Course: Securing and Integrating Components of your Application

Module 2: Using Pub/Sub to integrate components of your application

**Why Pub/Sub?**

* A fully managed messaging architecture
  + can be used to build loosely coupled micro services
    - services communicate asynchronously
  + resilient
  + distributed - the application components can sit on multiple different Clouds, or on hybrid architecture
  + can send and receive PubSub messages using client libraries for different programming languages, using open-source REST HTTP, gRPC service APIs and Apache kafka connector.
  + Some use cases:
    - real-time gaming
    - stream of click data to be ingested and processed
    - IoT sensor and device data processing for healthcare, auto and manufacturing
    - financial data
  + Characteristics of use cases:
    - can fan out messages to multiple subscribers
    - can ingest large volume of data
  + A key component of GCP data streaming pipeline
    - delivers data to Cloud Dataflow and BigQuery for data processing and analytics.
  + Concepts: topics, publishers, subscribers, push and pull subscriptions
  + Scales automatically depending on volume of messages



* ingest -> transform stream data -> save to analytical database -> get business insight and activate processes that enhance user experience
* e.g. users (client) play song -> music streaming service ->

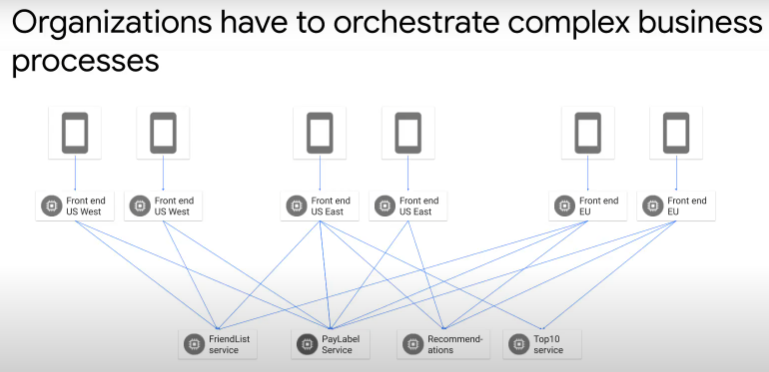
pays record company

live update to catelog, e.g. download counts

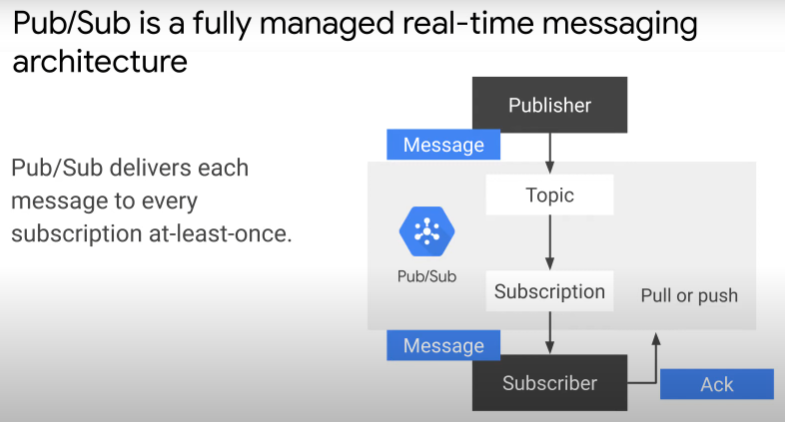
update song recommendation

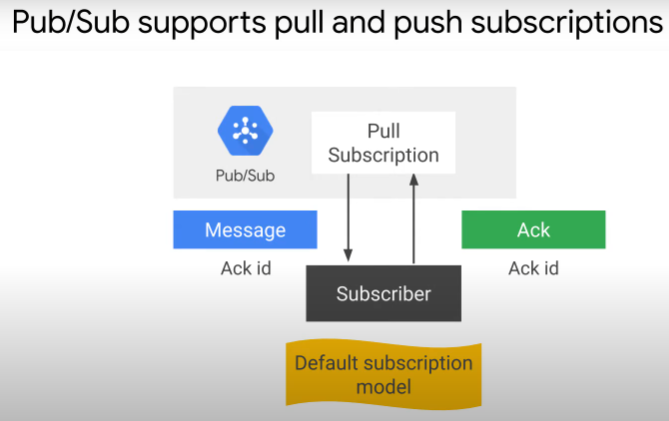
add interaction events

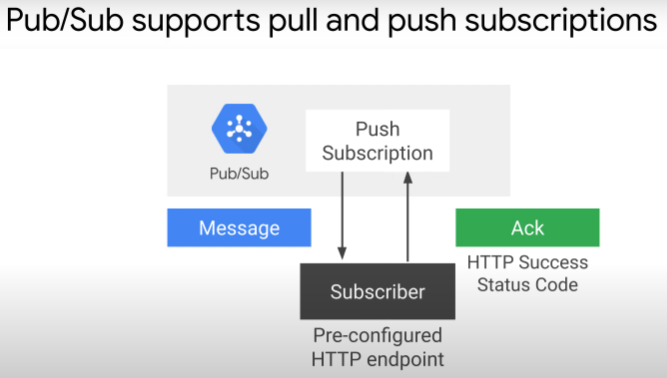
perform analytics on user actions

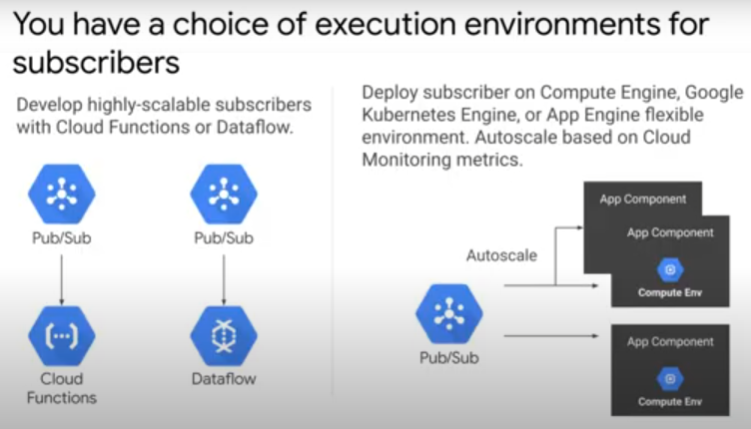


**Pub/Sub Concepts**

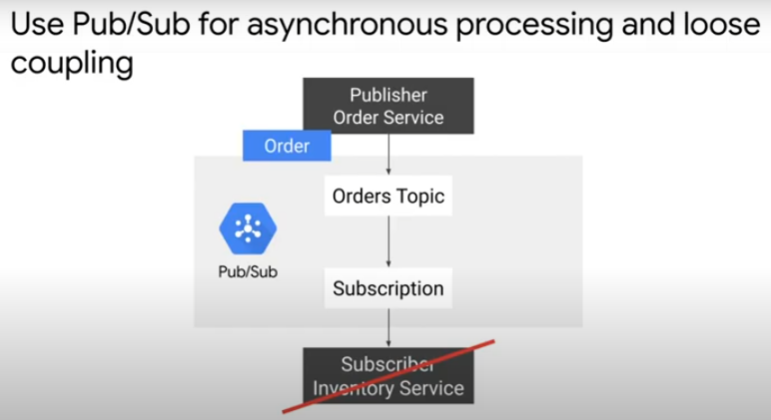


* publisher: application that creates and publishes messages to a topic
* subscriber: application that creates subscription to topic
  + - * + only receives messages published after subscription created
        + after a message is received, subscriber sends an acknowledgement back to Pub/Sub service
        + Pub/Sub removes acknowledged messages from subscription queue
        + if subscriber does not acknowledge a message before acknowledgement dateline, Pub/Sub will resend
        + Pub/Sub delivers every message at least once
* subscription can use push or pull method for message delivery
  + pull method: default
  + subscriber explicitly calls the pull method to request for delivery
    - Pub/Sub responds with message and acknowledgement ID
    - upon receipt, subscriber invokes acknowledged method using acknowledgement ID
    - subscriber uses cloud-client libraries to retrieve client messages
    - subscriber could be Dataflow
    - because subscriber initiates pull, it controls rate of message delivery
    - subscriber can modify acknowledgment dateline to allow more time to process messages
    - to process messages rapidly (large volume with high throughput)
      1. scale up the number of instances of subscribers. => Multiple subscribers pulling from same subscription. Massive parallel consumption.
      2. Batch delivery and acknowledgement.
* Benefits:
  + loose coupling between application components
  + a buffer to handle spikes in data volume
  + push subscription method
    - Pub/Sub sends each message as an HTTP request to a subscriber
    - HTTP endpoint pre-configured. Can be load balancer, App Engine standard application ..
    - Can configure a default acknowledgement dateline. If subscriber does not acknowledge before dateline, Pub/Sub will retry sending of message.
    - Subscriber endpoint acknowledges message by responding with HTTP success status code (200)
      * an error response code (e.g. 404) indicates that message must be resent
      * Pub/Sub dynamically adjusts rate of push requests based on rate at which it receives success HTTP responses.
    - subscriber does not need to implement Google Cloud library methods to request for messages
    - use cases:
    1. for situations where Google Cloud dependencies, such as credentials and client libraries, cannot be configured.
    2. Multiple topics are processed by same webhook.
    3. When an HTTP endpoint is common and will be invoked by Pub/Sub and some other applications. No cloud platform specific codes to pull messages. You have a choice of execution environment for subscribers.
       - subscriber codes can be deployed as Cloud Functions. Cloud Functions triggered when a new message arrive.
         * serverless approach
         * to build highly scalable systems
       - subscriber codes can be deployed on a compute environment (Compute Engine, GKE, App Engine flexible environment)
         * ↑ to multiple instances to split workload
         * automatically scale/shut down when few messages arrive
       - no need to handle scaling or concurrency in code

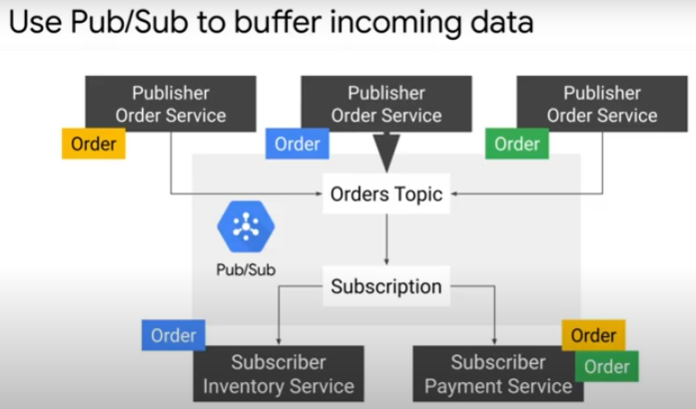




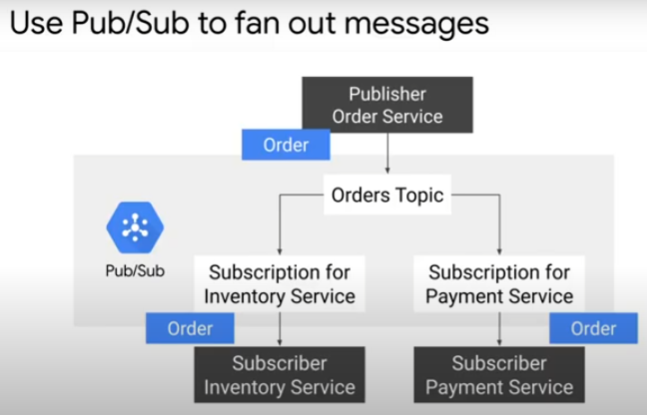
**Pub/Sub Use cases**



* if one component of application directly invoke another component, caller must wait for entire operation to finish. => synchronous call. Tightly coupled as components running together at same time.
* Pub/Sub allows asynchronous operations, loosely coupled services
* Example:
  + Publisher: Order Service
  + Message: order information
    - publisher creates and publishes message to Orders Topic and returns immediately
  + Subscriber: Inventory Service
    - subscription to Orders Topic
    - does not run in sync with Publisher
    - may be late, may even be down



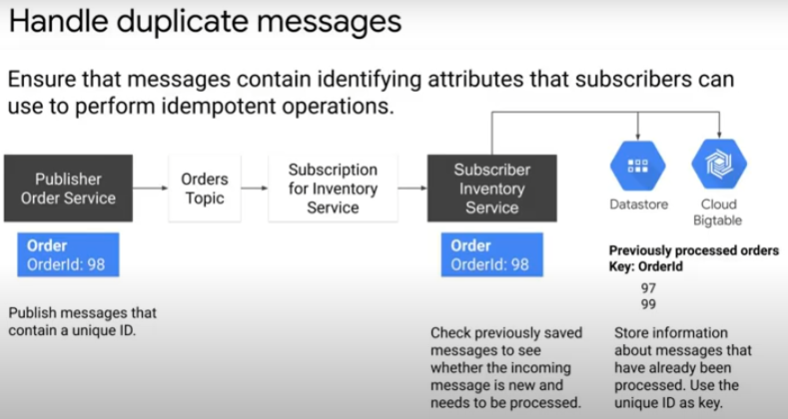
* + Topic: buffer to hold data that is coming in rapidly
    - multiple instances of Publisher are publishing unique messages
    - can scale up number of instances of Subscriber when volume of messages increase. Can scale down too.



Use Pub/Sub to fan out to multiple subscribers providing different services:

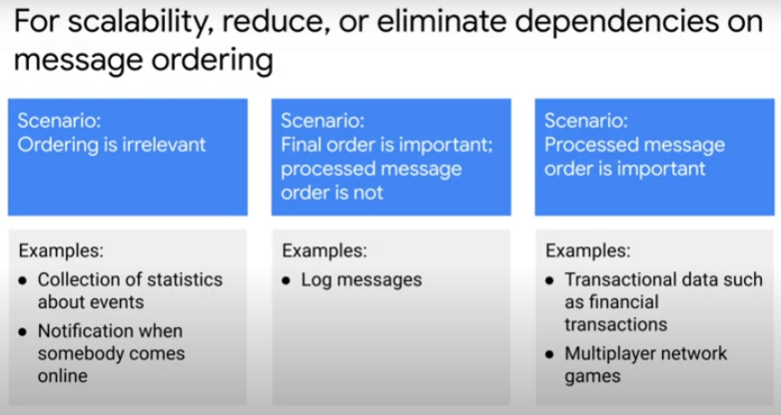
* Order Service generating large volume of data rapidly: Publisher
* publish messages to central Pub/Sub Topic where they are stored
* multiple different downstream services, e.g. Inventory Service and Payment Service, subscribe to Topic
* Instead of coding direct connection to every service that needs a piece of info (brittle, need to review publisher and subscriber codes if add subscriber, update codes).

How to handle duplicate messages?



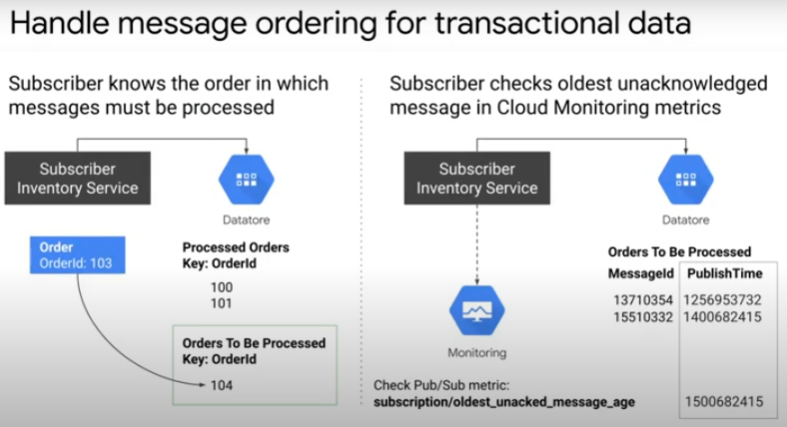
* Pub/Sub delivers each message at least once
  + So, subscriber may see duplicate messages
* Need to implement publisher and subscriber such that application can handle idempotent operations
  + idempotent – same effect, no additional effect if operation performed more than once with same input parameters
* Example: pusblish each message with unique ID
  + subscriber use Firestore, Datastore or BigTable to store info on messages already processed
  + use unique ID as key
  + subscriber checks its database to see if message processed
  + if processed, subscriber then discard message

Reduce or eliminate dependencies on message ordering for scalability



* message ordering not guaranteed for Pub/Sub
* Situation 1: message ordering not important for some use cases
  + e,g,when collecting statistics for some events
    - when counting number of participants online around the same time
* Situation 2: order of processing not important, but final order of messages important
  + e.g. log events with time stamps may stream in to logging service from various sources. Each event may be processed on a first-come-first-served basis, but final log will be ordered by time stamp.
* Situation 3: Messages must be processed in order they are published
  + e.g. financial transactions
  + e.g. event sequence in a multi-player game

How to handle message ordering



* First method:
  + Publish message with unique ID
  + Subscriber has logic for messsages to be processed in order
  + Subscriber stores info on messages using Datastore
    - processed queue
    - pending queue
  + When new message received, Subscriber checks if message to be processed immediately, discarded or there are other pending messages to be processed first
  + e.g. message with Order ID 103 received. Checks processed order queue and determines that 102 is pending. Stores 103 in pending queue behind 104. Process 103 and 104 after 102 arrives.
* Second method:
  + Subscriber stores all messages in a persistent data store (Datastore)
  + Subscriber compares timestamp of oldest unacknowledged message (means not received yet) in Cloud Monitoring/StackDriver.
  + Any message before that timestamp must be acknowledged (i.e. received by Subscriber)
    - These can be removed from Datastore for processing