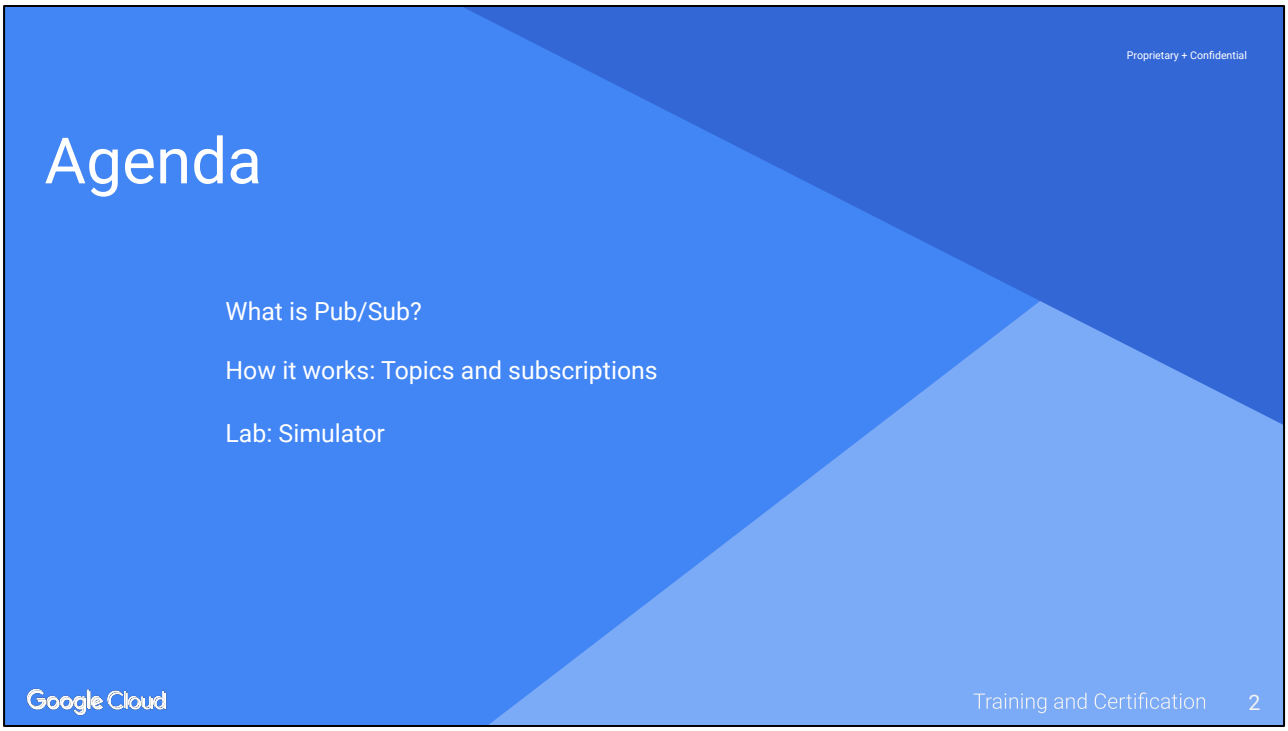
Google Cloud

Ingesting variable volumes

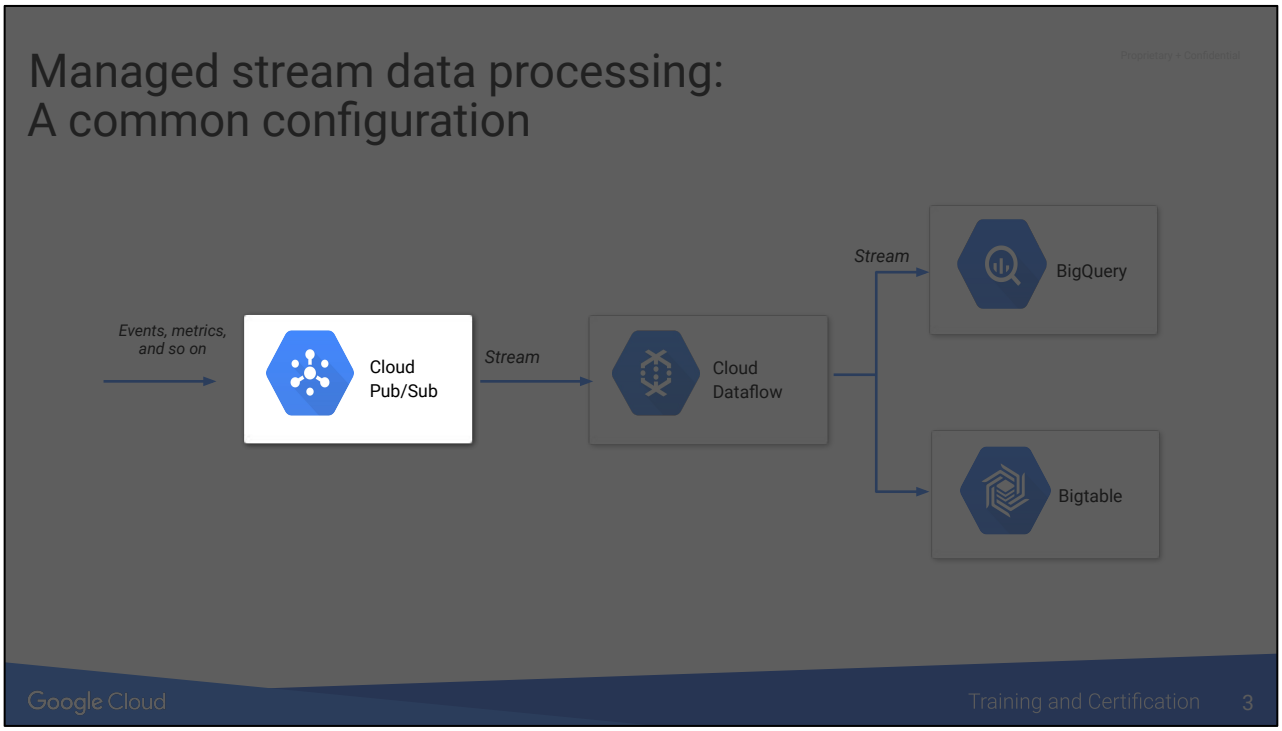
Data Engineering on Google Cloud Platform

1 hour, including lab



Notes:

First part of the pipeline is how to handle the variable volume.



Notes:

PubSub is your global message bus we will use to ingest our traffic data in the class hands on exercise.

It works very well with streaming data providing high ingest speed, durability, fault tolerance, no ops, and autoscaling.

Notes:

Discoverability: Pub/Sub handles discovery of subscribers who are interested in messages from specific publishers without having to worry about one finding the other directly.

- Moves can be handled
- Turndowns can be handled

Availability: Don't worry about whether or not subscribers are around to receive messages

- Pub/Sub absorbs the requirements of handing subscribers that can't keep up or go down

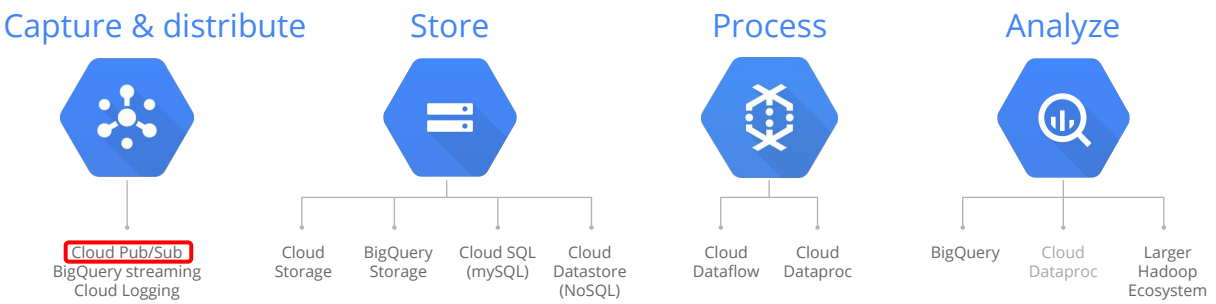
Scale points:

- Number of publishers
- Number of subscribers
- Size of messages
- Number of messages
- Throughput of messages

- Durability
  - Messages are saved to be delivered later, when subscribers are not around to receive them.

Discoverability img source:
https://pixabay.com/en/binoculars-looking-man-discovery-1209011/ (cc0)
Availability:
https://pixabay.com/en/neon-neon-sign-the-text-of-the-open-1191281/ (cc0)
Durability:
https://pixabay.com/en/aluminum-briefcase-business-case-1846345/ (cc0)
Latency: https://pixabay.com/en/clock-pocket-watch-movement-1205634/
(cc0)
Scalability image is from our training deck template

Cloud Pub/Sub connects applications and services through a messaging infrastructure



Notes:

Starts with capture….whether is transactions...or traffic data

It is serverless….global

Cloud Pub/Sub is a service used to distribute data, as messages, among applications or services. It is a single, global service, offering high, consistent throughput and latency.

Isn't the network "plumbing for applications?"

Without Pub/Sub

With Pub/Sub

WITH NETWORK ALONE, TO COMMUNICATE ALL APPLICATIONS MUST BE ONLINE + AVAILABLE ALL THE TIME.

Notes:

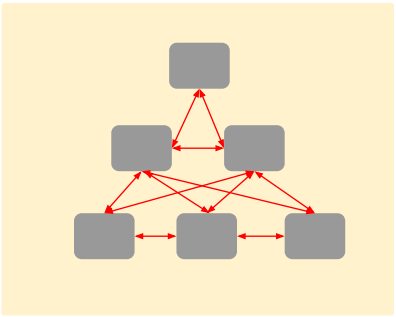
Without pub/sub: ... every components must be online at the same time to get anything done. If one is unavailable, everything grinds to a halt. It's the world of telephones.

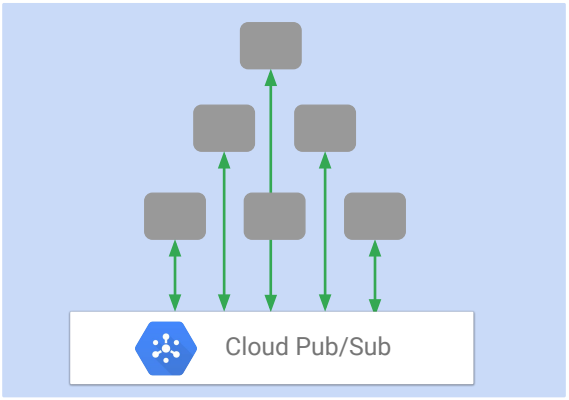
With pub/sub : .. as with email for people, no one has to be online at the same time. Pub/Sub delivers messages instantly if everyone is online & safely holds them until they can be delivered otherwise. Can deal with spikes

Telephon icon :
https://pixabay.com/en/phone-telephone-communication-160430/ (cc0)

Pub/Sub simplifies event distribution

- By replacing synchronous point-to-point connections with a single, high-availability asynchronous bus



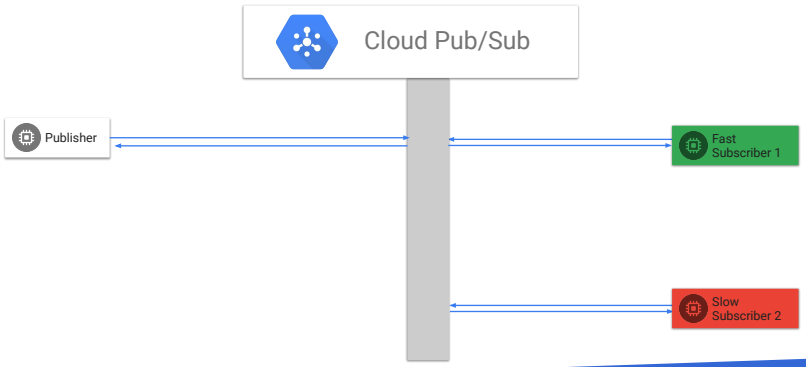


Notes:

Pub/Sub simplifies systems by removing the need for every component to speak to every component. Instead, each only has to know how to speak to the Pub/Sub system.

Aysnchronous → publisher never waits

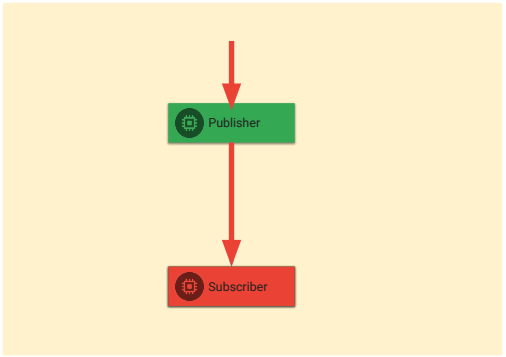
A subscriber can get the message now or anytime (within 7 days).

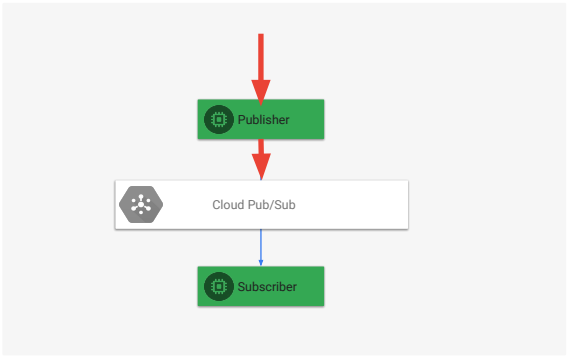


Notes:

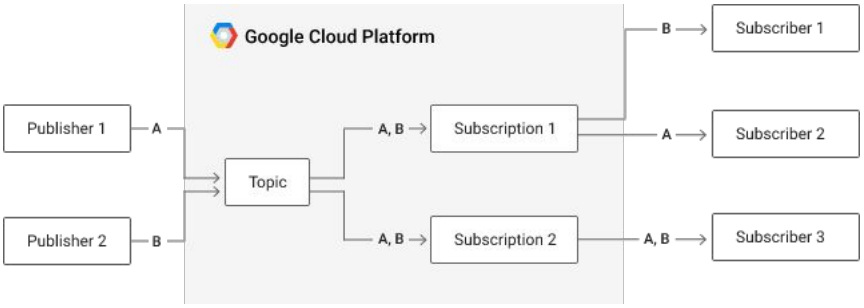
Applications are asynchronous: message are delivered instantly to online subscribers, but kept and retried for 7 days if a subscriber is down or overloaded. Question on SLA….may want to include.

Can avoid overprovisioning for spikes with Pub/Sub





How it works: Topics and subscriptions



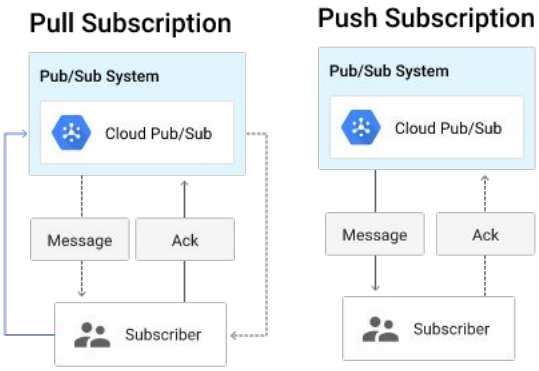
Notes:

- Pub/Sub implements the asynchronous **Publish/Subscribe** pattern:
  - **A publishers** send **messages** to a **topic**.
  - **Subscribers** receive messages in a topic by creating a subscription.
  - Each subscriber is guaranteed to get each message at least once.

- The decoupling of publishers and subscribers means neither side needs to worry about the other
- The message bus will handle everything!

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
- A subscriber can extend the deadline per message



Duplicate & out-of-order delivery is expected

Pub/Sub tracks which messages have been ACK’ed -> simpler subscriber code.

Once an ack has been received, PubSub will stop delivery for the subscription. To do the equivalent of Kafka worker groups, subscription + ack (and multiple subscribers) is the answer. PubSub attempts to deliver a message to each subscription until:

1. A subscriber acknowledges the msg, or
2. The message expires (7 days)

Once a subscriber has pulled a message, it's "out for delivery" until the ack deadline. PubSub will retry delivery to the subscription if msg hasn't been ack'd by the deadline. This ensures that only one worker in a pool receives any given msg.

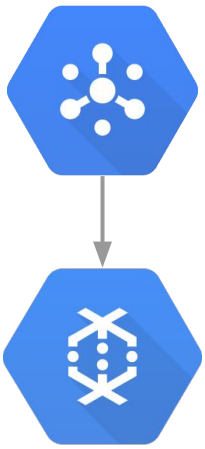
Subscribers can extend the ack deadline and tune flow control with maxOutstandingElementCount, etc.

Exactly once, ordered processing

Pub/Sub delivers at least once

Dataflow: deduplicate, order & window

Separation of concerns -> scale



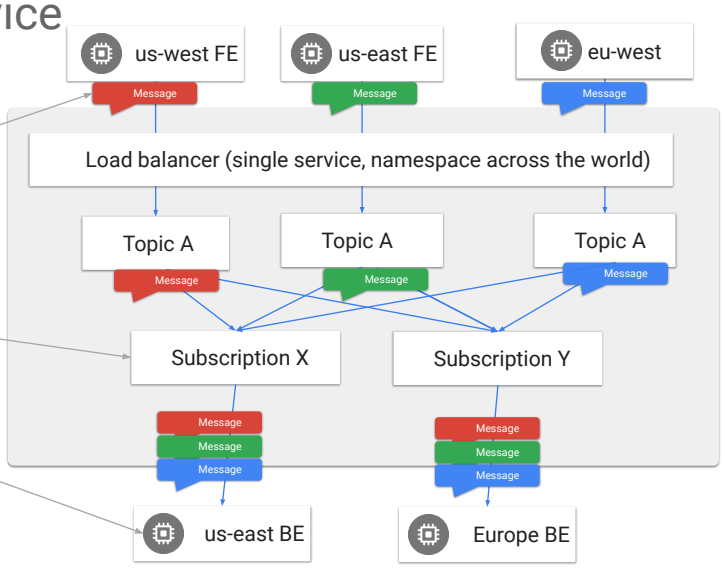
This is also how you do stateful processing of any kind in Pub/Sub too.
Pub/Sub is async and stateless.
Dataflow adds the ability to do stateful.

Pub/Sub is a global service

Messages stored in region closest to publisher (in multiple availability zones)

A Subscription collates a topic from different regions

Subscribers can be anywhere in world; no change of code.



Pub/Sub features

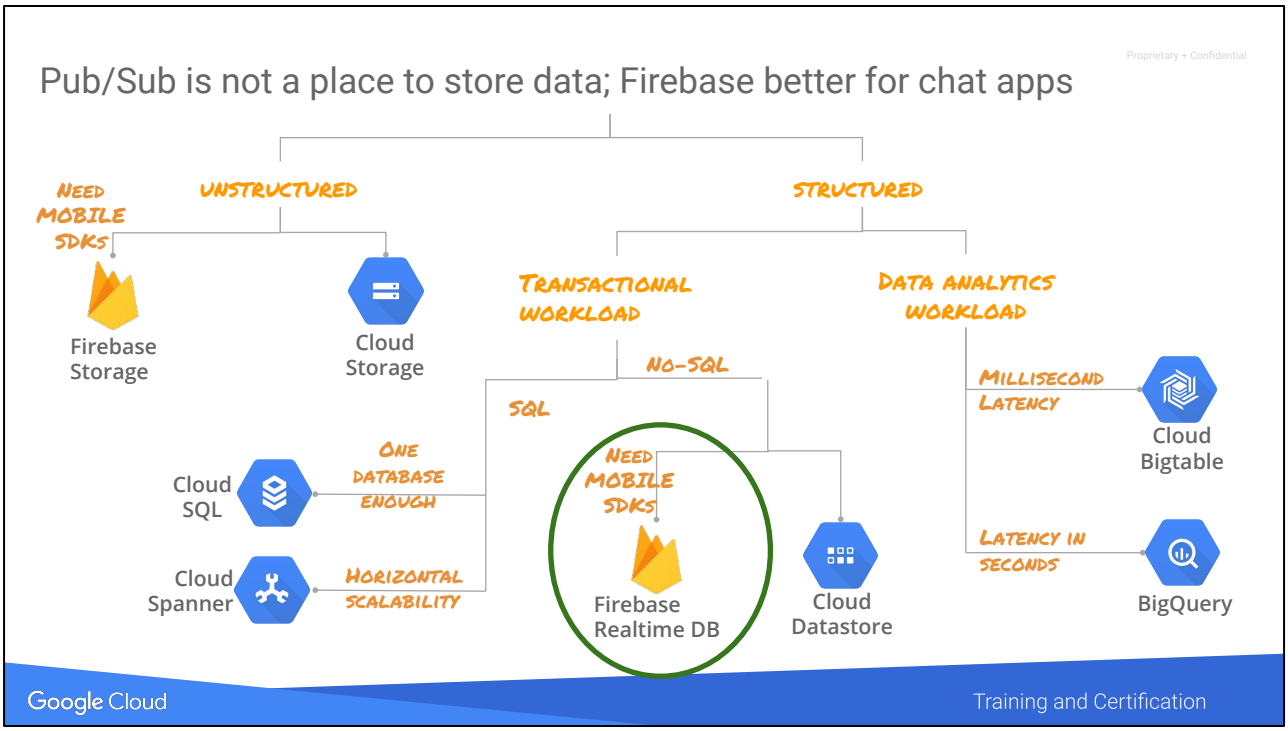
- Fast: order of 100s of milliseconds
- Fan-in, fan-out parallel consumption
- Push & pull delivery flows
- Client libraries:
  - Idiomatic, hand-built in Java, Python, C#, Ruby, PHP, Node.js
  - Auto-generated in 10 gRPC languages

Notes:

Fast: <150 ms 99.9% of publish, <1 s 99.9% end-to-end
Durable: 3x3 sync data replication before ACK, for 7 days
Secure: encryption in transit & rest, private network
Managed: round-the-clock ops team
Compliant: HIPPA compliance

Other features:

- Scalable from 1 KB/s to 100 GB/s, with consistent performance.
- Durable: 3x3 sync data replication before ACK, for 7 days
- Secure: encryption in transit & rest, private network
- Managed: round-the-clock ops team



Pub/Sub does not appear in this diagram. Why? Because it is not a database. It is for data ingest. Use Pub/Sub to ingest the data and stream it into a data warehouse/cloud-storage/bigquery etc.

Firebase is also the better option if your needs involve real-time person-person communication, games, chats, activity streams, etc.

Agenda

How it works: Topics and subscriptions

Create topic and publish message

```
gcloud pubsub topics create sandiego
```

```
gcloud pubsub topics publish sandiego "hello"
```

Python code

```
from google.cloud import pubsub
publisher = pubsub.PublisherClient()

event_type = publisher.topic_path(args.project,"sandiego")
publisher.create_topic(event_type)
publisher.publish("sandiego",b'hello')
```


Notes:

Realize that gcloud is simply making a REST API call. So, you can make that REST API call from any language … not just Python.

https://googlecloudplatform.github.io/google-cloud-python/latest/pubsub/

Other publish options

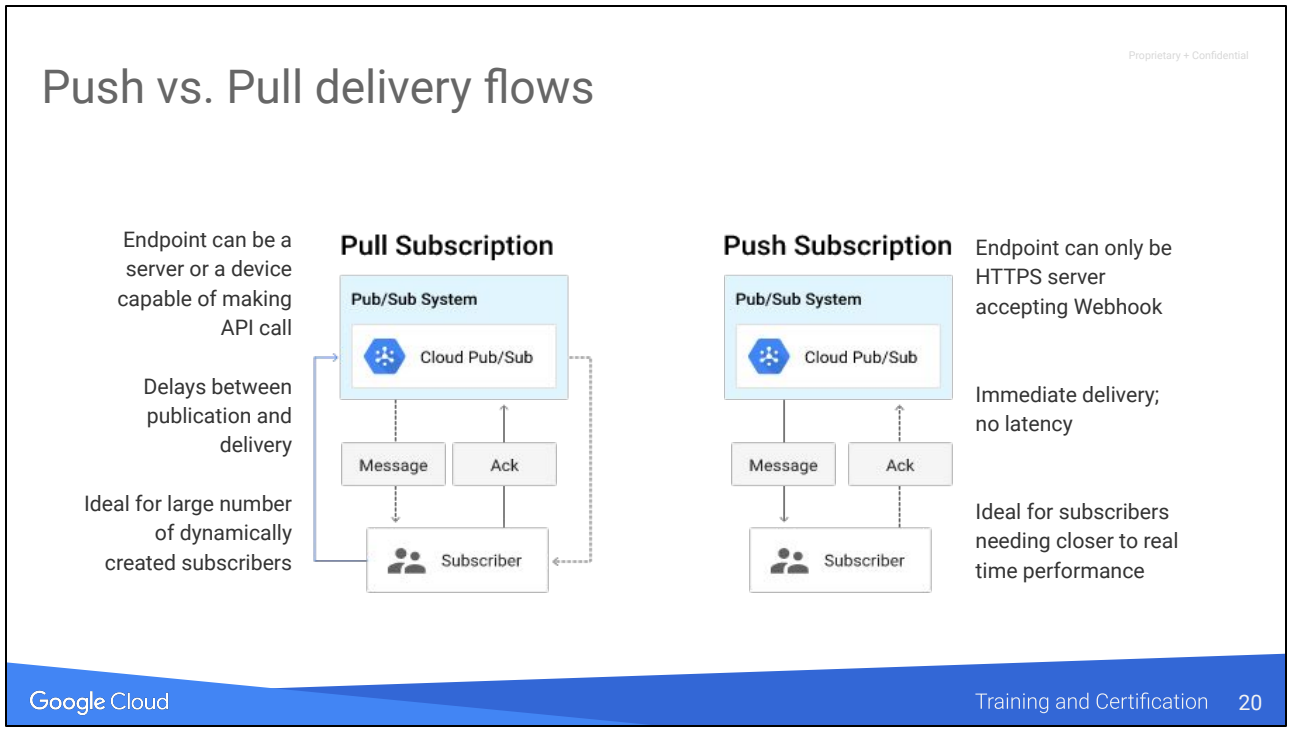
Python code

```
TOPIC = 'sandiego'
publisher.publish(TOPIC,b'This is the message payload')

#Publish a single message to a topic, with attributes:
publisher.publish(b'Another message payload', extra='EXTRA')

#Publish a set of messages to a topic (as a single request):
with publisher.batch() as batch:
        batch.publish(PAYLOAD1)
        batch.publish(PAYLOAD2, extra=EXTRA)
```


Notes:

If you want to include attributes, simply add keyword arguments, such as extra='EXTRA' in the example.

Messages only after the subscription is created. Earlier messages are lost. Doing batch is more cost effective….mention the san diego example.



Notes:

Ack is just http ack. For push...there is exponential backoff.....u can control retry

In push delivery, Pub/Sub initiates requests to your subscriber application to deliver messages. In pull delivery, your subscriber application initiates requests to the Pub/Sub server to retrieve messages.

In a push subscription, the Pub/Sub server sends a request to the subscriber application, at a preconfigured endpoint. The subscriber's HTTP response serves as an implicit acknowledgement: a success response indicates that the message has been successfully processed and the Pub/Sub system can delete it from the subscription; a non-success response indicates that the Pub/Sub server should resend it (implicit "nack"). To ensure that subscribers can handle the message flow, Pub/Sub dynamically adjusts the flow of requests and uses an algorithm to rate-limit retries.

In a pull subscription, the subscribing application explicitly calls the API `pull` method, which requests delivery of a message in the subscription queue. The Pub/Sub server responds with the message (or an error if the queue is empty), and an *ack ID*. The subscriber then explicitly calls the `acknowledge` method,

using the returned ack ID, to acknowledge receipt.

Create subscription, pull messages

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

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

Python code

```
subscription = topic.subscription(subscription_name)
subscription.create()

results = subscription.pull(return_immediately=True)

if results:
 subscription.acknowledge([ack_id for ack_id, message in
results])
```


Notes:

Namespace is within your project...not global namespace

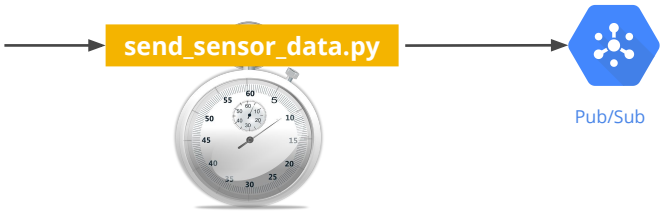
Messages only after the subscription is created. Earlier messages are lost.

Realize that gcloud is simply making a REST API call. So, you can make that REST API call from any language … not just Python

```
# Change return_immediately=False to block until messages are received.
```

Lab: Simulator

Lab 1: Publish Streaming Data into Pub/Sub



Notes:


cloud.google.com
Image by Connie Zhou