Build data products in a data mesh

To ensure that the use cases of data consumers are met, it's essential that data products in a data mesh are designed and built with care. The design of a data product starts with the definition of how data consumers would use that product, and how that product then gets exposed to consumers. Data products in a data mesh are built on top of a data store (for example, a domain data warehouse or data lake). When you create data products in a data mesh, there are some key factors that we recommend you consider throughout this process. These considerations are described in this document.

This document is part of a series which describes how to implement a data mesh on Google Cloud. It assumes that you have read and are familiar with the concepts described in Architecture and functions in a data mesh (/architecture/data-mesh) and Build a modern, distributed Data Mesh with Google Cloud (https://services.google.com/fh/files/misc/build-a-modern-distributed-datamesh-with-google-cloud-whitepaper.pdf)
.

The series has the following parts:

- Architecture and functions in a data mesh (/architecture/data-mesh)
- Design a self-service data platform for a data mesh (/architecture/design-self-service-data-platform-data-mesh)
- Describe and organize data products and resources in a data mesh (/architecture/describe-organize-data-products-resources-data-mesh)
- Build data products in a data mesh (this document)
- Discover and consume data products in a data mesh (/architecture/discover-consume-data-products-data-mesh)

When creating data products from a domain data warehouse, we recommend that data producers carefully design analytical (consumption) interfaces for those products. These consumption interfaces are a set of guarantees on the data quality and operational parameters, along with a product support model and product documentation. The cost of changing consumption interfaces is usually high because of the need for both the data producer and potentially multiple data consumers to change their consuming processes and applications. Given that the data consumers are most likely to be in organizational units which are separate to that of the data producers, coordinating the changes can be difficult.

The following sections provide background information on what you must consider when creating a domain warehouse, defining consumption interfaces, and exposing those interfaces to data consumers.

Create a domain data warehouse

There's no fundamental difference between building a standalone data warehouse and building a domain data warehouse from which the data producer team creates data products. The only real difference between the two is that the latter exposes a subset of its data through the consumption interfaces.

In many data warehouses, the raw data ingested from operational data sources goes through the process of enrichment and data quality verification (curation). In Dataplex (/dataplex)-managed data lakes, curated data typically is stored in designated curated zones. When curation is complete, a subset of the data should be ready for external-to-the-domain consumption through several types of interfaces. To define those consumption interfaces, an organization should provide a set of tools to domain teams who are new to adopting a data mesh approach. These tools let data producers create new data products on a self-service basis. For recommended practices, see Design a self-service data platform (/architecture/design-self-service-data-platform-data-mesh).

Additionally, data products must meet centrally defined data governance requirements (see Describe and organize data products and resources (/architecture/describe-organize-data-products-resources-data-mesh)). These requirements affect data quality, data availability, and lifecycle management. Because these requirements build the trust of data consumers in the data products and encourage data product usage, the benefits of implementing these requirements are worth the effort in supporting them.

Define consumption interfaces

We recommend that data producers use multiple types of interfaces, instead of defining just one or two. Each interface type in data analytics has advantages and disadvantages, and there's no single type of interface that excels at everything. When data producers assess the suitability of each interface type, they must consider the following:

- Ability to perform the data processing needed.
- Scalability to support current and future data consumer use cases.
- Performance required by data consumers.

- Cost of development and maintenance.
- Cost of running the interface.
- Support by the languages and tools that your organization uses.
- Support for separation of storage and compute.

For example, if the business requirement is to be able to run analytical queries over a petabyte-size dataset, then the only practical interface is a BigQuery view. But if the requirements are to provide near real-time streaming data, then a Pub/Sub-based interface is more appropriate.

Many of these interfaces don't require you to copy or replicate existing data. Most of them also let you separate storage and compute, a critical feature of Google Cloud analytical tools. Consumers of data exposed through these interfaces process the data using the compute resources available to them. There's no need for data producers to do any additional infrastructure provisioning.

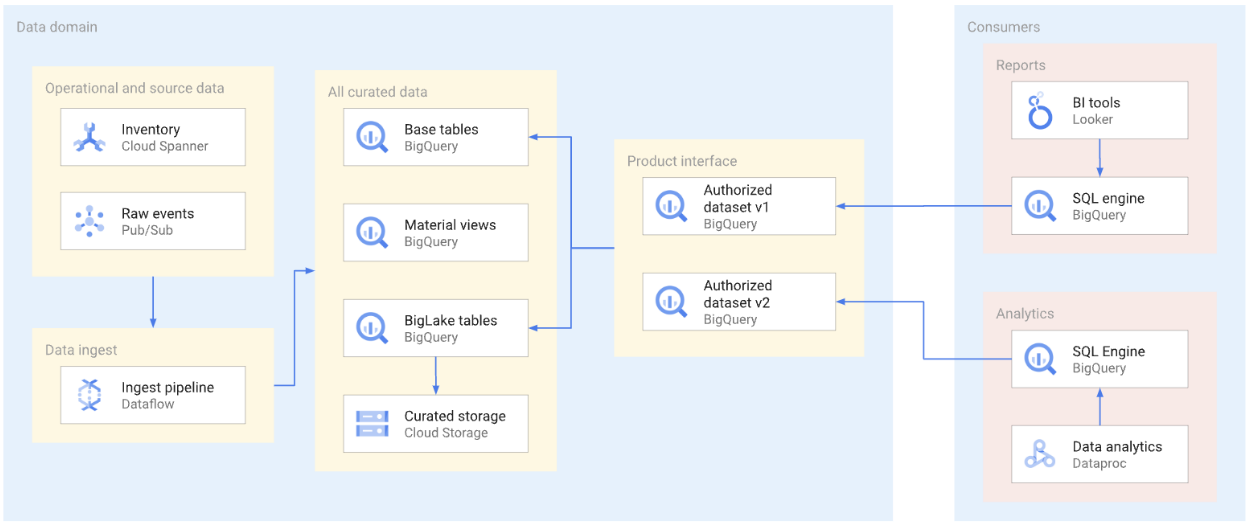
There's a wide variety of consumption interfaces. The following interfaces are the most common ones used in a data mesh and are discussed in the following sections:

- Authorized views and functions (#authorized_views_and_functions)
- Direct read APIs (#direct_read_apis)
- Data as streams (#data_as_streams)
- Data access API (#data_access_api)
- Looker blocks (#looker-blocks)
- Machine learning (ML) models (#ml_models)

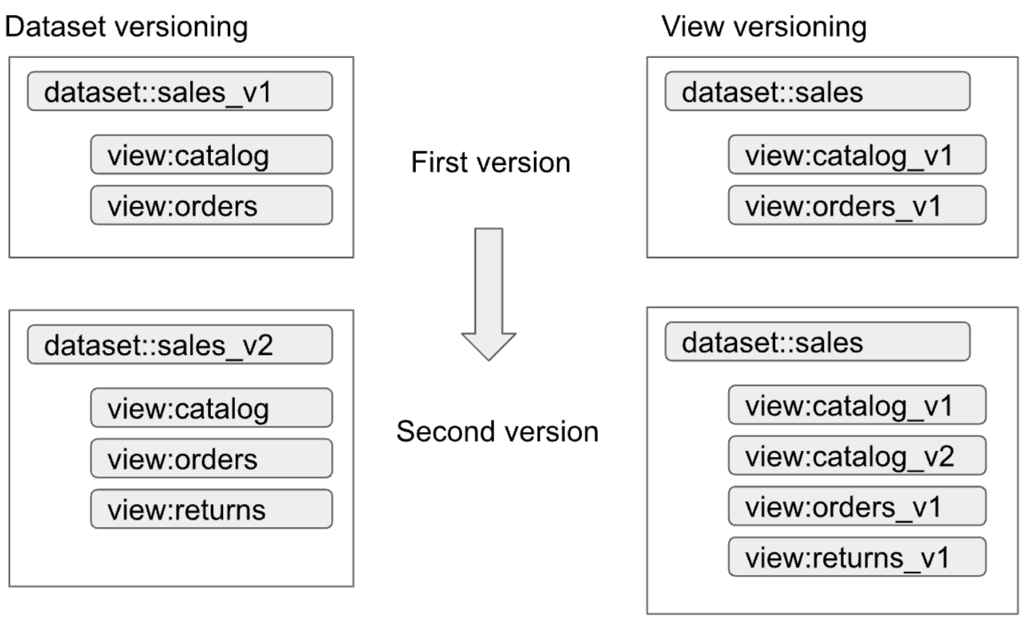
The list of interfaces in this document is not exhaustive. There are also other options that you might consider for your consumption interfaces (for example, Analytics Hub (/analytics-hub)). However, these other interfaces are outside of the scope of this document.

Authorized views and functions

As much as possible, data products should be exposed through authorized views (/bigquery/docs/authorized-views) and authorized functions, (/bigquery/docs/authorized-functions) including table-valued functions. Authorized datasets (/bigquery/docs/authorized-datasets) provide a convenient way to authorize several views automatically. Using authorized views prevents direct access to the base tables, and lets you optimize the underlying tables and queries against them, without affecting consumer use of these views. Consumers of this interface use SQL to query the data. The following diagram illustrates the use of authorized datasets as the consumption interface.



Authorized datasets and views help to enable easy versioning of interfaces. As shown in the following diagram, there are two primary versioning approaches that data producers can take:



The approaches can be summarized as follows:

- **Dataset versioning:** In this approach, you version the dataset name. You don't version the views and functions inside the dataset. You keep the same names for the views and functions regardless of version. For example, the first version of a sales dataset is defined in a dataset named `sales_v1` with two views, `catalog` and `orders`. For its second version, the sales dataset has been renamed `sales_v2`, and any previous views in the dataset keep their previous names but have new schemas. The second version of the dataset might also have new views added to it, or may remove any of the previous views.

- **View versioning:** In this approach, the views inside the dataset are versioned instead of the dataset itself. For example, the sales dataset keeps the name of `sales` regardless of version. However, the names of the views inside the dataset change to reflect each new version of the view (such as `catalog_v1`, `catalog_v2`, `orders_v1`, `orders_v2`, and `orders_v3`).

The best versioning approach for your organization depends on your organization's policies and the number of views that are rendered obsolete with the update to the underlying data. Dataset versioning is best when a major product update is needed and most views must change. View versioning leads to fewer identically named views in different datasets, but can lead to ambiguities, for example, how to tell if a join between datasets works correctly. A hybrid approach can be a good compromise. In a hybrid approach, compatible schema changes are allowed within a single dataset, and incompatible changes require a new dataset.

BigLake table considerations

Authorized views can be created not only on BigQuery tables, but also on BigLake (/bigquery/docs/biglake-intro) tables. BigLake tables let consumers query the data stored in Cloud Storage by using the BigQuery SQL interface. BigLake tables support fine-grained access control without the need for data consumers to have read permissions for the underlying Cloud Storage bucket.

Data producers must consider the following for BigLake tables:

- The design of the file formats and the data layout influences the performance of the queries. Column-based formats, for example, Parquet (https://parquet.apache.org/) or ORC (https://orc.apache.org/), generally perform much better for analytic queries than JSON or CSV formats.
- A Hive partitioned layout (/bigquery/docs/hive-partitioned-queries-gcs) lets you prune partitions and speeds up queries which use partitioning columns.
- The number of files and the preferred query performance for the file size must also be taken into account in the design stage.

If queries using BigLake tables don't meet service-level agreement (SLA) requirements for the interface and can't be tuned, then we recommend the following actions:

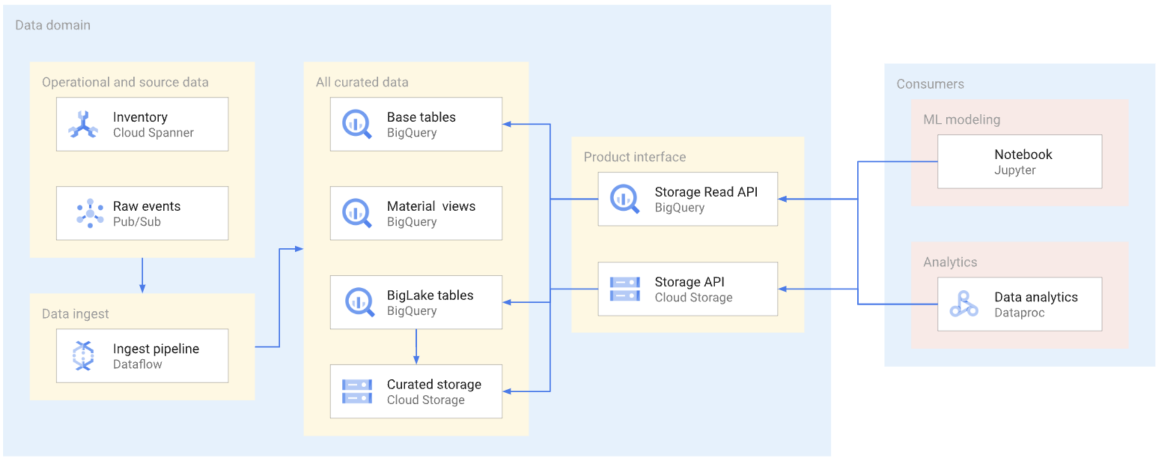
- For data that must be exposed to the data consumer, convert that data to BigQuery storage.
- Redefine the authorized views to use the BigQuery tables.

Generally, this approach does not cause any disruption to the data consumers, or require any changes to their queries. The queries in BigQuery storage can be optimized using techniques that aren't possible with BigLake tables. For example, with BigQuery storage, consumers can query materialized views that have different partitioning and clustering than the base tables, and they can use the BigQuery BI Engine.

Direct read APIs

Although we don't generally recommend that data producers give data consumers direct read access to the base tables, it might occasionally be practical to allow such access for reasons such as performance and cost. In such cases, extra care should be taken to ensure that the table schema is stable.

There are two ways to directly access data in a typical warehouse. Data producers can either use the BigQuery Storage Read API (/bigquery/docs/reference/storage), or the Cloud Storage JSON or XML APIs (/storage/docs/apis). The following diagram illustrates two examples of consumers using these APIs. One is a machine learning (ML) use case, and the other is a data processing job.



Versioning a direct-read interface is complex. Typically, data producers must create another table with a different schema. They must also maintain two versions of the table, until all the data consumers of the deprecated version migrate to the new one. If the consumers can tolerate the disruption of rebuilding the table and switching to the new schema, then it's possible to avoid the data duplication. In cases where schema changes can be backward compatible, the migration of the base table can be avoided. For example, you don't have to migrate the base table if only new columns are added and the data in these columns is backfilled for all the rows.

The following is a summary of the differences between the Storage Read API and Cloud Storage API. In general, whenever possible, we recommend that data producers use BigQuery API for analytical applications.

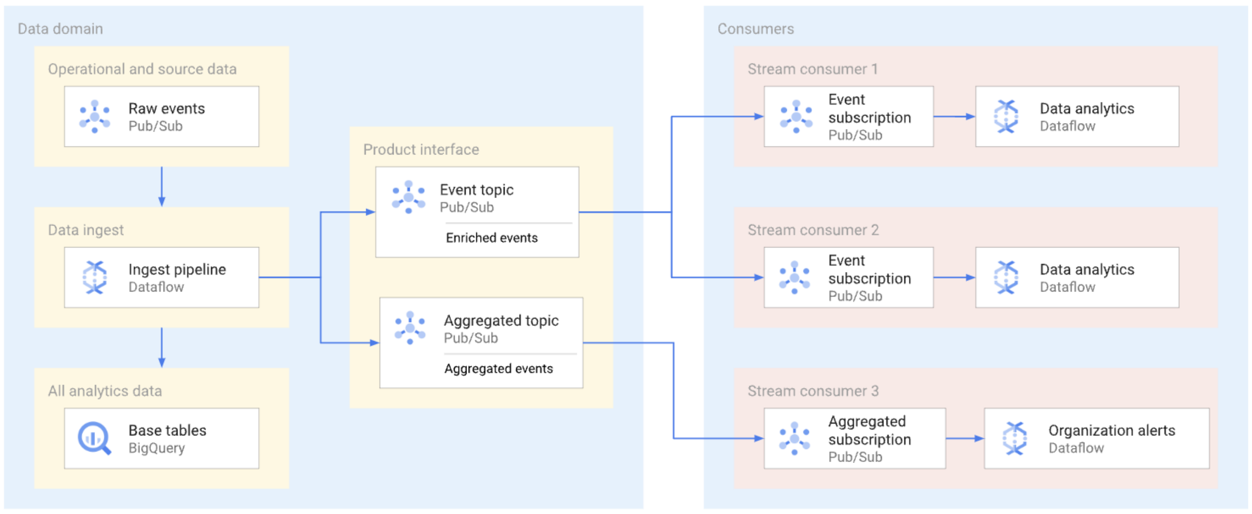
- **Storage Read API:** Storage Read API can be used to read data in BigQuery tables and to read BigLake tables. This API supports filtering and fine-grained access control, and can be a good option for stable data analytics or ML consumers.
- **Cloud Storage API:** Data producers might need to share a particular Cloud Storage bucket directly with data consumers. For instance, data producers can share the bucket if data consumers can't use the SQL interface for some reason, or the bucket has data formats that aren't supported by Storage Read API (/bigquery/docs/biglake-intro#limitations).

In general, we don't recommend that data producers allow direct access through the storage APIs because direct access doesn't allow for filtering and fine-grained access control. However, the direct access approach can be a viable choice for stable, small-sized (gigabytes) datasets.

Allowing Pub/Sub access to the bucket gives data consumers an easy way to copy the data into their projects and process it there. In general, we don't recommend data copying if it can be avoided. Multiple copies of data increase storage cost, and add to the maintenance and lineage tracking overhead.

Data as streams

A domain can expose streaming data by publishing that data to a Pub/Sub topic. Subscribers who want to consume the data create subscriptions to consume the messages published to that topic. Each subscriber receives and consumes data independently. The following diagram shows an example of such data streams.



In the diagram, the ingest pipeline reads raw events, enriches (curates) them, and saves this curated data to the analytical data store (BigQuery base table). At the same time, the pipeline publishes the enriched events to a dedicated topic. This topic is consumed by multiple subscribers, each of whom may be potentially filtering these events to get only the ones relevant to them. The pipeline also aggregates and publishes event statistics to its own topic to be processed by another data consumer.

The following are example use cases for Pub/Sub subscriptions:

- Enriched events, such as providing full customer profile information along with data on a particular customer order.
- Close-to-real-time aggregation notifications, such as total order statistics for the last 15 minutes.
- Business-level alerts, such as generating an alert if order volume dropped by 20% compared to a similar period on the previous day.
- Data change notifications (similar in concept to change data capture (https://en.wikipedia.org/wiki/Change_data_capture) notifications), such as a particular order changes status.

The data format that data producers use for Pub/Sub messages affects costs and how these messages are processed. For high-volume streams in a data mesh architecture, Avro or Protobuf formats are good options. If data producers use these formats, they can assign schemas (/pubsub/docs/schemas) to Pub/Sub topics. The schemas help to ensure that the consumers receive well-formed messages.

Because a streaming data structure can be constantly changing, versioning of this interface requires coordination between the data producers and the data consumers. There are several common approaches data producers can take, which are as follows:

- A new topic is created every time the message structure changes. This topic often has an explicit Pub/Sub schema. Data consumers who need the new interface can start to consume the new data. The message version is implied by the name of the topic, for example, `click_events_v1`. Message formats are strongly typed. There's no variation on the message format between messages in the same topic. The disadvantage of this approach is that there might be data consumers who can't switch to the new subscription. In this case, the data producer must continue publishing events to all active topics for some time, and data consumers who subscribe to the topic must either deal with a gap in message flow, or de-duplicate the messages.
- Data is always published to the same topic. However, the structure of the message can change. A Pub/Sub message attribute (/pubsub/docs/overview#core_concepts) (separate from the payload) defines the version of the message. For example, `v=1.0`. This approach removes the need to deal with gaps or duplicates; however, all data consumers must be ready to receive messages of a new type. Data producers also can't use Pub/Sub topic schemas for this approach.
- A hybrid approach. The message schema can have an arbitrary data section that can be used for new fields. This approach can provide a reasonable balance between having strongly typed data, and frequent and complex version changes.

Data access API

Data producers can build a custom API to directly access the base tables in a data warehouse. Typically, these producers expose this custom API as a REST or a gRPC API, and deploy on Cloud Run or a Kubernetes cluster. An API gateway like Apigee can provide other additional features, such as traffic throttling or a caching layer. These functionalities are useful when exposing the data access API to consumers outside of a Google Cloud organization. Potential candidates for a data access API are latency sensitive and high concurrency queries which both return a relatively small result in a single API and can be effectively cached.

Examples of such a custom API for data access can be as follows:

- A combined view on the SLA metrics of the table or product.
- The top 10 (potentially cached) records from a particular table.
- A dataset of table statistics (total number of rows, or data distribution within key columns).

Any guidelines and governance that the organization has around building application APIs are also applicable to the custom APIs created by data producers. The organization's guidelines and governance should cover issues such as hosting, monitoring, access control, and versioning.

The disadvantage of a custom API is the fact that the data producers are responsible for any additional infrastructure that's required to host this interface, as well as custom API coding and maintenance. We recommend that data producers investigate other options before deciding to create custom data access APIs. For example, data producers can use BigQuery BI Engine to decrease response latency and increase concurrency.

Looker Blocks

For products such as Looker, which are heavily used in business intelligence (BI) tools, it might be helpful to maintain a set of BI tool-specific widgets. Because the data producer team knows the underlying data model that is used in the domain, that team is best placed to create and maintain a prebuilt set of visualizations.

In the case of Looker, this visualization could be a set of Looker Blocks (prebuilt LookML data models). The Looker Blocks can be easily incorporated into dashboards hosted by consumers.

ML models

Because teams that work in data domains have a deep understanding and knowledge of their data, they are often the best teams to build and maintain ML models which are trained on the domain data. These ML Models can be exposed through several different interfaces, including the following:

- BigQuery ML models can be deployed in a dedicated dataset and shared with data consumers for BigQuery batch predictions.
- BigQuery ML models can be exported into Vertex AI to be used for online predictions.

Data location considerations for consumption interfaces

An important consideration when data producers define consumption interfaces for data products is data location. In general, to minimize costs, data should be processed in the same region that it's stored in. This approach helps to prevent cross-region data egress charges. This approach also has the lowest data consumption latency. For these reasons, data stored in multi-regional BigQuery locations is usually the best candidate for exposing as a data product.

However, for performance reasons, data stored in Cloud Storage and exposed through BigLake tables or direct read APIs should be stored in regional buckets.

If data exposed in one product resides in one region and needs to be joined with data in another domain in another region, data consumers must consider the following limitations:

- Cross-region queries that use BigQuery SQL are not supported. If the primary consumption method for the data is BigQuery SQL, all the tables in the query must be in the same location.
- BigQuery flat-rate commitments are regional. If a project uses only a flat-rate commitment in one region but queries a data product in another region, on-demand pricing applies.
- Data consumers can use direct read APIs to read data from another region. However, cross-regional network egress charges apply, and data consumers will most likely experience latency for large data transfers.

Data that's frequently accessed across regions can be replicated to those regions to reduce the cost and latency of queries incurred by the product consumers. For example, BigQuery datasets can be copied (/bigquery/docs/copying-datasets) to other regions. However, data should only be copied when it's required. We recommend that data producers only make a subset of the available product data available to multiple regions when you copy data. This approach helps to minimize replication latency and cost. This approach can result in the need to provide multiple versions of the consumption interface with the data location region explicitly called out. For example, BigQuery Authorized views can be exposed through naming such as `sales_eu_v1` and `sales_us_v1`.

Data stream interfaces using Pub/Sub topics don't need any additional replication logic to consume messages in regions that are not the same region as that where the message is stored. However, additional cross-region egress charges (/pubsub/pricing#egress_costs) apply in this case.

Expose consumption interfaces to data consumers

This section discusses how to make consumption interfaces discoverable by potential consumers. Data Catalog (/data-catalog/docs/concepts/overview) is a fully managed service which organizations can use to provide the data discovery and metadata management services. Data producers must make the consumption interfaces of their data products searchable and annotate them with the appropriate metadata to enable product consumers to access them in a self-service manner.

For information on how metadata for data products is created and shared in a central, searchable way in Data Catalog, see Describe and organize data products and data resources (/architecture/describe-organize-data-products-resources-data-mesh).

The following sections discuss how each interface type is defined as a Data Catalog entry.

BigQuery-based SQL interfaces

Technical metadata, such as a fully qualified table name or table schema, is automatically registered for authorized views, BigLake views, and BigQuery tables that are available through the Storage Read API. We recommend that data producers also provide additional information in the data product documentation to help data consumers. For example, to help users find the product documentation for an entry, data producers can add a URL to one of the tags that has been applied to the entry. Producers can also provide the following:

- Sets of clustered (/bigquery/docs/clustered-tables) columns, which should be used in query filters.
- Enumeration values for fields that have logical enumeration type, if the type is not provided as part of the field description.
- Supported joins with other tables.

Data streams

Pub/Sub topics are automatically registered with the Data Catalog. However, data producers must describe the schema in the data product documentation.

Cloud Storage API

Data Catalog supports the definition of Cloud Storage file entries and their schema. If a data lake fileset is managed by Dataplex, the fileset is automatically registered in the Data Catalog. Filesets that aren't associated with Dataplex are added using a different approach (/data-catalog/docs/how-to/filesets).

Other interfaces

You can add other interfaces which don't have built-in support from Data Catalog by creating custom entries (/data-catalog/docs/how-to/custom-entries).

What's next

- See a reference implementation of the data mesh architecture (https://github.com/GoogleCloudPlatform/data-mesh-demo).
- Learn more about BigQuery (/bigquery).
- Read about Dataplex (/dataplex).
- For more reference architectures, diagrams, tutorials, and best practices, explore the Cloud Architecture Center (/architecture).