

An insider's guide to BigQuery cost optimization

Introduction

Discover the best practices that help you optimize both cost and performance of BigQuery.

Analytics is at the heart of every business decision. In today's economic climate, it has become more important than ever—helping organizations monitor their health, sharpen their focus, and drive efficiencies. Yet these same economic conditions are putting pressure on organizations to optimize the cost associated with analytics tools and technologies.

To make sure return on investment remains as high as possible, leading organizations are looking to best practices and optimization techniques for the cloud technologies they use. If you use BigQuery, this paper has you covered on both counts.

First, it explores best practices around data ingestion, export, storage, and analysis; as well as storage models and how long-term storage helps reduce costs. It also explores the consumption models of BigQuery Analysis, best practices on workload and capacity management, and cost optimization techniques on ingestion, extraction, and analysis. Finally, it dives into industry use cases to help you see how others are using BigQuery for data warehousing, storage, and analysis.



Contents

01

BigQuery
architecture
and pricing
model

1.1 Architecture
1.2 Pricing model

02

Data ingestion
and extraction
in BigQuery

2.1 Ingestion
2.2 Extraction
2.3 Best practices for
ingestion and extraction

03

BigQuery
storage

3.1 How data is stored
3.2 Key features
3.3 Pricing
3.4 Best practices and
optimization

04

BigQuery
compute
best practices

4.1 Cost analysis, budget
alerts & custom cost controls
4.2 BigQuery compute
consumption models
4.3 Pick the right consumption
model for your workload
4.4 Best practices on
workload management
4.5 BigQuery compute
cost optimization

05

Different use
cases of
building data
warehouses

Use Case 1: Marketing
Data Warehouse
Use Case 2: Advertising
Data Pipelines
Use case 3: Mobile
Gaming Analytics Platform

06

Conclusion

Conclusion

01

BigQuery architecture and pricing model

An introduction to BigQuery's architecture and pricing model

Companies of all shapes and sizes are using BigQuery to perform data analysis and reveal invaluable insights business data—helping them make decisions in real time, streamline business reporting, and predict future business opportunities. As a completely serverless enterprise data warehouse, BigQuery stands out as one of the most cost-effective in the market today.

In this chapter:



BigQuery architecture

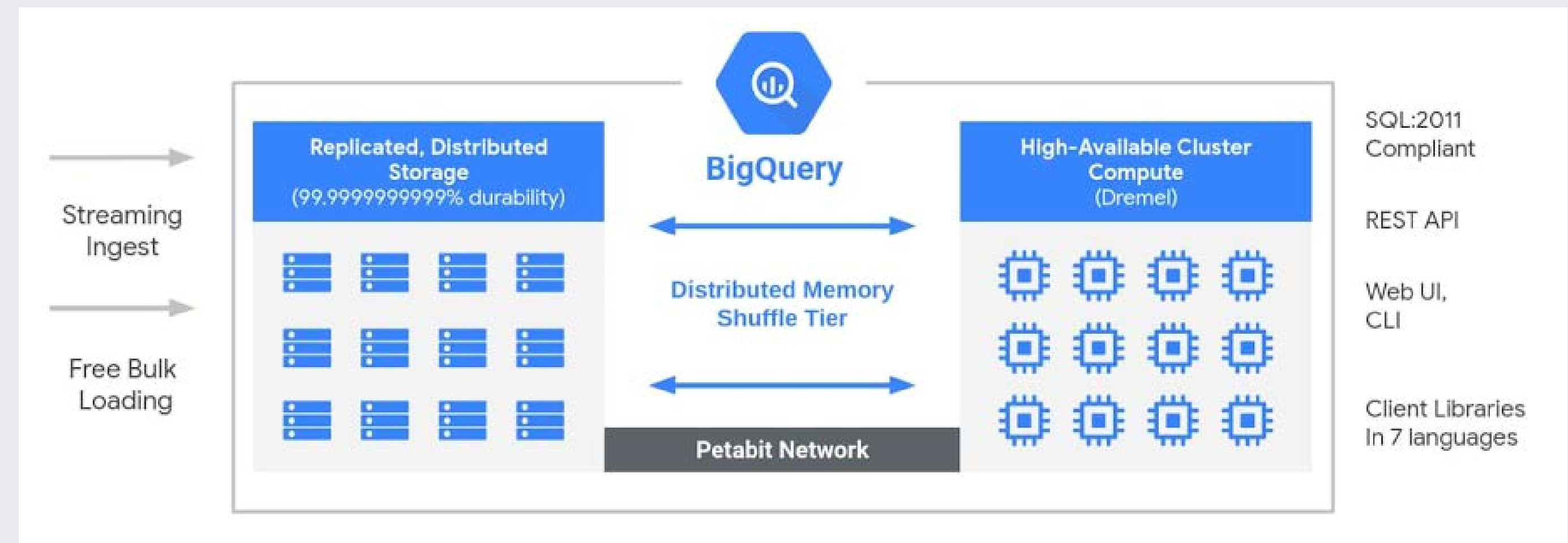


BigQuery pricing model

A serverless architecture

Google BigQuery was designed as a cloud-native data warehouse—built to address the needs of data-driven organizations in a cloud-first world.

Compared to traditional node-based cloud data warehouses or on-premise massively parallel processing (MPP) systems, BigQuery has a serverless architecture—which decouples storage and compute so both can scale independently and on demand. This structure offers immense flexibility and cost control, as customers don't need to keep expensive compute resources up and running all the time. It also means customers of any size can start analyzing their data using Standard SQL, without worrying about database operations and system engineering.



BigQuery's distributed architecture

A simple pricing model

As a serverless data analytics platform, you don't need to provision individual instances or virtual machines to use BigQuery. Instead, it automatically allocates computing resources as you need them—helping you reduce costs. For extra peace of mind, you can also reserve compute capacity ahead of time in the form of slots, or virtual CPUs.

When you run a BigQuery job, you do it from a project—and each project has a billing account attached to it. The project will accrue the cost of the job. Data is allocated to projects too. These projects will accrue storage costs, and allocate it to the attached billing account. Then, you can view BigQuery costs and trends in the Cloud Billing reports page in the Google Cloud console.

BigQuery pricing has two main components:

- 1 [Analysis pricing](#) is the cost to process queries which include SQL queries, user-defined functions, scripts, and certain data manipulation language (DML), BQML, and data definition language (DDL) statements that scan tables
- 2 [Storage pricing](#) is the cost to store the data you load into BigQuery

Things you can do for free in BigQuery

- [Batch loading data](#) into BigQuery
- [Automatic re-clustering](#) (which requires no setup and maintenance)
- [Batch exporting](#) data
- Deleting tables, views, partitions, functions, and datasets
- Metadata operations
- [Cached queries](#)
- Queries that result in error
- Storage for first 10 GB of data per month
- Query data processed for first 1 TB of data per month (advantageous to users on on-demand pricing)

How BigQuery pricing works

BigQuery pricing is based on analysis type, storage, additional services, and data ingestion and extraction. Loading and exporting data are free.

Services and usage	Subscription type	Price (USD)
Free tier	The BigQuery free tier gives customers 10 GB storage, up to 1 TB queries free per month, and other resources.	Free
Analysis	On-Demand. Pay-per-byte model. Generally gives you access to up to 2,000 concurrent slots, shared among all queries in a single project.	Starting at \$6.25 per TB. First 1TB per month is free (US multi-region)
	Standard Edition Reservation Pay-per-slot-hour model. Makes it easier to put a cap on spending.	Starting at \$40 for 100 slots per hour (US multi-region)
	Enterprise Edition Reservation builds on Basic Editions adding more features. It also permits fixed costs setups.	Starting at \$60 for 100 slots per hour (US multi-region) Discounts available for 1 or 3 years commitments.
	Enterprise Plus Edition Reservation builds on Enterprise Editions by adding multi-region redundancy(on roadmap) and higher compliance	Starting at \$100 for 100 slots per hour (US multi-region) Discounts available for 1 or 3 years commitments.
Storage	Active logical storage, based on the uncompressed bytes used in tables or table partitions modified in the last 90 days.	Starting at \$0.02 per GB. The first 10 GB is free each month
	Long-term logical storage, based on the uncompressed bytes in tables or partitions that have not been modified for 90 consecutive days	Starting at \$0.01 per GB. The first 10 GB is free each month
	Active physical storage, based on the compressed bytes used in tables or table partitions modified for 90 consecutive days.	Starting at \$0.04 per GB. The first 10 GB is free each month
	Long-term physical storage, based on compressed bytes in tables or partitions that have not been modified for 90 consecutive days.	Starting at \$0.02 per GB. The first 10 GB is free each month
Data ingestion	Batch loading, export table data to Cloud Storage.	Free when using the shared slot pool
	Streaming inserts, you are charged for rows that are successfully inserted. Individual rows are calculated using a 1 KB minimum.	\$0.01 per 200 MB
	BigQuery Storage Write API, data loaded into BigQuery, is subject to BigQuery storage pricing or Cloud Storage pricing .	\$0.025 per 1 GB. The first 2 TB per month are free
Data extraction	Batch export, export table data to Cloud Storage.	Free when using the shared slot pool
	Streaming reads, use the storage Read API to perform streaming reads of table data.	Starting at \$1.10 per TB read

02

Data ingestion and extraction in BigQuery

Data ingestion and extraction in BigQuery

To perform analytics, you need data. Structured, semi-structured, or unstructured, this data needs to be supplied—or ingested—into your data platform without blowing your budget. Likewise, it often needs to be extracted, too. Let's explore your options for cost-effectively moving data in and out of BigQuery.

In this chapter:

- Ways to ingest data
- Ways to extract data
- Pricing of data ingestion and extraction
- How to optimize costs and apply best practices

Ingesting data into BigQuery

By default, you are not charged for batch loading data from Cloud Storage or local files into BigQuery. These jobs use a shared pool of slots—yet BigQuery doesn't guarantee availability of this shared pool or the throughput you will see. If you are loading large amounts of data, your job might wait as slots become available. In that case, you have an option to obtain dedicated capacity by assigning your load jobs to [editions](#) reservation, but you lose access to the free pool and use reservation resources.

 [Learn more about assignments](#)

If the target dataset is co-located with a Cloud Storage dataset, you are not charged for network egress when loading from a Cloud Storage bucket in any other region.

 [Learn more about location considerations](#)

BigQuery offers two modes of data ingestion:

- 1 Batch loading of source data into one or more BigQuery tables in a single batch operation
- 2 Streaming data one record at a time or in small batches

Here’s an example of ingestion pricing in the US:

Operation	Pricing	Details
Batch Loading	Free using the shared slot pool.	Customers can choose Editions reservations for guaranteed capacity. Once the data is loaded into BigQuery, you are charged for storage.
BigQuery Storage Write API	\$0.025 per 1 GB	The first 2 TB per month are free.
Streaming inserts (tabledata.insertALL)	\$0.01 per 200 MB	You are charged for rows that are successfully inserted. Individual rows are calculated using a 1 KB minimum size.

Extracting data from BigQuery



BigQuery offers two modes of data extraction:

- 1 Batch export of table data to Cloud Storage
- 2 Streaming reads of table data using the Storage Read API

By default, you are not charged for batch exporting of data from BigQuery. Similar to batch loading, these jobs use a shared pool of slots—again, BigQuery doesn't guarantee availability or the throughput you will see. Alternatively, in case you want dedicated capacity for your export jobs, you can assign export jobs to [editions](#) reservation, they lose access to the free pool and use reservation resources.

The Storage Read API has an on-demand pricing model, which means BigQuery charges for the number of bytes processed (or 'bytes read'). On-demand pricing is solely based on usage, with a bytes read free tier of 300 TB per month for each billing account. Bytes scanned as part of reads from [temporary tables](#) are free and do not count towards the 300 TB. Associated egress cost is not included, either.

When it comes to Storage Read API charges, note that:

- You are charged according to the total amount of data read. This is calculated based on the type of data in the column, and the size of the data is calculated based on the column's data type.

 [Learn how data size is calculated](#)

- You are charged for any data read in a read session even if a ReadRows call fails.
- If you cancel a ReadRows call before the end of the stream is reached, you are charged for any data read before the cancellation. Your charges can include data that was read but not returned to you before the cancellation of the ReadRows call.
- To lower costs, use partitioned and clustered tables whenever possible. You can reduce the amount of data read by using a WHERE clause to prune partitions.

 [Learn more about querying partitioned tables](#)



Best practices for ingestion and extraction

Use shared pool resources to load or export data

By default, loading or unloading data from BigQuery is free. Use this free resource whenever you can—and only use [dedicated capacity](#) when you need to load large amounts of data or if your project is time-sensitive.

Use streaming wisely

One of BigQuery's key features — which sets it apart from competitors — is the ability to stream data directly using [Storage write API](#) or [legacy streaming API](#). For new projects, we recommend using the BigQuery Storage Write API instead of the legacy `tabledata.insertAll` method. The Storage Write API has lower pricing and more robust features, including exactly-once delivery semantics. Yet streaming comes at a cost. Only use streaming for real-time analytics use cases.

Choose the right data ingestion format

BigQuery natively supports JSON (newline-delimited), Avro, CSV, ORC, and Parquet file types from Cloud Storage into BigQuery using console, command line, BigQuery Data Transfer Service, and several other programming languages. With better performance on load, Avro and Parquet are perfect for large loading jobs. If you need to load hierarchical data with nested and repeated fields, opt for Avro, JSON, ORC, or Parquet file formats. And, while BigQuery supports star schema, if you are looking to denormalize the data, nested and repeated fields reduce the duplication.

Uncompress data before you load

Sometimes you need to compress data so you can transfer it faster over the network with low latency. Generally, compressed files take longer to load in BigQuery—so to optimize performance when loading, uncompress your data first. Just note that in the case of Avro files, compressed files load faster than uncompressed files.

Use native connectors to read and write data to BigQuery

Many customers use other tools along with BigQuery to process data. Before building custom integrations, explore native connectors—most of which use the BigQuery Storage Read/Write API at the backend. **Examples include:**

- [Dataproc](#) to read/write data from BigQuery
- Vertex AI's out-of-the-box integrations to:
 - Directly import BigQuery tables into Vertex AI to train your models
 - Directly access BigQuery datasets in Vertex AI Workbench
 - Export your model's test prediction results into BigQuery
 - Perform Vertex AI batch predictions using BigQuery as a source and destination for data
- [Pub/Sub](#) to stream data direct to BigQuery
- BigQuery Data Transfer Service to transfer data from AWS and Azure

03

Storage

of data in

BigQuery

Storage of data in BigQuery

There are some big benefits to using BigQuery storage instead of external storage for the data that drives decision-making in your business. Yet, as we'll explore below, storage may comprise a good portion of your overall costs when using BigQuery, so it pays to know the tricks of the trade for optimizing these costs—without compromising performance.

BigQuery stores data in compressed format and offers two storage billing models - Logical and Physical storage, choosing the right storage billing model helps you save the cost. In the Physical storage model, you are charged based on actual physical bytes stored, you can save a good amount on storage by opting for a physical storage model for your dataset if your data compresses well, we will go into a little more detail on this in the best practices section of this chapter.

In this chapter:

- BigQuery storage architecture and key features
- How storage pricing works
- How to optimize costs and apply best practices

How data is stored

Before diving into best practices and optimization on the storage front, let's take a look at the BigQuery storage architecture. A key feature is the separation of storage and compute, which allows BigQuery to scale both independently, based on demand.

When you run a query, the query engine distributes the work in parallel across multiple workers, which scan the relevant tables in storage, process the query, and then gather the results. BigQuery executes queries completely in-memory, using a petabit network to move data extremely fast to the worker nodes.



BigQuery architecture

Most data stored in BigQuery is table data, and you're billed for the storage used for these resources. There are different types of table:

- [Standard tables](#) contain structured data. Every table has a schema, and every column in the schema has a data type. BigQuery stores data in columnar format.

 [Learn about storage layout](#)

- [Table clones](#) (available as [preview](#)) are lightweight, writeable copies of standard tables. BigQuery only stores the delta between a table clone and its base table.
- [Table snapshots](#) are point-in-time copies of tables. They're read-only, but you can restore a table from a table snapshot. BigQuery only stores the delta between a table snapshot and its base table.
- [Materialized views](#) are precomputed views that periodically cache the results of the view query. The cached results are stored in BigQuery storage.

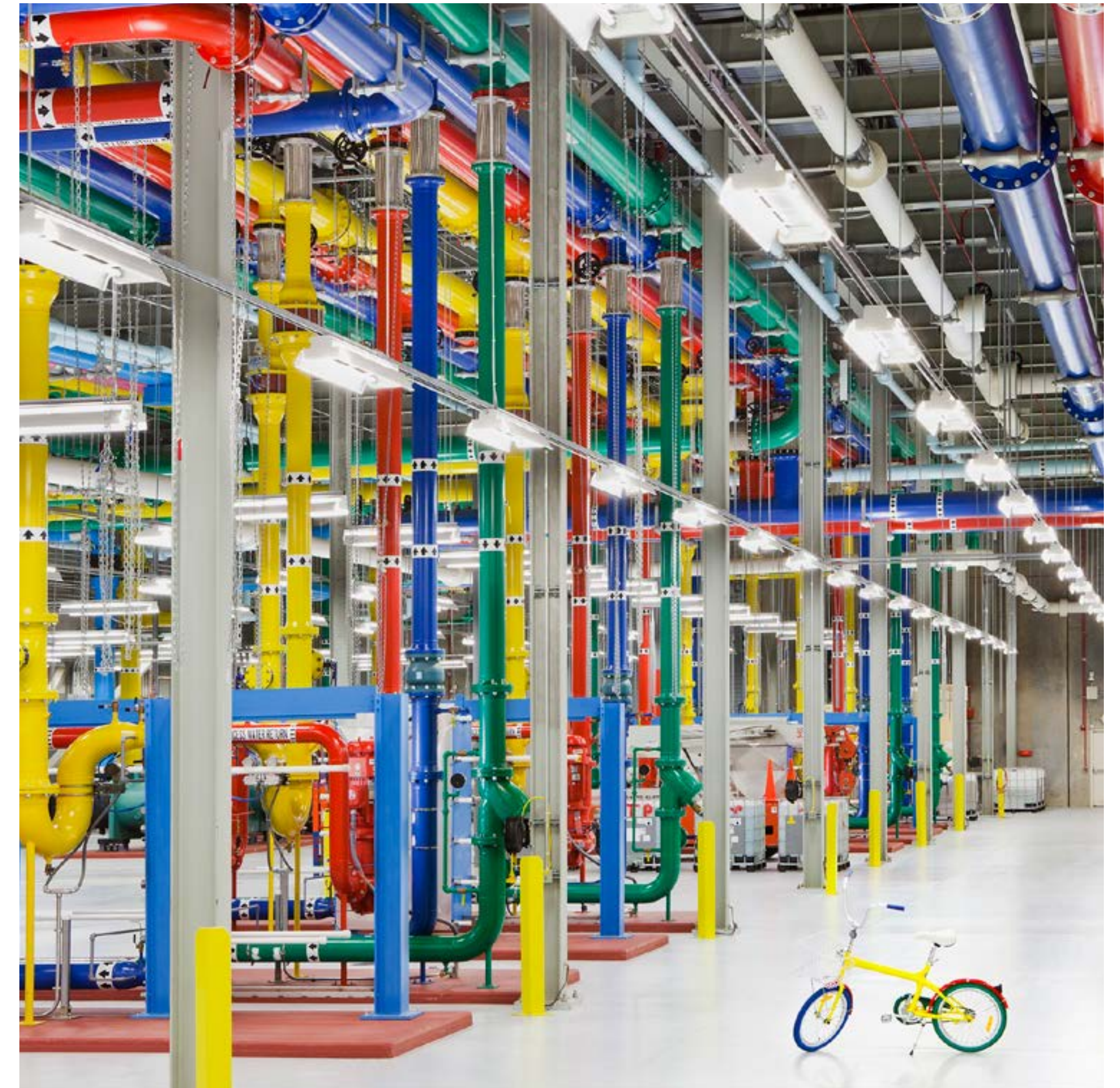


[Cached query results](#) are stored as temporary tables, and you aren't charged for this storage.

BigQuery also supports [external tables](#). Here, the data resides in a data store external to BigQuery, such as Cloud Storage. It has a table schema, just like a standard table, but the table definition points to the external data store. In this case, only the table metadata is kept in BigQuery storage. BigQuery does not charge for external table storage, although the external data store might come at a cost.





BigQuery organizes tables and other resources into logical containers called datasets. How you group your BigQuery resources affects permissions, quotas, billing, and other aspects of your BigQuery workloads.

 [Get recommended guidelines for organizing BigQuery resources](#)



Key features of BigQuery storage

BigQuery storage is a secure and trusted option for any data analysis workload. Here's why:

- **Managed.** As a completely managed service, you don't need to provision storage resources or reserve units of storage. BigQuery automatically allocates storage when you load data into the system—and you only pay for the amount of storage you use.
 [Learn more about BigQuery pricing](#)
- **Durable.** BigQuery storage is designed for 99.999999999% (11 9's) annual durability. BigQuery replicates your data across multiple availability zones to protect from data loss due to machine-level or [zonal](#) failures.
 [Learn more about availability and durability](#)
- **Encrypted.** BigQuery automatically encrypts all data before it is written to disk. You can provide your own encryption key or let Google manage the encryption key.
 [Learn more about encryption at rest](#)
- **Efficient.** BigQuery storage uses an efficient encoding format that is optimized for analytic workloads.
 [Learn more about columnar storage](#)
- **Compressed.** Industry leading compression, result of over a decade of innovation in storage optimization technology including proprietary columnar compression, automatic data sorting, clustering and compaction.



How storage pricing works

Typically, storage contributes ~20-30% to the overall cost of BigQuery—which may not seem much, but it still pays to understand how storage pricing works. This way, you can better optimize spending.

[Storage costs](#) relate to data loaded and stored in BigQuery.

BigQuery storage is billed based on logical bytes by default. However, you can choose to use physical bytes as a billing model for your dataset storage. If you do, you can't switch back to logical bytes. The cost of physical storage is higher than that of logical storage, but it can save you money if your data compresses well.

 [Learn more about data storage billing models](#)

There are two types:

- 1 Active storage includes any table or table partition that has been modified in the last 90 days.
- 2 Long-term storage includes any table or table partition that has not been modified for 90 consecutive days. It's around 50% cheaper, yet there is no difference in performance, durability, or availability between active and long-term storage.



Let's explore some pricing estimates using the Physical storage billing model, pricing varies slightly by region.

In the below examples, we use US-Multi Regional pricing (\$.04/GB/month for active storage and \$.02/GB/month for long-term storage).



Active storage pricing is prorated per MB, per second. For example, if you store:

- 100 GB for half a month, you pay \$2.00 ($$.04 \times 100 / 2$)
- 5 TB for half a month, you pay \$102.40 ($$.04 \times 1024 \times 5 / 2$)
- 1 PB for a full month, you pay \$41,943.04 ($$.04 \times 1024^2$)

Long-term storage is also prorated per MB, per second. For example, if you store:

- 100 GB for half a month, you pay \$1.00 ($$.02 \times 100 / 2$)
- 5 TB for half a month, you pay \$51.20 ($$.02 \times 1024 \times 5 / 2$)
- 1 PB for a full month, you pay \$20,971.52 ($$.02 \times 1024^2$)

Best practices for optimizing storage

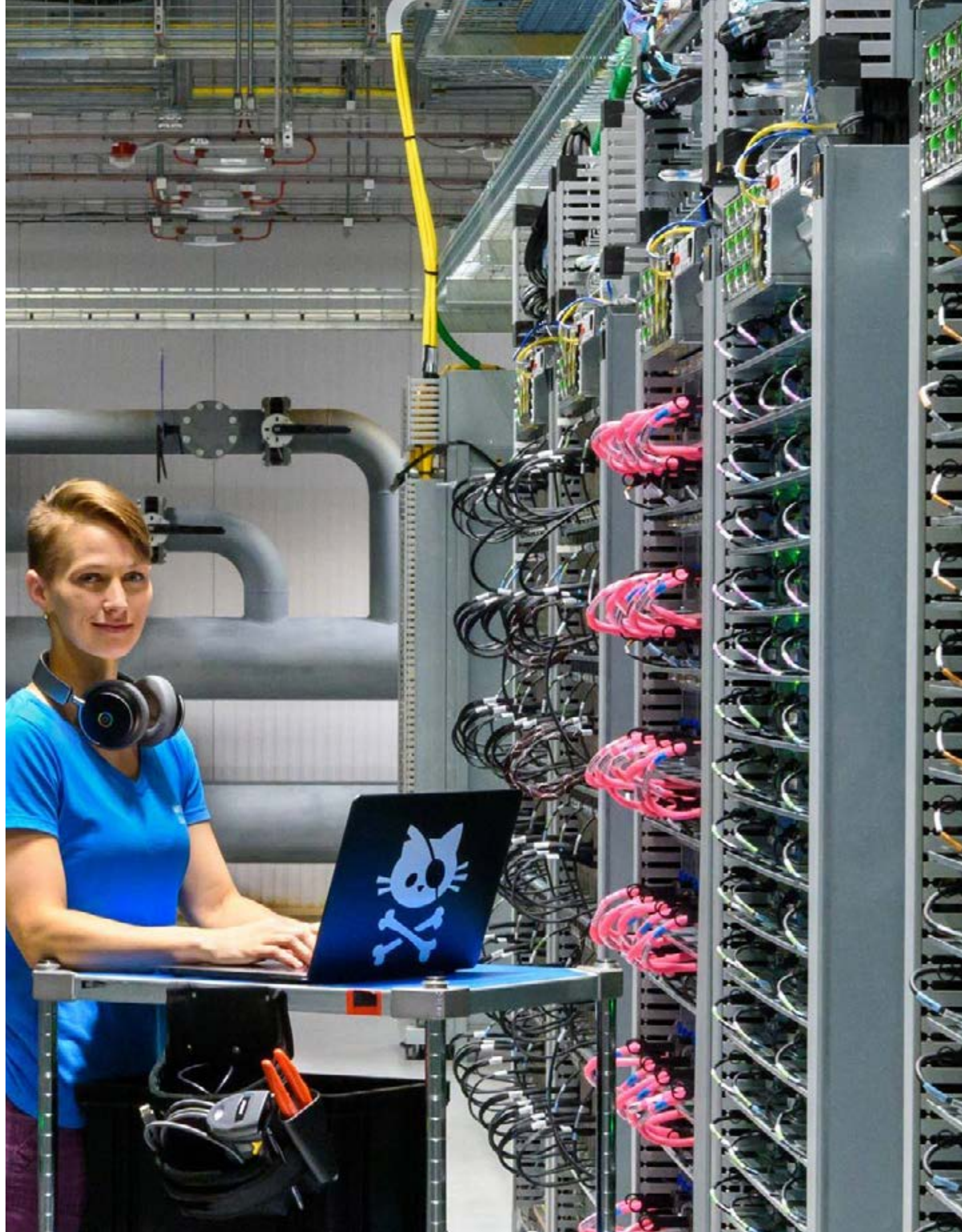
Use long-term storage

As we saw earlier, long-term storage is half the price of active storage. Table partitions help you tap into these savings. Each partition is considered separately for long-term storage pricing—so, if a partition hasn’t been modified in the last 90 days, the data in that partition is charged at the discounted price.

If a non-partitioned table or a table partition is edited, the price reverts back to the regular storage pricing, and the 90-day timer starts again.

Actions that reset the timer include:

Action	Details
Loading data into a table	Any load or query job that appends data to a destination table or overwrites a destination table.
Copying data into a table	Any copy job that appends data to a destination table or overwrites a destination table.
Writing query results to a table	Any query job that appends data to a destination table or overwrites a destination table.
Using data manipulation language (DML)	Using a DML statement to modify table data.
Using data definition language (DDL)	Using a CREATE OR REPLACE TABLE statement to replace a table.
Streaming data into the table	Ingesting data using the tabledata.insertAll API call.



Actions that don't reset the 90-day long term storage timer include:

- Querying a table
- Creating a view that queries a table
- Exporting data from a table
- Copying a table (to another destination table)
- Patching or updating a [table resource](#)

For non-partitioned tables and partitions that reach the 90-day threshold during a billing cycle, the price is prorated accordingly.

Use time travel wisely

Using [time travel](#), you can recover from mistakes like accidentally changing or deleting data, or dropping a table. Just note that storage to support time travel is charged at active storage pricing if you opt for a physical storage billing model for your dataset.



The window for time travel is set to seven days by default, but can be reduced to two days on a per-dataset basis.

Consider the most cost-effective ways to maintain data in a table. For example, it may be better to change certain partitions in a table instead of dropping and recreating the entire table daily (with lower time travel storage costs when changing a partition, versus maintaining entire copies of the data in the time travel window). Alternatively, if you need to drop and recreate a table daily, it would make sense to reduce the duration of the respective dataset's time travel window to two days—which means five fewer copies of the table exist in the time travel window.

For analysis purposes, the [table storage information schema](#) can be used to better

understand the storage usage of any table. This includes information about the physical size of the 'live' table and the respective table's time travel storage consumption.



In addition to time travel, BigQuery also offers a 7-day [fail-safe period](#). This means that if you accidentally delete data, it will be retained for additional 7 days in fail-safe storage after the time travel window. The fail-safe period is non-configurable and you cannot query or recover data directly from fail-safe storage. To recover data from fail-safe storage, contact [Cloud Customer Care](#).

Check if physical storage billing model is a good fit for you

As we mentioned earlier, by default your dataset storage is billed based on logical bytes used. However, you also have the option to use physical bytes as a [billing model](#) for your dataset storage. The cost of physical storage is higher than that of logical storage, but it is a good option if your data compresses well. This is because you will be billed based on the physical bytes stored in a dataset, regardless of how much logical space they take up.

When you set your storage billing model to use physical storage, the total storage costs you are billed for include the bytes used for [time travel](#) storage and [fail-safe](#) storage. As you set the storage billing model as dataset level, you could optimize more by having a mix of logical and physical bytes models in the organization, please look at [this example](#) to calculate the price difference per dataset using [Table Storage](#) view.

Configure 'time to live' for your transient data

If you don't need to look at data after initial analysis, apply an expiration date to its table or partition. You can configure the [default table expiration](#) for transient datasets, as well as the [expiration time for tables](#) and the [expiration time for partitions](#).

Use snapshots for longer backup

A BigQuery table snapshot preserves the contents of a table (called the base table) at a particular time. You can save a snapshot of a current table, or create a snapshot from any time within the time travel window (which covers the past seven days by default). A table snapshot can have an expiration date, and you can query a table snapshot as you would a standard table. While they're read-only, you can create (restore) a standard table from a table snapshot, and then modify the restored table.

With table snapshots, you can:

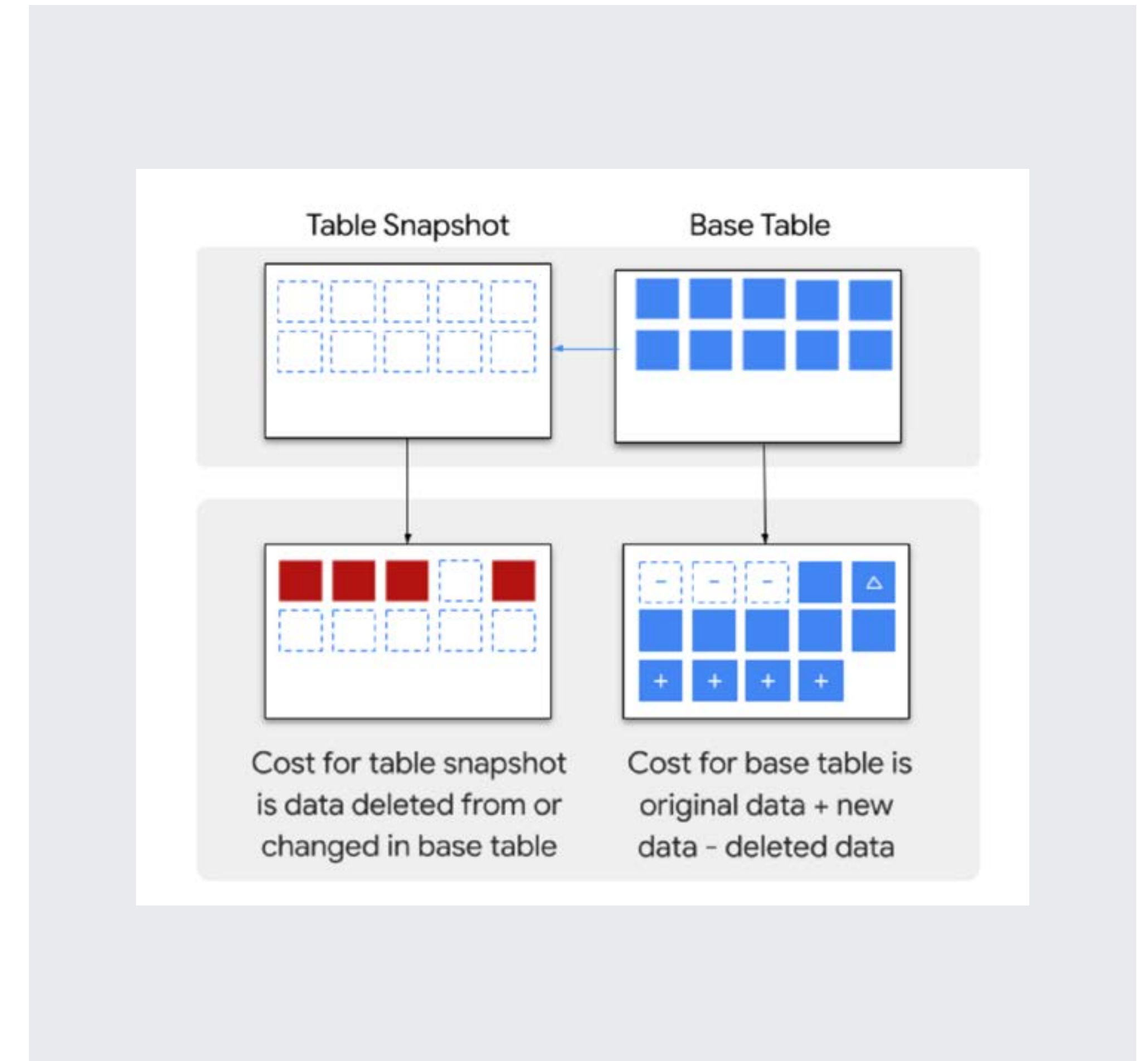
- Keep a record for longer than seven days. Time travel only provides access to data from the past seven days (by default). With table snapshots, you can preserve a table's data for as long as you want.
- Minimize storage cost. BigQuery only stores bytes that are different between a snapshot and its base table, so a table snapshot typically uses less storage than a full copy of the table.

[Storage costs](#) apply for table snapshots, but BigQuery only charges for data in a table snapshot that no longer exists in its base table, or that has changed in its base table.

Just remember:

- When a table snapshot is created, there is no initial storage cost for the table snapshot
- If you change or delete data in the base table—and this data also exists in the table snapshot—then you are charged for the table snapshot storage of the changed or deleted data

For example:



Use table clones for modifiable copies of production data

A [table clone](#) is a lightweight, writable copy of another table (called the base table). Other than the billing model for storage, and some additional metadata for the base table, a table clone is similar to a standard table—you can query it, make a copy of it, delete it, and so on.

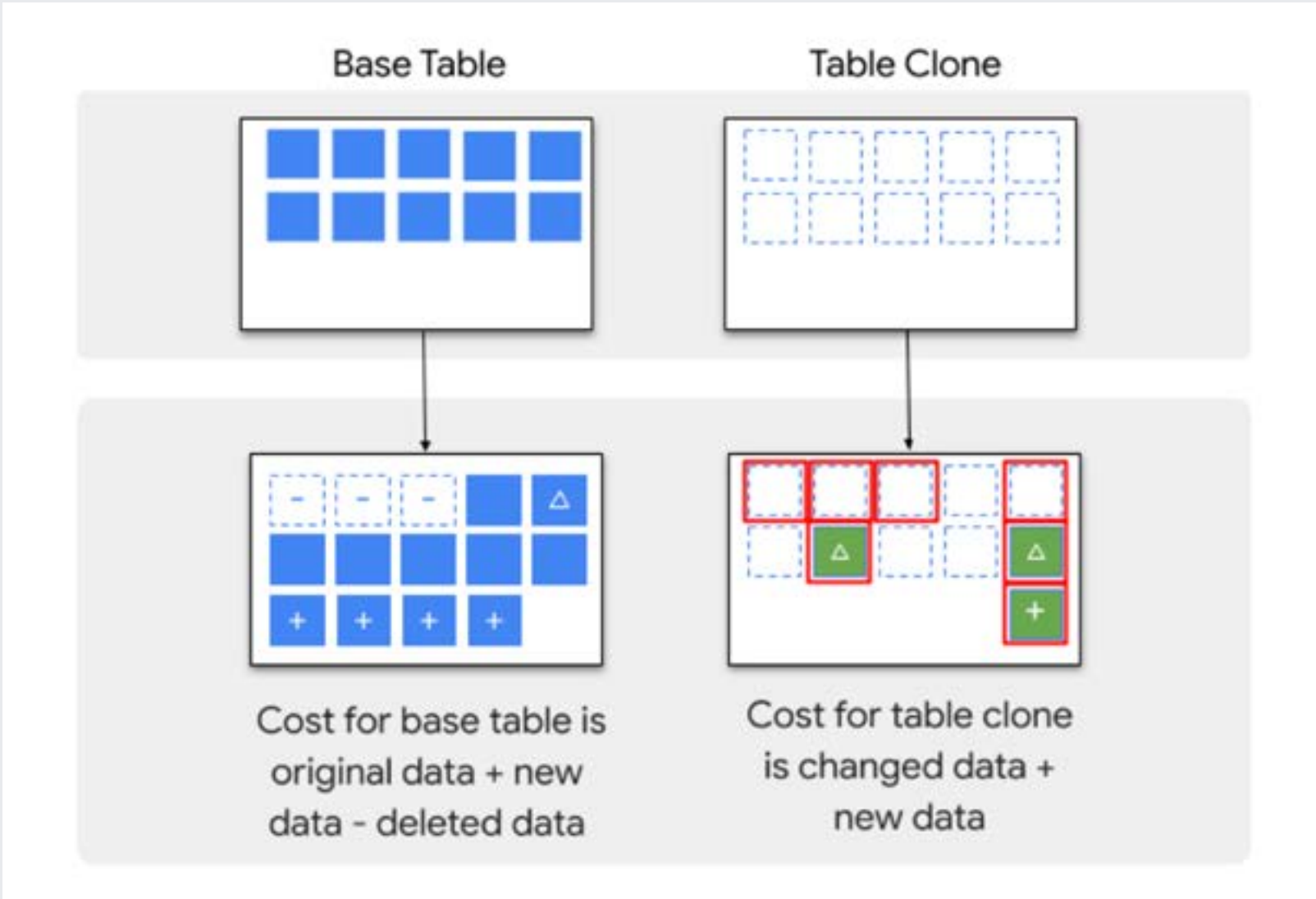
After you create a table clone, it is independent of the base table. Any changes made to the base table or table clone aren't reflected in the other.

[Storage costs](#) apply for table clones, but BigQuery only charges for the data in a table clone that differs from the table clone's base table.

Just remember:

- When a table clone is created, there is no initial storage cost for the table clone
- If data is added or changed in a table clone, then you are charged for the storage of the added or updated data
- If data is changed or deleted in the base table—and this data also exists in the table clone—then you are charged for the table clone storage of the changed or deleted data
- If data is added to the base table after the table clone was created, then you aren't charged for storage of that data in the table clone
- Partitions can help reduce storage costs for table clones—generally, BigQuery only makes a copy of modified data within a partition, instead of the entire table clone

For example:

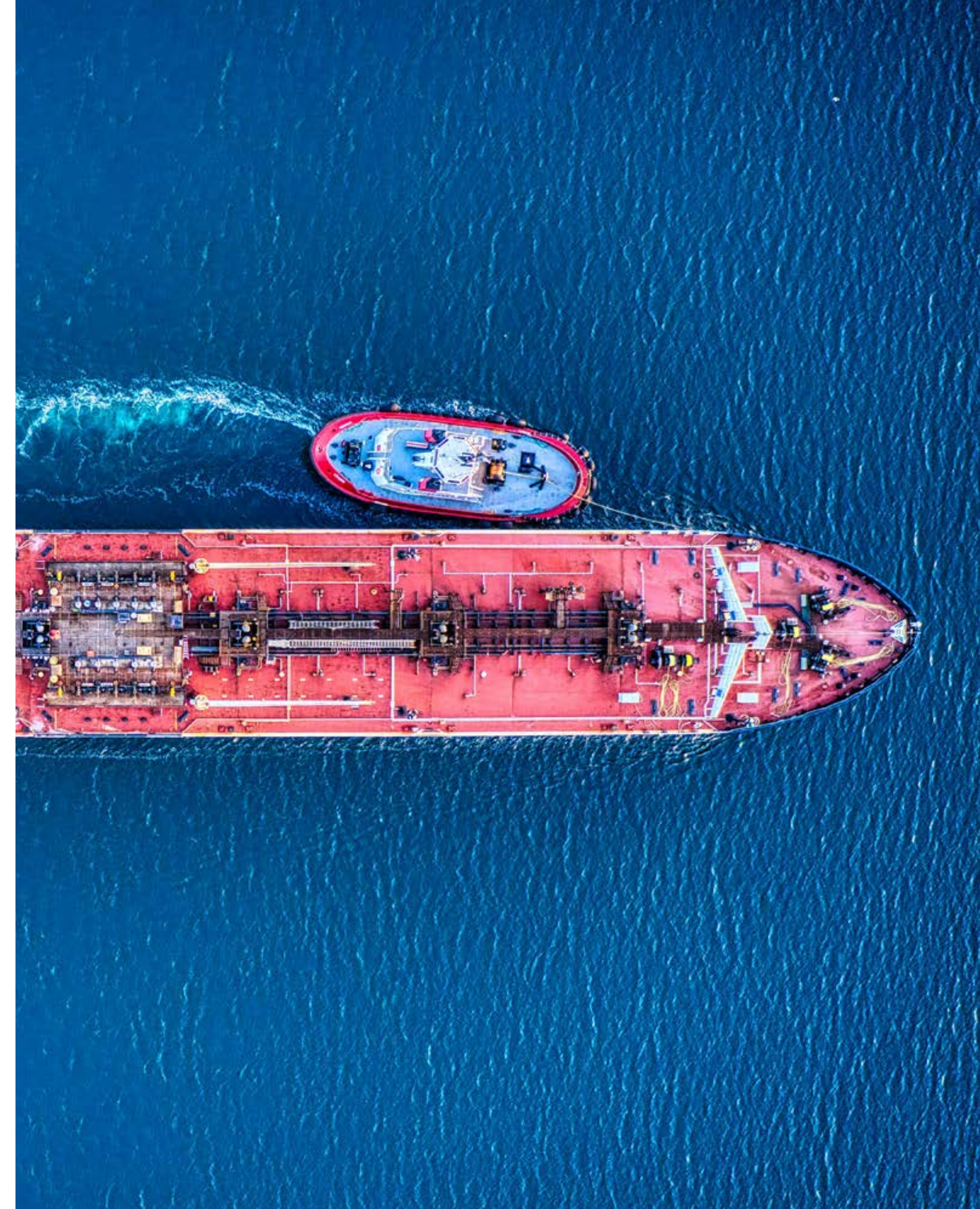


Archive data into a new BigQuery table

In certain scenarios, it may make sense to move older data into a new BigQuery table. This helps you avoid the additional overhead of scanning all rows in a table if a partition is not specified.

For example, you may have 10 years of sales history in a table, but 99% of queries only need data from the last three years. You could set up two views of the table—one that filters for just the last three years, and the other exposing all data—yet end users would still have access to the underlying table. In other words, they're not forced to use the views. Instead, copying the seven oldest years of data into a new BigQuery table may be better, particularly if query performance is the primary objective.

Just note that this can complicate ETL processes, which may have to maintain the 10 years of historical data—because two tables need to be maintained instead of a single table. Also, note that creating a snapshot isn't a valid option because you can't take a snapshot of a subset of a table.



Move ‘cold data’ to Google Cloud Storage

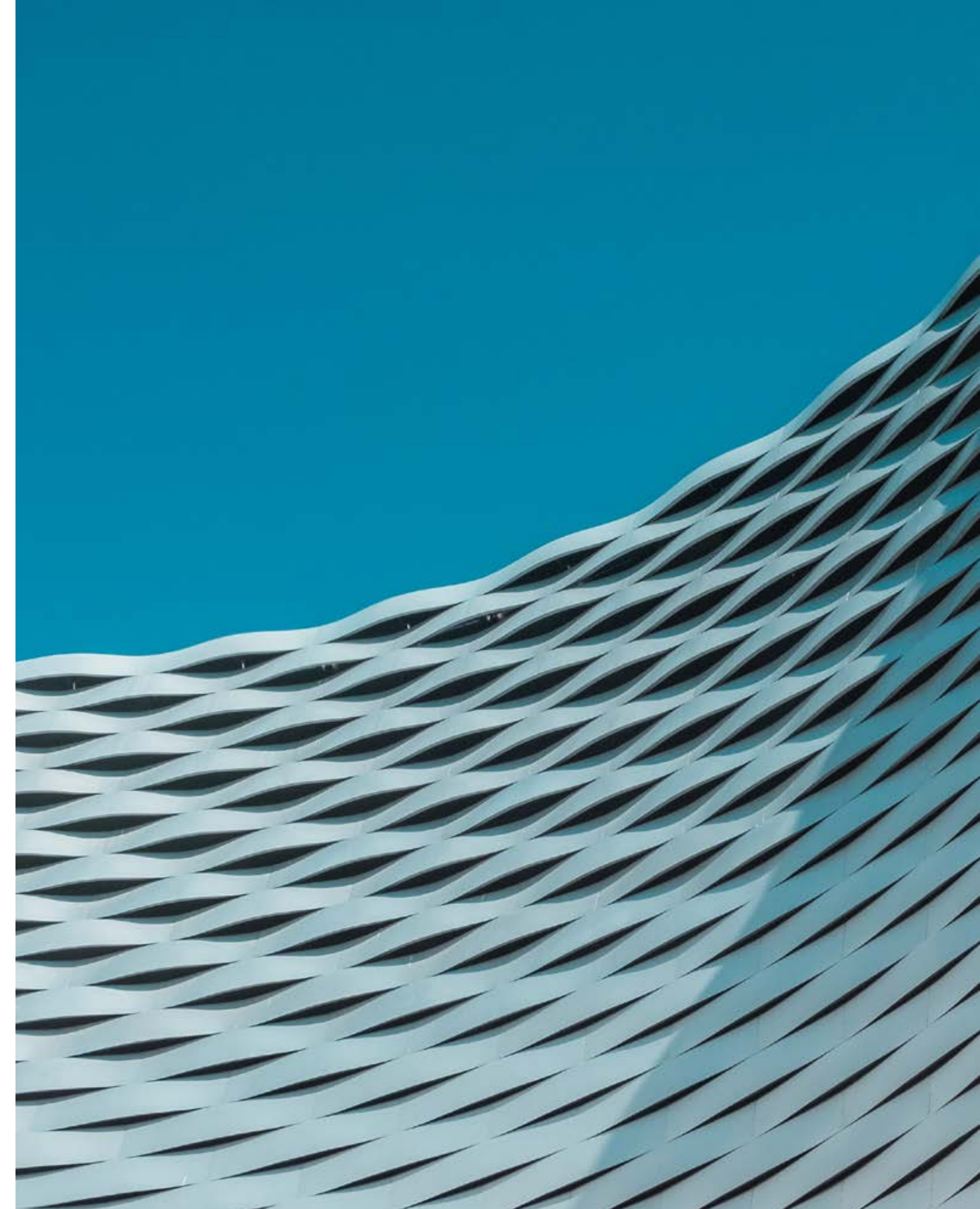
If you don’t want to use time travel, snapshots, cloning, or archiving, you could create a back-up by offloading data from BigQuery to Cloud Storage.

There are four tiers of [Cloud Storage pricing](#), applicable to multi-regional, dual-regional, and regional locations—Standard, Nearline, Coldline, and Archive. To optimize your costs, you need to consider three variables:

- **Retrieval cost:** This is free for Standard, with the price per GB scanned increasing as you move from Nearline through to Archive. Note that Nearline, Coldline, and Archive Storage have minimum retention periods.
- **Storage cost.** Standard is the most expensive (comparable to BigQuery, depending on the file type chosen) and Archive is the cheapest.
- **Performance.** Speed/Performance of retrieval is similar for all GCS storage classes; however, if speed of retrieval matters, consider offloading to a BigQuery table instead.

As a rule of thumb, the frequency of data retrieval is the top factor in choosing the optimal Cloud Storage for BigQuery data archival.

You can use the [BigQuery console, bq extract, export data statement or an extract job using BigQuery API or client libraries](#) to move data from BigQuery to Cloud storage. [Dataplex](#) provides an automated way to [tier data from BigQuery to Cloud Storage](#).



04

Compute costs

and best practices

in BigQuery

Compute costs and best practices in BigQuery

Now, let's turn our attention to compute costs, which usually make up the largest part of BigQuery spend. These costs are generated by processing queries, including SQL queries, user-defined functions, scripts, materialized views, and certain DML and DDL statements that scan tables.

In this chapter:


- Cost control basics
- Compute pricing models and how to mix them
- Best practices for managing workloads
- Ways to reduce costs

Keeping a close watch on your costs

Pre-emptive measures will help you control costs on BigQuery. Start with [budget alerts](#) in the Cloud Billing console. You can also [control cost](#) in other ways—such as knowing your cost before you run a query, and setting custom quotas at the project or user level to prevent accidental charges. These prove extremely useful with the on-demand consumption model where you pay for data processed by each query.

[Billing reports](#) give you deeper insights into analysis costs. And, for even deeper analysis, you can export your billing data into BigQuery and consume it with your favorite BI tool. A

detailed Looker Studio dashboard allows you to identify costly queries and then optimize for cost and query performance. It also provides insight into the usage patterns and resource utilization associated with your workload.

 Use this [step-by-step guide](#) for Looker Studio

[BigQuery admin resource charts](#) are also helpful in monitoring costs and performance, and optimizing your workload.



How compute costs are calculated

There are two pricing models for running queries:

- On-demand is the default model. With this pricing model, you are charged for the number of bytes processed by each query. The first 1 TB of query data processed per month is free.
- Editions [Reservations](#) charge for the number of slot_sec used by each query (a slot_sec equals one slot for one second). A slot is a unit of measure for BigQuery compute power (like vCPU for virtual machines). For example, a query using 100 slots for 60 seconds will accrue 6000 slot_sec.

As a simpler model, most customers begin with on-demand pricing. Over time, they may move some of their workloads to Editions, which enables them to use BigQuery more efficiently. The best part? One of BigQuery's unique features is the ability to combine the two different pricing models to optimize your costs. In other words, you can mix and match the models to suit different workloads.

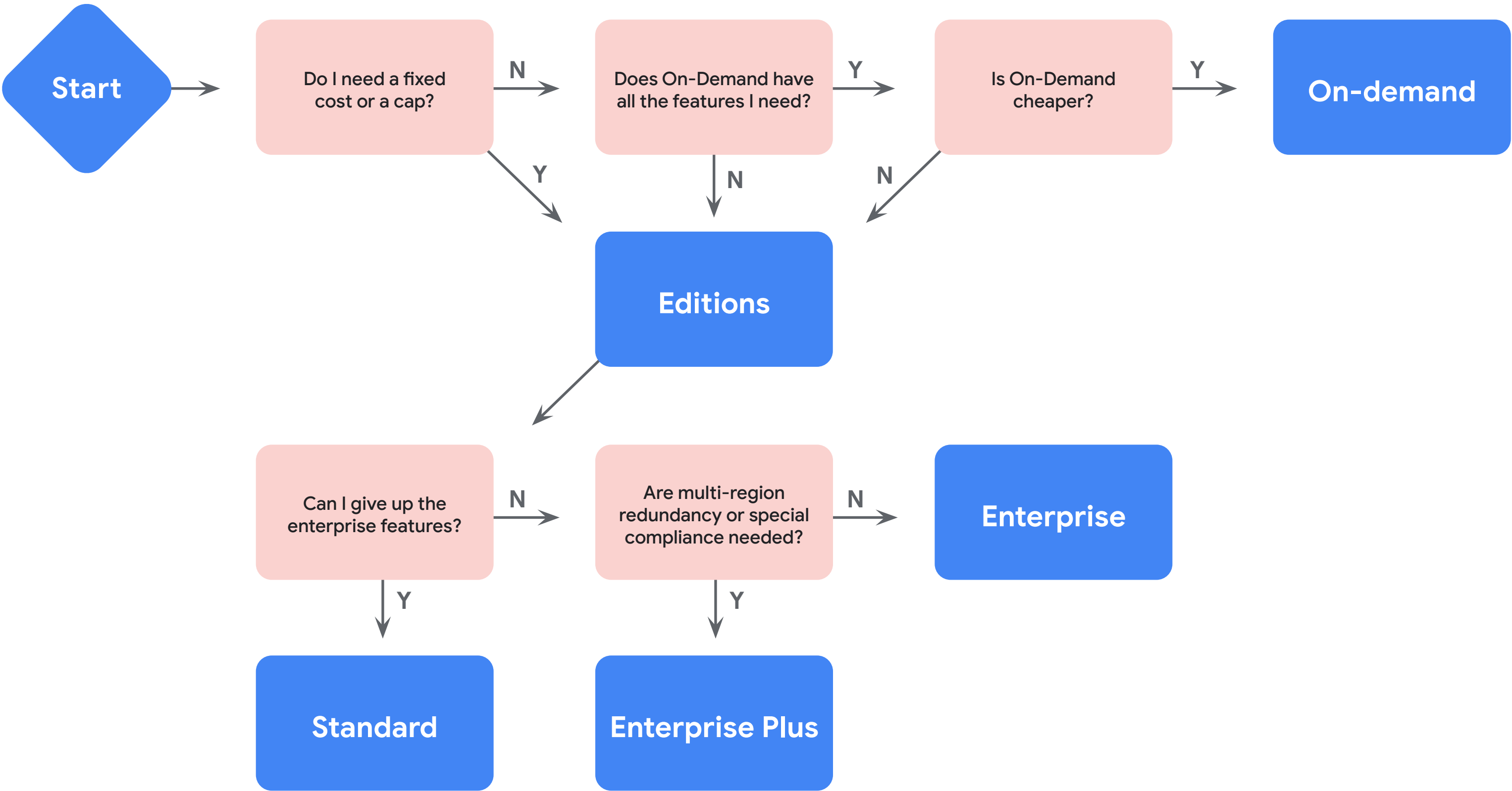


Choosing the right consumption model for your workload

How do you know which pricing model is best? It can depend on a number of factors, outlined in the diagram and explained below.

Control versus simplicity

On-demand is the easiest way to use BigQuery, but reservations give you more cost control. With the on-demand model, cost control is achieved by creating [custom quotas](#). For example, you can set the maximum amount of bytes that can be processed by a single user on a given billing project—when this limit is passed, the user gets a ‘quota exceeded’ error. It’s a hard stop that can only be resolved by raising the quota. Reservations, on the other hand, don’t stop the user from executing their queries even when the maximum amount of slots is in use. By setting the baseline equal to the maximum, Enterprise and Enterprise Plus Reservations allow for fixed cost setups.



Decision flow for the consumption model

Features

Next, you should check if on-demand has all the features required for your workload. The table below summarizes the various features available among on-demand and three editions.

	Standard Edition	Enterprise Edition	Enterprise Plus Edition	On-demand
Unit of measure	Slot-second (1 minute minimum)	Slot-second (1 minute minimum)	Slot-second (1 minute minimum)	Bytes processed (with free tier)
Cost control	Max slots	Max and Baseline slots	Max and Baseline slots	Quotas
Committed Use	-	Discounted 1y and 3Y commitments	Discounted 1y and 3Y commitments	-
Monthly Uptime SLO	99.9	99.99	99.99	99.99
Recovery Time Objective (RTO)	Best effort	0 (zonal failure) Best effort (regional failure)	0 (zonal failure) 5 min (regional failure, in roadmap)	0 (zonal failure) Best effort (regional failure)
Compute Redundancy Guarantee	No guarantee	Regional (two zones)	Multi-regional (2 regions, in roadmap)	Regional (two zones)
Compliance	Foundational compliance and HIPAA	Foundational compliance and HIPAA	Foundational compliance, HIPAA, Enhanced compliance through assured workloads	All compliance
VPC Service Controls	-	VPC-SC Support	VPC-SC Support	VPC-SC Support
Data Governance	-	Column security Row security Data masking	Column security Row security Data masking	Column security Row security Data masking
Storage Encryption	Google-managed keys	Google-managed keys	Customer-managed keys (CMEK) Google-managed keys	Customer-managed keys (CMEK) Google-managed keys

	Standard Edition	Enterprise Edition	Enterprise Plus Edition	On-demand
Materialized Views	Direct query of MVs	Create and manage MVs with automatic query rewrite	Create and manage MVs with automatic query rewrite	Create and manage MVs with automatic query rewrite
Business Intelligence	-	Query acceleration through BI Engine	Query acceleration through BI Engine	Query acceleration through BI Engine
Caching	User result set caching	Cross-user result set caching	Cross-user result set caching	User result set caching
Search	-	Query acceleration through search indexes	Query acceleration through search indexes	Query acceleration through search indexes
Unstructured Data	Query object tables	Object tables with ML inference	Object tables with ML inference	Query object tables
Machine Learning	-	BQML	BQML	BQML
Workload Management	Basic WLM Query Queues	Advanced WLM Idle capacity sharing Query Queues	Advanced WLM Idle capacity sharing Query Queues	Basic WLM Query Queues
Supported Assignment Types	QUERY, PIPELINE	QUERY, PIPELINE, ML_EXTERNAL, BACKGROUND, SPARK	QUERY, PIPELINE, ML_EXTERNAL, BACKGROUND, SPARK	-

Features

Typically, Standard Edition is suitable for dev environments, ad-hoc analysis, POCs and workloads that require low compute and no-redundancy. Majority of the compute heavy production workloads would fall under Enterprise Edition including machine learning, advanced workload management, workloads with zonal and multi-cloud requirements. Workloads with requirements of high compliance and multi-regional redundancy (on roadmap) are perfect fit for Enterprise Plus Edition. BigQuery provides an easy way to mix and switch editions for workloads within the same GCP organization, and you can do this without any movement in data.

Cost savings

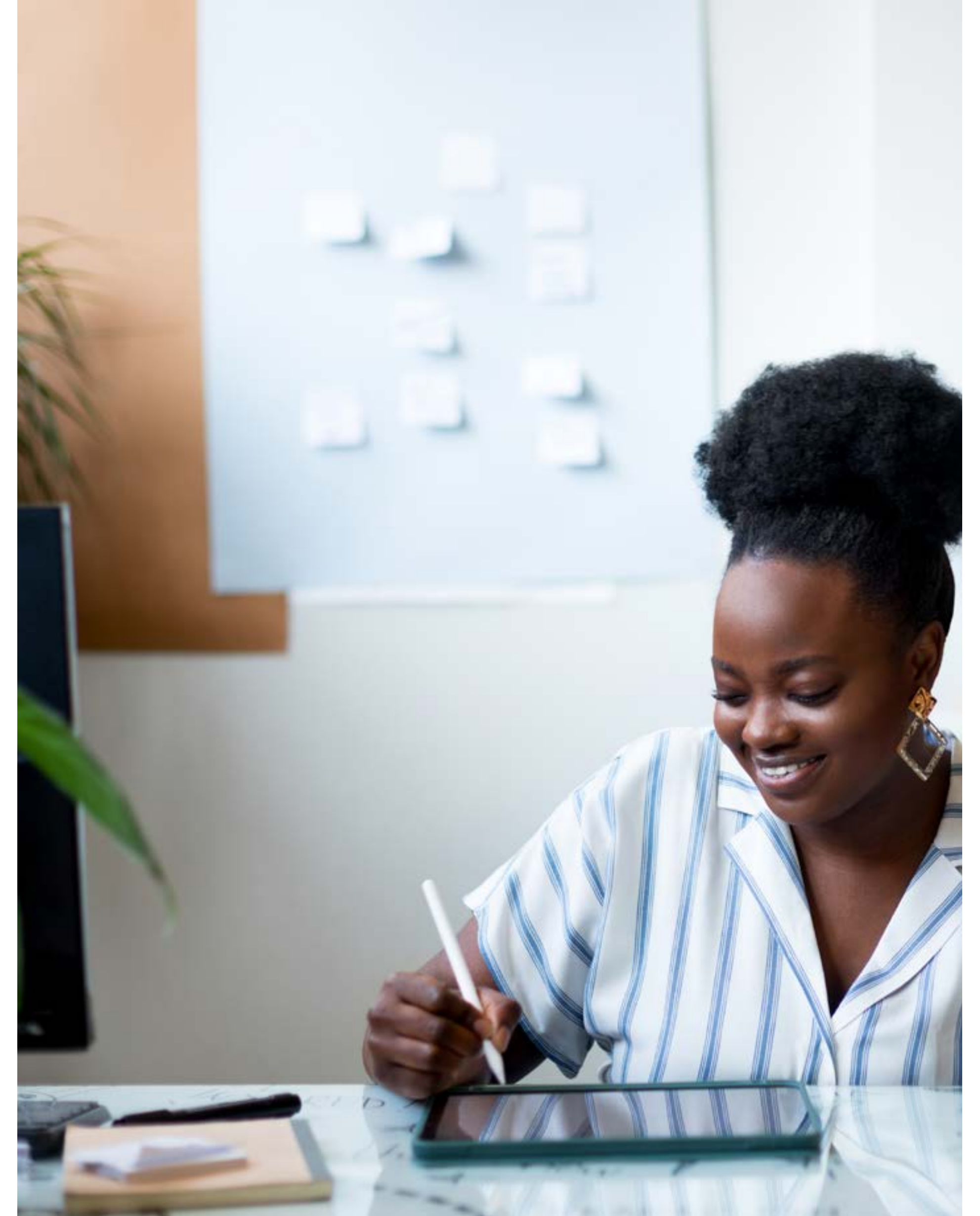
Depending on the query, one pricing model can be more cost efficient than the other. On-demand pricing, based on bytes processed, is IO-related. Autoscaling pricing, based on slots, is CPU-related. So a job that does a lot of IO but lightweight CPU processing can cost less under autoscaling pricing. Conversely, a CPU-intensive job with limited IO can cost less under on-demand pricing.

To calculate the cost of a job under both models, look at the `total_bytes_billed` and the `total_slot_ms`, which are both available in the [job information](#).


```
on_demand_cost = (total_bytes_billed *  
  cost\_per\_tib) / 2^40  
  
slot_based_cost = (total_slot_ms *  
  cost_per_slot_h) / (1000 * 60 * 60)
```

Note that `slot_based_cost` may be underestimated since it doesn't take into account the autoscaler behavior. Yet, if `on_demand_cost < slot_based_cost` this will be true even considering autoscaling. This can also be useful when [autoscaler behavior](#) can be ignored, for example when the target reservation has autoscaling disabled (`max=baseline`).

Slight changes in a query can impact the cost—as can data changes and query parameters. The same recurring query may be cheaper under the on-demand model on one occasion, and not another. Or, it may drift over time. Given this, optimization is not easy to achieve at scale—yet it is possible, as we will explore further down.



Best practices on workload management

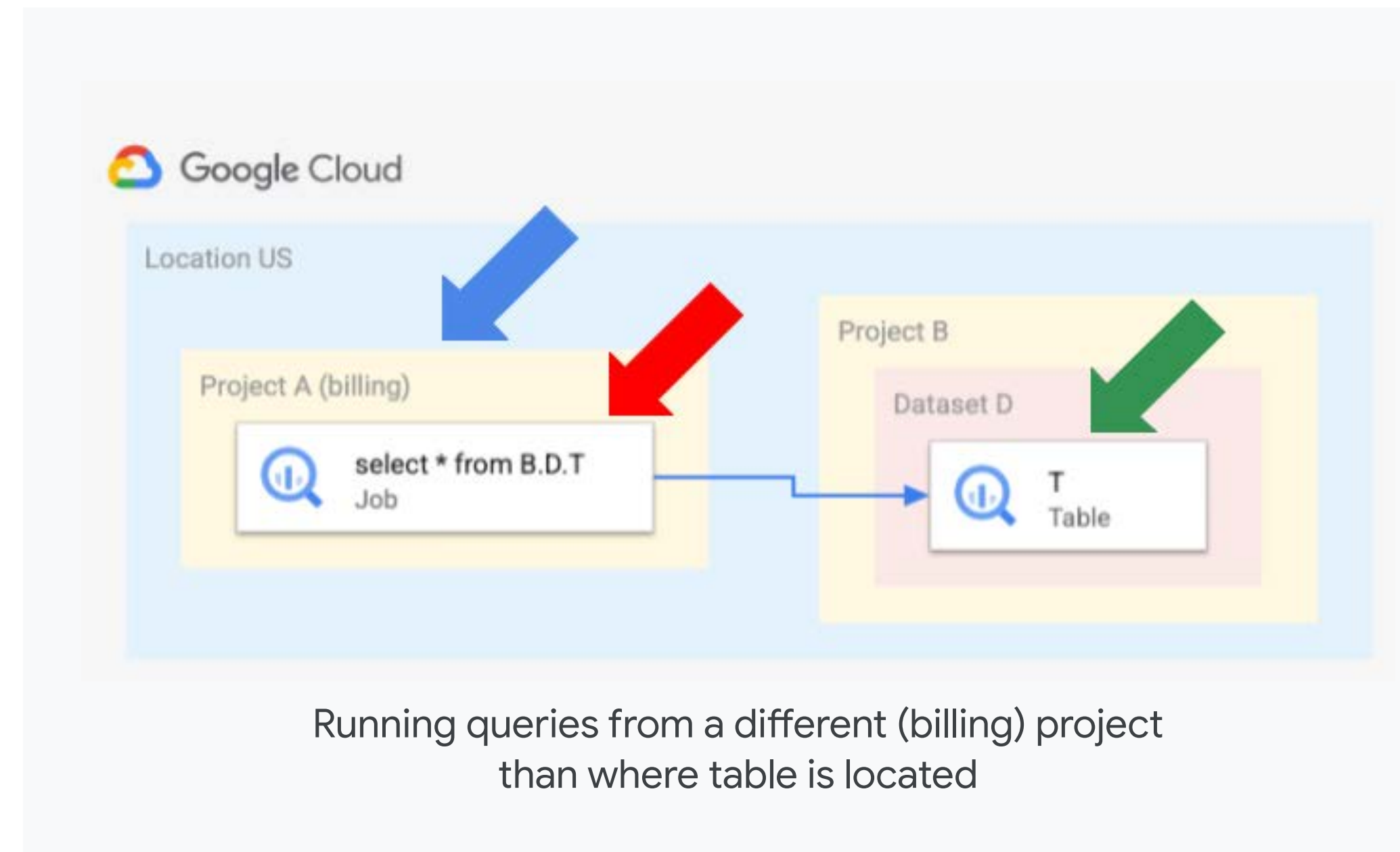
 Need a recap? Review our documentation on [reservations](#) and [BigQuery Editions](#)

Use multiple billing projects

In BigQuery, you can create datasets and tables in one project and run queries from others. A project where you run queries, known as a billing project, accrues the compute costs of a query. Think of it like the computing power source that will run the query. The billing project is specified when the query is launched and the query in itself doesn't depend syntactically on it. In other words, you can run the same query from different billing projects without changing the query, and easily move workloads between different billing projects just before launch.

```
bq --project_id=A ... 'SELECT ... FROM  
B.D.T GROUP BY ...'
```

By allocating jobs (queries) on different billing projects, you are actually allocating jobs on different cost centers. This not only simplifies reporting, but it also enables you to allocate and control expenditure across workloads. For example, you can assign [custom quotas](#) to each billing project.

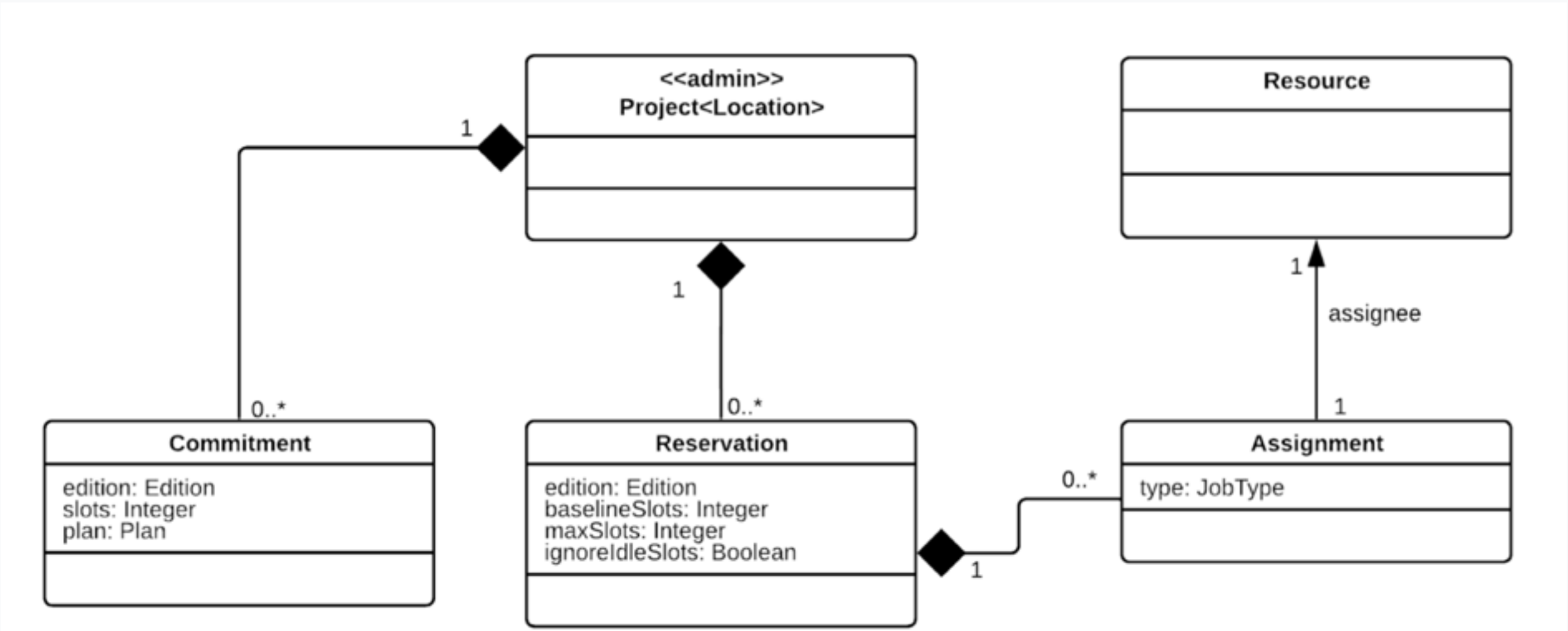


Mixing and switching pricing models

To better understand how to optimize pricing, it pays to know how reservations, commitments, and assignments are related.

As this diagram shows, an admin project is one with reservations. If we were to imagine slots as a liquid, they would flow through an admin project into reservations, which then split the input flow into output flows via assignments. An assignment can assign slots to a resource (which can be the organization, a folder, or a billing project).

With this in mind, it gets a little easier to grasp how the pricing models work. It all depends on whether the billing project has slots assigned, and where they come from.



Reservations and related concepts

As long as you have the right permissions, you can move workloads from one pricing model to another at any time by changing the billing project when you run the job.

Does an assignment point to the billing project* ?	What kind of reservation feeds the assignment?	Pricing model of the billing project
No	N.A.	On-demand
Yes	A ‘none’ reservation **	On-demand
Yes	A reservation	Editions

Assignments and the pricing model

* A project can also get slots indirectly, if an assignment points to one of its parent folders or to the organization.
** A ‘none’ reservation is a special reservation that imposes the on-demand model.

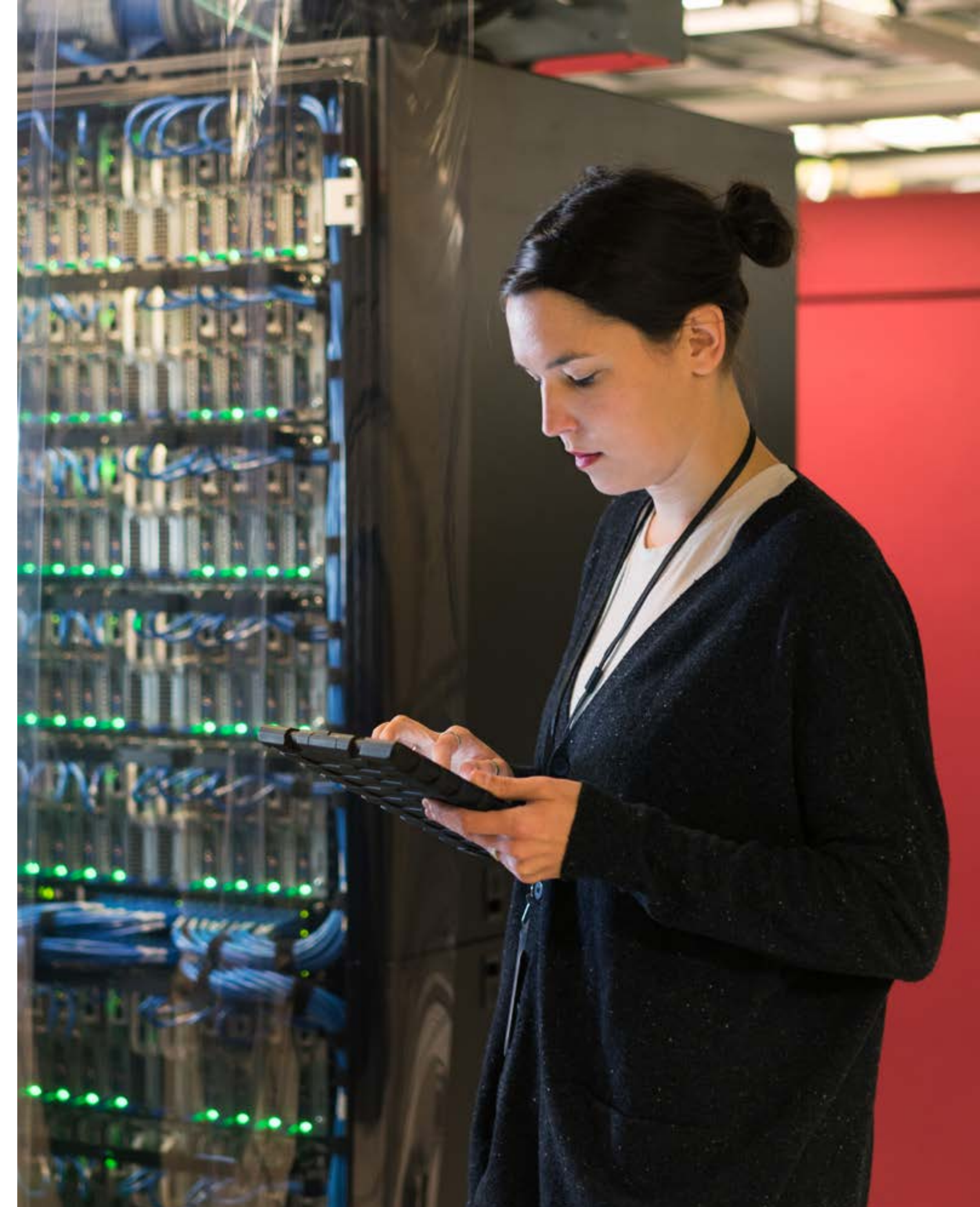
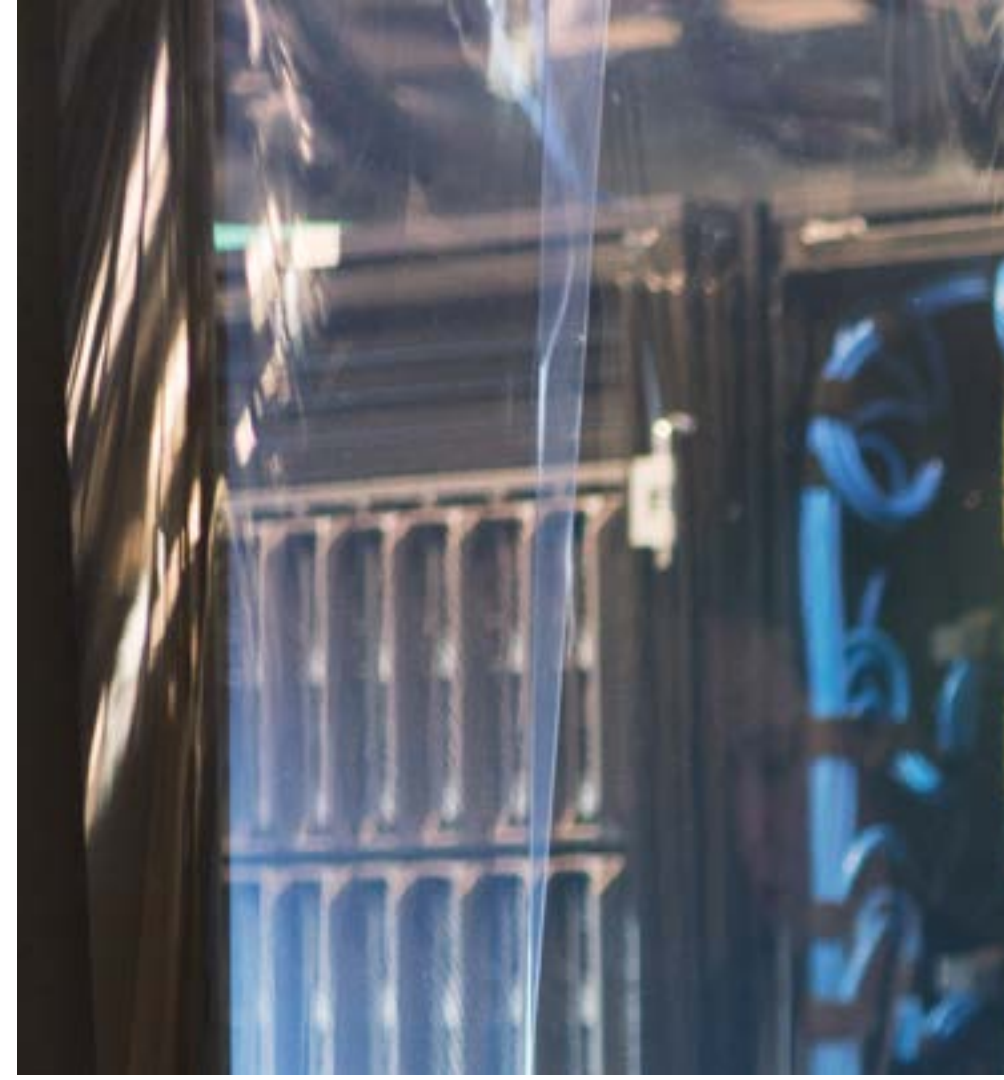
Know how many slots you need

It is important to understand how many slots are needed before configuring a reservation.

To help here, Google Cloud provides tools like:

- [Slot recommendations and insights](#) – use this to study existing on-demand usage
- [Slot estimator](#) – if you're already using reservations, this allows you to analyze at an organization or project level

Or, you can do it yourself. INFORMATION_SCHEMA tables will show your usage. The documentation contains some useful [examples](#). Metric [estimatedRunnableUnits](#) can be useful since it tells you if the query needs more slots or not.



Use separate reservations for compute intensive workload

To create an [Edition](#) reservation, you assign a reservation a maximum number of slots. Ideally, this would be as high as possible, because more slots means that queries are executed faster. Yet a very high number can be a liability, as a runaway query could burn a lot of money (slot_sec) before it's spotted and terminated. In other words, there's a tradeoff between performance and risk.

There's a good solution here. You can use more than one reservation—a high maximum reservation for critical jobs that have been thoughtfully tested, and a low maximum reservation for less-tested jobs and interactive queries. This way, risky queries run in a reservation where the potential damage is mitigated by the lower cap.

Multiple reservations also help allocate costs across the organization. For example, you can assign a certain number of slots to each business unit and set up their reservations accordingly. In this way each business unit will pay only what was allocated to them.



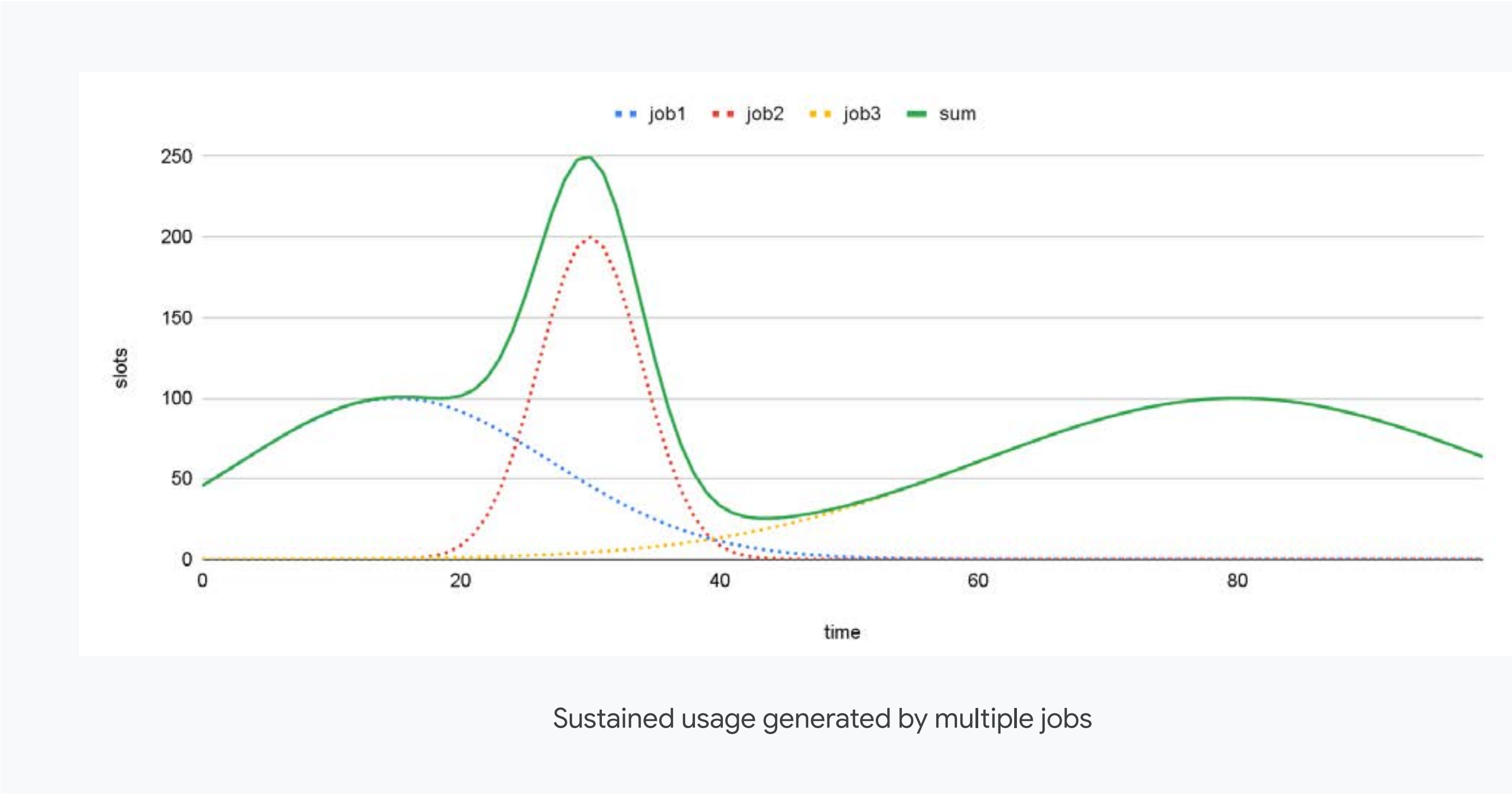
Use baseline slots for critical reservations

You can optionally specify a baseline number of slots (or ‘granted’ slots) per reservation, which can be useful in ensuring you get the slots you need for a project. The baseline effectively sets a minimum number of slots that will always be allocated to the reservation, and you will always be charged for them

Use commitments for sustained usage

Another way to get granted slots is by creating a commitment. Coming in at a discounted price, commitments make sense when you have sustained slot usage in an admin project. You can have multiple commitments on the same project, and you can add them as needed over time.

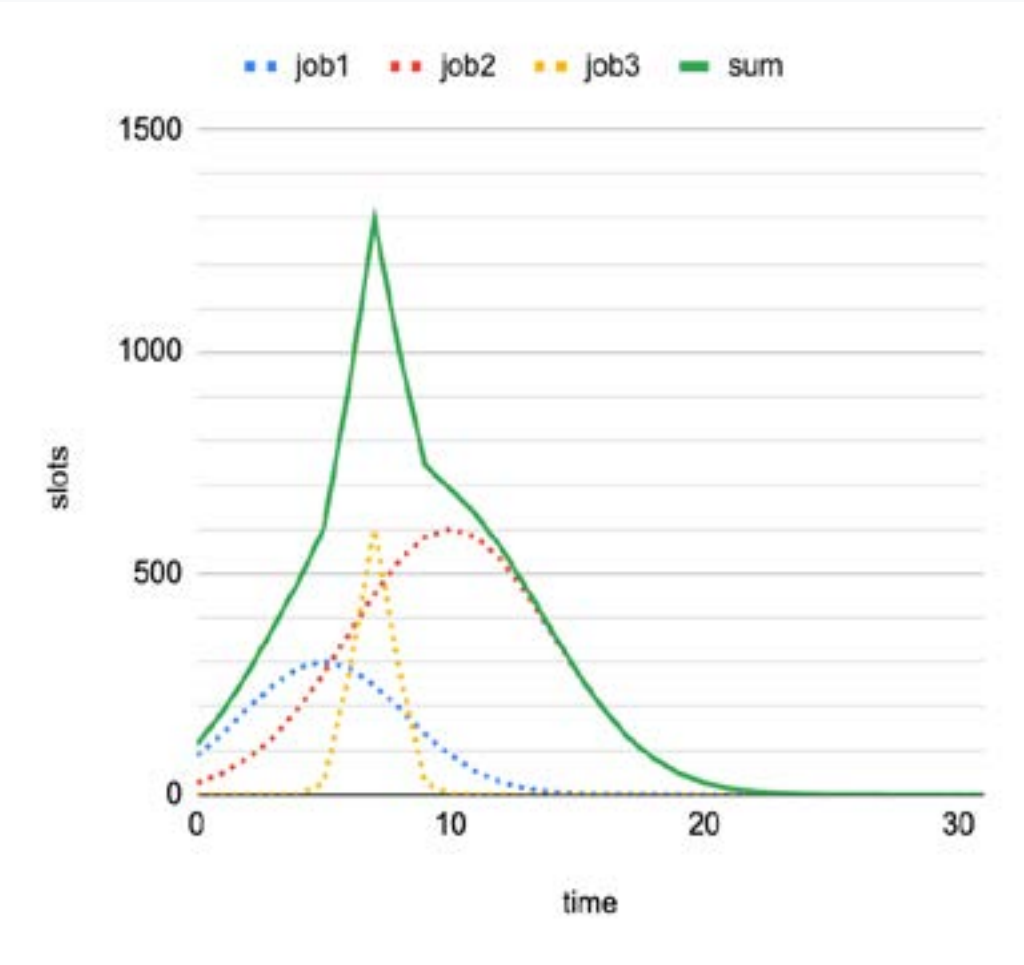
Just remember that you pay for commitments even when there’s no usage. But sustained usage can form organically from the superimposition of many jobs running on the same admin project.



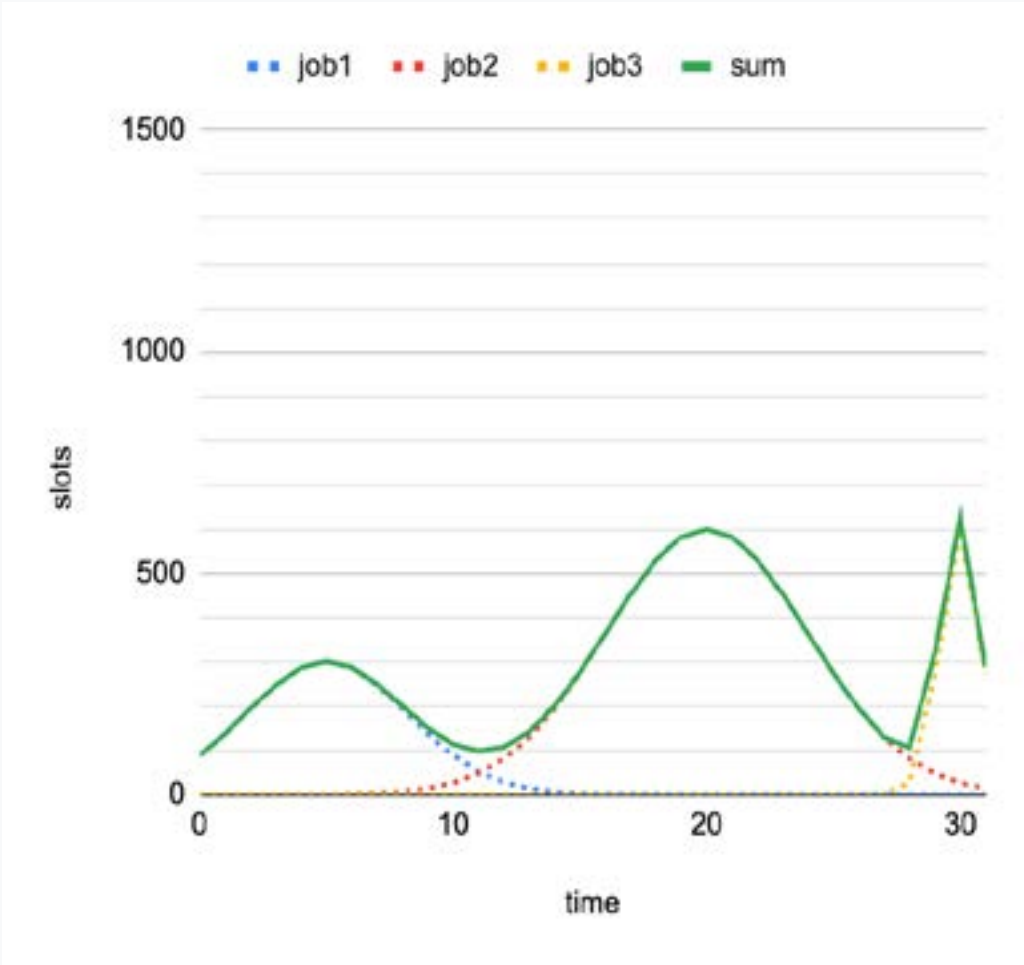
You can improve sustained usage by distributing the same jobs more evenly over time.

For example:

- Instead of computing all weekly jobs on a Monday, spread them over the week
- Instead of performing all steps of a complex computation together, pre-compute some sub-steps
- Instead of processing all data before a deadline, process incrementally as soon as they arrive



Multiple jobs running at the same time, using a high number of slots in peak period



Distributing same jobs over time to create sustained usage

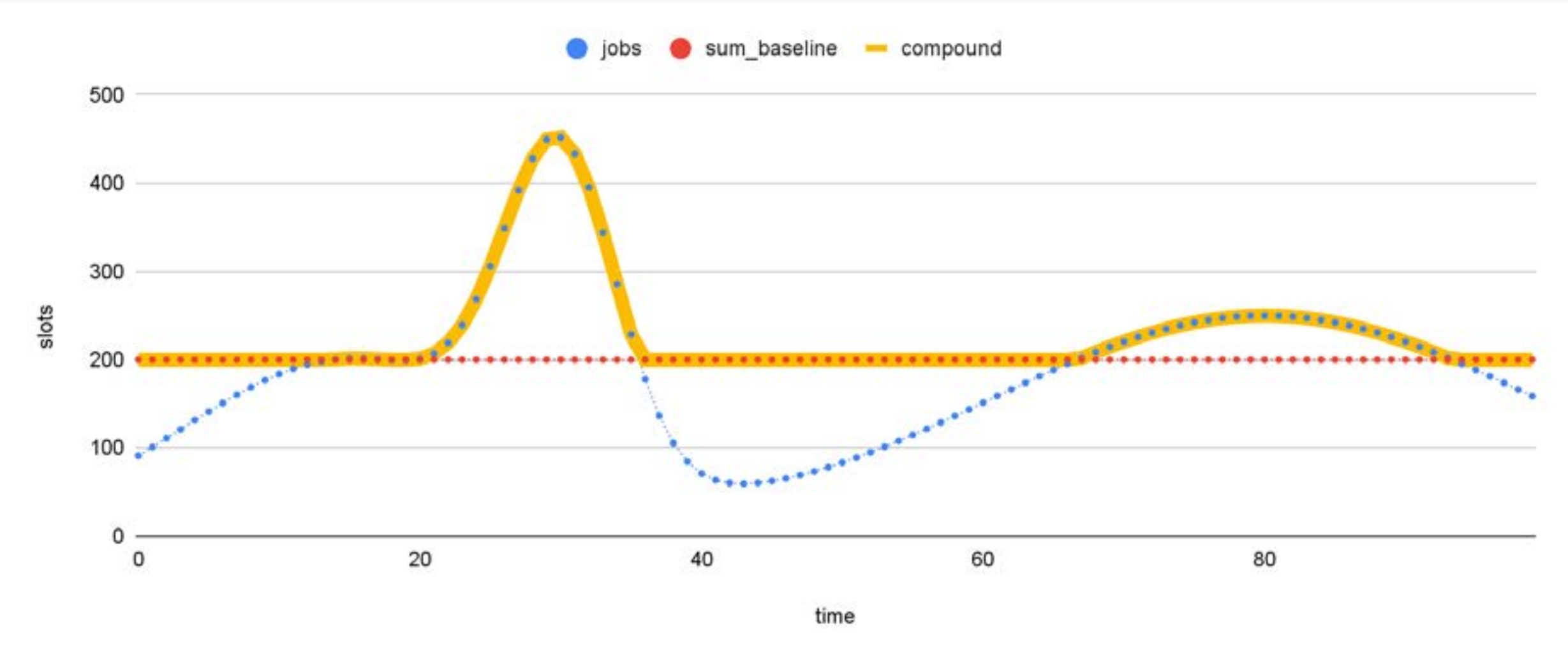
We define compound usage as:

$$\text{compound_usage} = \text{MAX}(\text{jobs_usage}, \text{SUM}(\text{baseline}))$$

Compound usage represents the amount of slots you are always consuming because of actual workloads or base lines. It makes sense to cover stable compound usage with Commitments to take advantage of their discounts. The example 1 shows a possible setup for the case illustrated by the graph.

- Admin project AP
 - Reservation R1: baseline 200 slots
 - Commitment C1: 200 slots

Example 1: Commitment and baseline



Compound usage over time

Take advantage of slots sharing

BigQuery can share idle slots across reservations, so you can allocate your budget while minimizing wasted resources. Let's unfold this topic using a sequence of examples.

Option 1: Share nothing

You could keep everything separated, with each group allocated a reservation and commitment to cover the observed sustained usage at discounted price.

- Admin project P1
 - Commitment C1: 100 slots
 - Reservation R1
- Admin project P2
 - Commitment C2: 200 slots
 - Reservation R2

Example 2: Share nothing with multiple admin projects

In this case, every workload group has its own resources. It's nice and clean—but there can be a better way.



Option 2: Shared admin project

By using a single admin project for more than one reservation, you gain access to idle slots sharing (only for Enterprise and Enterprise Plus reservations). This powerful feature of BigQuery lets you share baseline slots across the reservations of an admin project.

Let's build some terminology. First, we define the 'slots allocated' to an admin project. When a commitment is created or a baseline >0 is set up for a reservation, those slots are allocated permanently to the admin project from the shared pool. No-one else can use them—which means you're charged for them regardless of whether you use them or not.

```
slots_allocated = MAX(SUM(commitment),  
SUM(baseline))
```

The MAX comes from the fact that whether a slot is pushed by a commitment or drained by a baseline (or both), it's allocated specifically to that admin project.

We also define the 'slots requested' to an admin project as the sum of all slots consumed by the jobs running on the billing projects linked to the admin project.

```
slots_requested = SUM(job_slots)
```

Next, we define the 'slots autoscaled' to an admin project. This is the number of extra slots needed beyond the slots allocated. Please note that the autoscaler scales by multiples of 100. For example, if there is a commitment of 100 slots and jobs are consuming 150, then the autoscaler provides 100 extra slots.

```
slots_autoscaled = CEIL(MAX(0, slots_  
requested - slots_allocated), 100)
```

Finally, we define 'slots wasted' as the number of slots allocated or scaled but not used.

```
slots_wasted = MAX(0, slots_allocated +  
slots_autoscaled - slots_requested)
```

With the above definitions in mind, let's look at some scenarios where sharing slots can deliver savings.

Option 2: Shared admin project

Savings on baselines

Imagine a single admin project with multiple reservations.

Admin project AP

- Reservation R1: baseline 100 slots
 - Assignment A1: assignee is P1
- Reservation R2: baseline 0 slots
 - Assignment A2: assignee is P2

Billing project P1

- No jobs running

Billing project P2

- A job is consuming 50 slots

Example 3: Idle slots shared and no autoscaling

Looking at the above admin project, you may wrongly think that since no jobs were running in P1, all 100 slots of the baseline of R1 were wasted.

And, since P2 is bound to R2 and R2 has no baseline, 100 slots were allocated by the autoscaler.

In fact, by applying our previous definitions, we get:

- $\text{slots_allocated} = \text{MAX}(0, 100) = 100$
- $\text{slots_requested} = 50$
- $\text{slots_autoscaled} = \text{CEIL}(\text{MAX}(0, 50 - 100), 100) = 0$
- $\text{slot_wasted} = \text{MAX}(0, 100 + 0 - 50) = 50$

Note how no slots are autoscaled, and only 50 are wasted. Thanks to slots sharing, the 100 allocated slots are also available to R2—meaning there are more than enough to cover the 50 slots requested by the job running on P2.

Now let’s assume that the billing project P2 on the same job actually needed 150 slots.

- Admin project AP**
 - (as before)
- Billing project P1**
 - (as before)
- Billing project P2**
 - A job is consuming 150 slots

Example 4: Idle slots shared and autoscaling

By applying our previous definitions we get:

- $\text{slots_allocated} = \text{MAX}(0, 100) = 100$
- $\text{slots_requested} = 150$
- $\text{slots_autoscaled} = \text{CEIL}(\text{MAX}(0, 150 - 100), 100) = 100$
- $\text{slot_wasted} = \text{MAX}(0, 100 + 100 - 150) = 50$

In this case, all the 100 allocated slots are used, autoscaling kicks in to provide another 100 slots from the shared pool and 50 slots are wasted. This is better than the 200 slots allocated and 150 slots wasted if idle slots sharing was not used.

Both cases deliver savings. By sharing slots, baseline slots are used more often, waste is reduced, and autoscaling is minimized.

Option 2: Shared admin project

Savings on commitments

Let’s assume the owners of billing projects P1 and P2 both decide to take advantage of discounts with commitments. They create commitments C1 and C2 independently, yet put them in the same admin project.

Admin project AP

- Commitment C1: 100 slots
- Commitment C2: 100 slots
- Reservation R1: baseline 0 slots
 - Assignment A1: assignee is P1
- Reservation R2: baseline 0 slots
 - Assignment A2: assignee is P2

Billing project P1

- A job is consuming 80 slots

Billing project P2

- A job is consuming 220 slots

By applying our previous definitions we get:

- $\text{slots_allocated} = \text{MAX}(100+100, 0) = 200$
- $\text{slots_requested} = 80+220 = 300$
- $\text{slots_autoscaled} = \text{CEIL}(\text{MAX}(0, 300 - 200), 100) = 100$
- $\text{slot_wasted} = \text{MAX}(0, 200+100-300) = 0$

In this scenario, only 100 slots (slots_ autoscaled) are taken at full price from the shared pool. If the teams had created separate admin projects (as per example 2), then team P1 would have wasted 20 slots (100 slots committed but only 80 used) and team P2 would have paid 200 slots at full price and wasted 80 (100 slots committed, 200 autoscaled, 220 used). By sharing, the commitments slots are used more often— increasing discounts and reducing waste.

Example 5: Two commitments on same admin project

Option 2: Shared admin project

Using multiple assignments to prioritize shared slots

Now, let's dive a little deeper into multiple assignments. Compared to multiple reservations with just one assignment, multiple assignments per reservation are useful to prioritize shared slots consumption. This is especially interesting in situations where autoscaling is disabled (baseline = maximum) and only shared slots are in use—which means their prioritization is more important.

In this example, the job load changes over time.

Admin project AP

- Commitment C: 300 slots
- Reservation R1: baseline = max = 200 slots
 - Assignment A1: assignee is P1
 - Assignment A2: assignee is P2
- Reservation R2: baseline = max = 100 slots
 - Assignment A3: assignee is P3
 - Assignment A4: assignee is P4

At time t0, all projects have jobs in queue and are at steady state, consuming all the slots fairly (respectively 100, 100, 50, 50).

At time t1, P1 has no more jobs in queue. Since P1 and P2 are assigned from R1, P2 alone will use the capacity of R1. At steady state, the distribution becomes 0, 200, 50, 50.

Now, if there were four reservations instead of two (so, R1(100) → P1, R2(100) → P2, R3(50) → P3, R4(50) → P4), then at t1, 100 slots become shareable and equally distributed among all the other reservations—leading to a distribution of 0, 133, 83, 83. In other words, P2 has no priority claim over P1's slots. In short, placing two or more billing projects in the same reservation prioritizes sharing idle slots between them.

Example 6: Multiple assignments per reservation



Apply a dynamic approach to workload management

Since everything on Google Cloud is under an API, SDK, or CLI, you can dynamically change commitments, reservations, assignments, and billing projects to further optimize your spending.

For example, you could schedule configuration changes to reservations and/or assignments with [Cloud Workflows](#) over time.

Or, you could change the configuration of two reservations so that on Monday mornings most of the slots coming from a commitment go to the ELT processes, and for the rest of the week they go to interactive users. You can put this in your orchestrator, [Cloud Composer](#), to change the configuration before running a sequence of jobs and then restore the original configuration.

Above, we discussed how on-demand can be cheaper for a query, yet it can change with runs or drift over time. Dynamically switching pricing models can help here.

Imagine building a lookup table that the orchestrator uses to decide whether to launch a query under an on-demand or an autoscaling billing project. In pseudo code:

```
model = get_cheaper_model($query_id)

billing_project = get_billing_project($model)

query = get_query($query_id, $parameters)

bq --project_id=$billing_project ... $query
```

The recurring query is [labeled](#) with a query_id so it can be recognized when parameters change. The query must show a strong cheapness in one model, and the behavior must be stable over time. This study must be repeated for all queries of interest. [Job information](#) tables can be used to detect these behaviors at scale and produce a periodically refreshed look up table where each query has its query_id and cheapest model stored. This is an advanced optimization that requires effort and may make sense only for users with large compute spending.

How does BigQuery compute compare to other solutions?

BigQuery minimizes waste

Other solutions may have similar concepts, but lack BigQuery's compute sharing feature (slots sharing). This means that, while they may permit the creation of multiple warehouses (reservations in BigQuery), these warehouses can't share idle computing power. Indeed, they scale discreetly, adding more identical clusters and the last cluster is, by design, always partially in use. Moreover, each warehouse defines the common size of all its clusters and, since the size is constrained to be a power of two, there can be a lot of wasted computing power in large warehouses.

There's less waste in BigQuery thanks to slots sharing and the fact that commitments and baselines can change linearly in steps of 100 slots (instead of exponentially as with 2^N).

BigQuery slots are stateless

Other solutions, when scaling, suffer a 'cold start' when adding new computing resources. The empty caches (for example, local disks) in

these resources need to be filled before they can run efficiently. With BigQuery, slots are stateless and scaling is not a problem—new slots are immediately as efficient as the other slots already at work.

The statefulness of other solutions leads to cold starts when queries are moved across warehouses. If the query is accessing storage that hasn't been accessed previously, then the warehouse has to cache this data before being fully operative. On BigQuery you can change the billing project of a workload any time without cold start issues.

BigQuery scales fast

Some solutions scale by adding more clusters to a warehouse. This increases the number of queries running in parallel (concurrency) but it can't increase the speed of single query (latency)—because a query runs inside a cluster. In this instance, once the whole cluster's computing power has been allocated to a single query, the query can't go any faster—even if there are other clusters

available.

On BigQuery, slots are like a flock that collaborate in completing the query, so if more slots become available then the query can proceed faster. In other words, BigQuery scaling helps with both concurrency and latency, even for queries already in execution.

BigQuery compute cost optimization

Follow SQL best practices

The same query can be written in different ways, some more efficient than others. To optimize query performance, it's recommended you follow the best practices in this [documentation](#). In summary, though, you should:

- Apply partitioning and clustering [recommendations](#) to optimize BigQuery usage
- Select only columns that you need and curate [filtering, ordering](#) and [sharding](#)
- Denormalize if and where needed
- Choose the right function and pay attention to Javascript User Defined Functions
- Choose the right data types
- Optimize join and common table expressions
- Look for anti patterns



Use BI Engine to reduce compute costs

Unlike other solutions, BI Engine provides an in-memory caching layer that enables sub-second analytical queries. And, while many people think BI Engine is something you pay extra for—to get a performance boost—it can actually reduce your bill.

When BI Engine is enabled, data is cached between storage and compute layers. This means that under on-demand pricing, a cache hit grants you 0 bytes billed. Similarly, under reservations, a query accelerated by BI Engine is not only faster, but it also uses less slot_sec and is therefore cheaper.

BI Engine is enabled by adding BI Engine reservations directly to the billing project. This is different from autoscaling reservations, which can be bound to a billing project, folders, and organizations via associations.

Use materialized views to reduce costs

In BigQuery, [materialized views](#) (MVs) are pre-computed views that cache the results of a query for increased performance and efficiency. They can help you:

- Simplify development by avoiding update scripts
- Improve performance by pre-computing view's results
- Reduce compute costs by avoiding computations of view's results

When applicable, MVs are a better solution than a table and relative ELT pipeline. There's no need to write incremental load scripts or think about backfills and trigger updates. Plus, the data queried through MVs is always strongly consistent with the base table (unless you tweak max_staleness parameter). Just remember, they're not always the right solution and may actually increase costs without improving performance.

Keep these things in mind before using MVs:

- MVs are [limited in number](#). Currently, you can't have more than 20 MVs per base table. When you hit this limit, you should only keep the MVs that together give you the best return on investment. For example, it may make sense to replace two similar ad-hoc MVs with a single MV with a broader scope that [smart query tuning](#) will use in both cases.
- Try to write the MV so it allows [incremental updates](#). Then, each read can reuse part of the MV storage and performs only a portion of the computations done by an equivalent view. This makes scenarios like the ones below less problematic and can justify the MV.
- Be careful about building MVs on clustered base tables. Background reclustering of the base table will trigger further refreshes of the MV.

Manually refresh MVs

Automatic refresh of MVs works asynchronously. This doesn't fit 'read-after-write' scenarios where, for example, a burst of queries hit the MV just after one of the base tables have been updated. These queries will find the MV storage out of date and will use the base tables, making the MV useless.

To counter this issue, you can manage MV refreshes manually, with updates in tune with base table updates. So, why not just use a table? First, calling `BQ.REFRESH_MATERIALIZED_VIEW` is still much simpler than an incremental load pipeline. Second, the smart query tuning can leverage MV but not tables. Here's when manual refreshes can be applied.

Single update point

If 1) there is a single place where base tables are modified, 2) you can add code into this script, and 3) the MV will be queried more than once before the next execution, then it makes sense to [manually trigger the MV refresh](#) in the script just after updating the base tables. This ensures the query will find the MV up-to-

date (which is not guaranteed with automatic refresh). Why not use a view? Because any query after the first (point 3) is more efficient with MV.

Single query point

If 1) there is a single place where the MV is queried, 2) you can add code into this script, 3) the base tables are updated more than once before the next query, and 4) the MV is not incremental, then you can permanently [disable automatic refresh](#). Instead, you can manually refresh the MV in the script before querying it, which avoids wasted MV refreshes. If the MV supports incremental updates you may still want to use automatic refresh.

Long update

If you spend more than five minutes (the automatic refresh wait time) updating the base tables and it does not make sense to have the MVs updated until you finish, then you may want to [temporarily disable automatic refresh](#). For example, if you append a first set of rows

to the base table and then another set with a query lasting 15 minutes, it's possible that automatic refresh will run before the second append.

Conditional refresh

If updating an MV every time a certain base table is updated isn't important, or you can skip the refresh on certain detectable conditions while updating the base tables, then you can permanently disable automatic refresh and manually refresh the MV only when needed.

Control refresh priority and billing project

If you want to control the refresh job priority compared to other queries, or you want to use a different project supplying the slots to the refresh job to the one containing the MV, then you must manually refresh the MV.

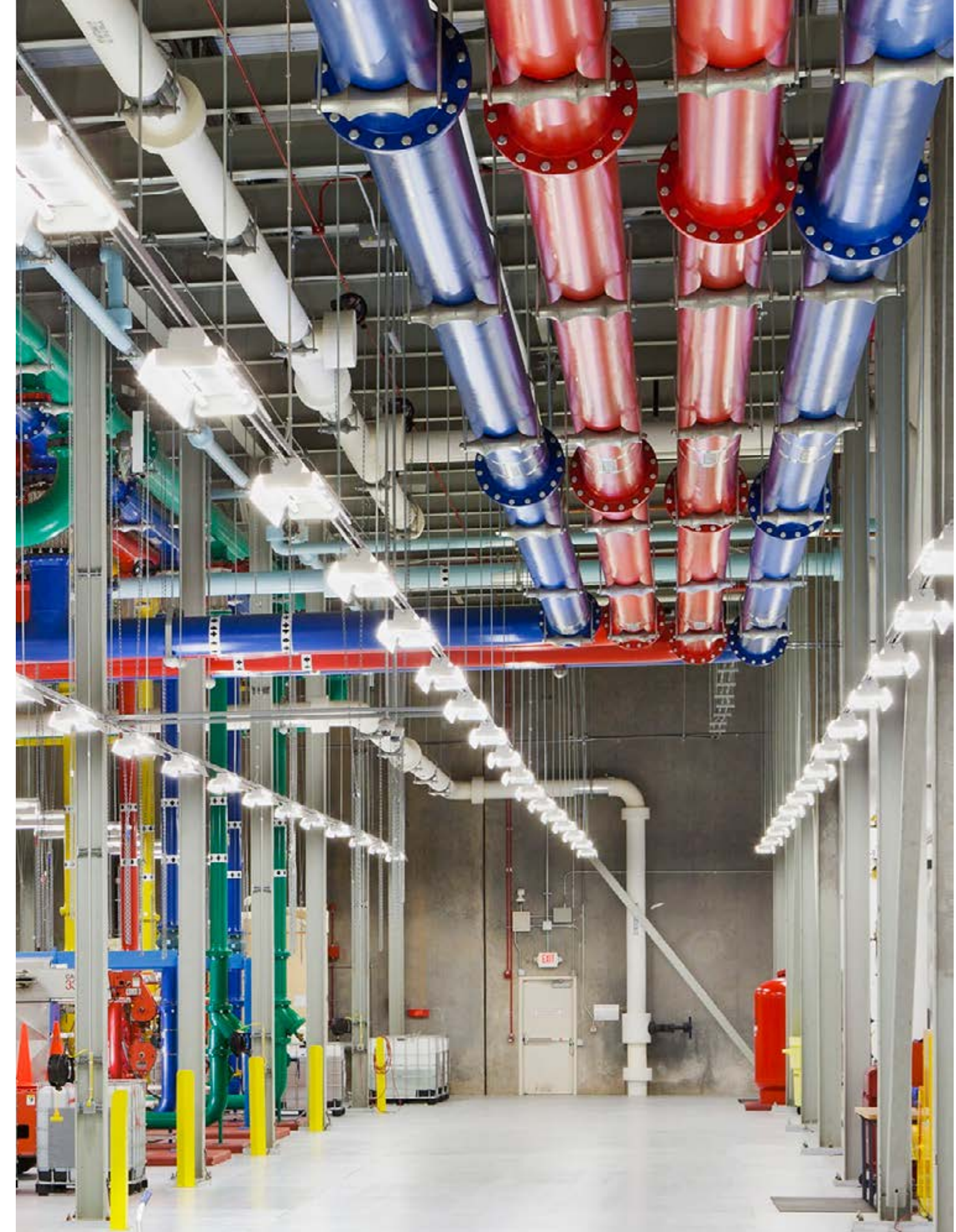
Otherwise, automatic refresh is treated similarly to a query with [batch](#) priority, subject to the slots available to the project containing the MV. The automatic refresh aims to refresh the MV within five minutes of a change in the base table, but latency could be higher if no slots are available.

Tune automatic refresh

When manual refresh is not an option, problems can occur when a not-incremental MV is often queried and the base table is often updated. In this case, the auto refresh uses power to update the MV and, since the MV is often out-of-date, the queries go straight to the base table.

In this case you can:

- Lower refresh costs by capping refresh frequency. This can be done by increasing the [refresh_interval_minutes](#) table option. It defaults to 30 minutes, but can be between one minute and seven days. Note this can increase the chances of querying the MV when out-of-date.
- Lower the query costs by accepting that you may get slightly stale results from the MV. This can be done by increasing the [max_staleness](#) table option (which defaults to zero, but can be any interval). Please note that smart-tuning ignores max_staleness because, by working behind the scenes, returning stale results to an unaware query could lead to data consistency problems.



05

Different use

cases of building

data warehouses

Use cases: Building data warehouses

In previous chapters, we studied the best ways to ingest, store and analyze data in BigQuery. Let's simulate some real-world scenarios of how organizations are using Google Cloud and BigQuery to unlock new insights from data.

In this chapter, we'll learn how organizations identify their data storage and analysis needs, and how they can implement Google Cloud solutions to meet those needs. We'll also explore some reference architectures for these use cases. And finally, we'll estimate the cost of storing and analyzing data in BigQuery.

In this chapter:



BigQuery architecture



BigQuery pricing model

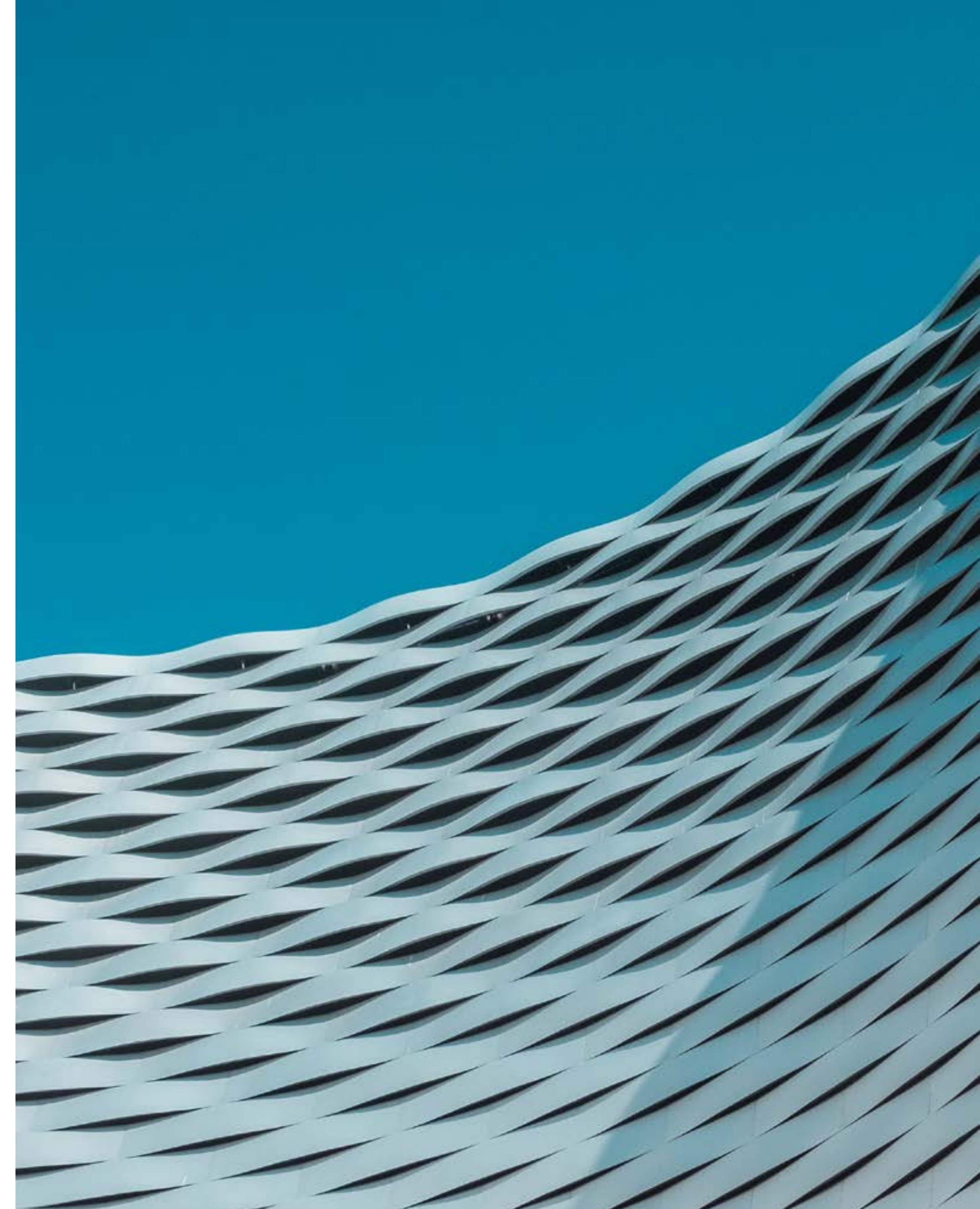
Use case 1:

Marketing data warehouse

Background

MB Healthcare is a fictional leading SaaS provider of electronic health record software to the medical industry. Customers include multinational medical offices, hospitals, and insurance providers. Experiencing exponential growth, the company needed a better way to scale and support continuous deployment, while also shoring up security and disaster recovery.

The company chose Google Cloud to replace current colocation facilities, with some legacy data remaining in AWS cloud storage.



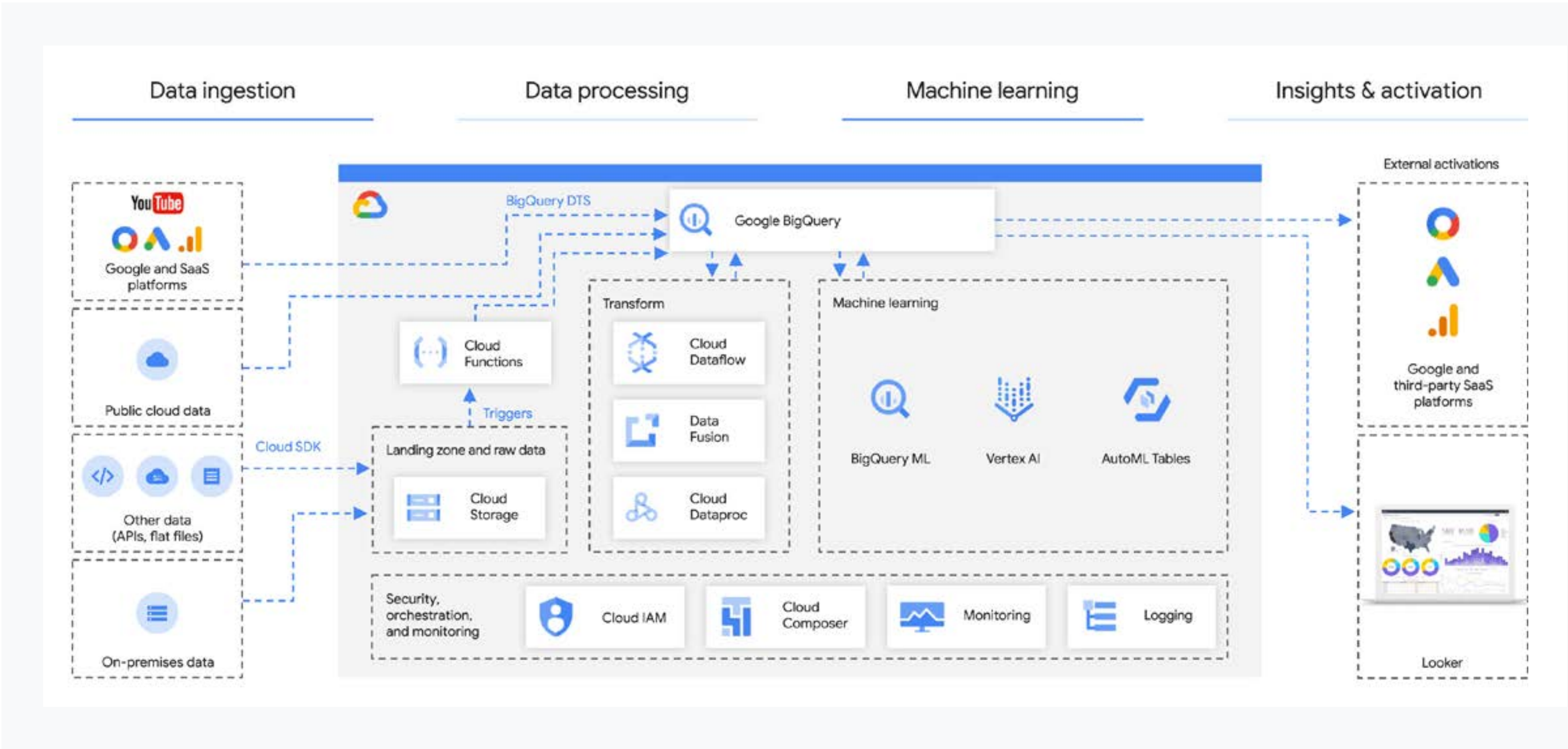
Technical requirements

MB Healthcare wanted to run predictive analytics on the lifetime value of specific users, and tailor its marketing campaigns accordingly.

To do this, it needed:

- ANSI compliant SQL engine to ingest and query 1 PiB of historical data and 20 TiB of daily updates data, with a 5x data compression ratio
- Automated data pipelines with GA360, Salesforce, Campaign Manager 360, and Google Ads
- SQL-based machine learning
- Machine learning training and batch predictions capabilities, with unpredictable workloads and specified SLAs

Here’s a typical marketing analytics reference architecture on Google Cloud that uses multiple data analytics and ML products.



Marketing Analytics reference architecture

Warehouse operations

Data ingestion

As a first step in building a marketing data warehouse, MB Healthcare needed to consolidate data in a central location. The following data sources were ingested into BigQuery:

- **Google and SaaS platforms**

Data sources like Google Analytics, Google Ads, and Google Marketing Platform can be ingested directly into the marketing data warehouse. To ingest data from sources like Salesforce, [SaaS connectors](#) are available in Google Cloud and through partners.

- GA360 has a direct native automatic export to BigQuery → [FREE](#)
- Google Ads could be automated via [BigQuery Data Transfer Service](#) → [FREE](#)
- Salesforce could be automated via [BigQuery Data Transfer Service](#) → [Partner pricing](#)

- **Public clouds**

[BigQuery Data Transfer Service](#) ingests data from other public clouds. For example, to move data from Amazon S3 to BigQuery, you can automatically schedule and manage recurring [load jobs](#). You can also use [BigQuery Omni](#), a flexible, multi-cloud analytics solution that lets you analyze data across Google Cloud and Amazon Web Services.

- Exporting logs from AWS S3 → [FREE](#) (MB Healthcare only paying for AWS Network egress)
- Analyzing in AWS S3 directly via [BigQuery Omni](#) → 1 reservation of Enterprise Edition with 100 slots, this would auto-scale based on the needs
- **APIs, flat files, and on-premises first-party data**

Data from sources like customer relationship management (CRM) and point of sale (POS)

systems can be ingested offline by using the [bq command-line tool](#) and [Cloud Storage](#).
→ [FREE](#)

Data storage

MB Healthcare required an estimated 1.5 PiB for the initial backfill and 30 TiB daily updates.

To optimize costs based on pattern usage, 75% of storage is long-term and 25% is active. Assuming that baseline storage with monthly cleaning tasks is ((1 PiB x 1024) + 30 TiB), then:

- Active storage (any table or table partition that has been modified in the last 90 days) → 215,859 GiB x 25% x [price per GiB](#)
- Long-term storage (any table or table partition that has not been modified for 90 consecutive days) → 215,859 GiB x 75% x [price per GiB](#)

Warehouse operations

Data processing

With data ingested and stored, MB Healthcare starts processing the data before running queries against it. Data processing includes cleaning and reformatting to provide consistency in big datasets. When considering an ELT (extract, load, and transform) approach or when performing subsequent transformations after the data has been loaded into the data warehouse, you can use BigQuery for the SQL transformations. It is managed and also provides user-defined functions.

For the initial phase, MB Healthcare pays only for the slots used using the Standard Edition pricing. Business analysts will most likely run queries during business hours (9AM - 5PM). With this in mind:

- Standard Edition pricing is \$0.04/slot hour in the US
- Number of hours per month = (8 hours per day) x (5 days per week) x (4 weeks per month) = 160 hours per month

- Average slots expected based is ~300 slots with up and down in the month
- Price: 300 slots x \$0.04/slot hour x 160 hours = \$1,920 per month

Machine learning

As described in the technical requirements, MB Healthcare is looking to do “SQL based machine learning” as they want to get to results faster, BigQuery ML lets you use SQL constructs to create, evaluate, and predict models. You can train and deploy many supported models, and execute machine learning workflows without moving data out of BigQuery.

As you can mix and match different BigQuery editions within the same GCP account, For this second phase, MB Healthcare decides to use Enterprise Edition for their machine learning workloads. MB Healthcare’s machine learning engineering team will most likely experiment during the day, and train their models overnight.

Enterprise Edition pricing is \$0.06/slot hour in the US

- **Experimentation:**
 - Number of hours per month = (8 hours per day) x (5 days per week) x (4 weeks per month) = 160 hours per month
 - No baseline needed (scaling from zero)
 - Average utilization is ~500 slots with up and down in the month
 - Price: 500 slots x \$0.06/slot hour x 160 hours = \$4,800 per month
- **Training:** We are assuming that MB Healthcare is retraining their models based on new data every night
 - Number of hours per month = (12 hours per night day) x (7 days per week) x (4 weeks per month) = 336 hours per month
 - Baseline expected is ~1000 slots
 - Average utilization is ~1700 slots with up and down in the month
 - Price: 1700 slots x \$0.06/slot hour x 336 hours = \$34,272 per month

After a few weeks, MB Healthcare understood its usage patterns and decided to create a commitment on their baseline for training and experimentation, with 1000 slots for three years. This led to significant savings, with the cost per slot hour falling by 40%, from \$0.06/slot hour to \$0.036/slot hour for the committed slots.

Use case 2:

Advertising data pipelines

Background

MB Adtech is an imaginary mobile-first advertising company that creates innovative campaigns for the world's best-known brands. Its comprehensive platform was built to work with both advertisers and publishers, providing SSP and DSP capabilities.

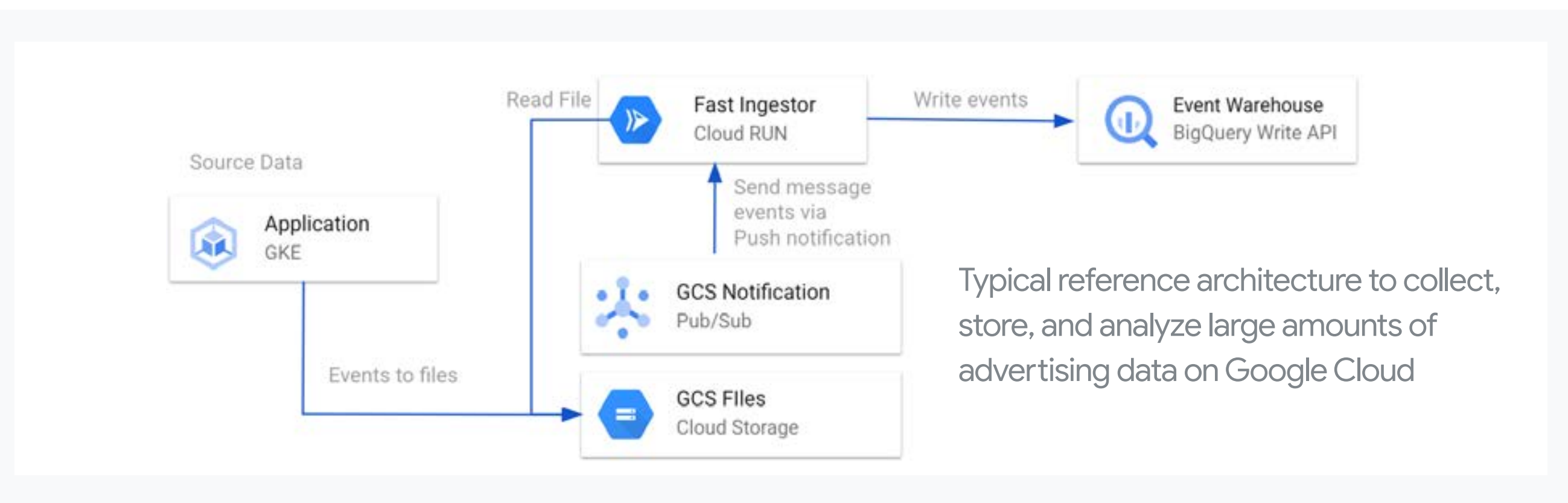
Requirements

As a data-driven business, MB Adtech had some specific requirements. It needed the ability to:

- Integrate with various advertising platforms (e.g. Google Ads, Facebook Ads, Amazon Advertising, etc.) to gather data from different sources

- Handle large volumes of data and perform real-time processing to provide up-to-date insights
- Easily scale, for easy addition or removal of data sources or changes to the pipeline's configuration
- Allow for data cleansing, transformation, and enrichment to improve data quality and usability
- Provide a secure and compliant environment for data storage and processing, including adherence to data privacy regulations such as GDPR and CCPA
- Ensure uninterrupted data processing and availability
- 10 TiB per day generated

This reference architecture example below provides a high level approach to collect, store, and analyze large amounts of advertising data on Google Cloud.



Data operations

Data ingestion

The cost for the fast micro-batching ingestion process, using the high-performance [BigQuery Storage Write API](#):

- 300,000 GiB x [price per GiB](#)

Data storage

For storage, MB Adtech estimated 1.5 PiB for the initial backfill and 30 TiB daily updates—with 80% in long-term storage and 20% in active storage.

- Total: $(1.5 \text{ PiB} \times 1024) + 30 \text{ TiB} \times 1024 = 1,603,584 \text{ GiB}$ (this would grow month over month)
- Active storage $\rightarrow 1,603,584 \times 20\% \times$ [price per GiB](#)
- Long-term storage $\rightarrow 1,603,584 \times 80\% \times$ [price per GiB](#)

Data processing

MB Adtech is mixing and matching different editions of BigQuery based on the use cases and budget allocated by teams:

- Enterprise Plus Edition is a great fit for their production pipeline's client-facing dashboards.
- Enterprise Edition is a great fit for their production pipeline's internal stakeholders.
- Standard Edition is the most cost-effective solution for their data and business analyst experimentation. Scaling from zero (without baseline) is not an issue for these users. Due to the different usage patterns, MB Adtech has created five reservations with Standard Edition for different groups of users.

This flexibility to mix and match different editions helps MB Adtech keep their costs under control.



Use case 3:

Mobile gaming analytics platform

Background

MB Games makes online, session-based, multiplayer games for mobile platforms. It was facing issues with scaling application servers, MySQL databases, and analytics tools—their old model wrote game statistics to files and sent them through an ETL tool, which loaded them into a centralized MySQL DB for reporting. They were in the process of building a new game, and they knew it would be a hit. So they wanted to reevaluate their data warehousing.

Mobile game applications generate a large amount of player-telemetry and game-event data, which provides insights into player

behavior and their engagement with the game. Yet the nature of mobile games—with massive numbers of devices, irregular and slow Internet connections, and battery issues—means that player telemetry and game event analytics face unique challenges.

On top of all this, MB Games had specific business requirements:

- Data must reside in the US only
- Some BI reports required sub-second query response times, yet they also wanted compute cost savings
- Replacing MySQL and moving to an autoscaling low latency, managed environment

- They wanted only fully managed servers—they didn't want to manage physical servers anymore
- KPIs to evaluate the speed and stability of the game and other metrics, providing deep insight into usage patterns to target users

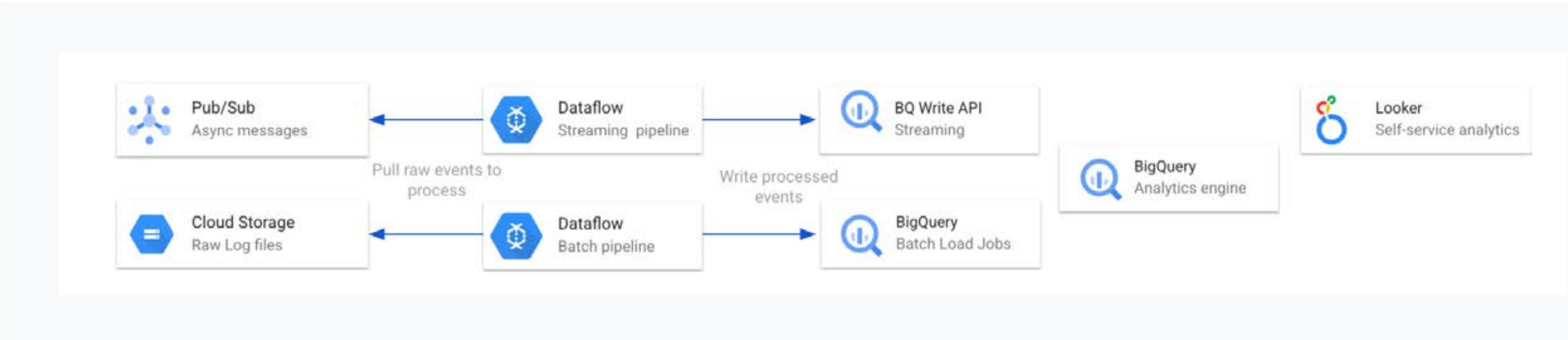
Technical requirements

- Scale up and down based on activity
- Process incoming data on the fly from game servers
- Process data that arrives late because of mobile networks
- Process files regularly uploaded by users mobile devices
- SQL Compatible engine to ingest and query ~1.5 PiB of historical data
- Daily updates data: 30 TiB

- In-memory analysis service must be fast and cover up to 250 GiB

This reference architecture provides a high level approach to collect, store, and analyze large amounts of player-telemetry data on Google Cloud.

- Real-time processing of individual events using a streaming processing pattern
- Bulk processing of aggregated events using a batch processing pattern



High level architecture to collect, store, and analyze large amounts of player telemetry data on Google Cloud

Warehouse operations

Data ingestion

Costs are estimated for both bulk and real-time processing:

- **Bulk processing**

BigQuery offers a free shared pool of slots for batch loading the data into BigQuery Storage (aka BQ Load). This is complementary and will not consume any allocated slot time.

- 1.5 PiB backfill historical data → using the free pool of slots

- **Real-time processing**

[BigQuery Storage Write API](#) offers a complimentary 2 TB per month (free tier), then charges [price per GiB](#).

- 30 TiB daily → 30 TiB x 1,024 x [price per GiB](#)
- Alternatively, MB Games could use our Streaming Inserts legacy API, but BQ Storage Write API is highly recommended for the high throughput and lower latency.

Data storage

MB Games needed an estimated 1.5 PiB for the initial backfill and 30 TiB daily updates. Based on pattern usage for this scenario, 80% will be in long-term storage and 20% in active storage.

- Total: $(1.5 \text{ PiB} \times 1024) + 30 \text{ TiB} \times 1024 = 1,603,584 \text{ GiB}$ (this would grow month over month)
- Active storage → $1,603,584 \times 20\% \times$ [price per GiB](#)
- Long-term storage → $1,603,584 \times 80\% \times$ [price per GiB](#)

Data analysis

MB Games wanted predictable pricing for their SQL transformation and aggregation, model training and prediction, and BI reporting. They also required ongoing ad-hoc batch query analysis to be completed overnight within specific SLAs.

As such, they sized the environment to:

- BigQuery Enterprise Edition with 2000 slots as the baseline and 3000 for the maximum capacity to control the cost during the day (MB Games could wait for the query to run longer during the day)
- Automated BigQuery Reservation API, using Cloud Composer (aka Airflow), an automatic increase of their maximum capacity in their existing reservation (from 3000 to 4500 slots) in the evening and decrease in the morning (from 4500 slots to 3000 slots)

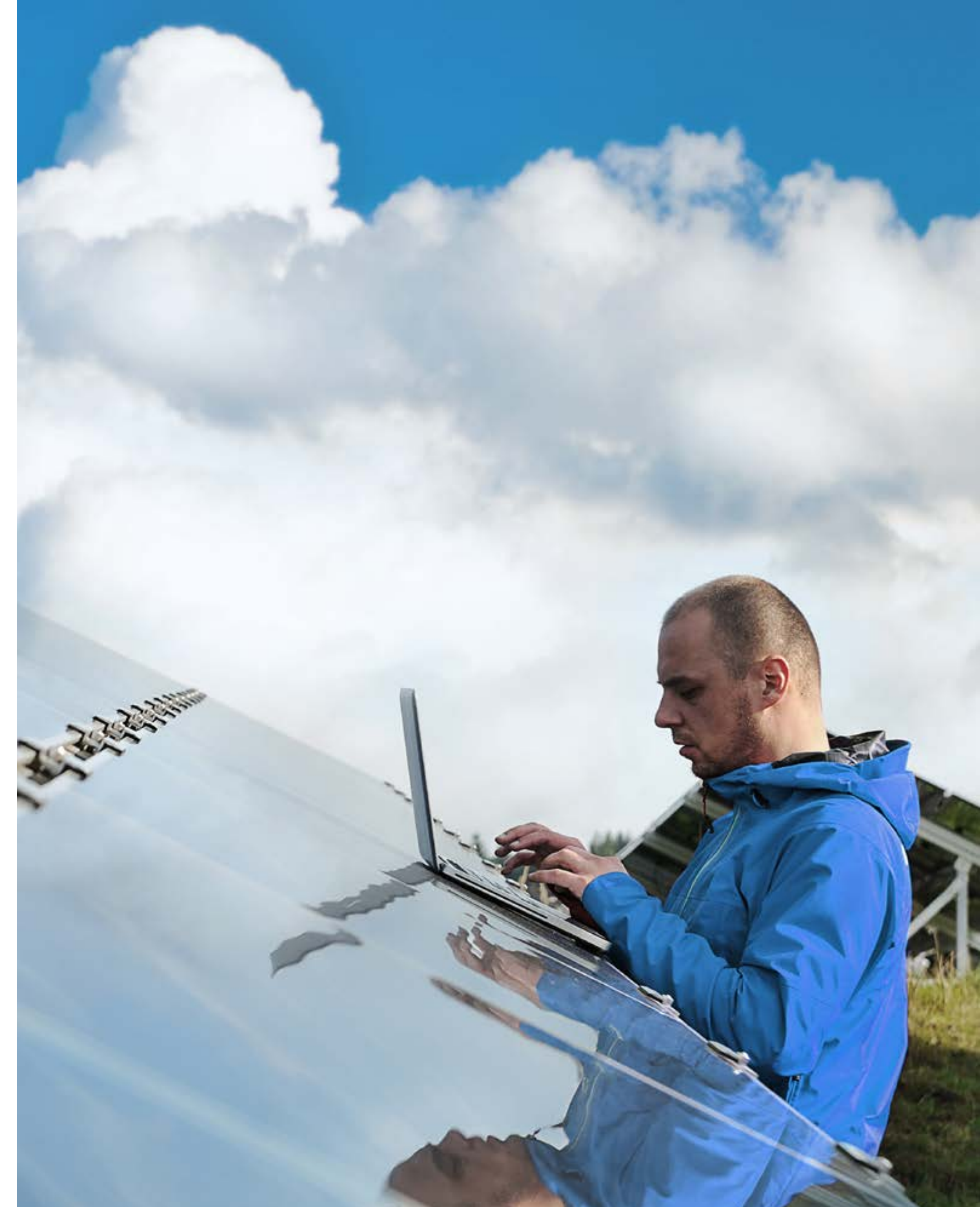
For data visualization, MB Games is using Looker:

- BigQuery does not charge any network egress fee
- BI Engine SQL Interface helps improve their query performance and reduce their slot usage, MB Games created a reservation of 150 GiB during the work hours:
 - 50 GiB x 40 hours x 30 days x [price per GiB](#)

Warehouse operations

Wary of hidden costs and fees with data warehousing, MB Games' CTO was reassured upon learning that the following operations are completely free with BigQuery:

- Batch load data
- Copy data
- Export data
- Delete operations
- Metadata operations
- Read pseudo columns
- Read meta tables
- Creating, replacing, or invoking persistent user-defined functions (UDFs)



06

Conclusion

Conclusion

Pricing is an important factor when choosing a data warehouse for analytics. Yet, as we've explored above, there's more to pricing than meets the eye. The ability to optimize costs based on different workloads and needs can make a big difference to the bottom line.

With BigQuery, you're charged for storage and analysis (compute). Recognizing that the compute component is the most costly, BigQuery provides several options that you can easily mix and match based on the use case. For example:

- The on-demand model can be used for any kind of workload, with pricing based on the number of bytes processed by your queries.
- Three BigQuery Editions provide performance-based pricing by scaling up and down based on pending workload

while also adhering to the limits that you can control—and there is no under- or over-utilization of resources within those limits (which is the case with some other cloud data warehouses). While the majority of the production workloads would fall under Enterprise Edition, Standard Edition is suitable for smaller workloads including ad-hoc analysis and proof-of-concepts. Enterprise Plus Edition is perfect for highly compliant and mission critical workloads.

Compute cost optimizations pay high dividends. Things like partitioning and clustering help reduce the compute resources you need for your queries, which in turn delivers performance gains and cost savings. Materialized views offer another way to save, by persisting results of frequently used queries. And then there are further discounts with annual

commitments on baseline capacity for slots autoscaling.

As well as flexibility around compute costs, you can also optimize storage to help reduce costs and boost performance. For example, you could partition the data to reduce the active storage volume, set the appropriate [data storage billing model](#) for your datasets, configure table expirations for certain datasets or tables, and choose an appropriate time travel option based on your needs.