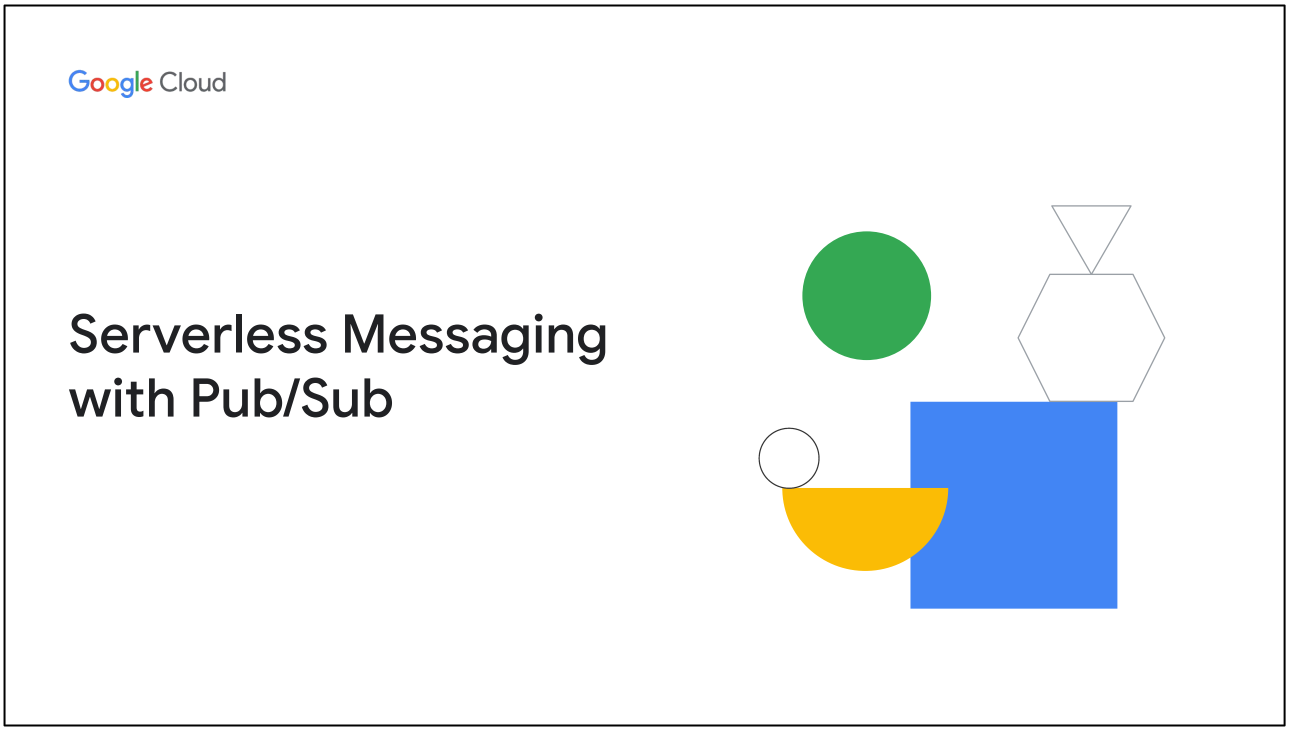
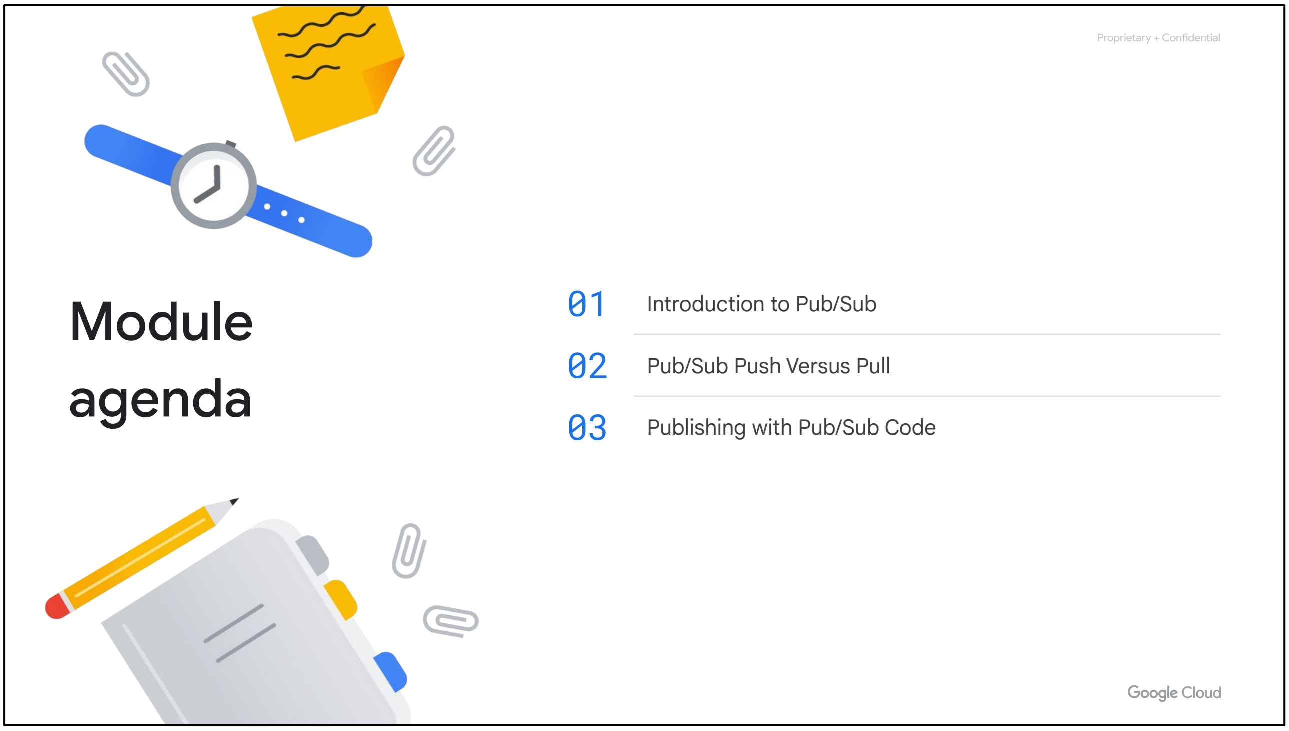


Google Cloud

Serverless Messaging with Pub/Sub

Now that we have a good understanding of the process of streaming data, let's dive into Pub/Sub to see how it works.



Module agenda

01 Introduction to Pub/Sub

02 Pub/Sub Push Versus Pull

03 Publishing with Pub/Sub Code

Google Cloud

We'll start by understanding how Pub/Sub works and is used in decoupling systems.

Then, we'll discuss the distribution of messages and different patterns through push and pull delivery models.

Finally, we'll focus on actual implementation - looking at how things are set up in Pub/Sub. This will include a hands-on lab, where you will publish streaming data into Pub/Sub.

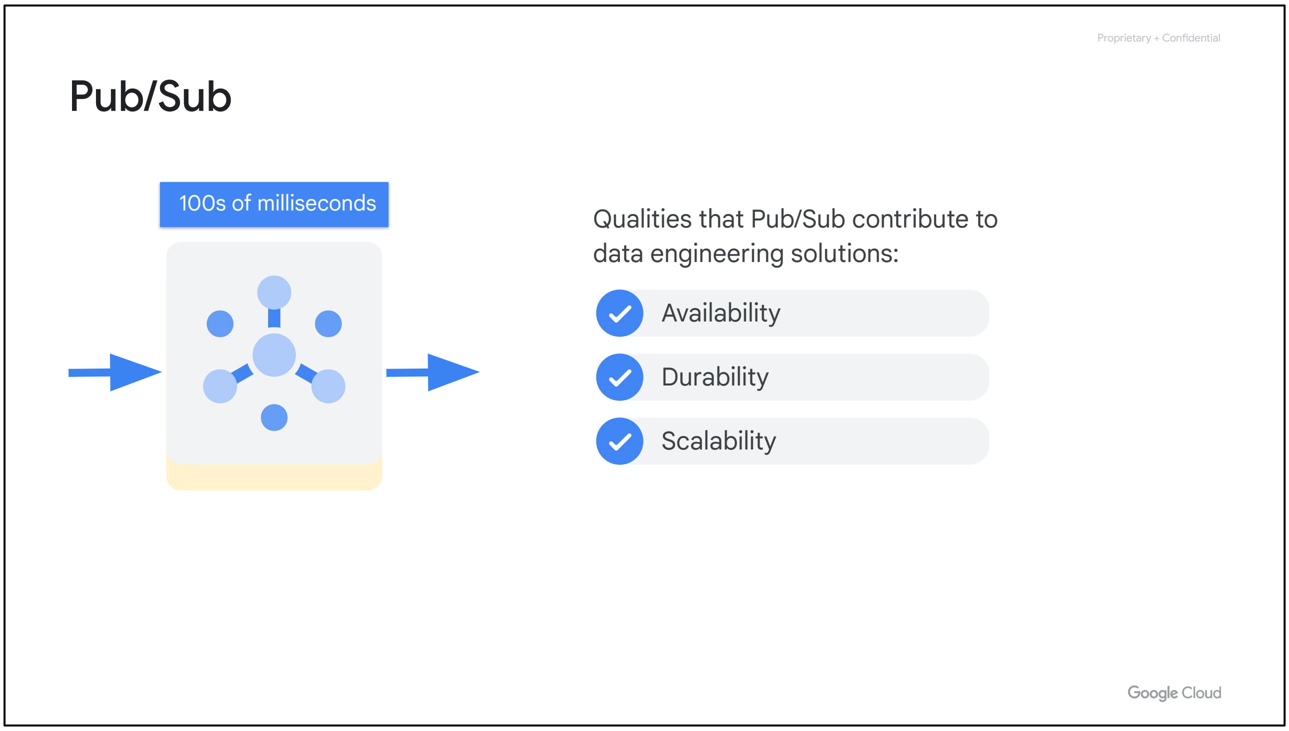
Proprietary + Confidential

01

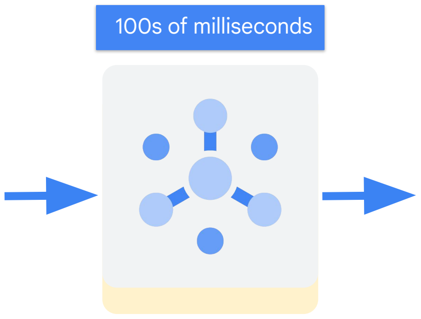
Introduction to Pub/Sub

Google Cloud

As we begin this module, I would ask you to keep your mind open to new ways of doing things. Pub/Sub does streaming differently than probably anything you have used in the past. It may be a different model than what you have seen before.



Pub/Sub



Qualities that Pub/Sub contribute to data engineering solutions:

- ✓ Availability
- ✓ Durability
- ✓ Scalability

Pub/Sub provides a fully managed data distribution and delivery system. It can be used for many purposes. It is most commonly used to loosely-couple parts of a system. You can use Pub/Sub to connect applications within Google Cloud, and with applications on premise or in other clouds to create hybrid Data Engineering solutions. With Pub/Sub the applications do not need to be online and available all the time. And the parts do not need to know how to communicate to each other, but only to Pub/Sub, which can simplify system design.

First, Pub/Sub is not software; it is a service. So, like all of the other serverless services we have looked at, you don't install anything to use Pub/Sub.

Pub/Sub client libraries are available in C#, GO, Java, Node.js, Python, and Ruby. These wrap REST API calls which can be made in any language.

It is also highly available.

Pub/Sub offers durability of messages. By default it will save your messages for seven days. In the event your systems are down and not able to process them. Finally, Pub/Sub is highly scalable as well. Google processes about 100 million messages per second across their entire infrastructure. This was actually one of the use cases for Pub/Sub early on at Google, to be able to distribute the search engine and index around the world because we keep local copies of search around the world, as you might imagine, in order to be able to serve up results with minimal latency.

If you think about how you would architect that, we are crawling the entire world wide web, so we need to send the entire world wide web around the world. Either that or we would need to have multiple crawlers all over the world, but then you have data consistency problems; they would all be getting different indexes.

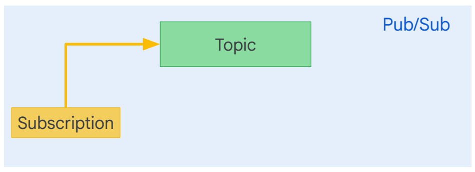
Therefore, what we do is use Pub/Sub to distribute. As the crawler goes out, it grabs every page from the world wide web, and we send every single page as a message on Pub/Sub and it gets picked up by all local copies of the search index so it can be indexed.

Currently, Google indexes the web anywhere from every two weeks, which is the slowest, to more than once an hour, for example on really popular news sites. So, on average, Google is indexing the web three times a day. Thus, what we are doing is sending the entire world wide web over Pub/Sub three times a day. This should explain how Pub/Sub is able to scale.

Pub/Sub is a HIPAA-compliant service, offering fine-grained access controls and end-to-end encryption. Messages are encrypted in transit and at rest. Messages are stored in multiple locations for durability and availability.

You control the qualities of your Pub/Sub solution by the number of publishers, number of subscribers, and the size and number of messages. These factors provide tradeoffs between scale, low latency, and high throughput.

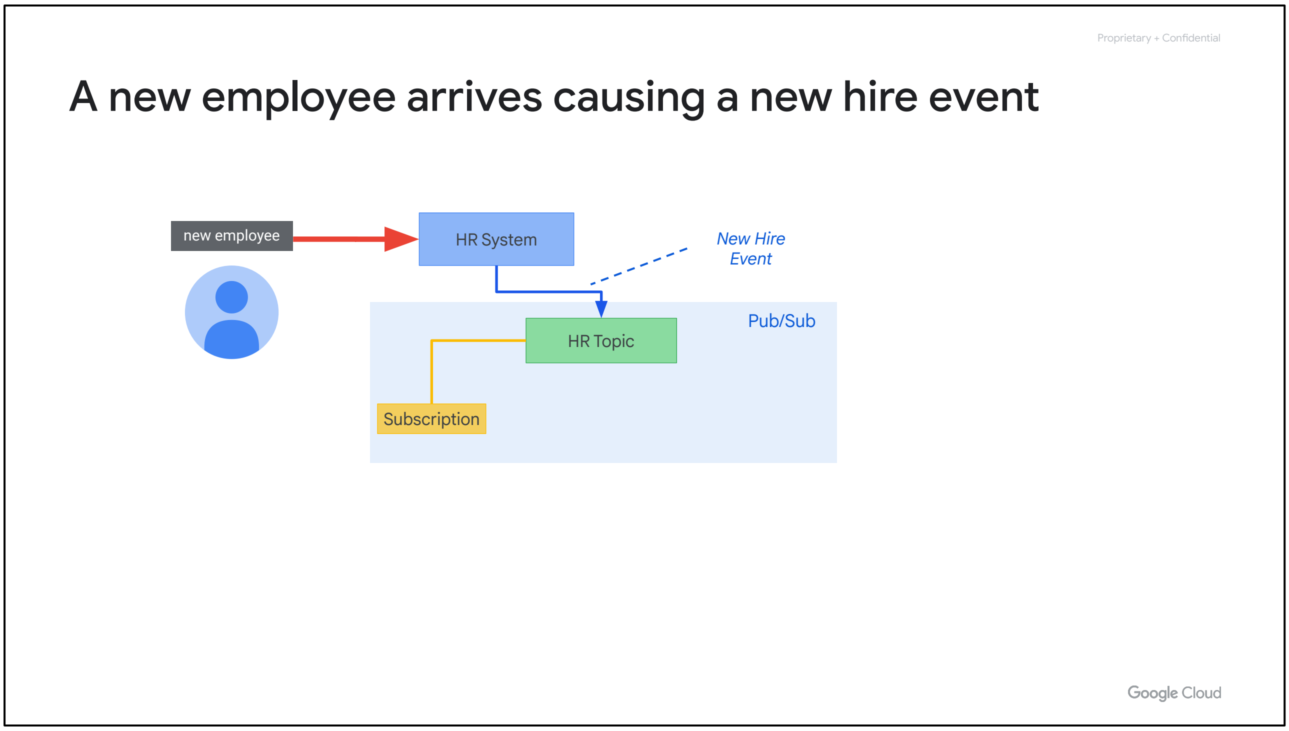
Example of a Pub/Sub application



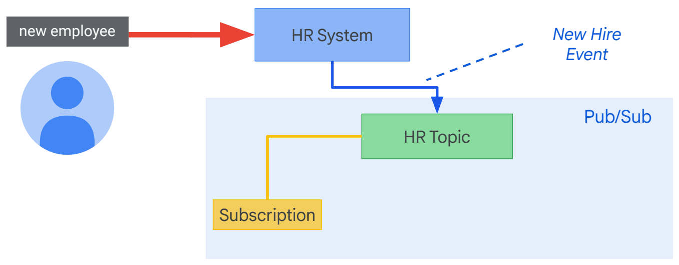
How does Pub/Sub work? The model is very simple. The story of Pub/Sub is the story of two data structures, the Topic and the Subscription. Both the Topic and the Subscription are abstractions which exist in the Pub/Sub framework independently of any workers, subscribers, etc. The Pub/Sub client that creates the Topic is called the Publisher. And the Pub/Sub client that creates the Subscription is called the Subscriber.

In this example, the Subscription is subscribed to the Topic.

To receive messages published to a topic, you must create a subscription to that topic. Only messages published to the topic after the subscription is created are available to subscriber applications. The subscription connects the topic to a subscriber application that receives and processes messages published to the topic. A topic can have multiple subscriptions, but a given subscription belongs to a single topic.

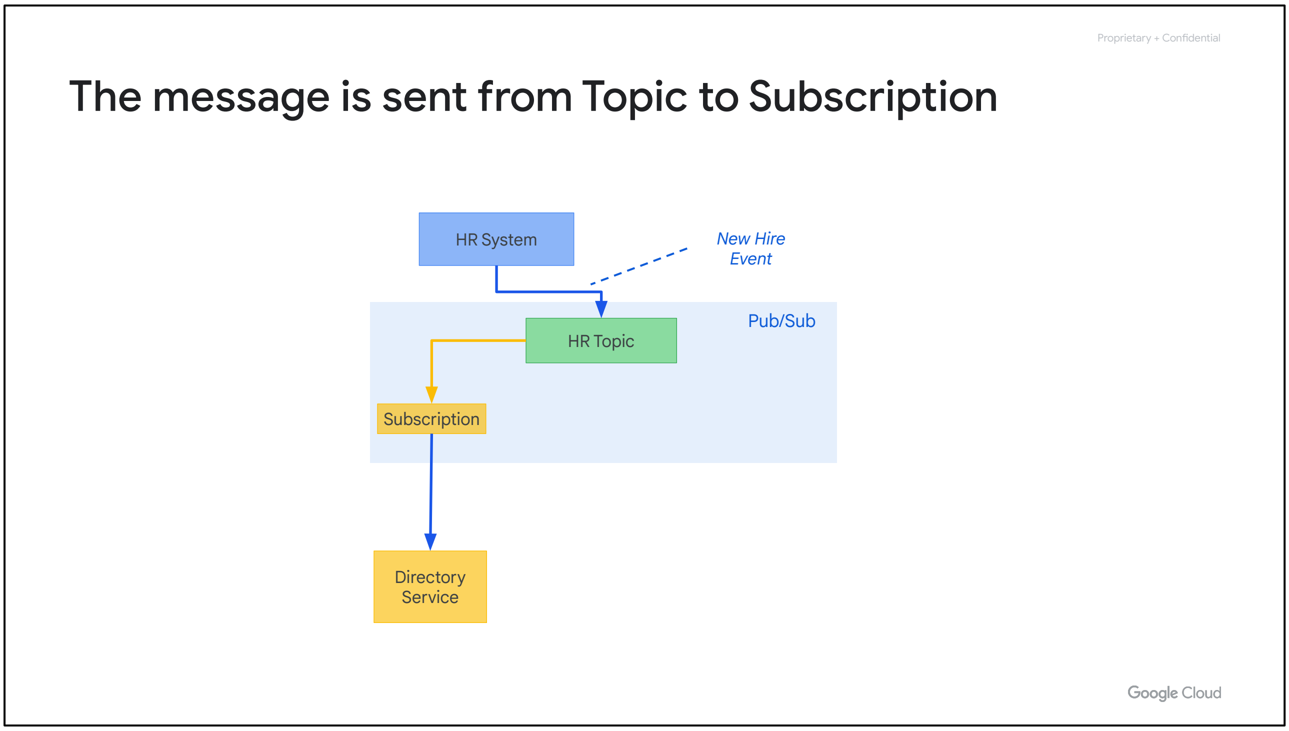


A new employee arrives causing a new hire event

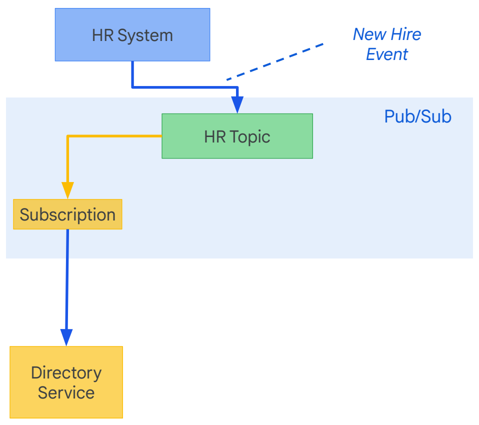


In essence, it’s an enterprise message bus. So, how does it work?

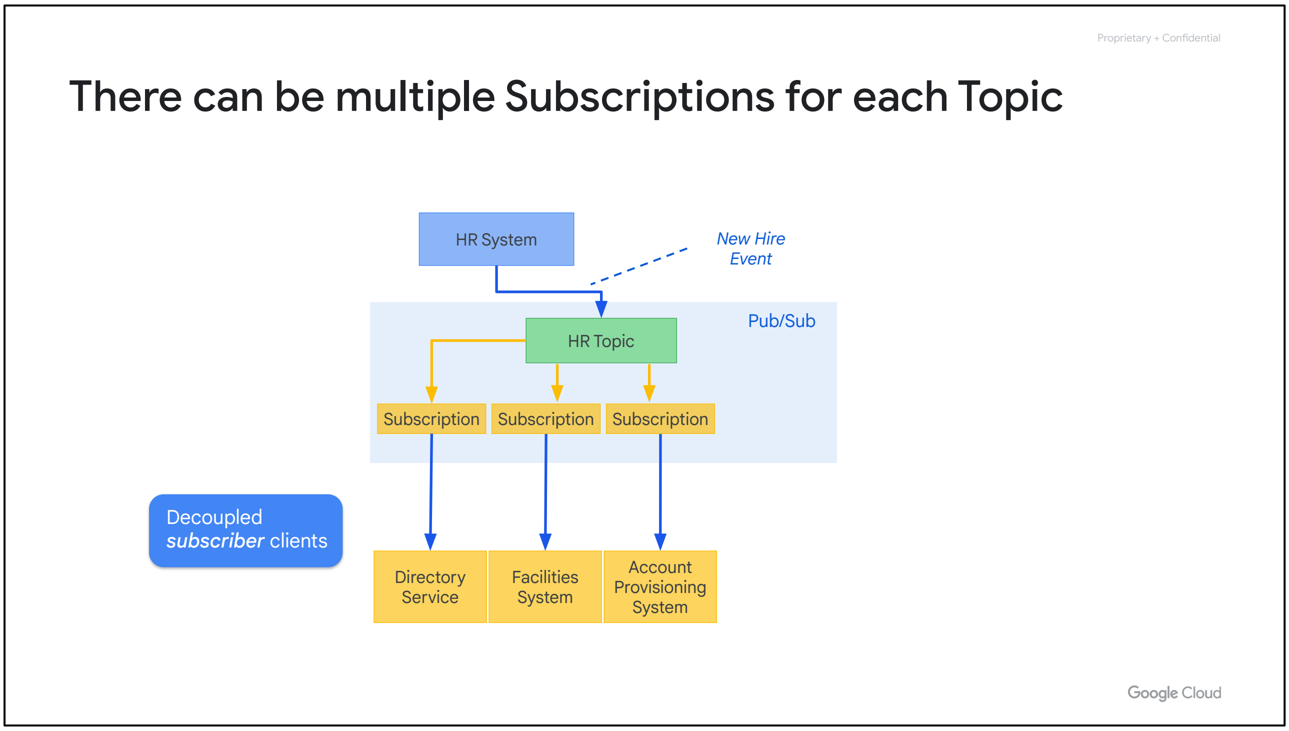
As you see here, there’s an HR Topic that relates to New Hire Events. For example, a new person joins your company and this notification should allow other applications that need to be notified about a new user joining to subscribe and get that message. What applications could tell you that a new person joined?



The message is sent from Topic to Subscription



One example is the Company Directory. This is a client of the Subscription also called a Subscriber. However, Pub/Sub is not limited to one Subscriber or one Subscription.



Proprietary + Confidential

There can be multiple Subscriptions for each Topic

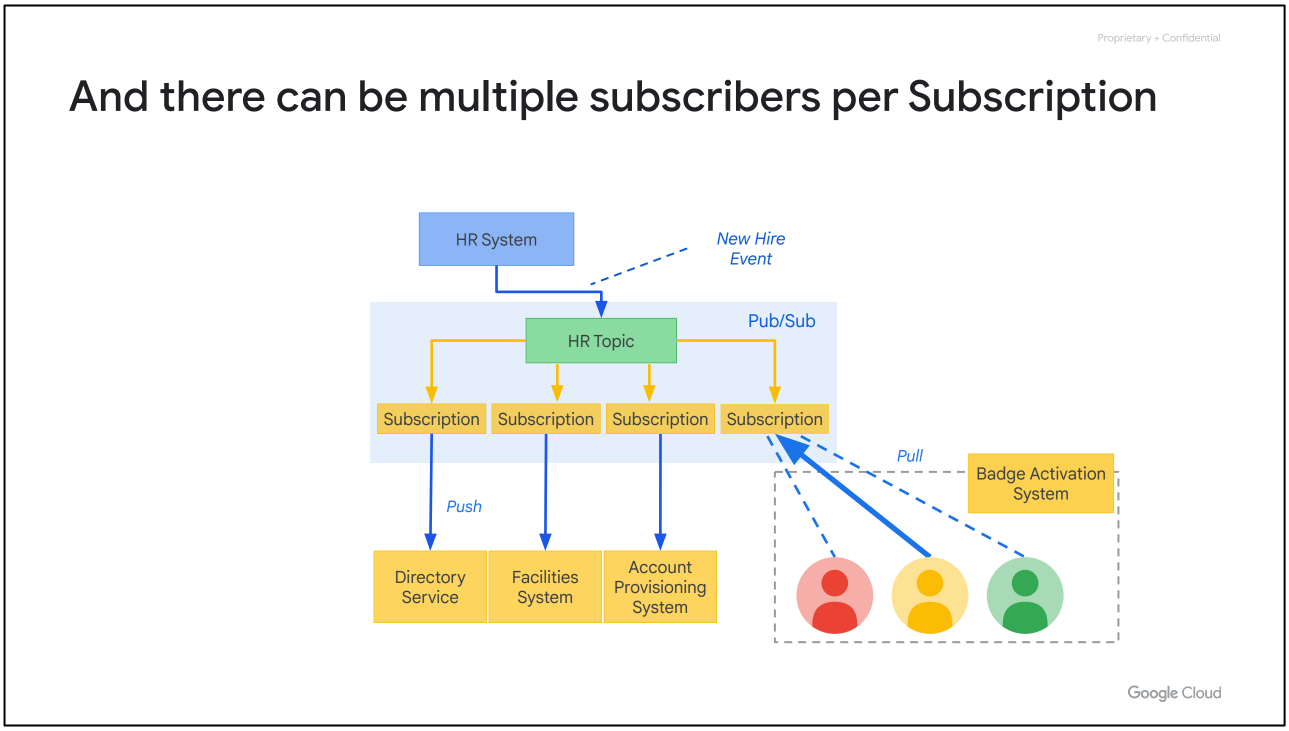
Google Cloud

Here there are multiple Subscriptions and multiple Subscribers. Maybe the Facilities System needs to know about the new employee for badging, and the accounting provisioning system needs to know for payroll.

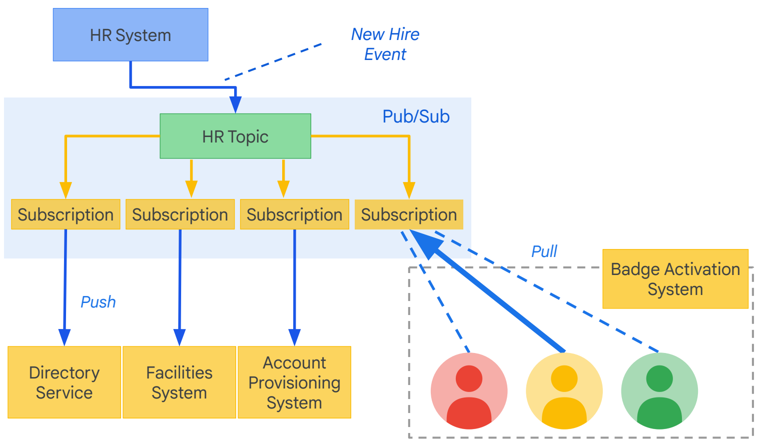
Each Subscription guarantees delivery of the message to the service.

These subscriber clients are decoupled from one another and isolated from the publisher. In fact, we will see later that the HR System could go offline after it has sent its message to the HR Topic, and the message will still be delivered to the subscribers.

These examples show one Subscription and one Subscriber. But you can actually have more Subscribers for a single Subscription.

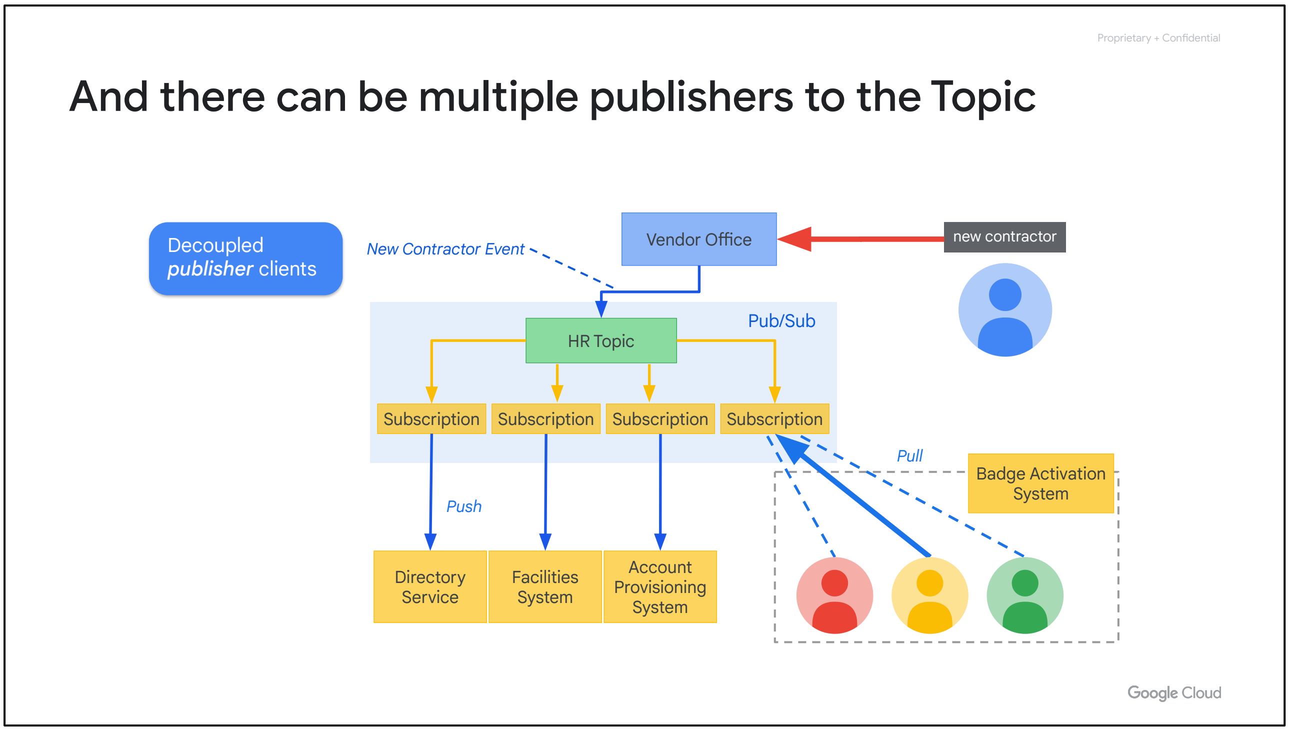


And there can be multiple subscribers per Subscription



In this example, the Badge Activation System requires a human being to activate the badge. There are multiple workers, but not all of them are available all the time.

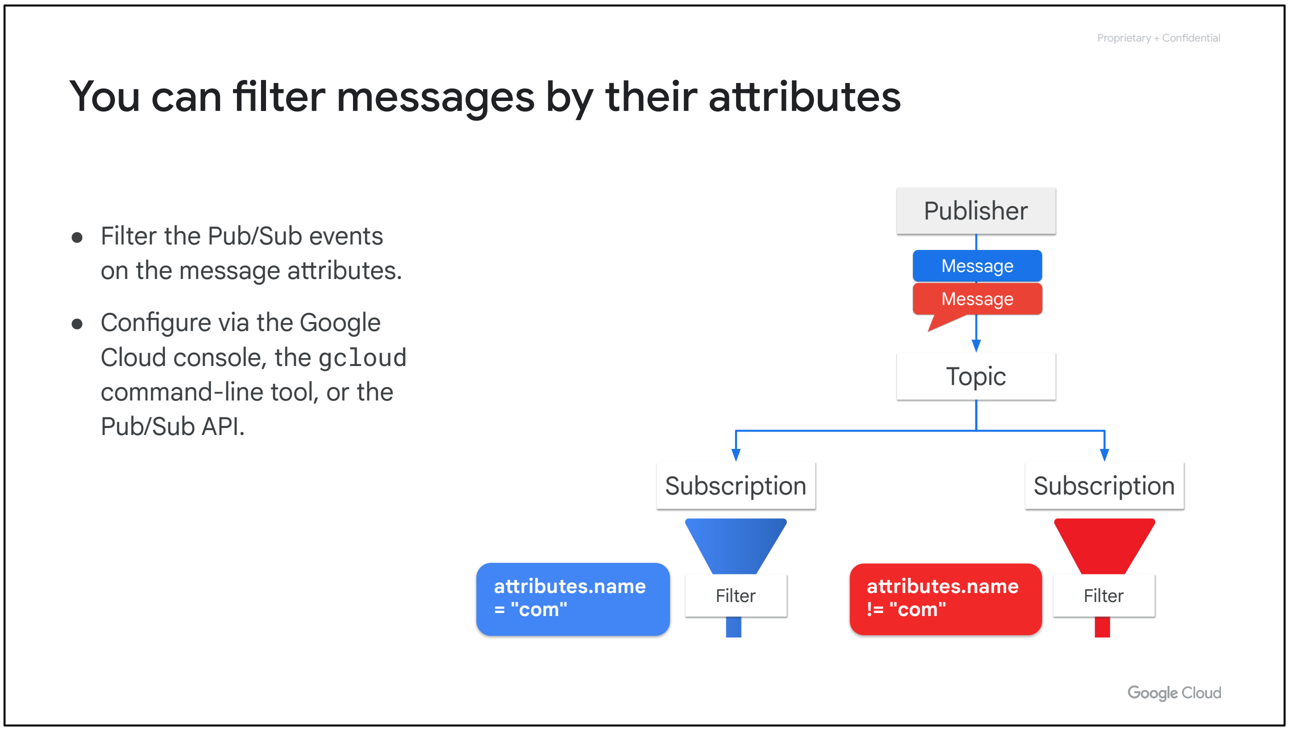
Pub/Sub makes the message available to all of them. But only one person needs to fetch the message and handle it. This is called a Pull Subscription. The other examples are Push Subscriptions.



And there can be multiple publishers to the Topic

Now, a new contractor arrives. Instead of entering through the HR System, they go through the Vendor Office. The same kinds of actions need to occur for this worker. They need to be listed in the company directory. Then, Facilities need to assign them a desk. Account provisioning needs to set up their corporate identity and accounts. And the badge activation system needs to print and activate their contractor badge. A message can be published by the Vendor Office to the HR Topic. The Vendor Office and the HR System are entirely decoupled from one another but can make use of the same company services.

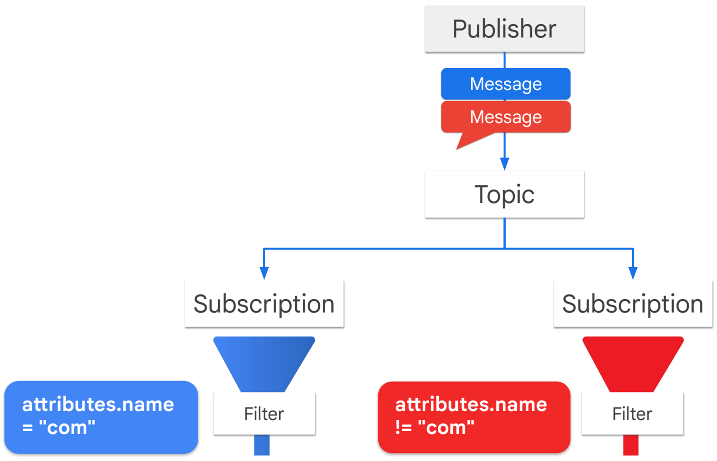
You can see from this illustration how important Pub/Sub is. Therefore, it gets the highest priority.



Proprietary + Confidential

You can filter messages by their attributes

- Filter the Pub/Sub events on the message attributes.
- Configure via the Google Cloud console, the `gcloud` command-line tool, or the Pub/Sub API.



Google Cloud

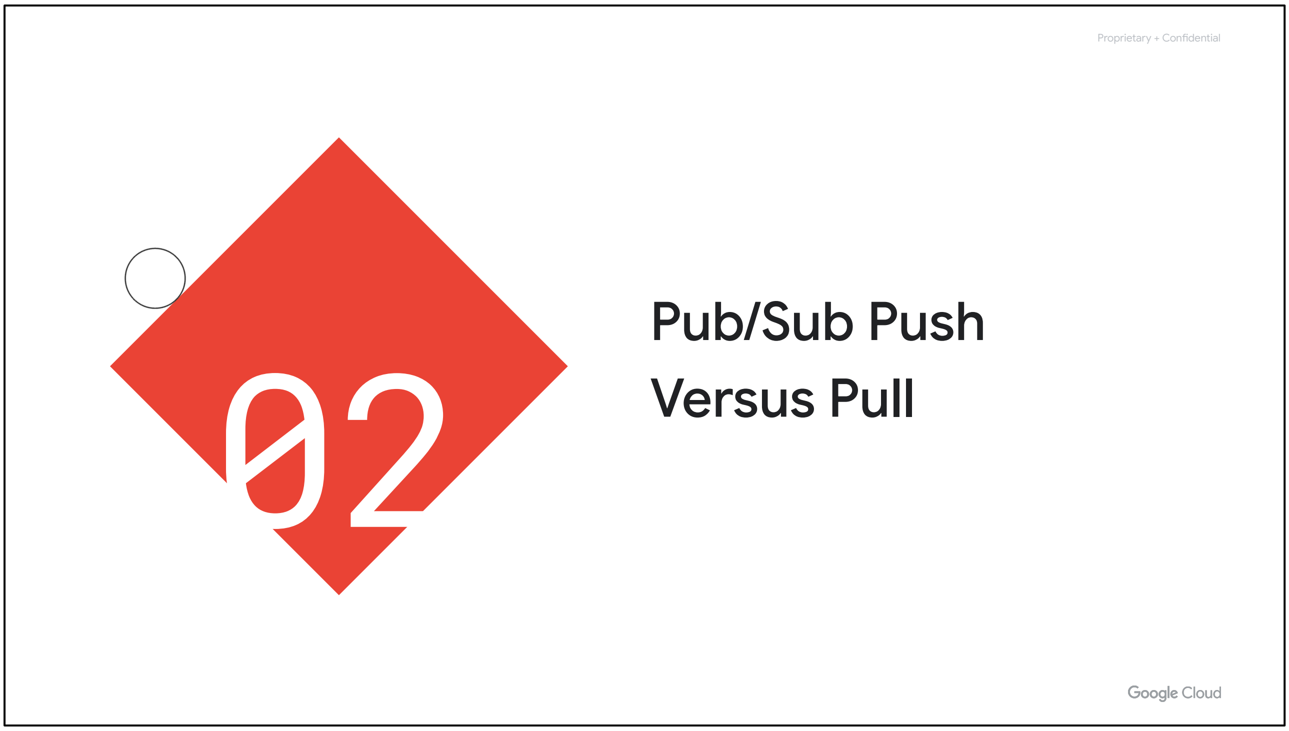
When you receive messages from a subscription with a filter, you only receive the messages that match the filter. The Pub/Sub service automatically acknowledges the messages that don't match the filter. You can filter messages by their attributes.

In the example, the filter sorts messages between those with the name attribute and the value of "com" and those without.

You don't incur egress fees for the messages that Pub/Sub automatically acknowledges. You incur message delivery fees and seek-related storage fees for these messages.

You can create a subscription with a filter using the Google Cloud console, the gcloud command-line tool, or the Pub/Sub API.

You have learned generally what Pub/Sub does. Next, you will learn how it works and many of the advanced features it provides.

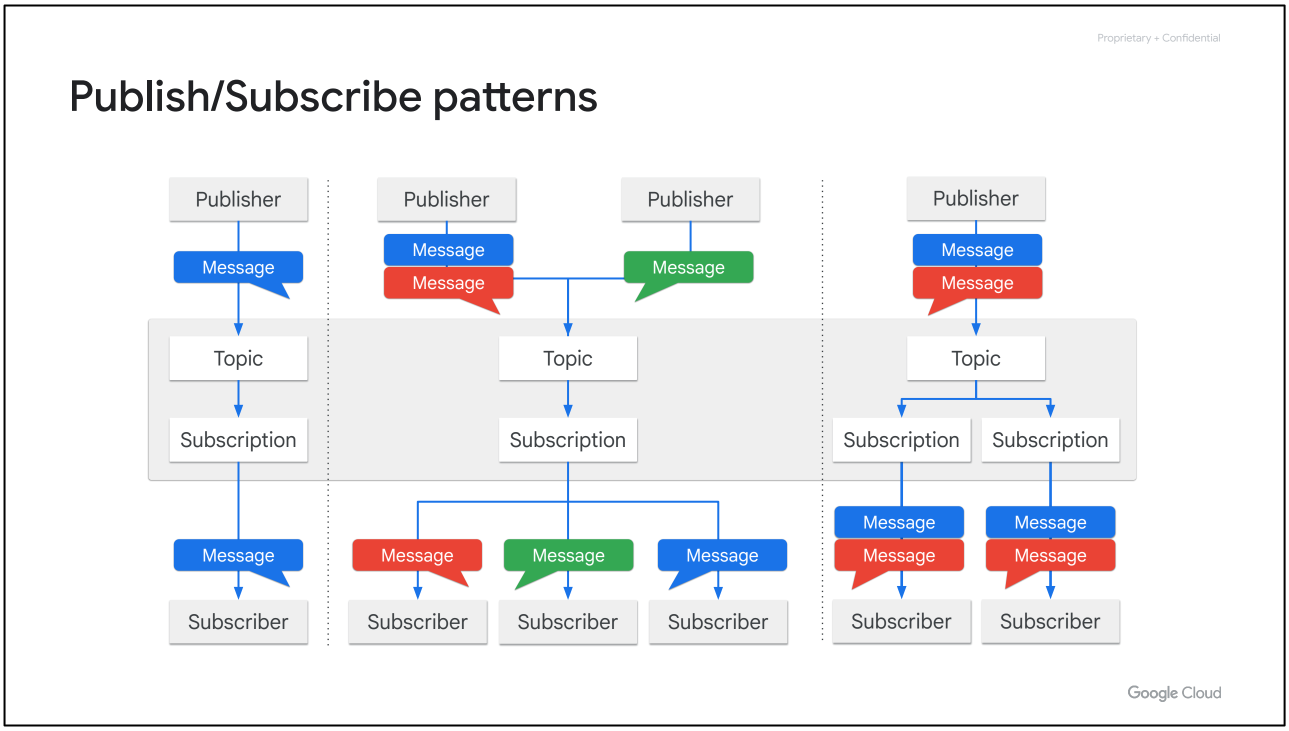


02

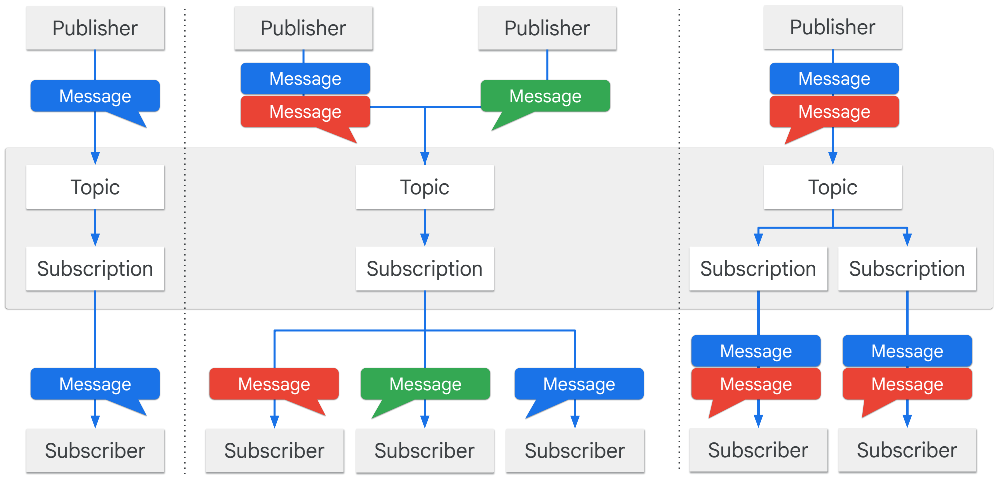
Pub/Sub Push Versus Pull

Google Cloud

From the HR analogy previously, you saw how Pub/Sub works and how it's used in decoupling systems. Let's now discuss the technical details associated.

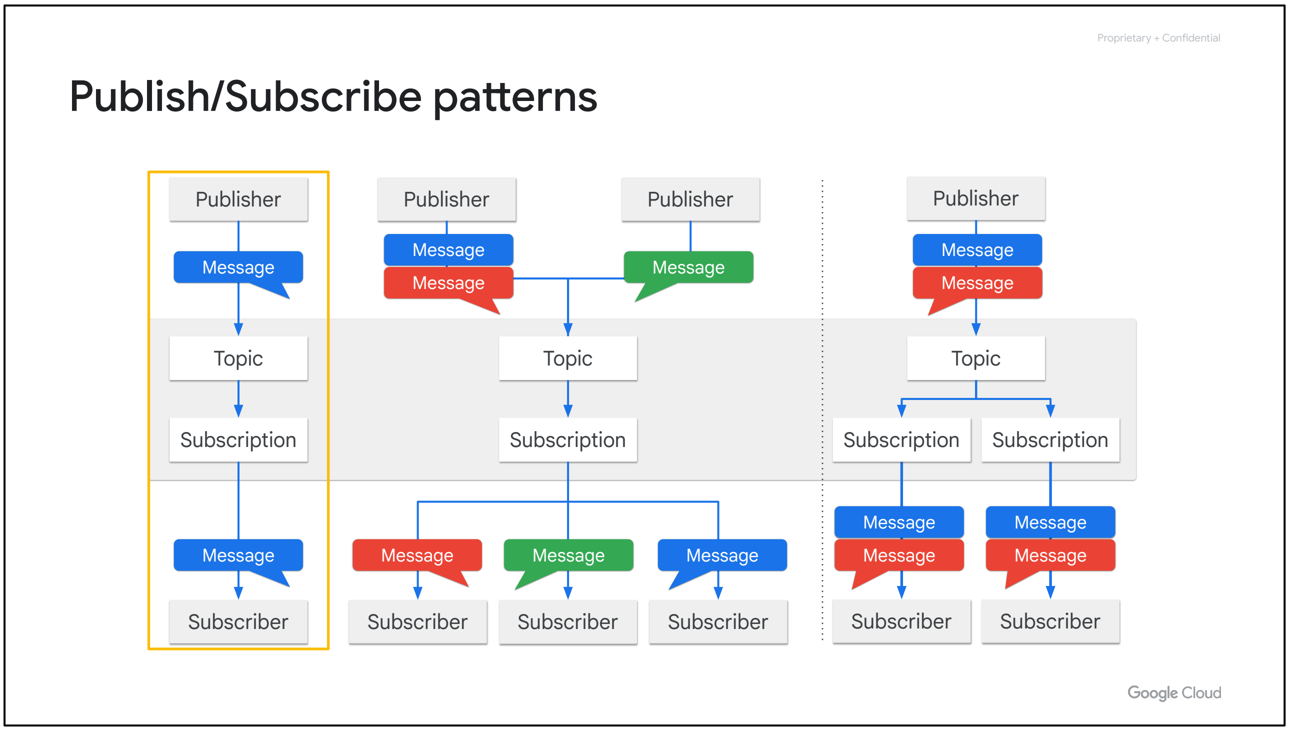


Publish/Subscribe patterns

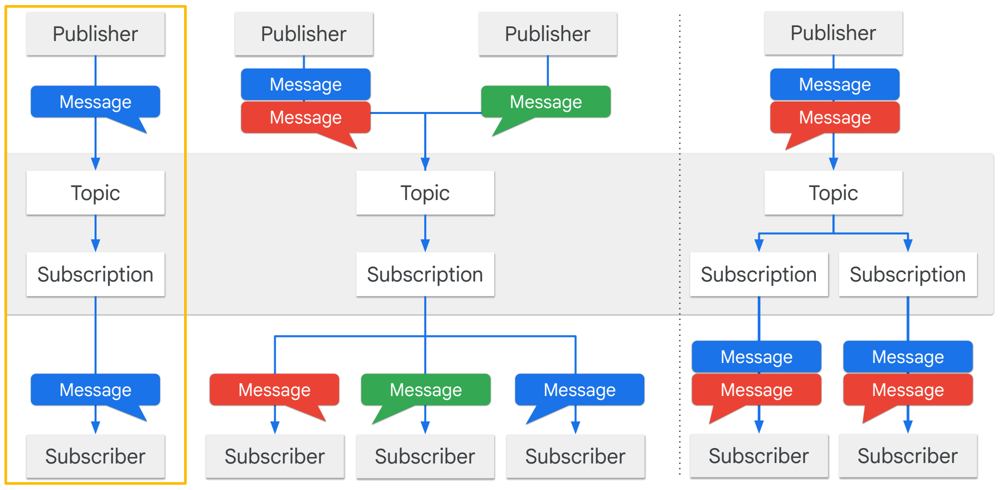


Google Cloud

We'll start by understanding the distribution of messages and different patterns. Here, the different colors represent different messages.

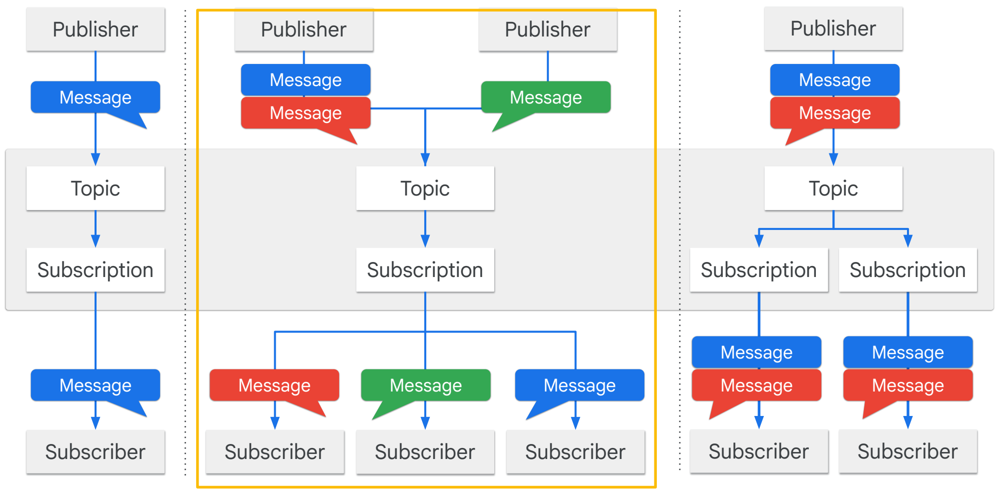


Publish/Subscribe patterns



The first pattern is just a basic straight through flow, where one publisher publishes messages into a topic, which then get consumed by the one subscriber through the one subscription.

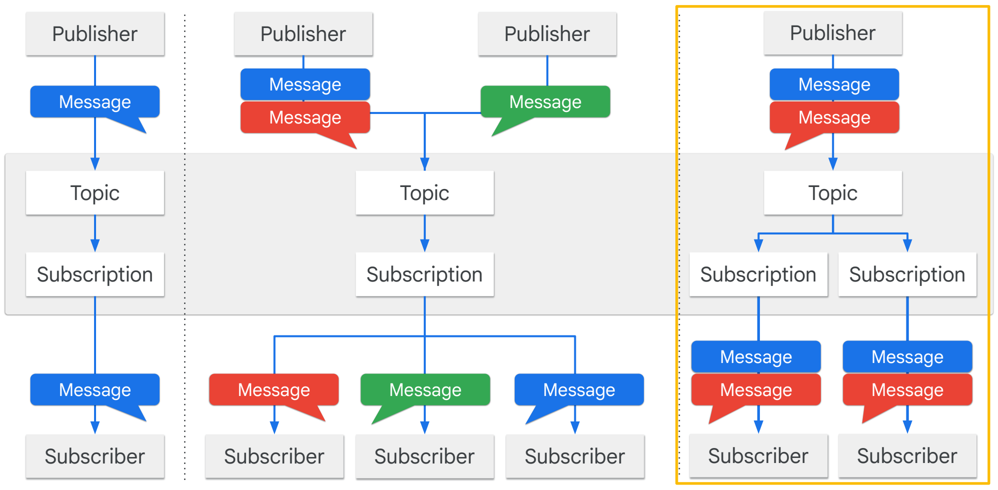
Publish/Subscribe patterns



The second pattern is fan in or load balancing. Multiple publishers can publish the same topic, and multiple subscribers can pull from the same subscription, leveraging parallel processing. What you see here are two different publishers sending three different messages all on the same topic. That means the subscription will get all three messages.

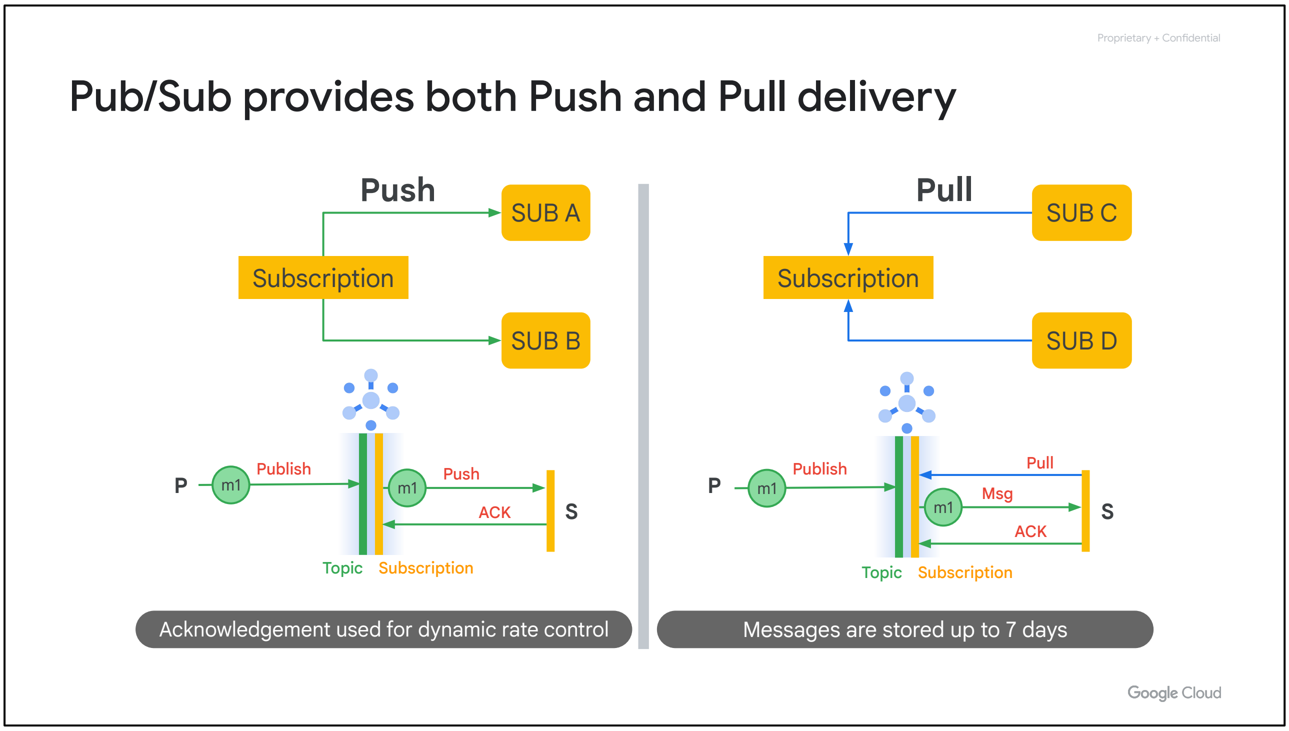
Proprietary + Confidential

Publish/Subscribe patterns



Google Cloud

The third pattern is fan out, where you have many use cases for the same piece of data, and all data is sent to many different subscribers. As you can see here, we have two subscriptions. So both are going to get the messages, both the red message and the blue message.



Pub/Sub provides both Push and Pull delivery

Acknowledgement used for dynamic rate control

Messages are stored up to 7 days

Pub/Sub allows for both Push and Pull delivery.

The Pull model

In the Pull model, your clients are subscribers and will be periodically calling for messages and Pub/Sub will just be delivering the messages since the last call.

In the Pull model, you're going to have to acknowledge the message as a separate step. So, what you see here is we initially make the call to the subscribers, it pulls the messages, it gets a message back, and then, separately, it acknowledges that message.

The reason for this is because the pull queues are often used to implement some kind of queueing system for work to be done, so you don't want to acknowledge the message until you firmly have the message and have done the processing on it, otherwise you might lose the message if the system goes down.

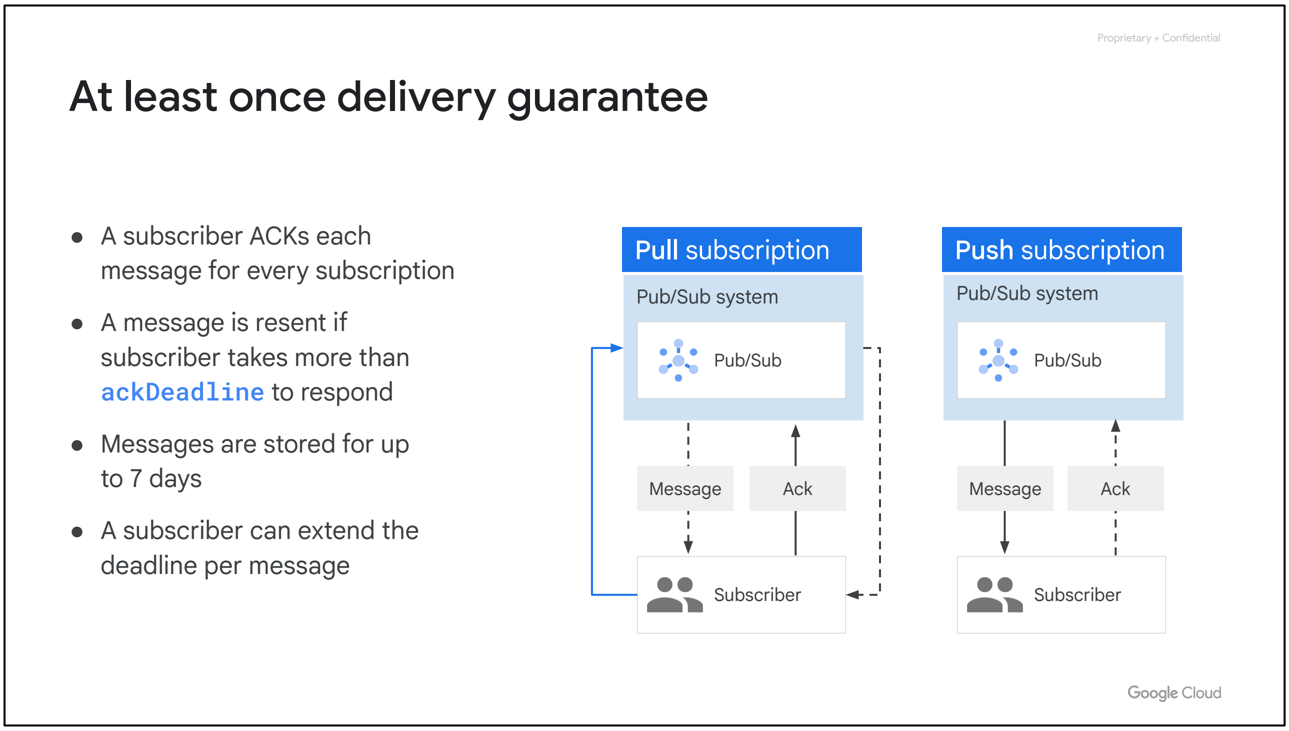
Therefore, we generally recommend you wait to acknowledge until after you have gotten it.

In the Pull model, the messages are stored for up to seven days.

The Push model

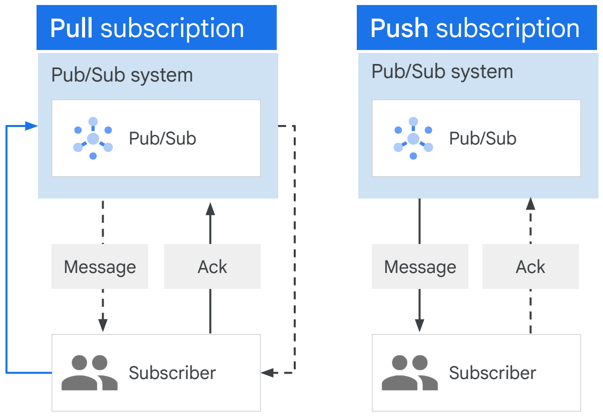
In the Push model, it actually uses an HTTP endpoint. You register a webhook as your subscription, and Pub/Sub infrastructure itself will call you with the latest messages.
In the case of Push, you just respond with ‘status 200 ok’ for the HTTP call, and that tells Pub/Sub the message delivery was successful.

It will actually use the rate of your success responses to self limit so that it doesn’t overload your worker.



At least once delivery guarantee

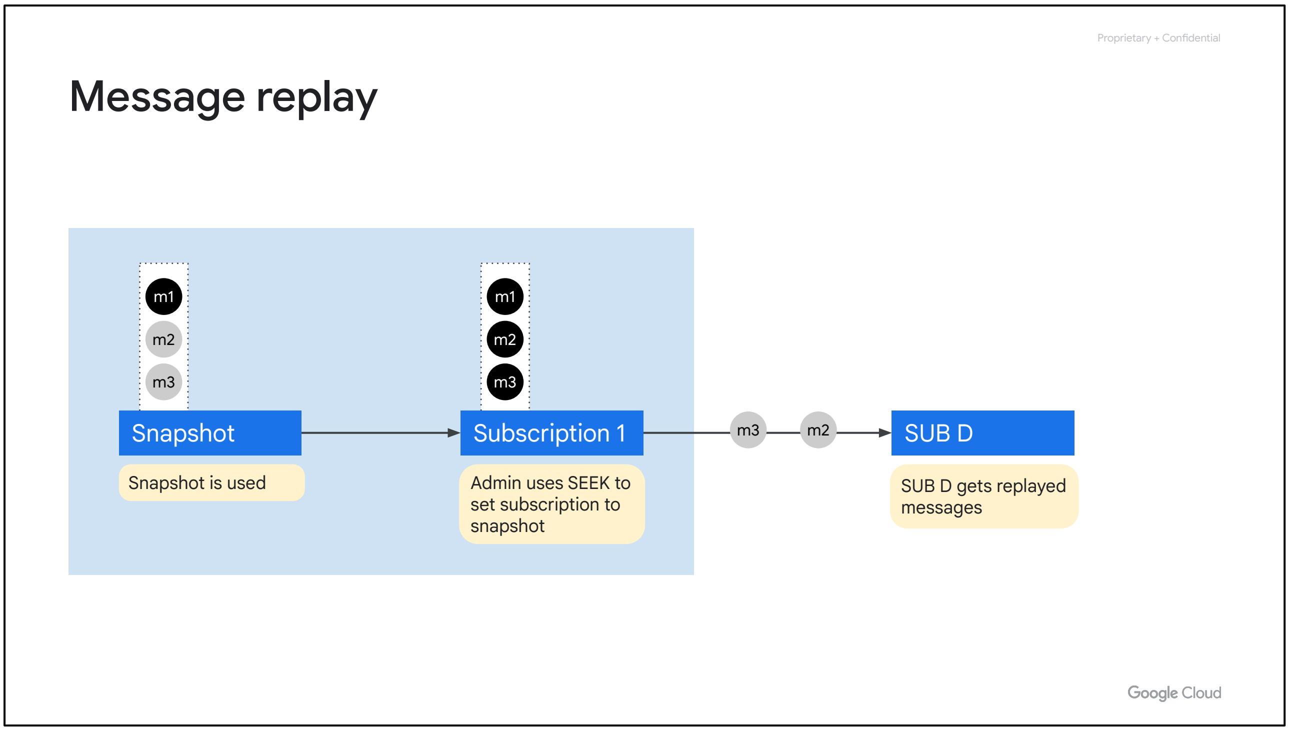
- A subscriber ACKs each message for every subscription
- A message is resent if subscriber takes more than `ackDeadline` to respond
- Messages are stored for up to 7 days
- A subscriber can extend the deadline per message



The way the acknowledgements work is to ensure every message gets delivered at least once. What happens is when you acknowledge a message, you acknowledge on a per subscription basis. So, if you have two subscriptions, you have one acknowledge and the other one doesn't, the one that acknowledged will continue to get the messages. Pub/Sub will continue to try to deliver the message for up to seven days until it is acknowledged.

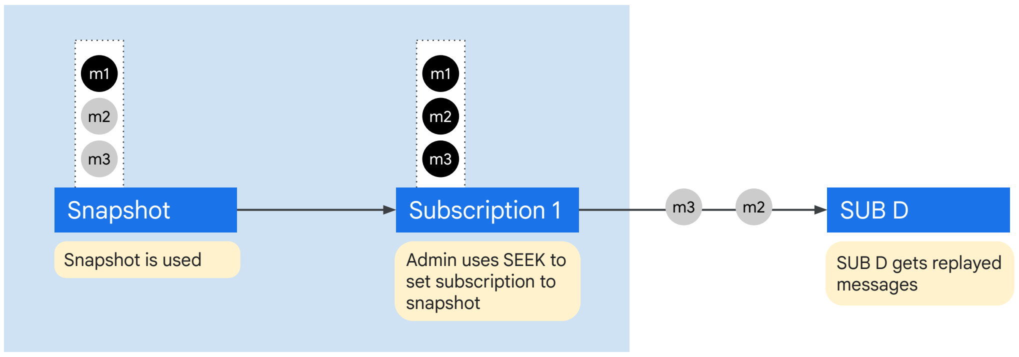
There is a replay mechanism as well that you can rewind and go back in time and have it replay messages, but in any case, you will always be able to go back seven days.

You can also set the acknowledgement deadline and do that on a per subscription. So if you know that on average it takes you 15 seconds to process a message in your work queue, then you might set your acknowledgement deadline to 20 seconds. This will ensure it doesn't try to redeliver the messages.



Proprietary + Confidential

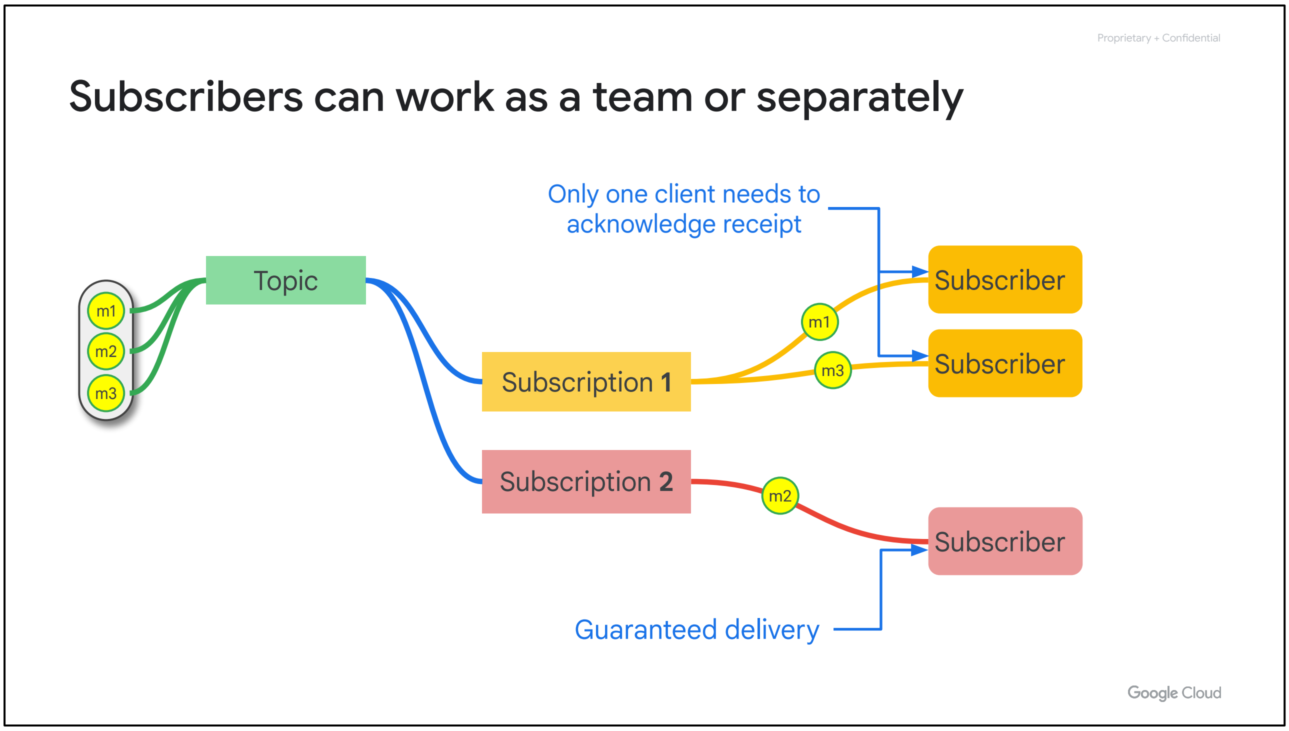
Message replay


m1
m2
m3
Snapshot
Snapshot is used
m1
m2
m3
Subscription 1
Admin uses SEEK to set subscription to snapshot
m3
m2
SUB D
SUB D gets replayed messages

Google Cloud

Configuring a topic with message retention gives you more flexibility, allowing any subscription attached to the topic to seek back in time and replay previously-acknowledged messages. Topic message retention also allows a subscription to replay messages published before a subscription was created. Snapshots are utilized to make replay highly efficient.

If topic message retention is enabled, storage costs for the messages retained by the topic are be billed to the topic's project.



Subscribers can work as a team or separately

Subscribers can work individually or as a group. If we have just one subscriber, it is going to get every message delivered through that subscription. However, you can set up worker pools by having multiple subscribers sharing the same subscription.

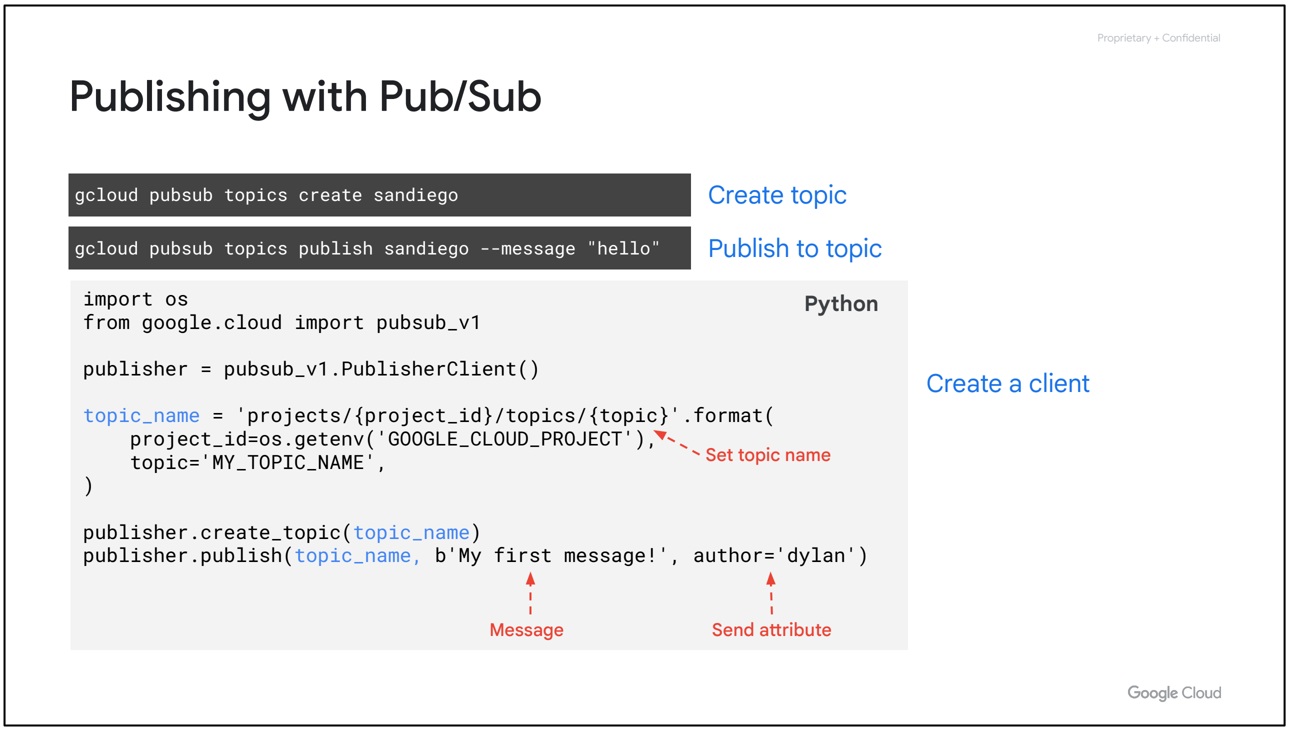
In this case, it is going to distribute the message, so one and three go to Subscription 1, and two goes to Subscription 2. And it is just random based on when it pulls from messages throughout the day.

In the case of a Push subscription, you only have one web inpoint, so you will only have one subscriber typically. But, that one subscriber could be an App Engine standard app, or Cloud Run container image, which autoscales. So, it is one web endpoint, but it can have autoscale workers behind the scenes. And that is actually a very good pattern.

03

Publishing with Pub/Sub Code

With the theoretical background covered in the previous lesson, let's now shift our focus on actual implementation. How are things set up in Pub/Sub?



Publishing with Pub/Sub

```
gcloud pubsub topics create sandiego
```
Create topic

```
gcloud pubsub topics publish sandiego --message "hello"
```
Publish to topic

Python

```
import os
from google.cloud import pubsub_v1

publisher = pubsub_v1.PublisherClient()

topic_name = 'projects/{project_id}/topics/{topic}'.format(
    project_id=os.getenv('GOOGLE_CLOUD_PROJECT'),
    topic='MY_TOPIC_NAME',
)

publisher.create_topic(topic_name)
publisher.publish(topic_name, b'My first message!', author='dylan')
```

Let's look at a little bit of code now. This example is using the Client Library for Pub/Sub.

If we want to publish the message, we first create the topic.

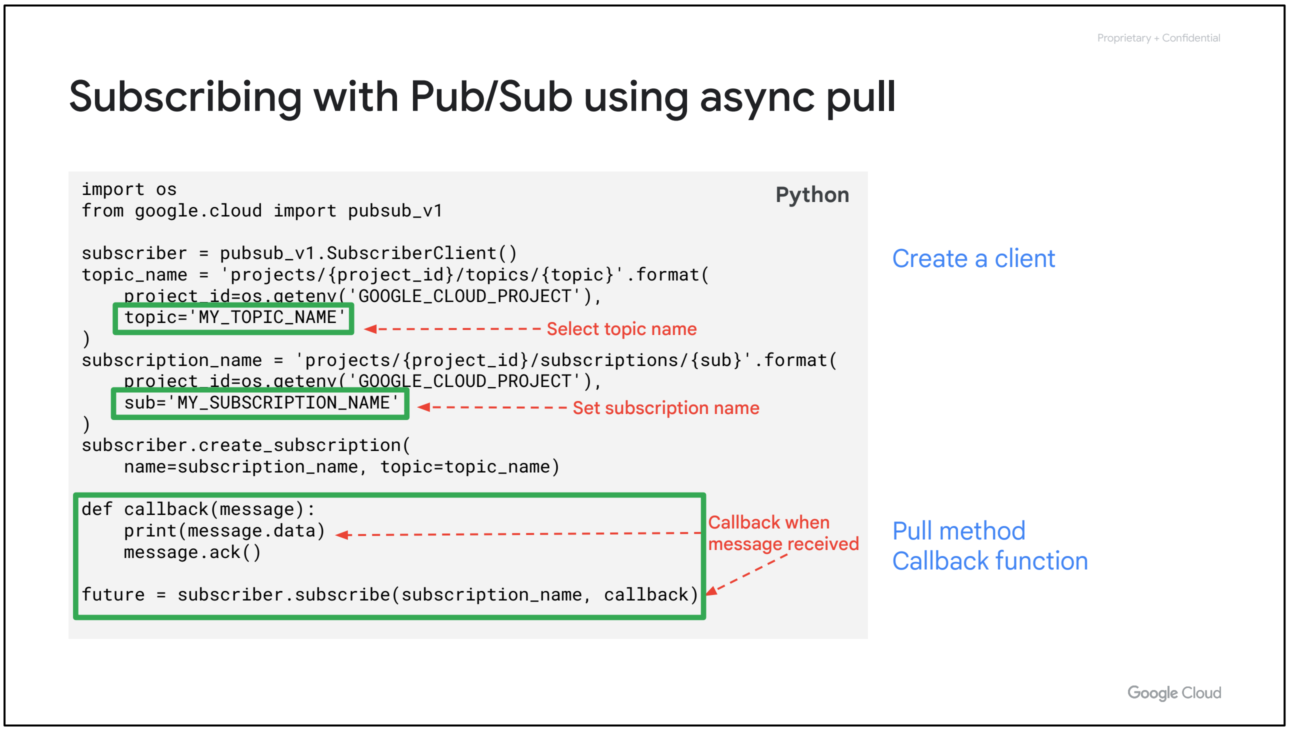
Then, we can publish the topic on the command line.

More commonly, it will be done in code. Here we get a PublisherClient, create a topic, and publish the message.

Notice the letter b, in front of 'My first message!' This is because Pub/Sub just sends raw bytes. This means that you aren't restricted to just text. You can send other data, like images if you wanted to. The limit is 10 MB.

There are also extra attributes that you can include in messages. In this example, you see author equals dylan. Pub/Sub will keep track of those attributes to allow your downstream systems to get metadata about your messages without having to put it all in the message and parse it out. So instead of serializing and de-serializing, it will just keep track of those key value pairs.

Some of them have special meaning. We will see some of those shortly.



Subscribing with Pub/Sub using async pull

Python

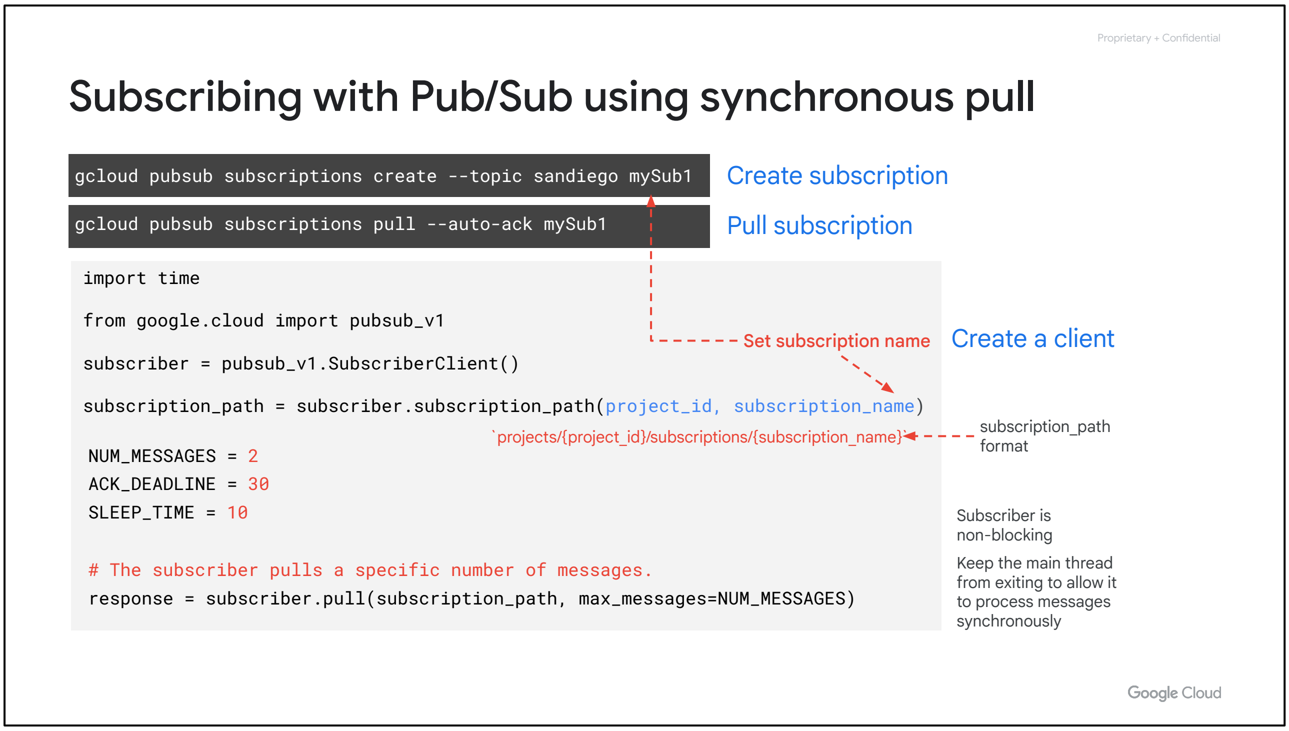
```
import os
from google.cloud import pubsub_v1

subscriber = pubsub_v1.SubscriberClient()
topic_name = 'projects/{project_id}/topics/{topic}'.format(
    project_id=os.getenv('GOOGLE_CLOUD_PROJECT'),
    topic='MY_TOPIC_NAME'
)
subscription_name = 'projects/{project_id}/subscriptions/{sub}'.format(
    project_id=os.getenv('GOOGLE_CLOUD_PROJECT'),
    sub='MY_SUBSCRIPTION_NAME'
)
subscriber.create_subscription(
    name=subscription_name, topic=topic_name)

def callback(message):
    print(message.data)
    message.ack()

future = subscriber.subscribe(subscription_name, callback)
```

Create a client

Pull method
Callback function

To subscribe with Pub/Sub using the Pull method, the code is similar.

- Select the topic.
- Name the subscription.
- This is a pull subscription, so we will define a callback.



Subscribing with Pub/Sub using synchronous pull

```
gcloud pubsub subscriptions create --topic sandiego mySub1
```

Create subscription

```
gcloud pubsub subscriptions pull --auto-ack mySub1
```

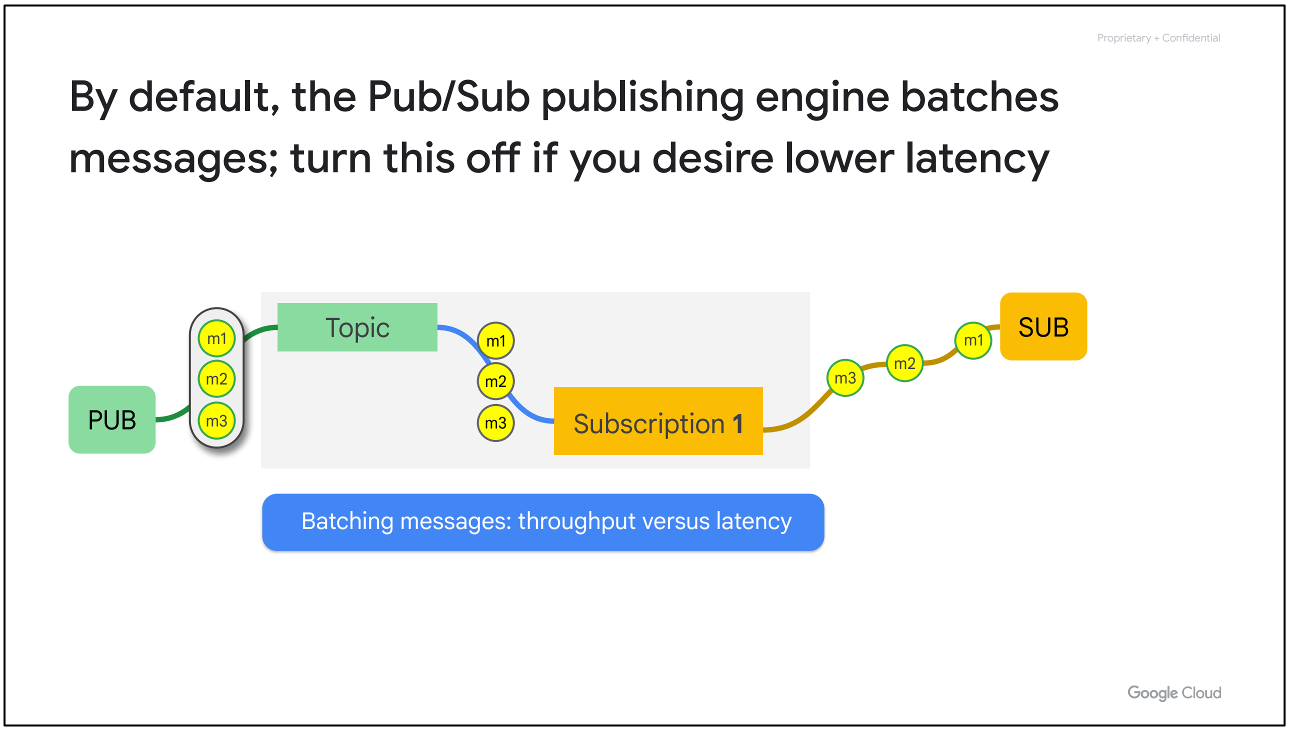
Pull subscription

```
import time

from google.cloud import pubsub_v1

subscriber = pubsub_v1.SubscriberClient()

subscription_path = subscriber.subscription_path(project_id, subscription_name)

 NUM_MESSAGES = 2
 ACK_DEADLINE = 30
 SLEEP_TIME = 10

 # The subscriber pulls a specific number of messages.
 response = subscriber.pull(subscription_path, max_messages=NUM_MESSAGES)
```

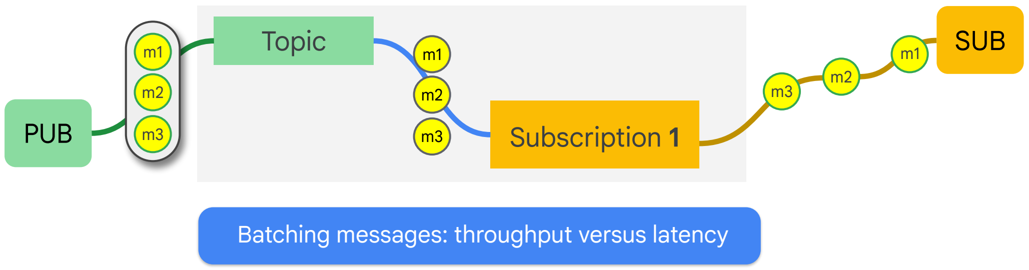
Subscriber is non-blocking

Keep the main thread from exiting to allow it to process messages synchronously

When you are doing a pull subscription, it looks like this. You can pull messages from the command line. You will see this in the lab. By default it will just churn one message, the latest message, but there is a dash-dash limit you can set. Maybe you want 10 messages at a time, you can try that in the lab.

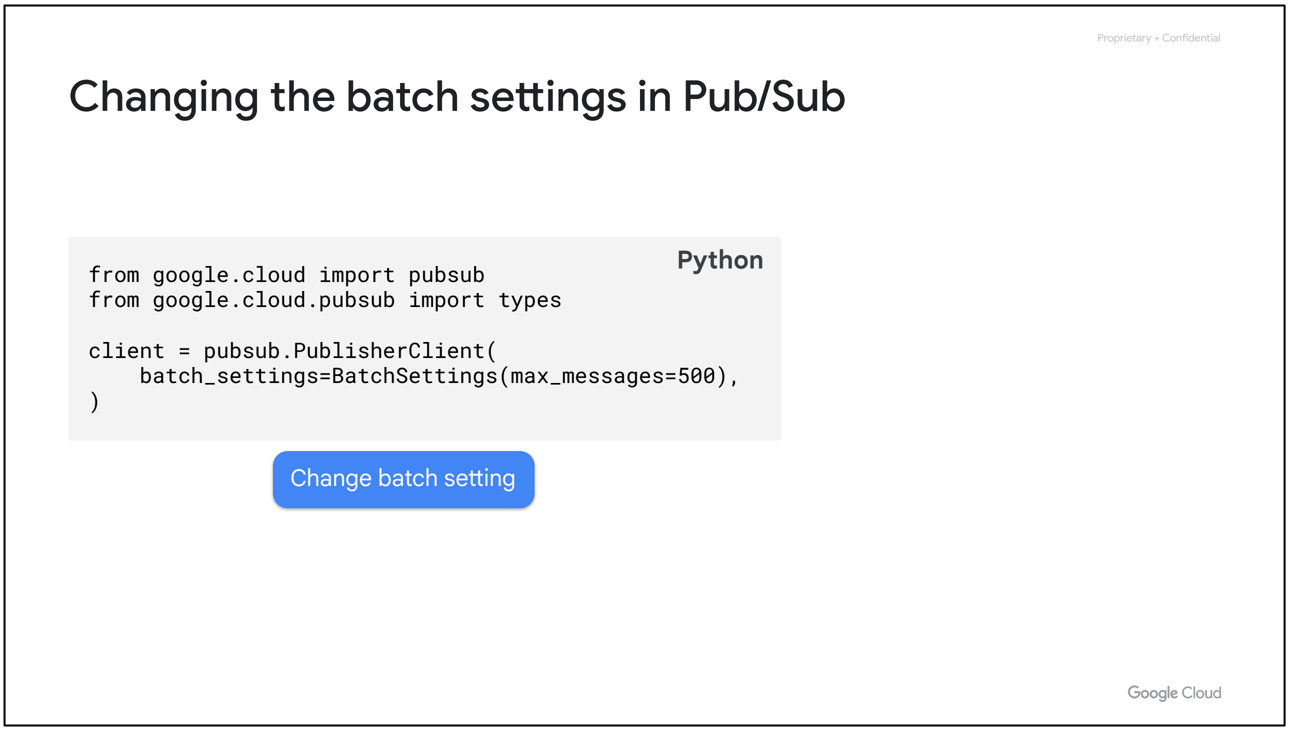


By default, the Pub/Sub publishing engine batches messages; turn this off if you desire lower latency



You can also batch publish messages. This just prevents the overhead of the call for individual messages on the publishing side. This allows the Pub/Sub publishing engine to wait and send 10, or 50, at a time. This increases efficiency. However, if you are waiting for 50 messages, this means the first one now has latency associated with it. So, it is a trade off in your system. What do you want to optimize? But, in any case, even if you batch publish, they still get delivered one at a time to your subscribers.

We will practice this technique in the lab.



Changing the batch settings in Pub/Sub

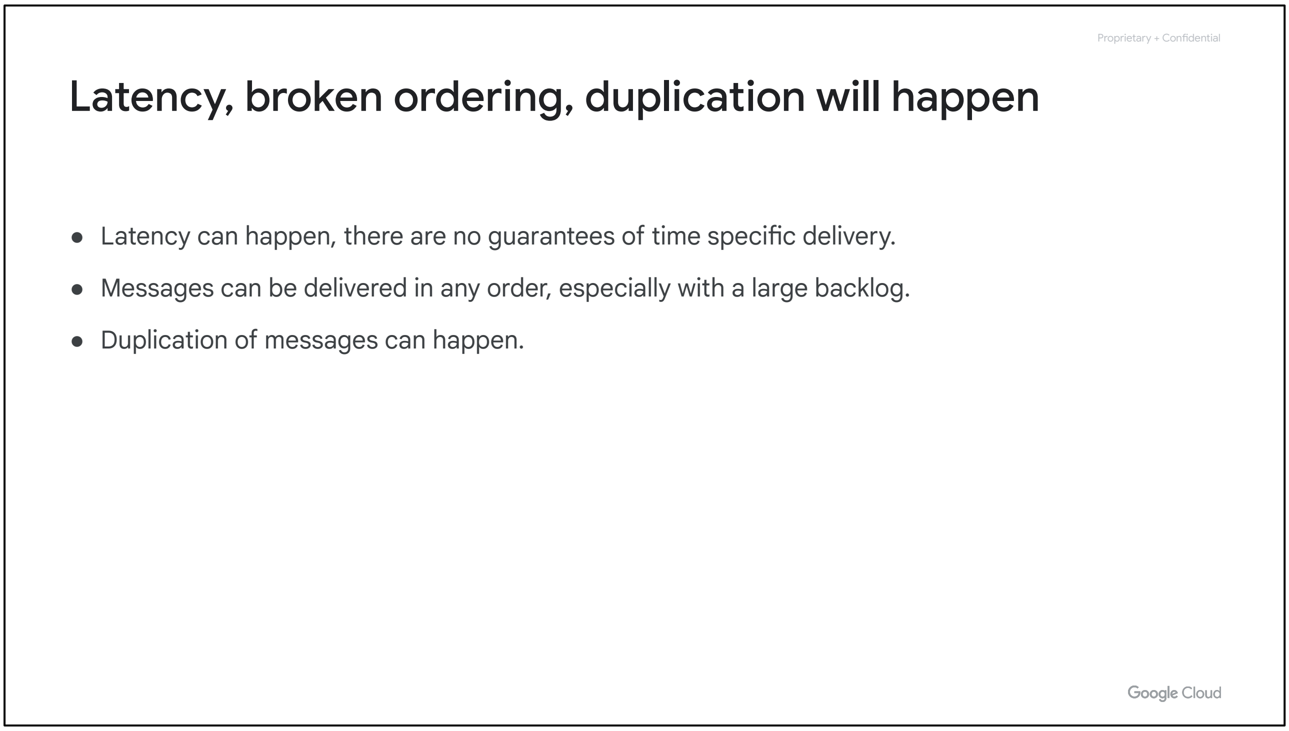
Python

```
from google.cloud import pubsub
from google.cloud.pubsub import types

client = pubsub.PublisherClient(
    batch_settings=BatchSettings(max_messages=500),
)
```

Change batch setting

Here is how you would set the batch in Python code.

Explore the documentation for other command options and settings.

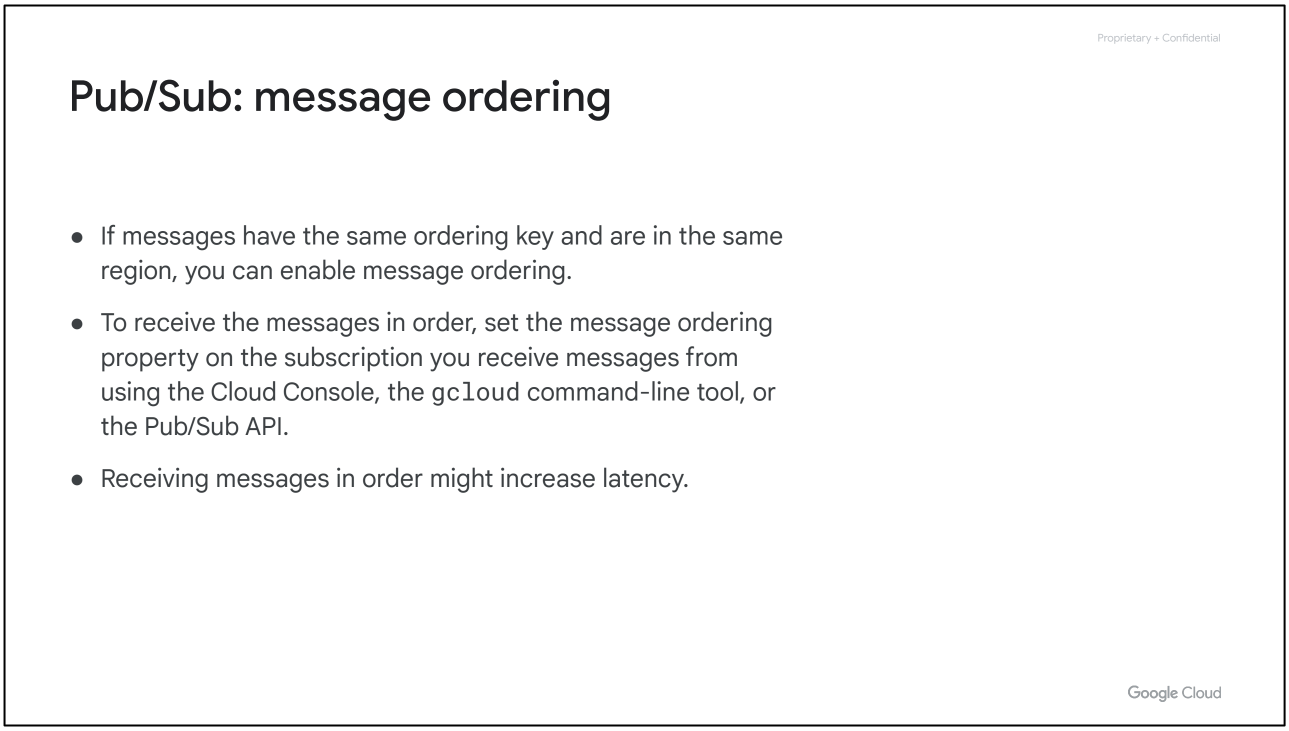


Latency, broken ordering, duplication will happen

- Latency can happen, there are no guarantees of time specific delivery.
- Messages can be delivered in any order, especially with a large backlog.
- Duplication of messages can happen.

By default, Pub/Sub ensures at least once delivery, but doesn't guarantee that a message is sent only once unless exactly-once delivery is configured. Please note that exactly-once delivery is only available for pull messages.

There can be latency and duplications. Messages can be delivered to subscribers in any order especially when there is a large backlog.



Proprietary + Confidential

Pub/Sub: message ordering

- If messages have the same ordering key and are in the same region, you can enable message ordering.
- To receive the messages in order, set the message ordering property on the subscription you receive messages from using the Cloud Console, the `gcloud` command-line tool, or the Pub/Sub API.
- Receiving messages in order might increase latency.

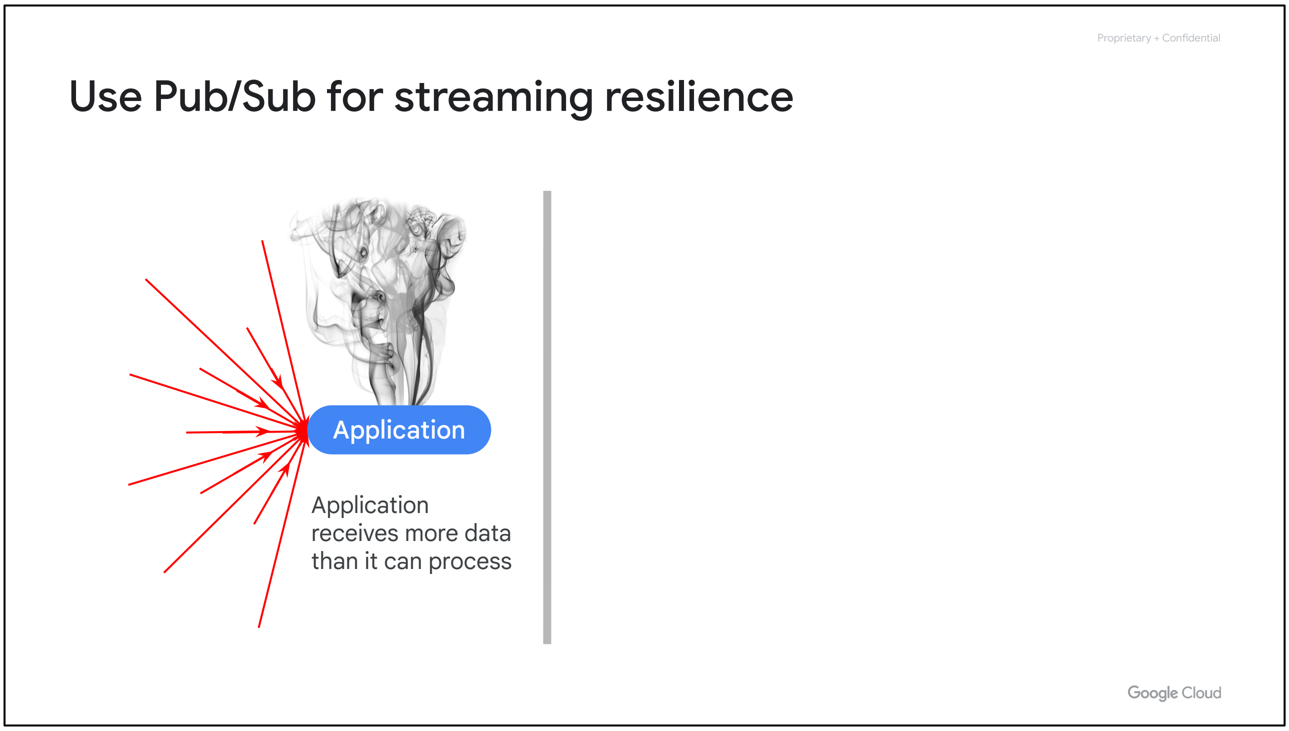
Google Cloud

If messages have the same ordering key and are in the same region, you can enable message ordering and receive the messages in the order that the Pub/Sub service receives them.

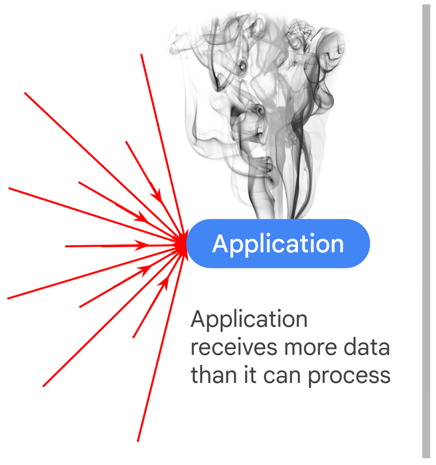
When the Pub/Sub service re-delivers a message with an ordering key, the Pub/Sub service also redelivers every subsequent message with the same ordering key, including acknowledged messages. If both message ordering and a dead-letter topic are enabled on a subscription, the ordering may not be true, as Pub/Sub forwards messages to dead-letter topics on a best-effort basis.

To receive the messages in order, set the message ordering property on the subscription you receive messages from. Please note that receiving messages in order might increase latency.

You can set the message ordering property when you create a subscription using the Cloud Console, the gcloud command-line tool, or the Pub/Sub API.

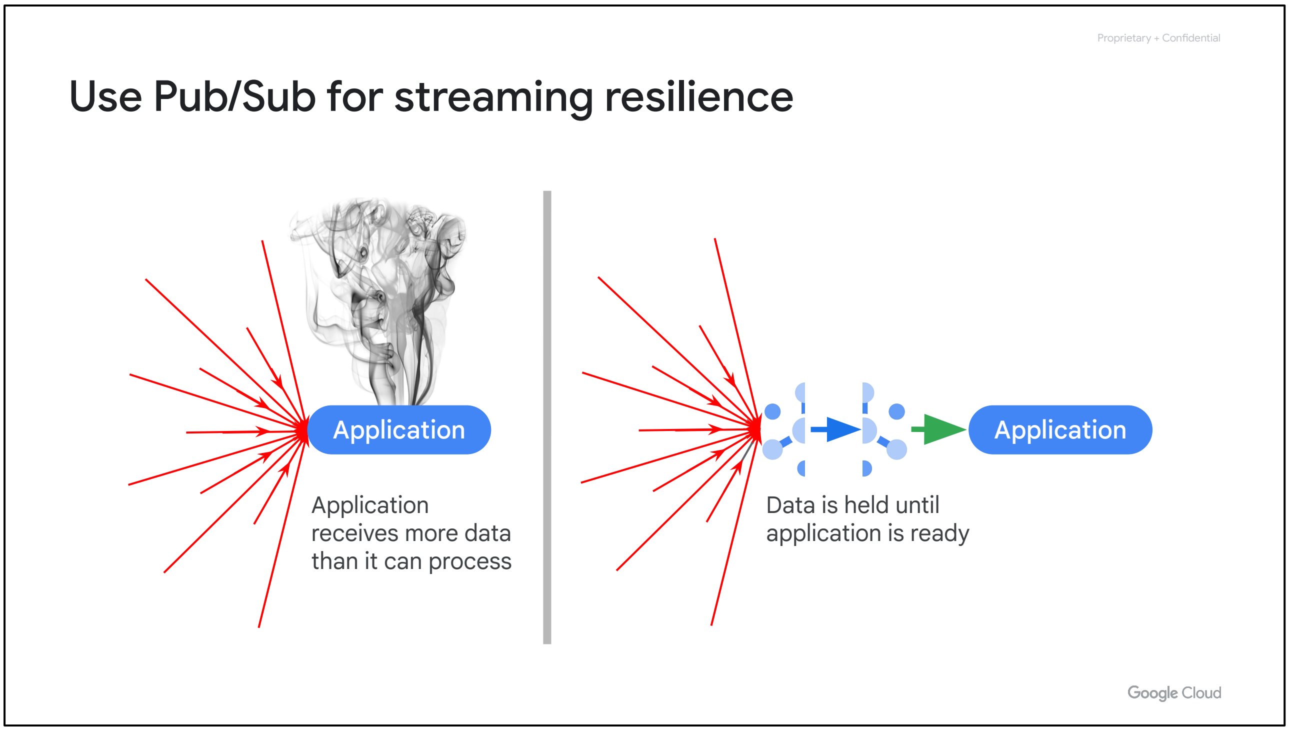


Use Pub/Sub for streaming resilience

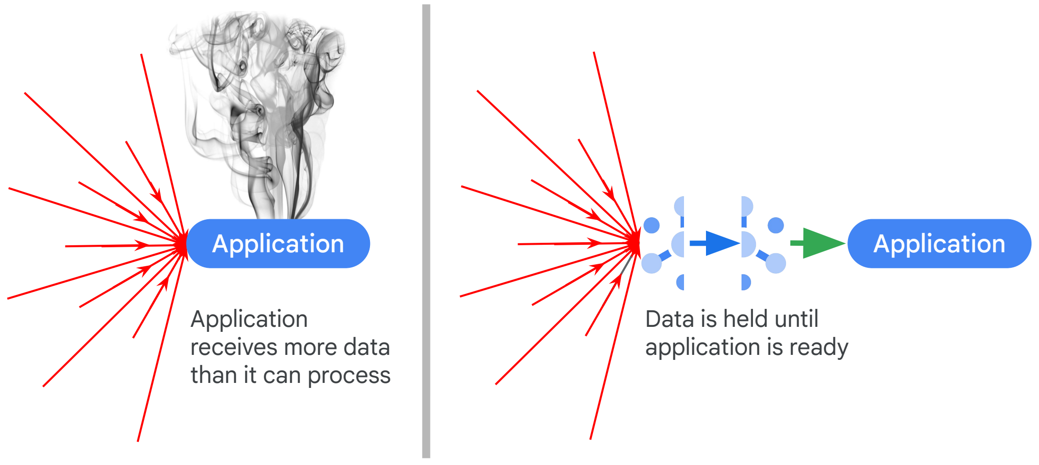


Application receives more data than it can process

Pub/Sub is also going to help us with streaming resilience, or buffering. What happens if your systems get overloaded with large volumes of transactions, like Black Friday? What you really need is some sort of buffer or backlog so that you can feed messages only as fast as the systems are able to process them. Pub/Sub has this as a built-in capability.

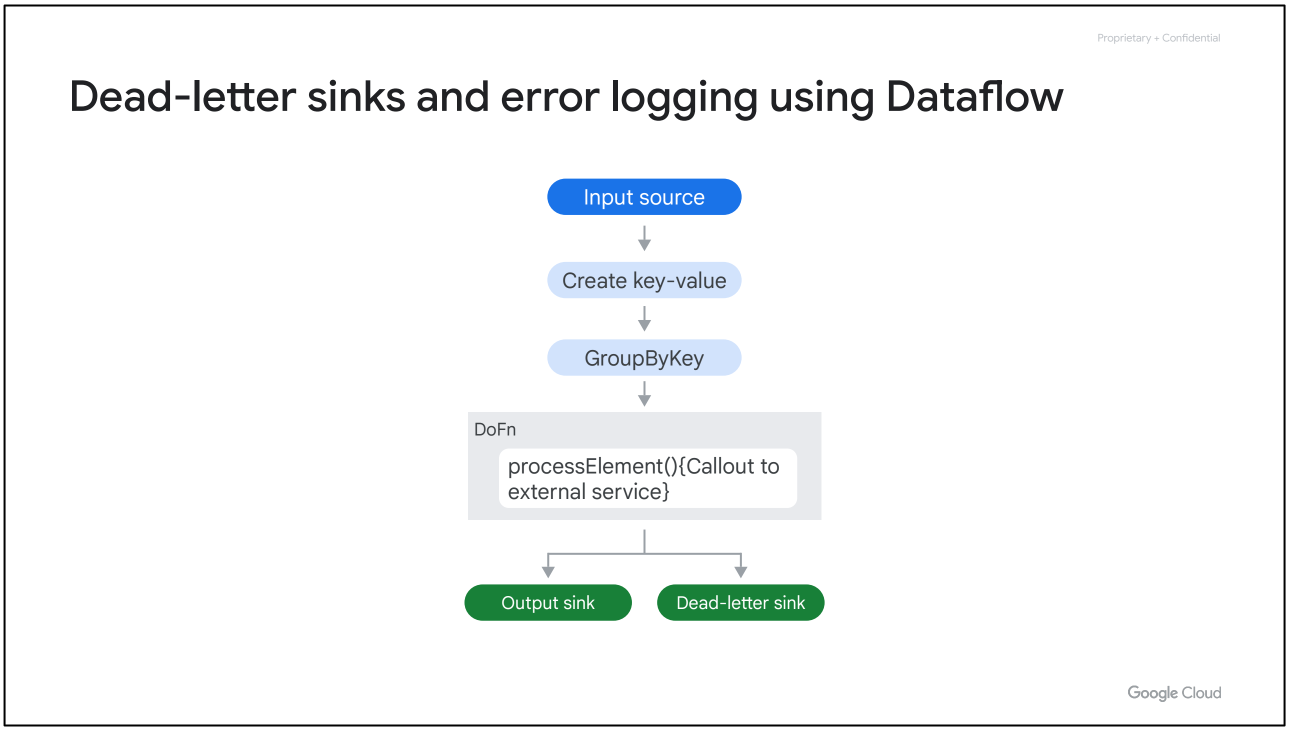


Use Pub/Sub for streaming resilience

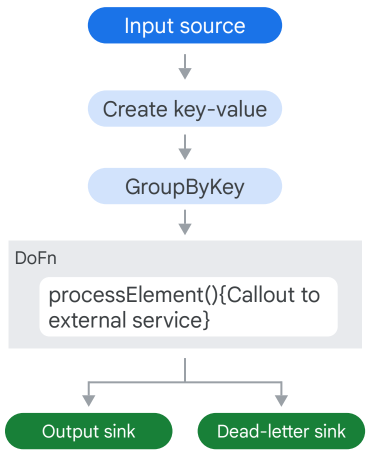

Application
Application receives more data than it can process
Data is held until application is ready
Application

Let’s recap this then. When you look at the example on this slide, in the example on the left, an overload of arriving data causes a traffic spike. This overdrives the resources of the application as illustrated by the smoke. One solution to this problem is to size the application to handle the highest traffic spike plus some additional capacity as a safety buffer. This is not only wasteful of resources, which must be retained at top capacity even when not being used, but it provides a recipe for a distributed denial of service attack by creating an upper limit at which the application will cease to behave normally and will exhibit non-deterministic behavior.

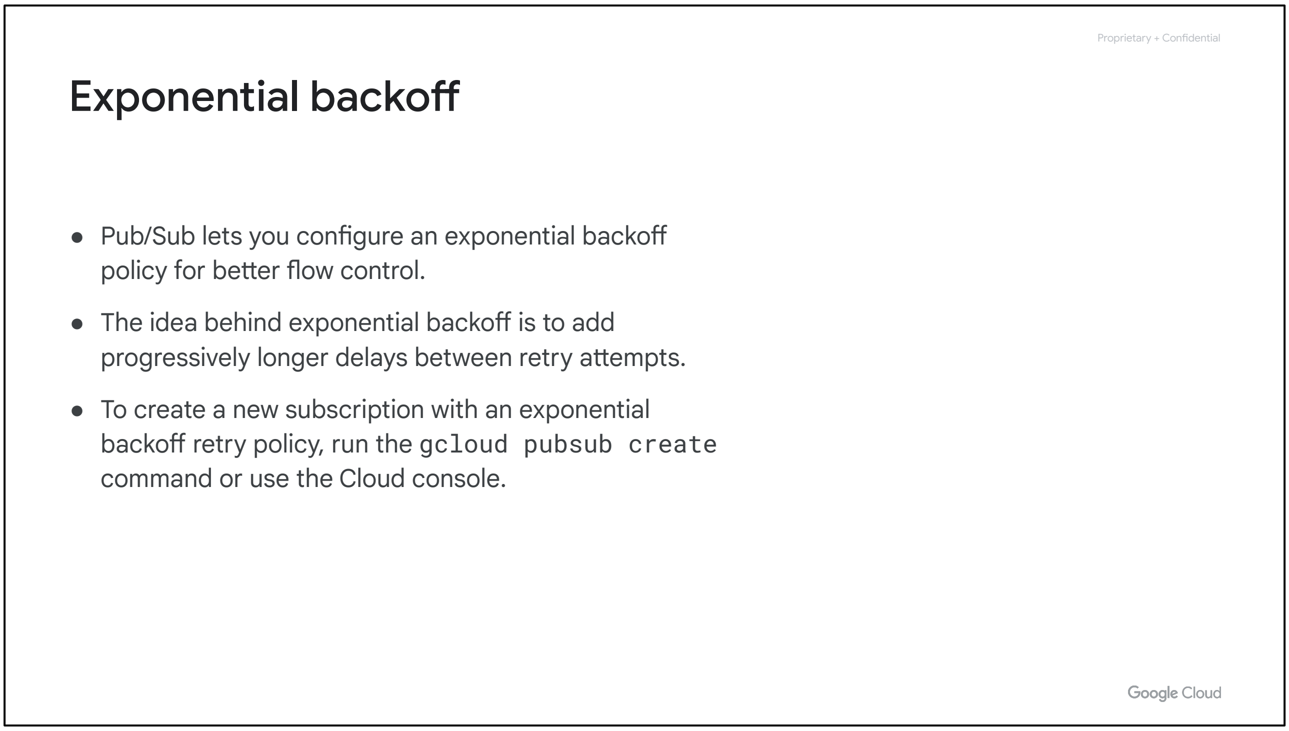
The solution on the right uses Pub/Sub as an intermediary, receiving and holding data until the application has resources to handle it, either through processing the backlog of work, or by autoscaling to meet the demand.



Dead-letter sinks and error logging using Dataflow



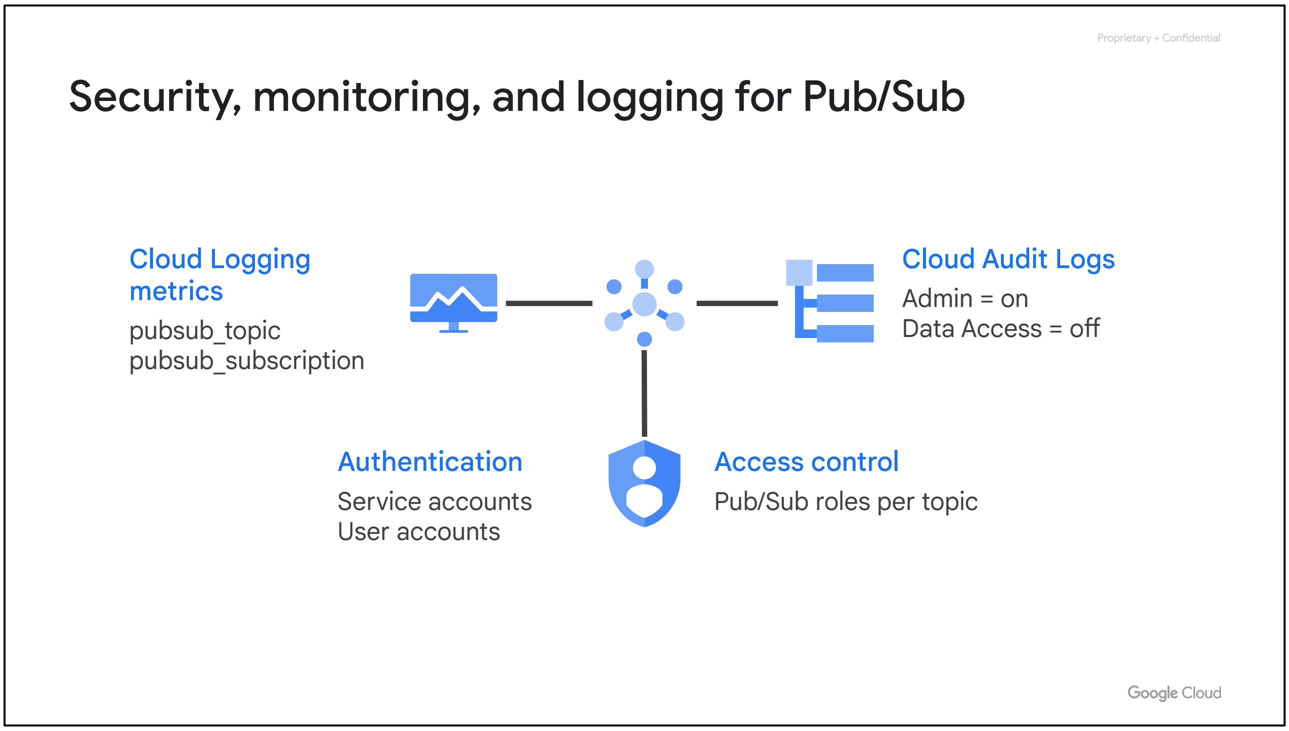
Erroneous records may cause your pipeline to get stuck or fail outright. We highly recommend implementing a dead-letter queue and error logging to prevent these failure modes. These can help catch problems in user code and/or data shape.



Exponential backoff

- Pub/Sub lets you configure an exponential backoff policy for better flow control.
- The idea behind exponential backoff is to add progressively longer delays between retry attempts.
- To create a new subscription with an exponential backoff retry policy, run the `gcloud pubsub create` command or use the Cloud console.

The idea behind exponential backoff is to add progressively longer delays between retry attempts. After the first delivery failure, Pub/Sub will wait for a minimum backoff time before retrying. For each consecutive failure on that message, more time will be added to the delay, up to a maximum delay. The maximum and minimum delay intervals are not fixed, and should be configured based on local factors to your application.



Security, monitoring, and logging for Pub/Sub

Cloud Audit Logs
Cloud Audit Logs maintains three audit logs for each Google Cloud project, folder, and organization: **Admin Activity**, **Data Access**, and **System Event**.

Admin Activity audit logs contain log entries for API calls or other administrative actions that modify the configuration or metadata of resources.

Data Access audit logs contain API calls that read the configuration or metadata of resources, as well as user-driven API calls that create, modify, or read user-provided resource data.

Admin Activity audit logs are always written; you can't configure or disable them.

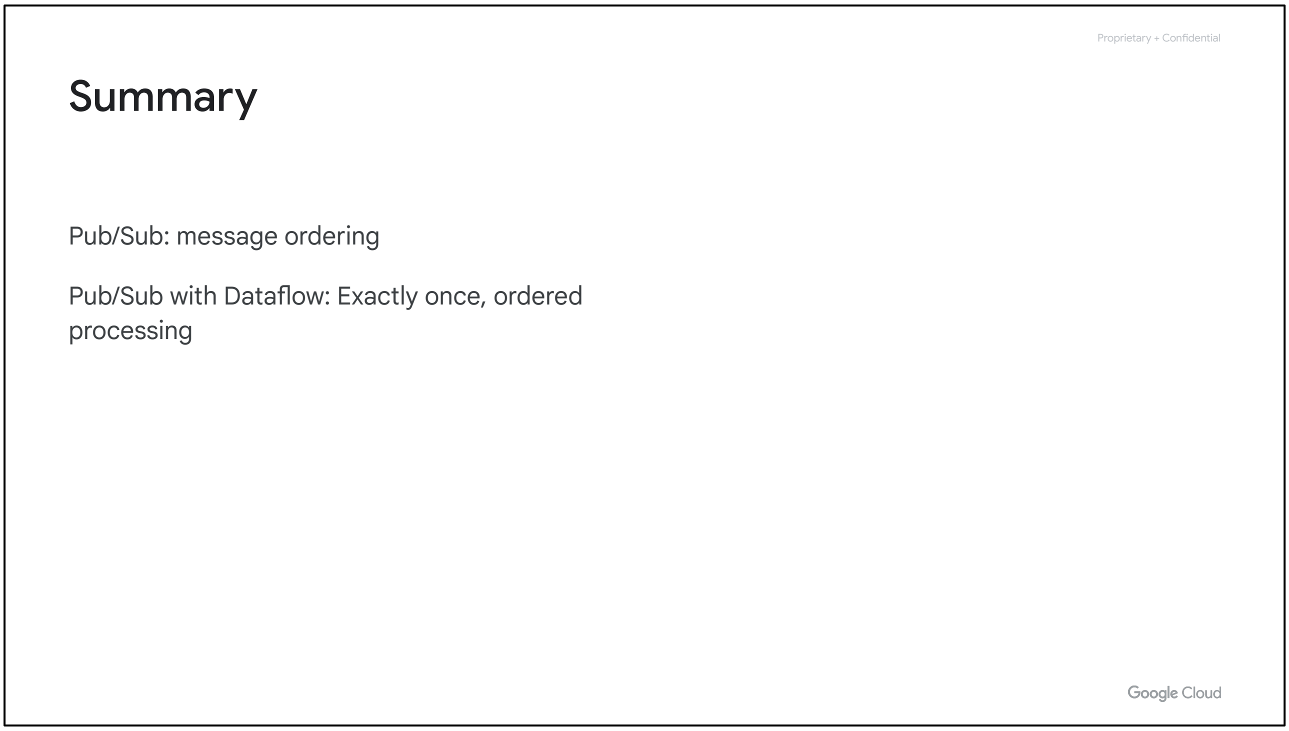
Data Access audit logs are disabled by default because they can be quite large; they must be explicitly enabled to be written.

Cloud Logging metrics
Pub/Sub reports metrics to Cloud Logging, including pubsub_topic and pubsub_subscription, which you can monitor against service quota utilization in a dashboard, and for setting notifications and alerts.

Authentication
Authentication is provided by service accounts. You can also directly authenticate

users by their user accounts and their identity is reported in audit logs. But user account authentication is not recommended.

Access control
Access control is provided by IAM.



Summary

Pub/Sub: message ordering

Pub/Sub with Dataflow: Exactly once, ordered processing

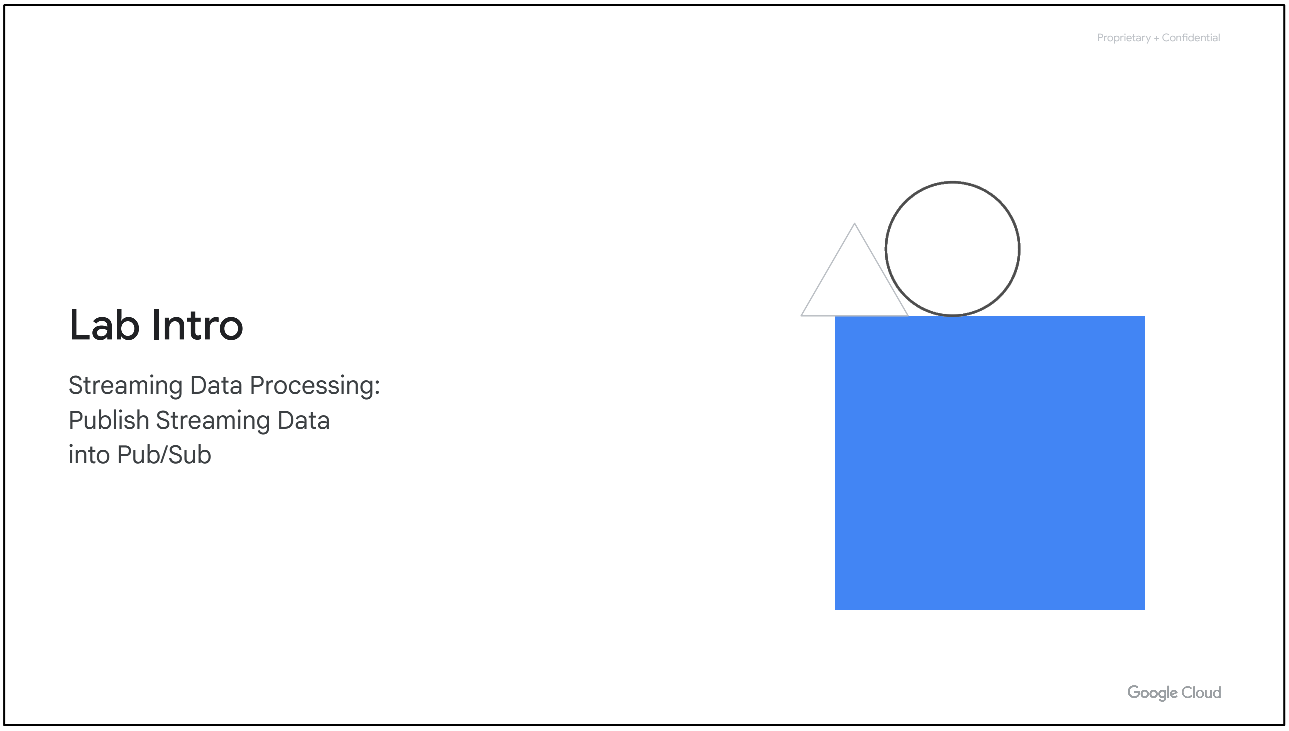
When working with Pub/Sub there are few things to keep in mind. First, data may be delivered out of order, but that doesn't mean you have to process the data that way. You could write an application that handles out of order and replicated messages. This is different from a true queuing system.

In general, they will be delivered in order but you can't rely on that with Pub/Sub. This is one of the compromises made for scalability especially since it's a global service. We have a mesh network, so a message might take another route, and if it happens to be a slower route, you could have an earlier message arriving later. For example, you wouldn't use this to implement a chat application because it would be awkward when messages arrive out of order. Therefore, we will handle ordering using other techniques.

Finally, we need to be ready for duplication.

You can use Dataflow in conjunction with Pub/Sub to solve some of the problems we just discussed. Dataflow will de-duplicate messages based on the message ID because in Pub/Sub, if a message is delivered twice, it will have the same ID in both cases.

BigQuery can also be used for this purpose but has limited capabilities. Dataflow will not be able to order in the sense of providing exact sequential order of when messages were published. However, it will help you deal with late data. Using Pub/Sub and Dataflow together allows you to get a scale that wouldn't be possible otherwise. In the next module, you’ll look at Dataflow's streaming capabilities in greater detail.



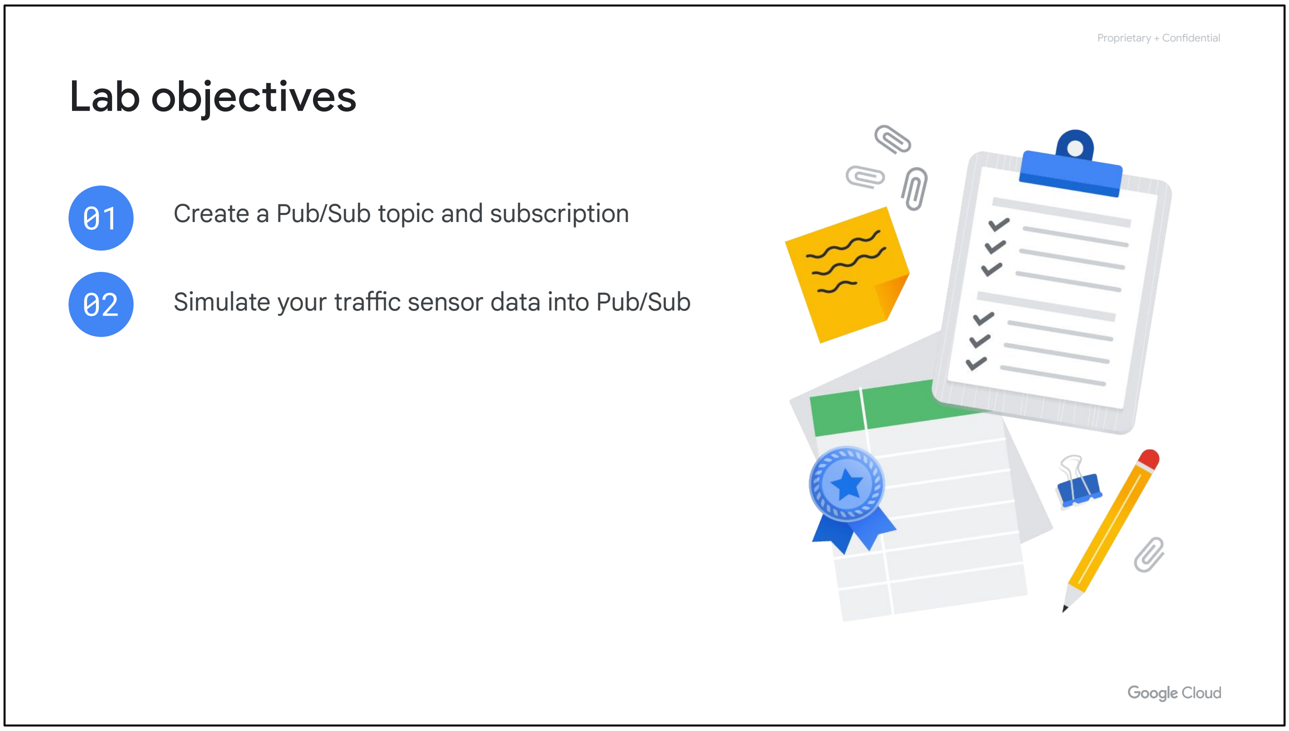
Proprietary + Confidential

Lab Intro

Streaming Data Processing:
Publish Streaming Data
into Pub/Sub

Google Cloud

Now, let's practice publishing streaming data into Pub/Sub.



Lab objectives

01 Create a Pub/Sub topic and subscription

02 Simulate your traffic sensor data into Pub/Sub

In this lab, you create a Pub/Sub topic and subscription and simulate San Diego traffic data into Pub/Sub.