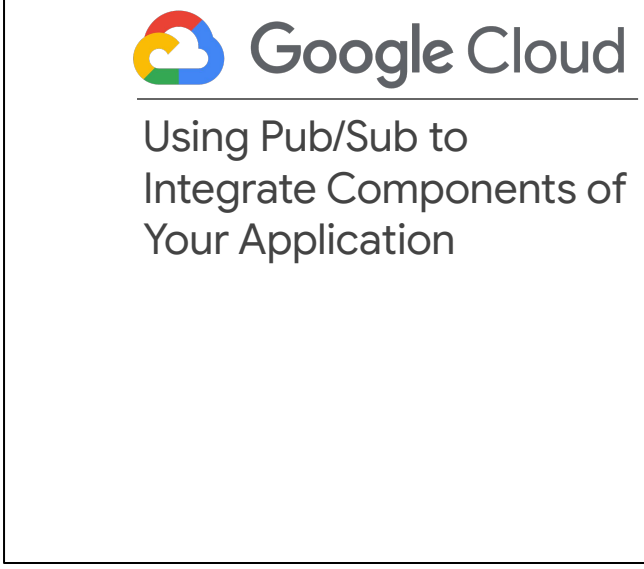


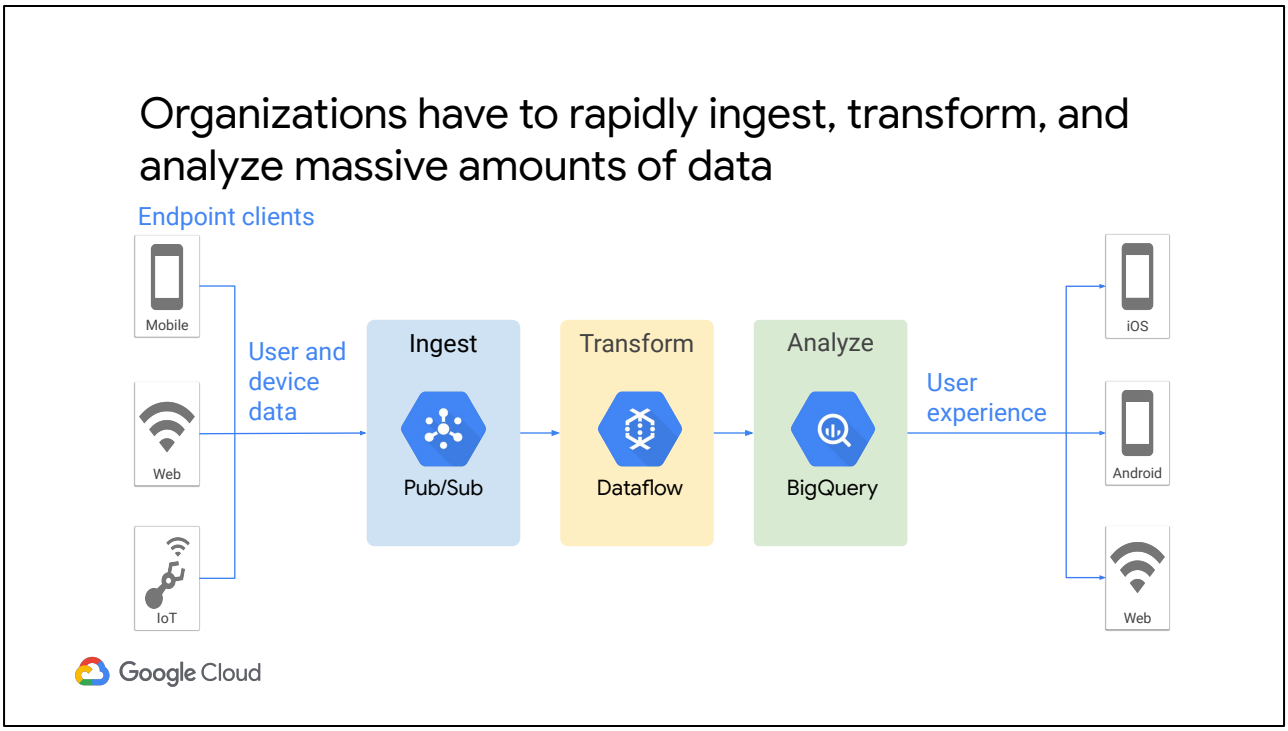
Pub/Sub is a fully managed messaging architecture that enables you to build loosely coupled microservices that can communicate asynchronously. You can use Pub/Sub to integrate components of your application. Pub/Sub also enables you to build your application on open multi-cloud or hybrid architectures. You can send and receive Pub/Sub messages using client libraries for popular programming languages, open REST/HTTP and gRPC service APIs, and an open source Apache Kafka connector.

Pub/Sub is ideal for use cases such as realtime gaming applications, clickstream data ingestion and processing, device or sensor data processing for healthcare and manufacturing, and integrating various data sources in financial applications.

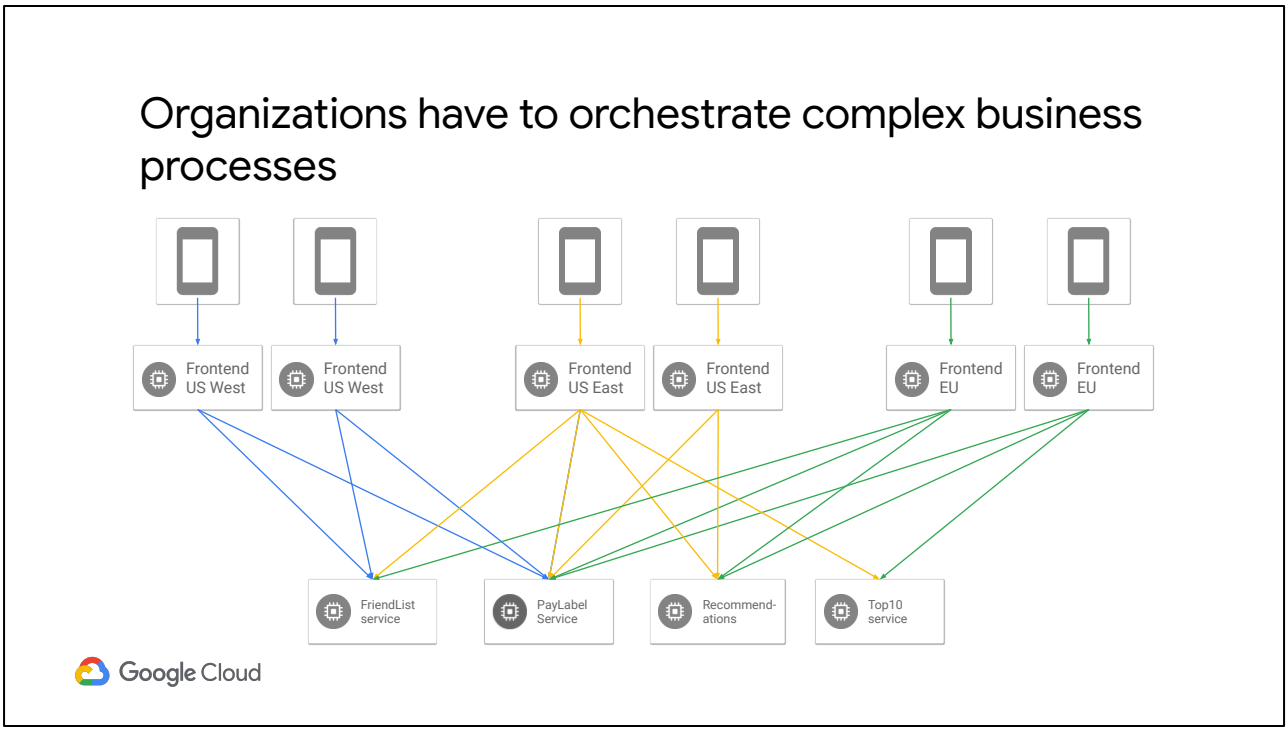
Pub/Sub is a key component of Google Cloud's streaming data pipeline. Pub/Sub can rapidly ingest large amounts of data and deliver the data reliably to Dataflow and BigQuery for data processing and analytics.

Pub/Sub scales automatically depending on the volume of messages and enables you to securely integrate distributed systems.

In the module, Using Pub/Sub to Integrate Components of Your Application, you will learn and apply concepts about Pub/Sub topics, publishers, subscribers, and push and pull subscriptions. You will learn the ideal use cases for Pub/Sub. For example, use Pub/Sub to build loosely coupled applications, fan-out messages to multiple subscribers, or to rapidly ingest large volumes of data.



Across industry verticals, a common scenario is that organizations have to rapidly ingest, transform, and analyze massive amounts of data. For example, a gaming application might receive and process user engagement and clickstream data. In the shipping industry, IoT applications might receive large amounts of sensor data from hundreds of sensors. Data processing applications transform the ingested data and save it in an analytics database. You can then analyze the data to provide business insights and create innovative user experiences.

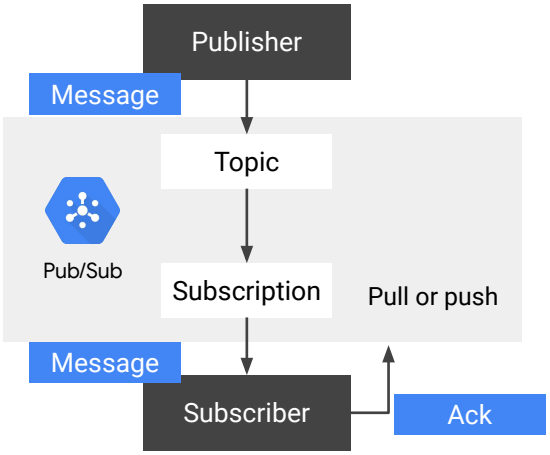


Organizations often have complex business processes that require many applications to interact with each other. For example, when a user plays a song, a music streaming service must perform many operations in the background. There might be operations to pay the record company, perform live updates to the catalog, update song recommendations, handle ad interaction events, and perform analytics on user actions. Such complex application interactions are difficult to manage with brittle point-to-point application connections.

Pub/Sub enables you to scalably and reliably ingest large volumes of data. It also enables you to design loosely coupled microservices that can streamline application interactions.

Pub/Sub is a fully managed real-time messaging architecture

Pub/Sub delivers each message to every subscription at-least-once.



Google Cloud

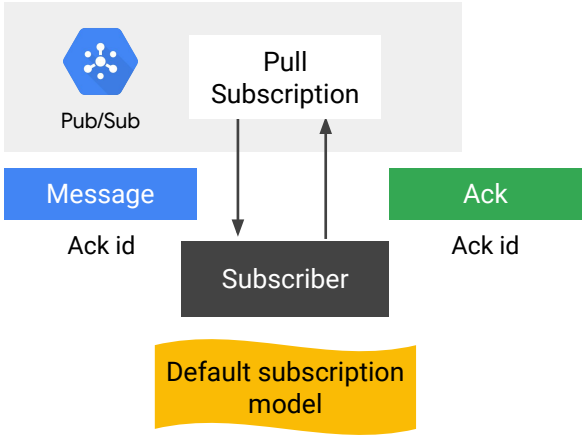
Let's take a look at some Pub/Sub concepts.

Pub/Sub is a fully managed real-time messaging architecture.

An application that publishes messages is called a publisher. A publisher creates and publishes messages to a topic. To receive messages, a subscriber application creates a subscription to a topic. A subscription can use either the push or pull method for message delivery. The subscriber only receives messages that are published after the subscription was created. After receiving and processing each pending message, the subscriber sends an acknowledgement back to the Pub/Sub service. Pub/Sub removes acknowledged messages from the subscription's queue of messages. If the subscriber doesn't acknowledge a message before the acknowledgement deadline, Pub/Sub will resend the message to the subscriber. Pub/Sub delivers each message to every subscription at least once.

Pub/Sub enables loosely coupled integration between application components, acts as a buffer to handle spikes in data volume, and also supports other use cases. Pub/Sub supports pull and push subscriptions.

Pub/Sub supports pull and push subscriptions



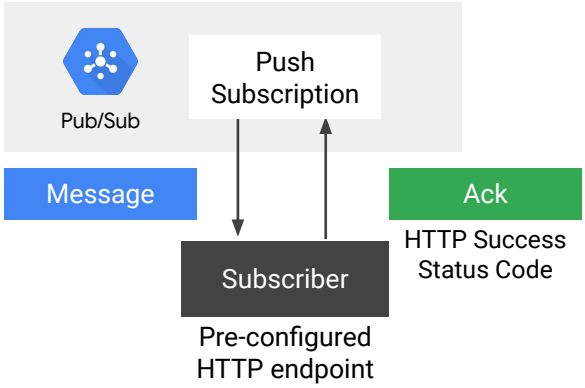
Google Cloud

In a **pull subscription**, the subscriber explicitly calls the pull method to request messages for delivery. Pub/Sub returns a message and an acknowledgement I.D. To acknowledge receipt, the subscriber invokes the acknowledged method by using the acknowledgement I.D. This is the default subscription model.

In a pull subscription model, the subscriber can be Dataflow or any application that uses Cloud Client Libraries to retrieve messages. The subscriber controls the rate of delivery. A subscriber can modify the acknowledgement deadline to allow more time to process messages. To process messages rapidly, multiple subscribers can pull from the same subscription. The pull subscription model enables batch delivery and acknowledgements as well as massively parallel consumption.

Use the pull subscription model when you need to process a very large volume of messages with high throughput.

Pub/Sub supports pull and push subscriptions



Google Cloud

A **push subscriber** doesn't need to implement Google Cloud client library methods to retrieve and process messages.

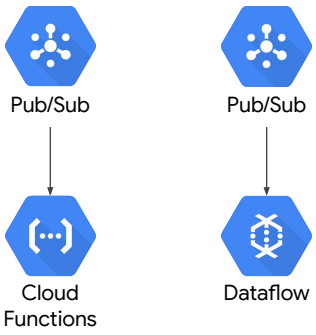
In a push subscription model, Pub/Sub sends each message as an HTTP request to the subscriber at a pre-configured HTTP endpoint. The push endpoint can be a load balancer or App Engine standard application. The endpoint acknowledges the message by returning an HTTP success status code. A failure response indicates that the message should be sent again. Pub/Sub dynamically adjusts the rate of push requests based on the rate at which it receives success responses. You can configure a default acknowledgement deadline for push subscriptions. If your code doesn't acknowledge the message before the deadline, Pub/Sub will resend the message.

Use the push subscription model in environments where Google Cloud dependencies, such as credentials and the client library, can't be configured, or multiple topics must be processed by the same webhook. A push subscription is also ideal when an HTTP endpoint will be invoked by Pub/Sub and other applications. In this scenario, the push subscriber should be able to process message payloads from Pub/Sub and other callers. You have a choice of execution environments for subscribers.

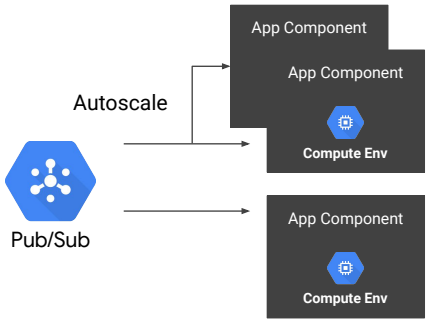
For more information about pull and push subscriptions, see https://cloud.google.com/pubsub/docs/subscriber#push_pull.

You have a choice of execution environments for subscribers

Develop highly-scalable subscribers with Cloud Functions or Dataflow.



Deploy subscriber on Compute Engine, Google Kubernetes Engine, or App Engine flexible environment. Autoscale based on Cloud Monitoring metrics.



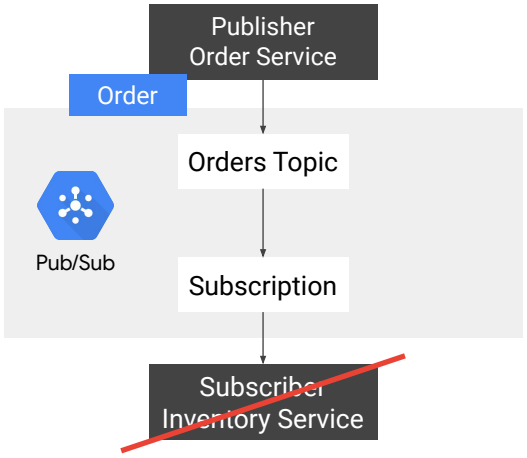
Google Cloud

You can deploy your subscriber code as Cloud Functions. Your Cloud Functions will be triggered whenever a new message is received. This method enables you to implement a serverless approach and build highly scalable systems.

Alternatively, you can deploy your subscriber application on a compute environment such as Compute Engine, Google Kubernetes Engine (GKE), or App Engine flexible environment. Multiple instances of your application can spin up and split the workload, by processing the messages and the topic. These instances can be automatically shut down when there are very few messages to process. You can enable elastic scaling using Pub/Sub metrics that are published to Cloud Monitoring.

With either approach, you don't have to worry about developing code to manage concurrency or scaling. Your application scales automatically depending on the workload.

Use Pub/Sub for asynchronous processing and loose coupling



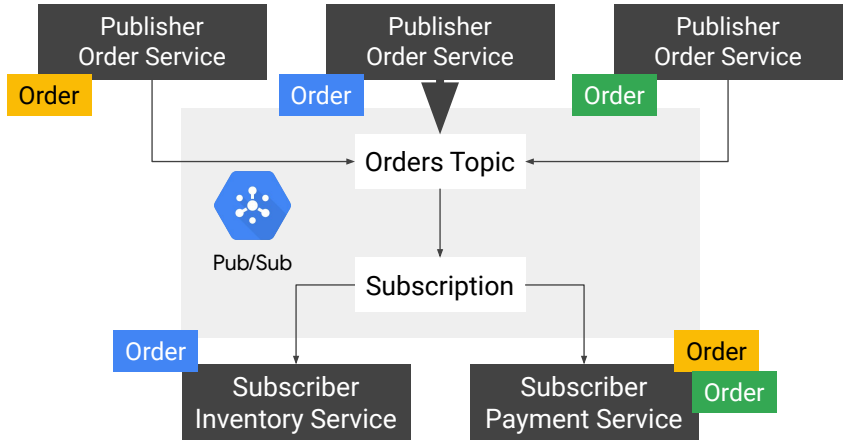
Google Cloud

Let's dive into the use cases for Pub/Sub.

Pub/Sub enables your application to perform operations asynchronously. Traditionally, one service or component of your application has to directly invoke another component to perform an operation. The call is synchronous, so the caller has to wait for the entire operation to finish. The application components are tightly coupled, because both must be running at the same time.

With Pub/Sub, you can design your services to make asynchronous calls and be loosely coupled. For example, in the diagram, the Order service acts as the publisher. It publishes messages that contain order information to the orders topic and returns immediately. The Inventory service acts as the subscriber. It might even be down when the publisher sends messages. The subscriber can start running a little later and process the messages it has subscribed to.

Use Pub/Sub to buffer incoming data



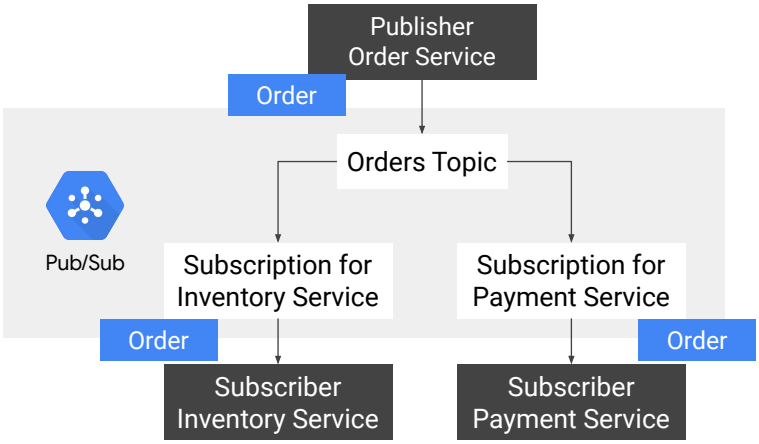
Google Cloud

Pub/Sub enables you to use a topic as a buffer to hold data that is coming in rapidly. For example, you might have multiple instances of a service generating large volumes of data. A downstream service that needs to process this data might become overwhelmed with the high velocity and volume of data.

To address this issue, instances of your data generating service can act as publishers and publish the data to a Pub/Sub topic. Pub/Sub can reliably receive and store large volumes of rapidly incoming data. The downstream service can then act as a subscriber and consume the data at a reasonable pace. Your application can even automatically scale up the number of instances of the subscriber to handle increased volumes of data.

The diagram shows many messages with order information coming in from various sources. The inventory service can subscribe to these messages and process them at a reasonable pace as appropriate.

Use Pub/Sub to fan out messages



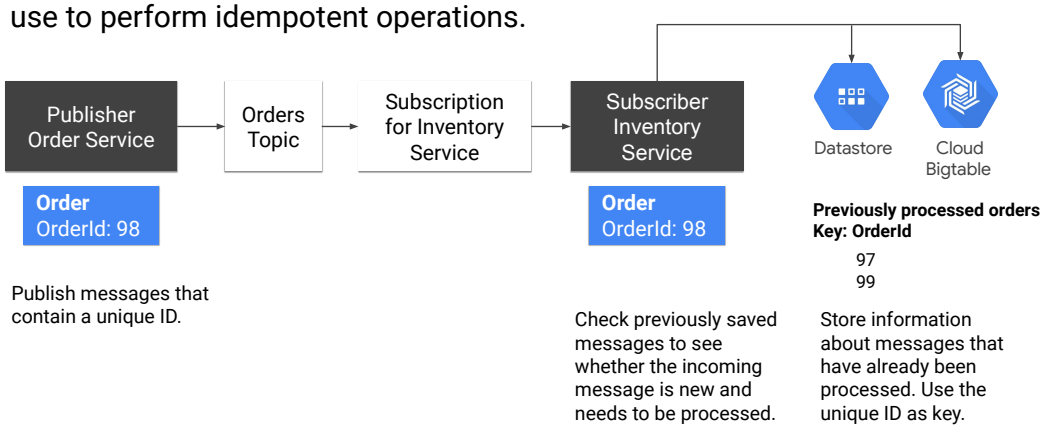
Google Cloud

With Pub/Sub, your application can fan out messages from one publisher to multiple subscribers. Instead of being aware of all services that might be interested in a particular piece of data and making brittle point-to-point connections with all the services, the publisher can push messages to a centralized Pub/Sub topic. Services that are interested in the information can simply subscribe to the topic.

For example, in the diagram, information about orders is fanned out to the inventory and payment services.

Handle duplicate messages

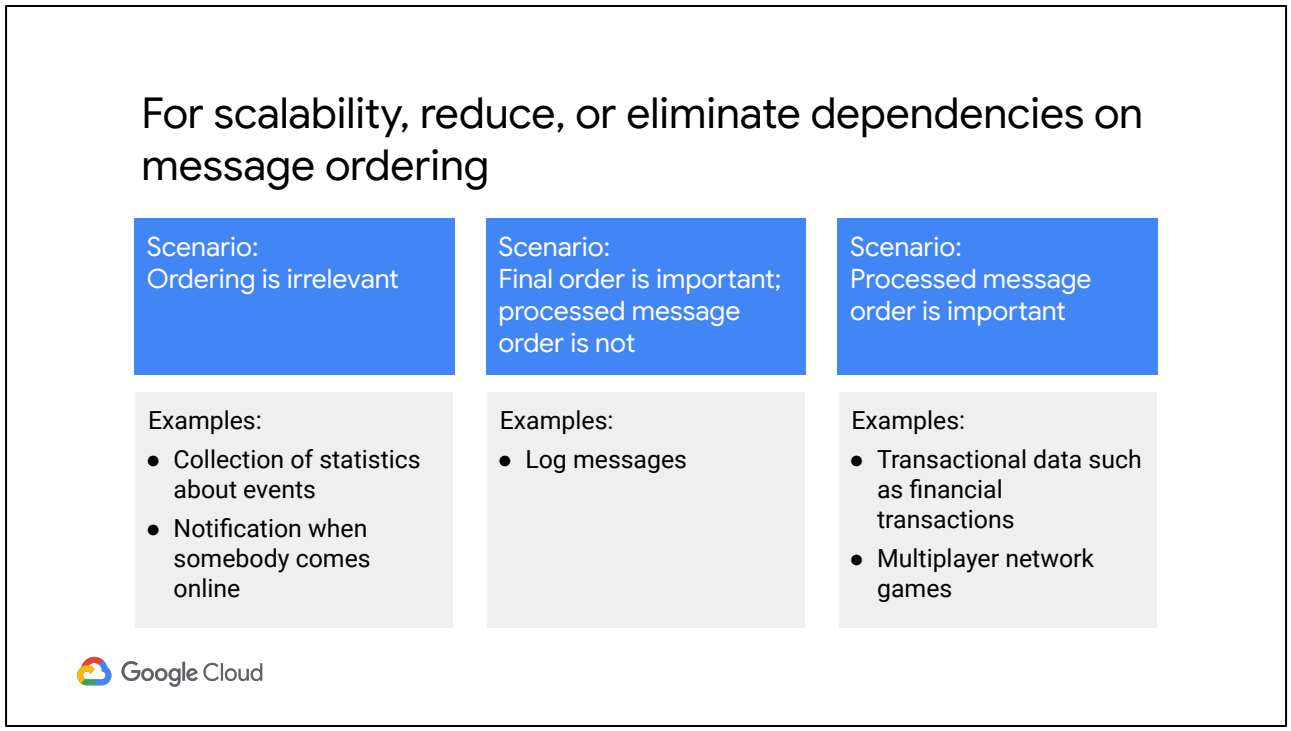
Ensure that messages contain identifying attributes that subscribers can use to perform idempotent operations.



Google Cloud

Pub/Sub delivers each message at least once. This means that a subscriber can sometimes see duplicate messages. Implement your publisher and subscriber in such a way that the application can perform idempotent operations.

The publisher can publish messages with a unique I.D. Your subscriber can use Datastore, Firestore, or Cloud Bigtable to store information about messages that have already been processed. Use the unique I.D. as key. Whenever a message is received, the subscriber can check previously saved messages to see whether the incoming message is new and needs to be processed. If the message has already been processed, the subscriber can just discard the duplicate message.



Pub/Sub provides a highly scalable messaging architecture. The scalability comes with a trade-off: message ordering is not guaranteed. Where possible, design your application to reduce or even eliminate dependencies on message ordering.

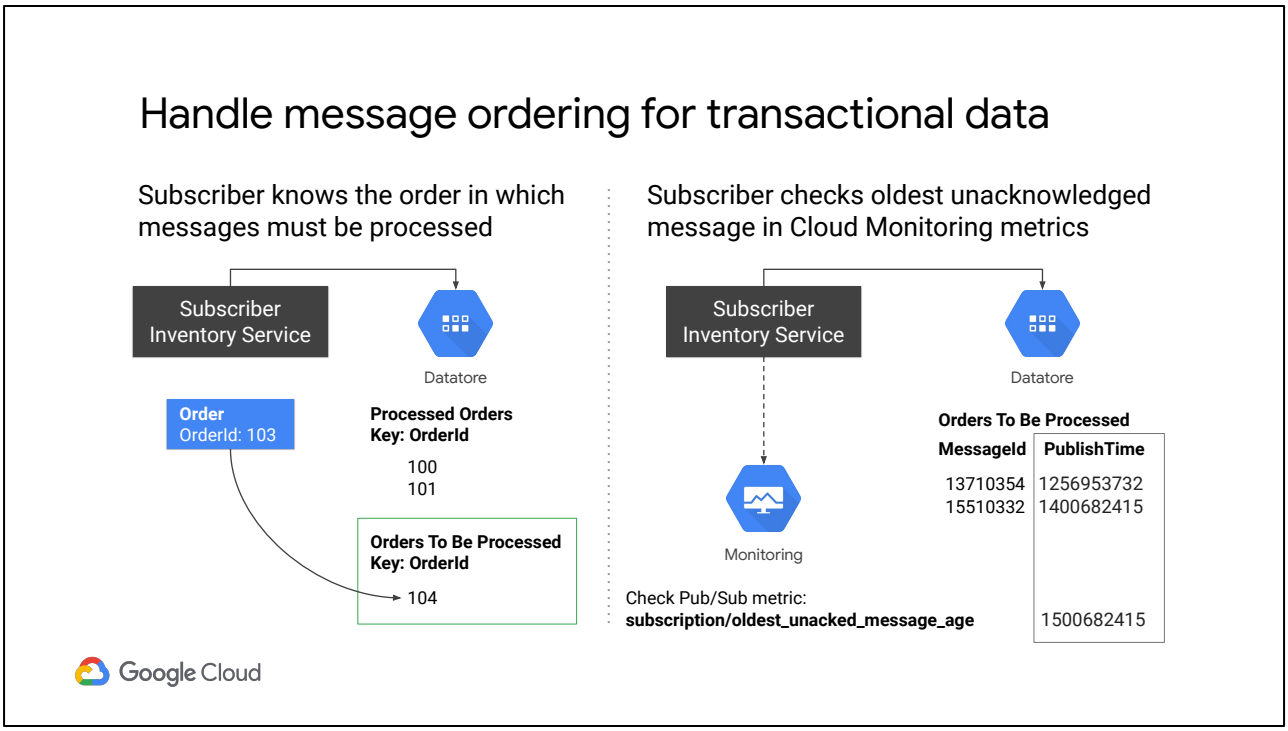
For example, message ordering is not relevant in use cases where your application is collecting statistics about events. You don't need to process the statistics for related events in order. Similarly, if, say, five of your contacts come online around the same moment in time, you don't need to be notified about their presence in the exact same order that they came online.

Then there are scenarios in which the final order of messages is important, but the order in which the individual messages are processed is not important. For example, log events with timestamps may stream into the logging service from various sources. The order of processing each log event is not important. However, eventually, your system should enable you to view log events ordered by timestamp.

Pub/Sub works well naturally for use cases where order of message processing is not important.

Finally, there are scenarios where messages must absolutely be processed in the order in which they were published. For example, financial transactions must be

processed in order. Similarly, in a multiplayer online game, if a player jumps over a wall, lands on a monster, and recovers the lost jewel, other players must see the events in exactly the same order.



You can implement ordered processing of messages in one of the following ways.

In the first scenario, the subscriber knows the order in which messages must be processed. The publisher can publish messages with a unique I.D. Your subscriber can use Cloud Datastore to store information about messages that have been processed so far. When a new message is received, the subscriber can check whether this message should be processed immediately or the unique I.D. indicates that there are pending messages that should be processed first. For example, the inventory service knows that orders should be processed in sequence using the order I.D. as key. When messages with order I.D.s 103 and 104 are received, it checks the previously processed orders and determines that 102 is pending. It temporarily stores orders 103 and 104 and processes them all in order when order 102 is received.

In the second scenario, the subscriber checks oldest unacknowledged message in Cloud monitoring metrics. The subscriber can store all messages in a persistent datastore. It can check [indistinct] subscription/oldest_unacked_message metric that Pub/Sub publishes to Stackdriver. The subscriber can compare the timestamp of the oldest unacknowledged message against the published timestamps of the messages in Cloud Datastore. All messages published before the oldest unacknowledged message are guaranteed to have been received, so those messages can be removed from the persistent datastore and processed in order.

For more information about message ordering, see https://cloud.google.com/pubsub/docs/ordering.

Developing a Backend Service

Duration: 75 minutes

In the lab, Developing a Backend Service, you will enhance the online Quiz application by developing a backend service to process user feedback and save scores.

Lab objectives

Create a Pub/Sub topic

Publish messages to the topic

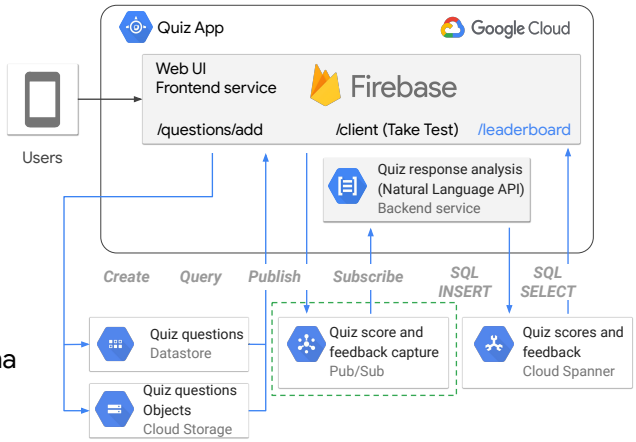
Subscribe to the topic to receive messages in a separate worker application

Harness the Natural Language API

Create a Cloud Spanner database instance

Architect a Spanner database schema

Insert data into a Spanner database



Google Cloud

You will **create** a Pub/Sub topic, **publish** messages containing quiz responses and free-form feedback text, and **subscribe** to messages in the topic. You will **invoke** the Natural Language API to process the feedback and determine a sentiment score.

You will **create** a Spanner database and **store** each user's quiz responses and sentiment score in the database.

Quiz

Pub/Sub dynamically adjusts the rate of push requests based on the rate at which it receives _____ responses

A. Failure

B. Success

Google Cloud

Pub/Sub dynamically adjusts the rate of push requests based on the rate at which it receives _____ responses.

a. failure
b. success

Quiz

Pub/Sub dynamically adjusts the rate of push requests based on the rate at which it receives _____ responses

A. Failure

B. Success

Google Cloud

Pub/Sub dynamically adjusts the rate of push requests based on the rate at which it receives **success** responses. (B)

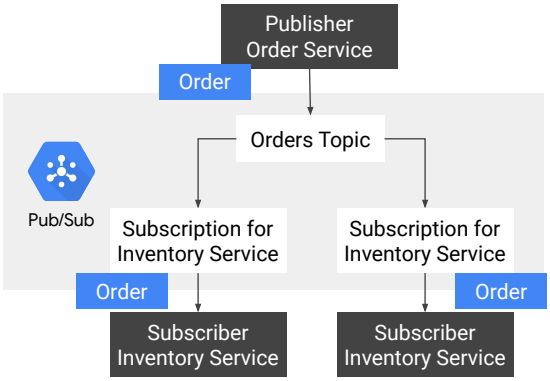
Quiz

What messaging use case is shown?

A. Message ordering

B. Handling duplicate messages

C. Message fan-out



Google Cloud

What messaging use case is shown?

a. Message ordering
b. Handling duplicate messages
c. Message fan-out

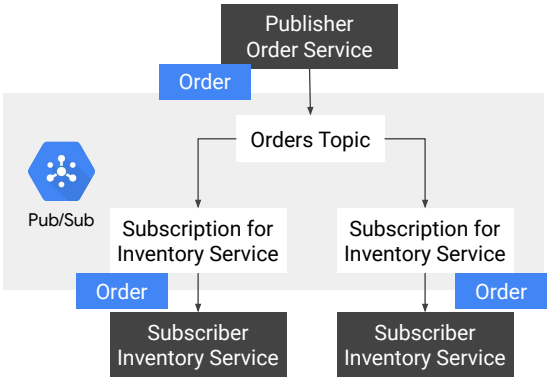
Quiz

What messaging use case is shown?

A. Message ordering

B. Handling duplicate messages

C. Message fan-out



Google Cloud

This messaging use case shown is a Message fan-out. (C)

Summary

In general, you should avoid point-to-point communications between applications. Such integrations can make your overall system brittle and difficult to manage. Use Pub/Sub to simplify integrations between scalable distributed systems. You can build a more scalable and resilient architecture when the components of your application are loosely coupled and communicate asynchronously.

Use Pub/Sub to rapidly and reliably ingest large volumes of incoming data and to fan out messages to multiple subscribers.

You can run your subscriber application in any execution environment, such as Compute Engine, GKE, or App Engine. You can also perform event-driven processing of Pub/Sub messages by using Cloud Functions.

With Pub/Sub, you can build highly scalable and resilient architectures.


Google Cloud