Pub/Sub supports both push and pull message delivery. For an overview and comparison of pull and push subscriptions, see the Subscriber Overview (/pubsub/docs/subscriber). This document describes pull delivery. For a discussion of push delivery, see the Push Subscriber Guide (/pubsub/docs/push).

Using asynchronous pulling provides higher throughput in your application, by not requiring your application to block for new messages. Messages can be received in your application using a long running message listener, and acknowledged one message at a time, as shown in the example below. Java, Python, .NET, Go, and Ruby clients use the streamingPull service API to implement the asynchronous client API efficiently.

Not all client libraries support asynchronously pulling messages. To learn about synchronously pulling messages, see Synchronous Pull (/pubsub/docs/pull#synchronous_pull).

For more information, see the API Reference documentation (/pubsub/docs/reference/libraries#additional_resources) in your programming language.

This sample shows how to pull messages asynchronously and retrieve the custom attributes from metadata:

This sample shows how to handle errors that arise when subscribing to messages:

Your subscriber client might process and acknowledge messages more slowly than Pub/Sub sends them to the client. In this case:

- It's possible that one client could have a backlog of messages because it doesn't have the capacity to process the volume of incoming messages, but another client on the network does have that capacity. The second client could reduce the subscription's backlog, but it doesn't get the chance to do so because the first client maintains a lease on the messages that it receives. This reduces the overall rate of processing because messages get stuck on the first client.
- Because the client library repeatedly extends the acknowledgement deadline for backlogged messages, those messages continue to consume memory, CPU, and bandwidth resources. As such, the subscriber client might run out of resources (such as memory). This can adversely impact the throughput and latency of processing messages.

To mitigate the issues above, use the flow control features of the subscriber to control the rate at which the subscriber receives messages. These flow control features are illustrated in the following samples:

More generally, the need for flow control indicates that messages are being published at a higher rate than they are being consumed. If this is a persistent state, rather than a transient spike in message volume, consider increasing the number of subscriber client instances.

Support for concurrency depends on your programming language. For language implementations that support parallel threads, such as Java and Go, the client libraries make a default choice for the number of threads. This choice may not be optimal for your application. For example, if you find that your subscriber application is not keeping up with the incoming message volume but is not CPU-bound, you should increase the thread count. For CPU-intensive message processing operations, reducing the number of threads might be appropriate.

The following sample illustrates how to control concurrency in a subscriber:

Support for concurrency depends on your programming language. Refer to the API Reference documentation (/pubsub/docs/reference/libraries#additional_resources) for more information.

The Pub/Sub service has two APIs for retrieving messages:

- Pull (/pubsub/docs/reference/rpc/google.pubsub.v1#pullrequest)
- StreamingPull (/pubsub/docs/reference/rpc/google.pubsub.v1#streamingpullrequest)

Where possible, the Cloud Client libraries use StreamingPull (https://grpc.io/docs/guides/concepts.html) for maximum throughput and lowest latency. Although you may never use the StreamingPull API directly, it is important to understand some crucial properties of StreamingPull and how it differs from the more traditional pull method.

The Pull method relies on a request/response model:

1. The application sends a request for messages.
2. The server replies with zero or more messages and closes the connection.

The StreamingPull service API relies on a persistent bidirectional connection to receive multiple messages as they become available, send acknowledgements, and modify acknowledgement deadlines:

1. The client sends a request to the service to establish a connection.
2. The client uses that connection to exchange message data.
3. The request (that is, the bidirectional connection) is terminated either by the client or the server.

StreamingPull streams are always terminated with an error code. Note that unlike in regular RPCs, the error here is simply an indication that a stream has been broken, not that requests are failing. Therefore, while the StreamingPull API may have a seemingly surprising 100% error rate, this is by design.

Because StreamingPull streams always terminate with an error, it isn't helpful to examine StreamingPull request metrics while diagnosing errors. Rather, focus on StreamingPull message operation metrics. Look for these errors:

- `FAILED_PRECONDITION` errors can occur in these cases:
  - Pub/Sub attempts to decrypt a message with a disabled Cloud KMS key.
  - The stream shuts down due to a suspended subscription. Subscriptions can be temporarily suspended if there are messages in the subscription backlog that are protected by a disabled Cloud KMS key.
- `UNAVAILABLE` errors

The gRPC StreamingPull stack is optimized for high throughput and therefore buffers messages. This can have some consequences if you are attempting to process large backlogs of small messages (rather than a steady stream of new messages). Under these conditions, you may see messages delivered multiple times and they may not be load balanced effectively across clients.

The buffer between the Pub/Sub service and the client library user space is roughly 10MB. To understand the impact of this buffer on client library behavior, consider this example:

- There is a backlog of 10000 1KB messages on a subscription.
- Each message takes 1 second to process sequentially, by a single-threaded client instance.
- The first client instance to establish a StreamingPull connection to the service for that subscription will get a buffer of the entire 10K messages.
- It takes 10000 seconds (almost 3 hours) to process the buffer.
- In that time, some of the messages exceed their acknowledgement deadline and are re-sent to the same client, resulting in duplicates.
- When multiple client instances are running, the messages stuck in the buffer will not be available to any instances other than the first.

This situation will not occur if the messages are arriving at a steady rate, rather as a single large batch. The service never has the entire 10MB of messages at a time and so is able to effectively load balance messages across multiple subscribers.

To address this situation, either use a push subscription or a pull API, currently available in some of the Cloud Client Libraries (see the Synchronous Pull section) and all API Client libraries. To learn more, see the Client Libraries documentation (/pubsub/docs/reference/libraries).

There are cases when the asynchronous pull is not a perfect fit for your application. For example, the application logic might rely on a polling pattern to retrieve messages or require a precise cap on a number of messages retrieved by the client at any given time. To support such applications, the service supports a synchronous Pull method.

Here is some sample code to pull (/pubsub/docs/reference/rpc/google.pubsub.v1#google.pubsub.v1.PullRequest) and acknowledge (/pubsub/docs/reference/rpc/google.pubsub.v1#modifyackdeadlinerequest) a fixed number of messages:

Note that to achieve low message delivery latency with synchronous pull, it is important to have many simultaneously outstanding pull requests. As the throughput of the topic increases, more pull requests are necessary. In general, asynchronous pull (/pubsub/docs/pull#asynchronous-pull) is preferable for latency-sensitive applications.

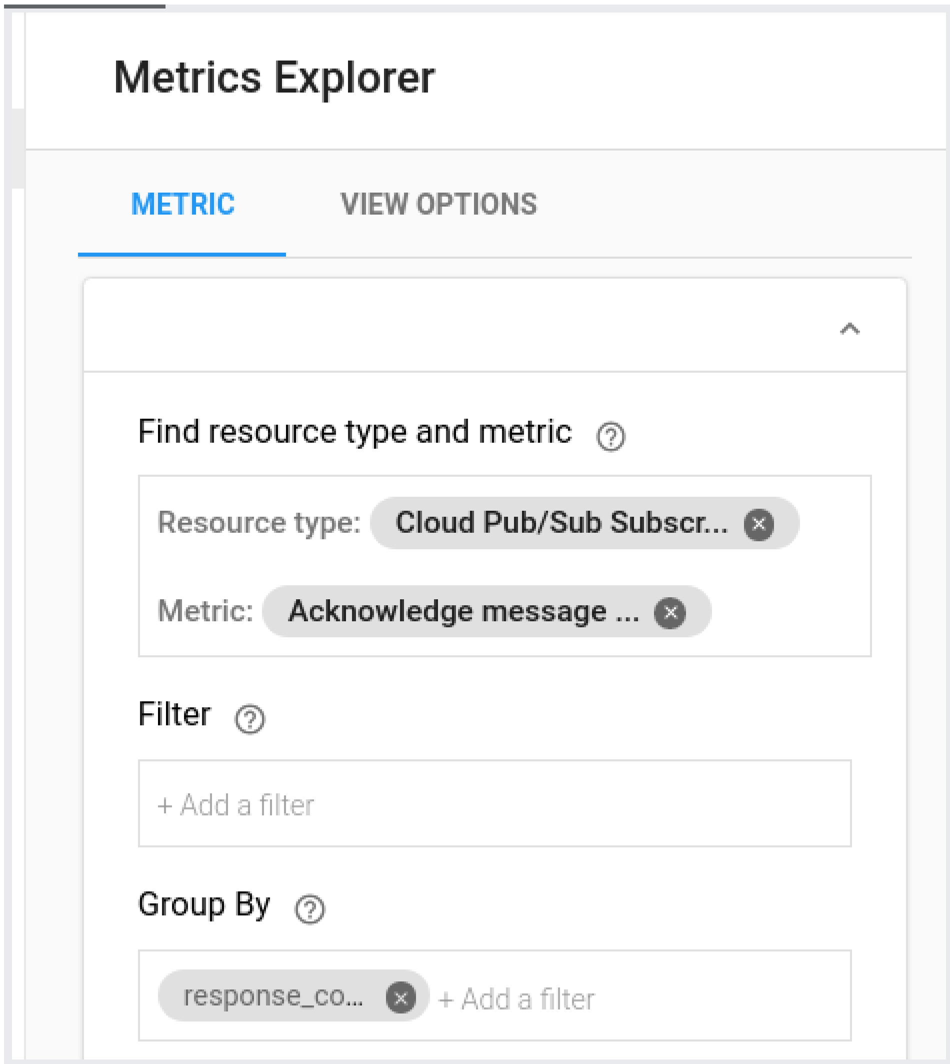
An individual message's processing may exceed the preconfigured acknowledgement deadline, also known as the lease. To avoid redelivery on these messages, the client libraries provide a way to reset their acknowledgement deadlines (except for the Go client library, which automatically modifies the acknowledgement deadlines for polled messages), as shown by the samples below:

You may need to implement a scaling mechanism for your subscriber application to keep up with message volume. How to do this depends on your environment, but it will generally be based on backlog metrics offered through the Stackdriver (/monitoring) monitoring service. For details on how to do this for Compute Engine, see Scaling based on Cloud Monitoring Metrics (/compute/docs/autoscaler/scaling-cloud-monitoring-metrics).

Go to the Pub/Sub section of the GCP Metrics List (/monitoring/api/metrics_gcp#gcp-pubsub) page to learn which metrics can be monitored programmatically.

Finally, as with all distributed services, expect to occasionally retry every request.

When you do not acknowledge a message before its acknowledgement deadline (/pubsub/docs/subscriber#delivery) has expired, Pub/Sub resends the message. As a result, Pub/Sub can send duplicate messages. Use Stackdriver to monitor (/pubsub/docs/monitoring) acknowledge operations with the `expired` response code to detect this condition. To get this data, select the **Acknowledge message operations** metric, then group or filter it by the `response_code` label. Note that `response_code` is a system label on a metric—it is not a metric.


Metrics Explorer

METRIC VIEW OPTIONS

Find resource type and metric

Resource type: Cloud Pub/Sub Subscr...

Metric: Acknowledge message ...

Filter

+ Add a filter

Group By

response_co... + Add a filter

To reduce the duplication rate, extend the message deadline.

- Client libraries handle deadline extension automatically, but you should note that there are default limits on the maximum extension deadline that can be configured.
- If you are building your own client library, use the `modifyAckDeadline` method to extend the acknowledgement deadline.

Alternately, to force Pub/Sub to retry a message, set `modifyAckDeadline` to 0.