicate how to write the data resulting from the calculation in a file saved on *Google Cloud Storage* by specifying the format and in the case of CSV the separator character. The starter of the data unmasking pipeline is listening on the topic outgoing from the Exporter (in case of successful execution) and will unmask the data present in the generated files before sending them to the target system.

6 Operation e monitoring

In the first part of this chapter we will analyze how the new developments of components or new configurations are released in the Production environment. The deployment on Production is completely automated and must comply with a series of steps and standards that guarantee the correctness and backwards compatibility of the new solution. To implement these requirements, **CI/CD** solutions have been implemented that allow you to analyze the code/configurations present on the repositories (GitHub) and then upload the new software to the Google Cloud platform without creating any kind of service interruptions.

In the second part of this chapter we will instead analyze how the platform is monitored in all its aspects to discover in the shortest possible time the presence of errors or blocks that are causing a disservice to customers. To achieve that, a system of **dashboarding** of performance metrics and a system of **alerting** based on thresholds has been developed which automatically alerts one or more people in the on-call team of the presence of situations to be monitored or restore.

6.1 CI/CD Pipeline

The CI/CD is a method for *frequent distribution* of applications, which involves the introduction of **automation** in the development stages of the application. Primarily, it is based on the concepts of continuous integration, distribution and deployment. More specifically, the CI/CD method introduces constant automation and continuous monitoring throughout the application lifecycle, from integration and testing to deployment.

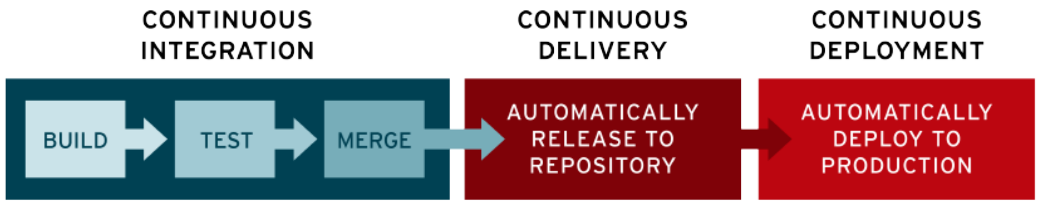


Figure 21: CI/CD Flow

By adopting **continuous integration**, developers can roll back code changes to a single branch that is shared more frequently, sometimes even daily. Once merged, the changes are validated by automatically compiling the application and running several levels of self-testing, typically unit and integration tests to ensure that the changes have not caused any error.
Following build automation and CI integration and unit testing, **continuous delivery** automates the release of validated code to a repository. As a result, for the continuous delivery process to be effective, it is important that the CI is already integrated into the development flow. The goal of continuous distribution is to have a base code that is always ready to be deployed in a production environment.

The final stage of a mature CI/CD stream is **continuous deployment**. As an extension of continuous delivery, which automates the release of a production ready build to a code repository, continuous deployment automates the release of the app into production. With no manual blocking of the flow stages prior to production, continuous deployment must rely on well-designed test automation. In practice this means that the change made by a developer to the application can become active within *a few minutes* of its writing, provided it passes the automated testing phase. Thanks to this method, receiving and integrating the feedback sent by users on a constant basis is easier.

The tool with which the CI/CD pipelines within the data lake were built is **Jenkins**[15]. Jenkins is an open source continuous integration server written in Java to orchestrate a chain of actions to achieve the continuous integration process in an automated way. Jenkins supports the entire software development lifecycle from software creation, testing, documentation, distribution and other phases of the software development lifecycle.

All the components that have been developed within the data lake in addition to the source code have **Unit tests**, that is, runs of the entire component code or a particular functions to verify its execution.

[15]Jenkins: https://www.jenkins.io/

In this way, when a code is patched by running these tests, it is possible to quickly and effectively verify the backward compatibility of the changes. Obviously, when adding features to the code, new unit tests must be created in order to cover as many lines of code as possible.

All these checks are performed with the **SonarQube** software[16]. SonarQube is an open source platform for code quality management. It can be summarized as a web application that produces reports on duplicate code, programming standards, unit tests, code coverage, complexity, potential bugs, comments, design and architecture. In our case the SonarQube analysis is triggered when a new **Pull Request** of a branch on GitHub is opened to merge it into the master. This analysis runs all unit tests of the modified component and also evaluates if the newly inserted lines of code are tested or not. In the event that at least one test is not successfully passed or if the test coverage of the new code is below the 80%, the Pull Request cannot be merged into the master and therefore it will be necessary to fix the problem or add new tests before be able to merge the PR.

In addition to the unit tests in the Pull Request phase, the **Integration tests** are also performed through Jenkins jobs. In this case, the new version of the code or configuration is deployed in the *Dev* environment and triggers the execution of the entire pipeline of the data lake to verify that a given input corresponds to the expected output. The possibility of merging is also subject to the successful termination of the integration test. In this way you can be sure that following a modification to a component or a configuration, the interfaces between the various modules of the platform will continue to work correctly.

When all the automatic tests have been passed, a member of the development team (other than the one have implemented the change to be merged) analyzes the Pull Request and if after this review the code is considered correct and mature to go into Production, the PR is approved and merged.
Now begins the phase of *continous delivery* proceeding with the creation of

[16]SonarQube: https://www.sonarqube.org/

the versioned code tag and with the build of the components. This new version is then automatically deployed in the *Test* environment where all the integration tests are once again performed but in addition, being an environment similar to the production one, the **Performance Tests** are also executed of the new version of the components to make sure that the performance of the data lake has not suffered a degradation. The metrics evaluated for these tests are the data ingestion rate, the write rate to GCS or BigQuery, and the data average elapsed time in the entire pipeline.

At the end of the tests in the *Test* environment, if passed successfully, you are ready to deploy in the Production environment. Before this last step, however, it is necessary to obtain **approval** at the release by the Product Owner of the product. Depending on the risk level of the change, one to three approvals are required. This approval is requested through a ticketing system such as **ServiceNow** on which a *Change Request* is automatically created at the end of the tests with the release note of the changes to be released.
When all the requested owners have given their approval, the *continous deploy* process is automatically triggered in the Production environment, this ending the CI/CD process that allows you to bring in a completely automatic way and without intervention manual by the development team members the code and new component configurations from the developer's local development environment to the Production cloud environment minimizing the possibility of introducing bugs or regressions into the platform. Below the schematic of the workflow:

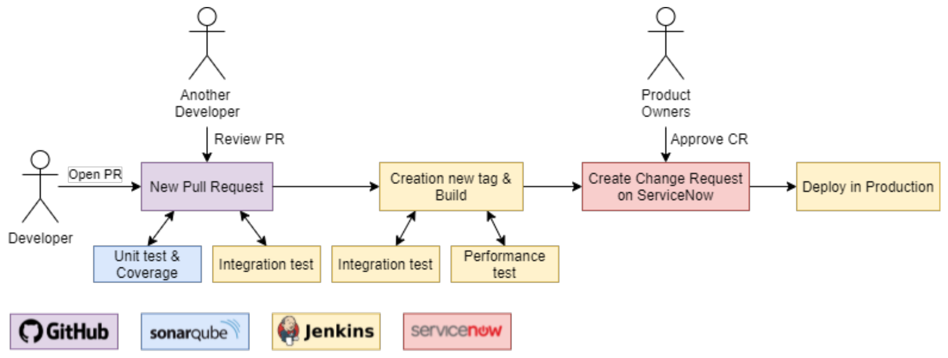


Figure 22: Data Lake's CI/CD Workflow

To interface the Jenkins jobs with the Google Cloud tools, the APIs made available by Google with the Python clients for Google Cloud Storage, Pub-/Sub and Big Query were used. As for the automation on the infrastructure and on the management of resources in the cloud, **Terraform**[17] was used as a tool that allows you to manage the deployment of the infrastructure through descriptive files.

6.2 Monitoring and alerting

An important pillar of a platform of this type, on which business choices are based, is the monitoring of the performance and status of the database when the presence of incomplete or, worse, incorrect data can lead the business users to make error in strategic choices or in incorrect marketing campaign. For this reason, each component that makes up the data lake is monitored at the level of performance and errors generated in order to intercept in the shortest possible time the presence of any disservice and communicate them to users. In addition to a technical control of the components, it is also present a semantic control on the data to ensure that all the calculated KPIs exploit updated data, a fundamental requirement for all use cases that require data in real time.

As already said in the previous chapter, all components log the details of their executions and some useful metrics to monitor the status of the platform on the Google **Logging** tool. In this way all the logs coming from the various components are stored in a single point and easy to find and consult. This tool also allows you to sink the logs on other tools of the GCP platform. In our case, all these logs are sinked on BigQuery tables in order to store them in a queryable way through SQL language.

Each component exposes different metrics depending on its goal and work. One of the most important metrics (and which is exposed by all components) is the processing time of a given file (identified through the **transaction_id** information common to all components for the same file). In this way, by linking through SQL queries all the execution times of the various components for the same transaction_id, it is possible to define the total elapsed time of a file within the entire pipeline. Using this information, analyzing it

[17]Terraform: https://www.terraform.io/

on all the processed files, it is possible to understand if a slowdown is being created on some component. When this crossing value, as we will see later, exceeds a critical level, automatically is triggered an alert that engages one of the people in the on-call team in order to analyze the situation and implement a solution to resume regular processing.

On the *Data Model Mapper* component the other metrics exposed, besides the elapsed time, are:

- number of lines written by a load job: useful to understand if the workload on the component is constant or if there is a spike that requires attention;
- number of partitions modified by a load job: useful for checking the achievement of the quota set by Google on the maximum number of daily changes that can be made on a partitioned table;
- maximum value of the field used to temporally partition the timeseries contained in the processed file: this information is useful to check if the data we are receiving from the source systems is in real time or if there is a delay.

As for the *Snapper*, instead, the metrics exposed are the following:

- job execution time: necessary to understand if the duration of the query necessary to materialize a snapshot or to calculate a KPI increases or remains constant;
- number of lines written: to monitor if the cardinality of the table remains constant or increases making it unmanageable and difficult to maintainable;
- quantity of bytes processed by the query: useful for estimate the execution costs of a materialization job (BQ jobs billing is based on the amount of data processed).

The monitoring architecture, in addition to the Logging and BigQuery, is based on two other tools (external to the GCP platform) such as *Prometheus* and *Grafana*.

Prometheus is an open source monitoring tool that allows you to store in an internal database the time series that represent the value of a metric at a given moment. These DBs can then be queried in real time allowing the analysis of these data. All the previously analyzed metrics are then parsed from the logs present in BigQuery and saved in Prometheus.
Starting from these time series, dashboards have been created using **Grafana** which show the real-time status of the platform, and also allowing to analyze the status of the same metrics at a specific time or in a past period.

Here two examples of dashboards created:

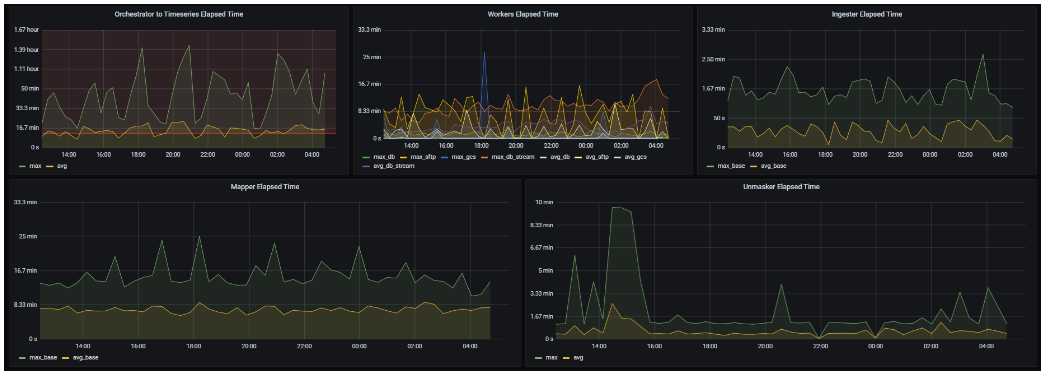
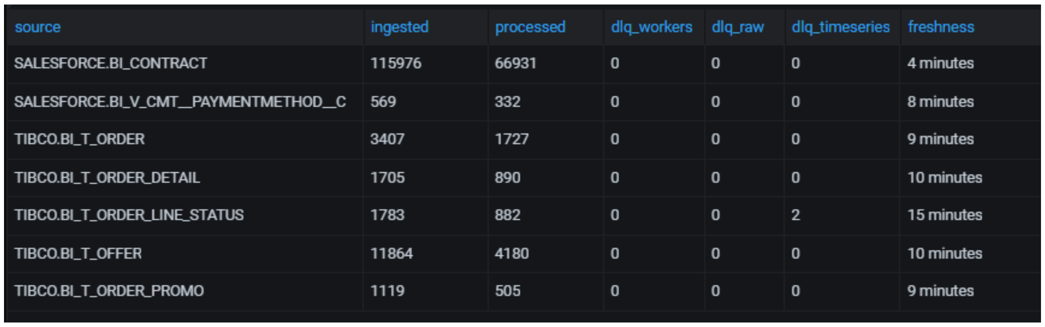


Figure 23: Example of dashboard to check component metrics



source	ingested	processed	dlq_workers	dlq_raw	dlq_timeseries	freshness
SALESFORCE.BI_CONTRACT	115976	66931	0	0	0	4 minutes
SALESFORCE.BI_V_CMT__PAYMENTMETHOD__C	569	332	0	0	0	8 minutes
TIBCO.BI_T_ORDER	3407	1727	0	0	0	9 minutes
TIBCO.BI_T_ORDER_DETAIL	1705	890	0	0	0	10 minutes
TIBCO.BI_T_ORDER_LINE_STATUS	1783	882	0	0	2	15 minutes
TIBCO.BI_T_OFFER	11864	4180	0	0	0	10 minutes
TIBCO.BI_T_ORDER_PROMO	1119	505	0	0	0	9 minutes

Figure 24: Example of dashboard to check data ingestion

The first dashboard is an example of monitoring, through a time series, of the value assumed by a metric exposed by the various components. In this case the monitored metric is the data processing time in the various components (*Workers, Ingester, Mapper and Unmasker*) and the total average elapsed time of the entire pipeline. In this way it is possible to analyze how this metric changes over time and analyze any useful patterns to better manage the scalability of the components. Furthermore, through the Grafana functions it is also possible to filter a subset of sources to be analyzed, a useful feature when you want to analyze the processing times of all the data coming from a given source.

The second dashboard, on the other hand, does not analyze technical aspects of the component but allows to check internal aspects of the data that is being ingested in the datalake, such as the number of read/written lines, how many lines have been discarded by the various components and the freshness of the data for the various flows involved.

To monitor the costs of the platform, dashboards were created using the tool integrated in GCP **Looker**. Looker is a Business Intelligence platform that allows you to perform analyzes and create dashboards on data saved in different tools of the GCP platform (Cloud Storage, BigQuery, MySQL, metadata of the different tools), but also in third parties storage such as S3 buckets on the platform *AWS*.
In our case through Looker we collect the metadata of the various tools to monitor the storage costs on Google Cloud Storage and BigQuery, the processing costs on BigQuery (execution of queries and scheduled jobs) and the costs of using the Virtual Machines necessary for the Composer instance, the different Dataflow jobs and Kubernetes cluster to manage Prometheus and Grafana. In this way we have the opportunity to monitor the costs of the platform and understand where to optimize to reduce costs.

In addition to monitoring via dashboard, for the most sensitive metrics, there is also a **alerting** system that automatically notifies one of the members of the on-call team of the presence of an anomalous situation. This system is based on thresholds applied to the metrics which, once exceeded for a certain period of time, trigger the alarm. This management is done through a dedicated Prometheus component, **Alertmanager**.

When an alarm is triggered, this is notified on the Slack channels of the teams responsible for the component that generated it, and a ticket of the type *Incident* is also opened on the **ServiceNow** platform with a description of the problem and where the solver will detail all the steps that were necessary to bring the metric below the critical level. When a ticket is created, one of the engineers of the on-call team responsible for the component on which the alarm was triggered is also notified by telephone, guaranteeing a control on the platform 24/7.

The overall architecture of the monitoring and alarm system is as follows:

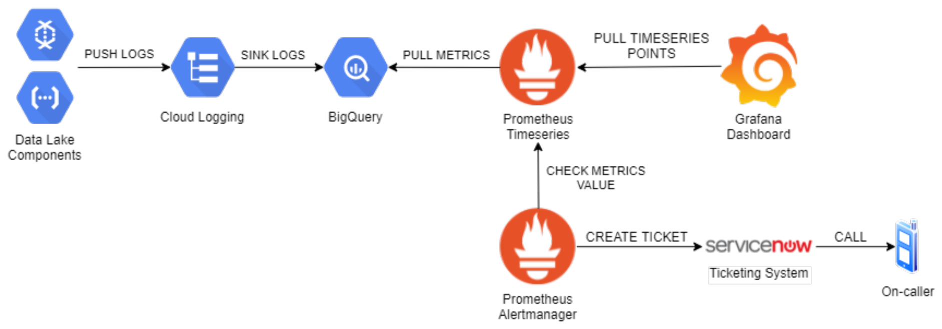


Figure 25: Data Lake's monitoring architecture

7 Conclusions

In this thesis I have analyzed my work experience in NTT Data to create a Data Lake in the Cloud by exploiting the potential and overcoming the limits of Google Cloud Platform. The project led to the creation of a complex infrastructure and architecture that allows to comply with all the functional and technical requirements requested by the client.

Here some statistics of the data lake at the time of writing the thesis:

- source systems: 90
- structures is RAW layer: 658
- structures in timeseries: 315
- snapshot tables: 71
- enriched tables: 105
- export flows: 90

All these configurations have been deployed in Production in about a year of the life of the project. This was made possible thanks to the simplicity of component customization and the deployment speed made available by the CI/CD pipeline. In fact, in a few days of work it is possible to pass from the requirements defined by the users together with the team of analysts to the release in production of the new flow.
In a project of this type, where the software created having a fundamental strategic importance in the choice of business decisions of the client, the speed and immediacy in responding to needs is fundamental.

These requirements within the project have been centered by creating highly configurable software that therefore requires a limited number of software changes when there are new requirements to be implemented and above all thanks to the potential of the cloud. In fact, by exploiting the scalability of the resources made available by the Google Cloud provider (workers in job Dataflows or Cloud Function executions) it is possible at any time to add new processing flows without any problem abount the resource availability.

Another fundamental aspect for platforms of this type is reliability. Business users who use the data present in the data lake for their strategic business choices must have access to the data at any time and be sure of the correctness and freshness of the data they are querying.
Also in this aspect, the use of a Cloud environment simplifies the management of the infrastructure and the availability of resources is demand to the provider. Furthermore, in the cloud environment, if one of the workers on which, for example, our Dataflow job is running should have a fault and no longer be usable, Dataflow will pull up a new worker and automatically resume execution.

In addition to this, however, it is still necessary to monitor the situation of the platform at all times to detect possible issue as quickly as possible with the aim of informing business users of the disruption and at the same time bringing the data lake back into a consistent situation. To achieve this goal, an architecture has been implemented that allows us to monitor component operation metrics and check data reliability. When a problem is detected, a member of the on-call team (available 24/7) is called with the aim of analyze and solve the problem.

Despite this there are several points of improvement for the platform and the two main ones we are focusing on at the moment are the Lineage of the data and the control and containment of costs.

With the passage of time and with the increase of the information available, the management and control of the data present is becoming increasingly complex, leading to situations such as the replication of some information or, worse, the difficulty in knowing what informations are content in some structures at the Enriched level. To solve this situation, Data Lineage solutions are being implemented to keep track of how the data ”travel” within the Data Lake and to know where the data comes from in every single table of the data lake. To do that, a data catalog is being created using the Collibra tool which, for each field of all the structures of the data lake, traces the source system from which it derives and all the structures that replicate it. Furthermore, for the key fields of the tables it also allows you to keep track of all the relationships that can be built with other tables apart from them.

The second problem, as already said, is the increase in the costs of the

platform, in particular the costs associated with the BigQuery service. The costs are partly due to technical aspects and partly to how the service is used by users.
As regards the technical part, solutions are being studied to delete old data in order to limit the amount of data stored in the tables (for example by setting retention on the partitions) and to move less data during the processing phase (by tuning the fields table partitioning and clustering).
As for the costs deriving from the method of use, the client is training its users to use the features made available by BigQuery to limit costs such as filtering the data for a single partition or limiting the columns selected in a query thus avoiding the *SELECT ** statements.

Working on this project I had the opportunity to understand what are the complexities of a project to create a strategic data lake for the customer's business and how modern software design and implementation techniques allow to guarantee all the requirements requested. I was able to put into practice sectors such as CI/CD, functional programming and the use of resources in the cloud to obtain maximum flexibility and reactivity from the platform to react to the needs and requirements of the business and support them in their choices, always guaranteeing the maximum accuracy of the data.

References

[cloud function] Google Cloud. *Cloud Functions Overview*. URL: `https://cloud.google.com/functions/docs/concepts/overview`.

[cloud events] Google Cloud. *Events and Triggers*. URL: `https://cloud.google.com/functions/docs/concepts/events-triggers`.

[bigQuery] Google Cloud. *What is BigQuery?* URL: `https://cloud.google.com/bigquery/docs/introduction`.

[composer] Google Cloud. *What is Cloud Composer?* URL: `https://cloud.google.com/composer/docs/concepts/overview`.

[gcs] Google Cloud. *What Is Cloud Storage?* URL: `https://cloud.google.com/storage/docs/introduction`.

[pub/sub] Google Cloud. *What Is Pub/Sub?* URL: `https://cloud.google.com/pubsub`.