Application Development with Cloud Run

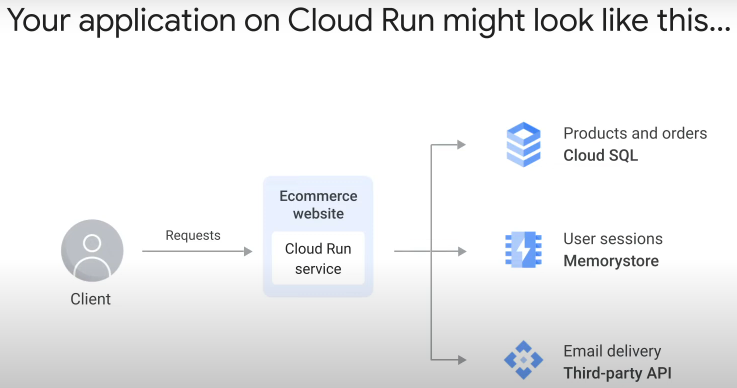
### Service-to-Service Communication

#### Introduction

Modern application is split into multiple services.

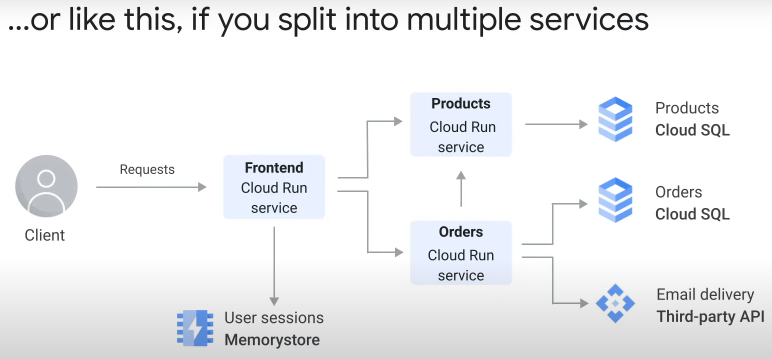
* Each service does 1 thing.
* Services need to communicate with one another

Architecture:



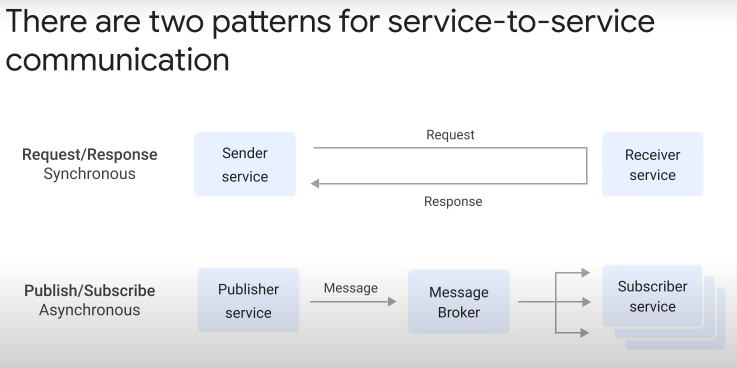
* Cloud Run frontend connected to multiple backends and APIs

Micro-service architecture:

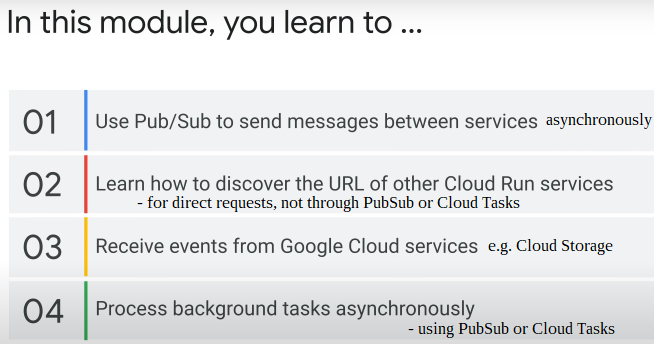
Advantages:

* improved scalability – every service can scale independently
* improved collaboration and development speed – distributed teams build and deploy services independently
* improved robustness – if one service fails, other services continue to work

2 service-service communication patterns:



1. Synchronous request-response
   * Sender service will wait (block) until response successfully received, before continuing
2. Asynchronous PubSub
   * Publisher/Sender service does not expect response, does not wait (block)
     + Publishes/sends message to broker
     + immediately continues next lines of code
   * message broker copies payload (request body) and sends to multiple subscribers

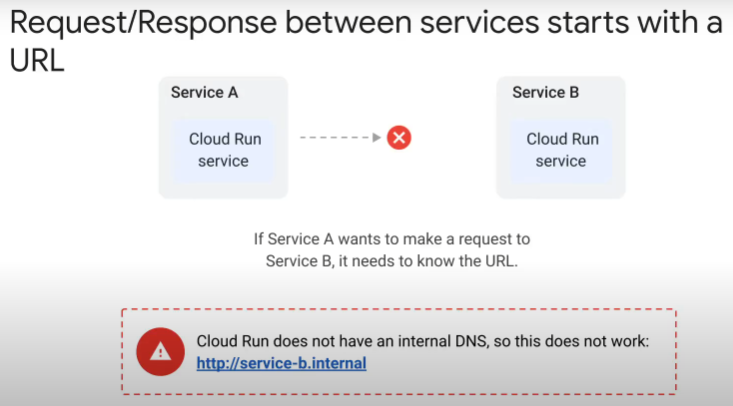


#### Request/Response

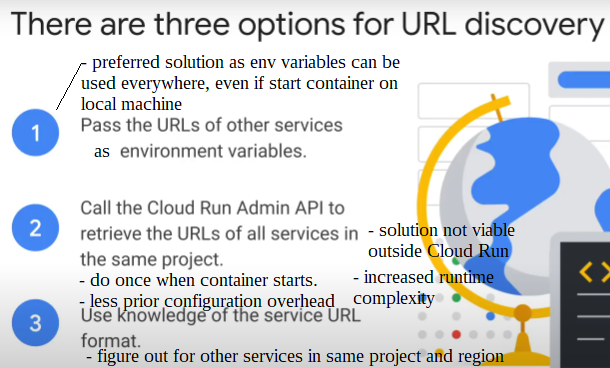
* Direct Cloud Run service to Cloud Run service
* not through PubSub
* **synchronous**

To send requests to a deployed Cloud Run service, need to know target URL

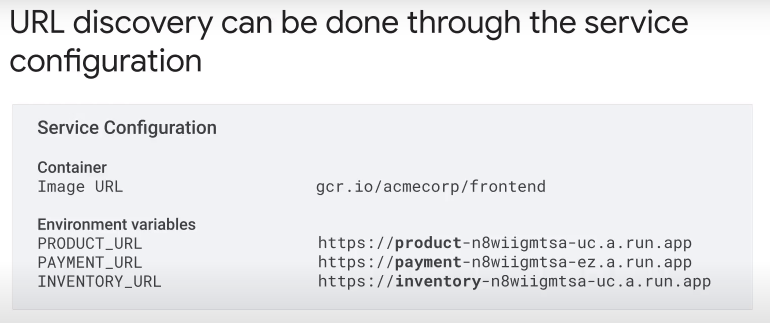
* can use stable, unique auto-generated URL for each deployed service
* format: revision tag---service name-project ID-region code.run.app
  + but not fixed and logic for its generation may change
* only this external default URL. No internal DNS for Cloud Run.
* https://cloud.google.com/run/docs/triggering/https-request



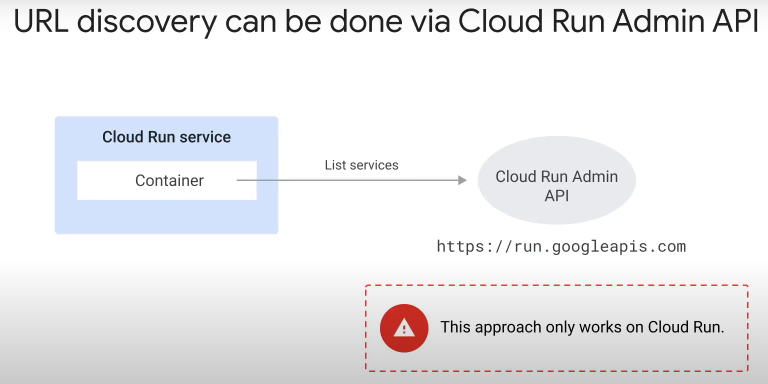
3 ways to discover URL of other Cloud Run services:



Recommended solution: save URL as service configuration (environment variables)

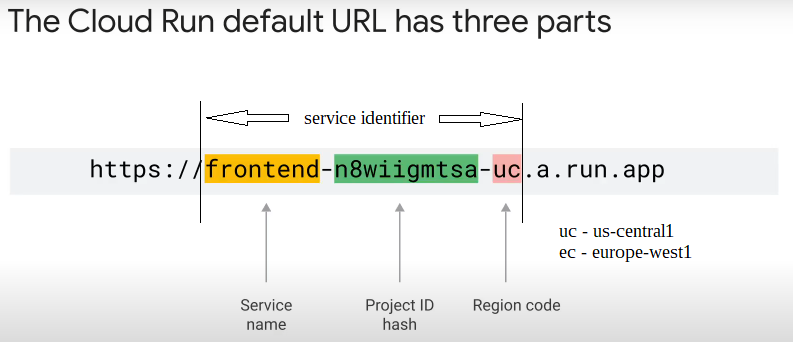
* in application code, access environment variable when need to use URL of target Cloud Run service
* most portable solution of 3 options
* one benefit is can easily change the URL when environment changes, e.g. testing to production, on local machine.
* can use environment variables to pass connection parameters for other services, e.g. database connection string, URL of 3rd-party API.
* Do not save API keys and passwords in environment variables
  + security risk
  + use Secret Manager
* if there are many services (Cloud Run, databases, 3rd-party), list of env variables will become long and difficult to maintain
  + can use Infrastructure-as-a-Code tool, e.g. Terraform, to automate

Alternative: Call Cloud Run Admin API from application code running in Cloud Run container



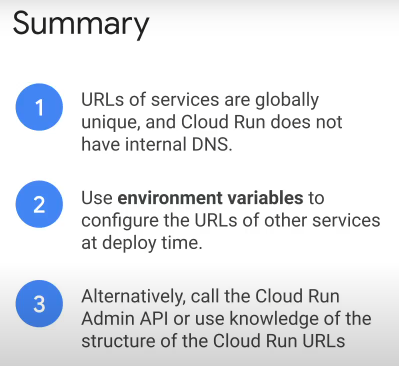
* there is a REST endpoint that lists all the services in project
* use GET method
* global base URL: https://run.googleapis.com
* https://cloud.google.com/run/docs/reference/rest
* drawbacks:
  + runtime complexity increases
    - Cloud Run Admin API becomes one more dependency in code
  + need to limit Cloud Run IAM permission to minimum required
    - run.services.list
    - https://cloud.google.com/run/docs/reference/iam/permissions
  + solution does not work outside Cloud Run
  + method does not work directly for other service endpoints, e.g. database connections, 3rd-party services.

Fast but risky solution: Use knowledge of structure of Cloud Run default URL



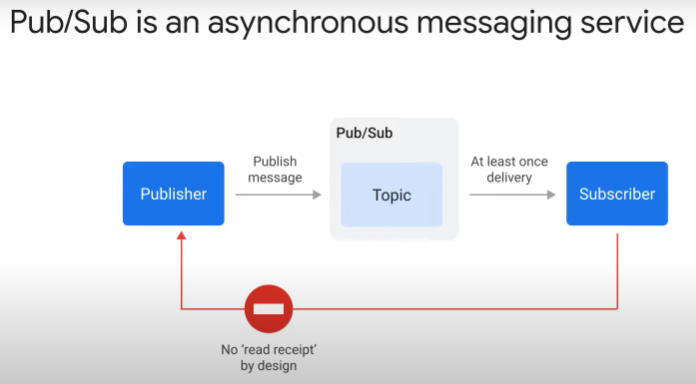
Infer URL of other Cloud Run services

* project ID is globally unique for a project, same for all services in a project
* drawback:
  + solution does not work for other service endpoints
  + service identifier structure not fixed, logic may change in future



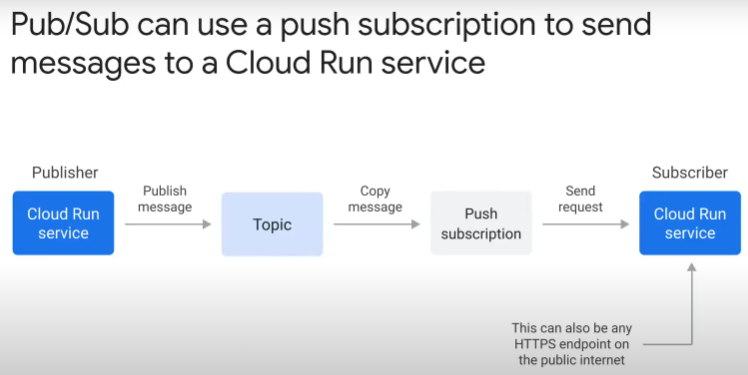
#### PubSub

* **Asynchronous, Scalable, Available, Low latency, Reliable**
* https://cloud.google.com/pubsub/docs/overview

General outline:

* publisher sends message (data) to topic (named entity that represents a feed of messages)
* PubSub writes message to storage and acknowledges receipt to publisher.
* At the same time, it delivers message at least once, to all attached subscriptions.
* After processing, subscribers respond to PubSub. For each subscription, if at least 1 subscriber response positively, PubSub deletes the message.
* publisher/producer is any process that can access PubSub API
* subscriber/consumer is any HTTPS endpoint (push subscription), or any process that can access PubSub API (pull subscription)
* publishers and subscribers are **decoupled**.
  + Publishers do not wait for response from subscribers
* publishers and subscribers can be (but need not be) Cloud Run services

Architecture: 1 PubSub topic > push subscriptions > to multiple subscribers



* each subscriber is configured independently using “subscription” (delivery type: push).
* https://cloud.google.com/run/docs/triggering/pubsub-push
* in subscription, can configure
  + endpoint URL (for PUSH delivery type only)
  + whether subscription to expire if inactive, or never expire
  + message retention duration (option to enable for acknowledged messages too)
  + message ordering (whether must receive in same order as published. Ordering may increase latency)
  + filter to apply (if any)
  + deadline for message to be acknowledged (max 10 min)
  + retry policy (immediate / exponential back off delay)
  + enable dead lettering (DLQ)?

Steps:

1. add codes to handle PubSub

* need to get request entity body, and send response HTTP code

1. create topic
2. create Service Account to associate with subscription

* SA identity must be given “Cloud Run Invoker” role

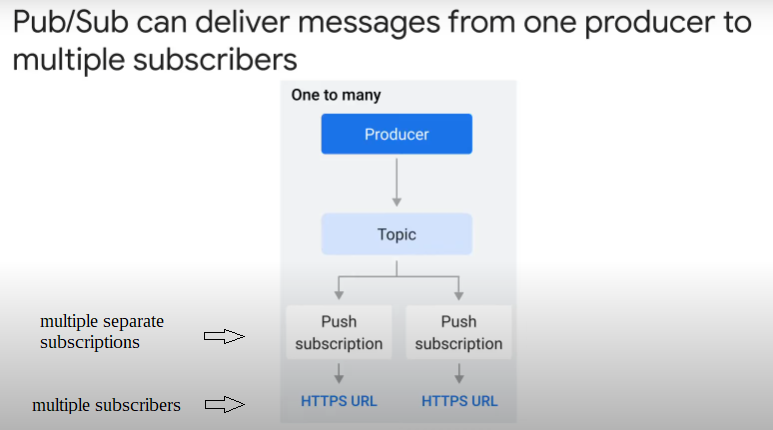
1. create subscription

* associate with topic, Service Account created and endpoint URL of subscriber
* configure push subscription

Process during runtime:

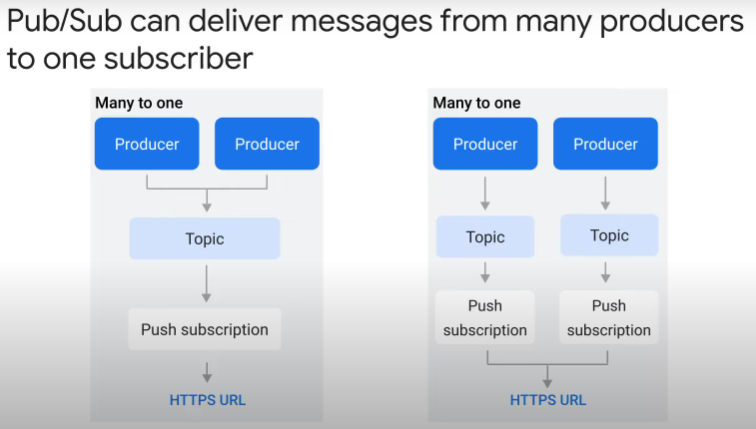
* Publisher publishes message to topic
* Message queues
* PubSub copies message to all subscriptions attached to topic
* HTTP POST requests to subscribing Cloud Run services
  + payload in request body

Communication pattern: One to Many



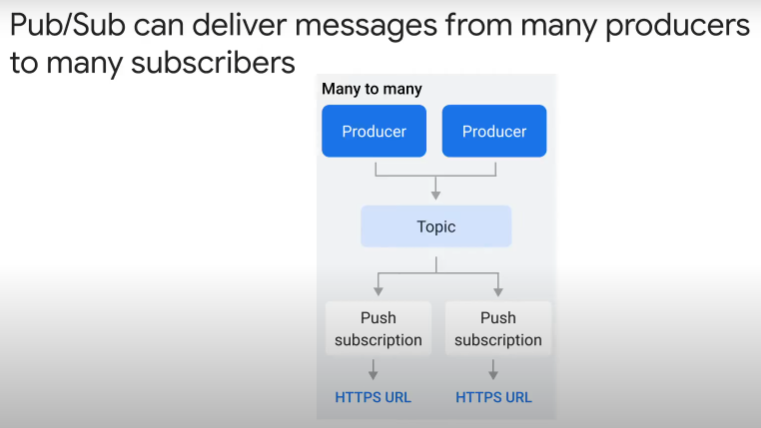
* messages will be load-balanced across subscribers

Communication pattern: Many to One

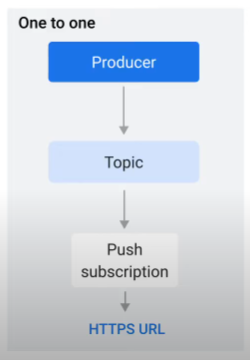


* multiple publishers POST message/event > one subscriber
* 2 scenarios:
  + One topic
    - e.g. multiple services publishing events > one subscriber aggregating events into a data store
  + Multiple topics
    - e.g. same Cloud Run subscriber but different endpoint paths
    - e.g. different subscription configurations for different publishers

Communication pattern: Many to Many



* many services POST and read events from same topic
* different subscriptions apply different filtering



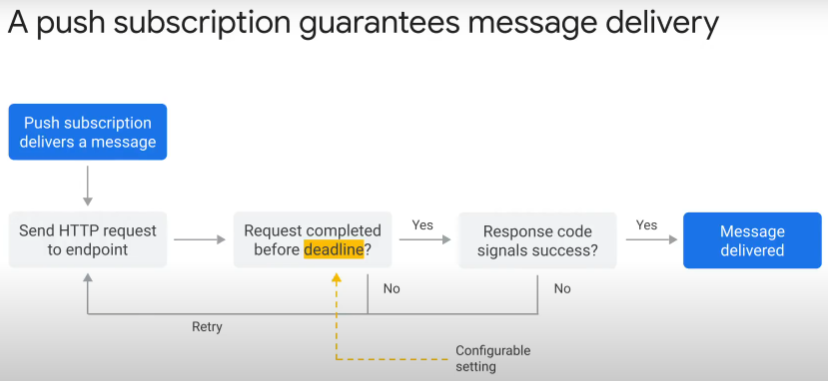
Using PubSub for **direct invocation**

* create topic and subscription for 2 Cloud Run services to communicate with each other.

Advantages of using PubSub:

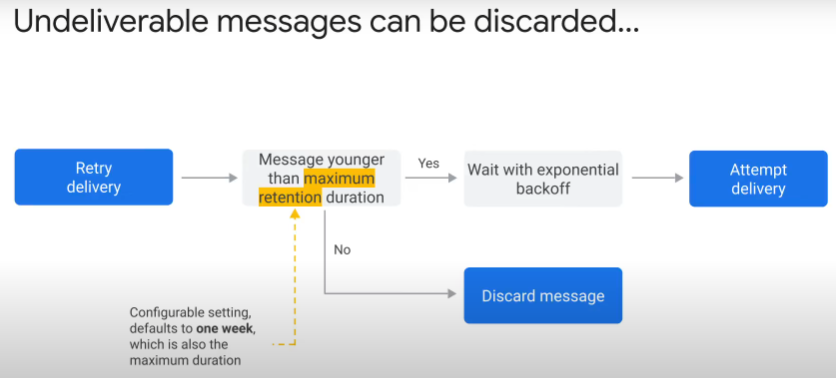
* publisher does not need to block/wait
* Publisher no need error handling and retry logic in code
* PubSub guarantees message delivered at least once

How PubSub decides if message delivery is successful:

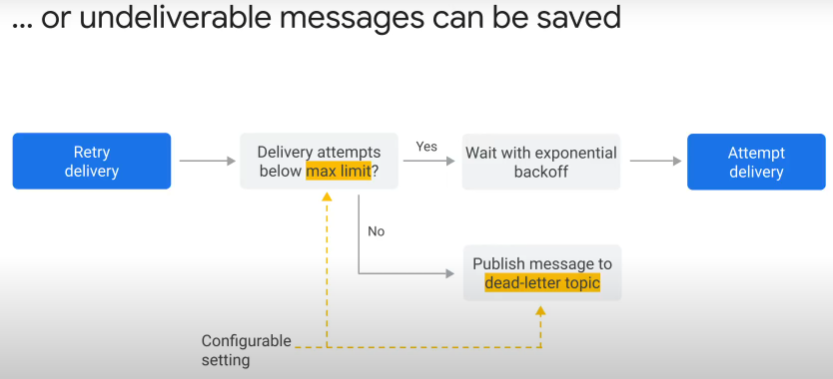


* PubSub POST message
* deadline for message to be acknowledged is 10s (default) to 10 min
* if request times out, retries
* if subscriber responds with HTTP 4XX (client error codes), retries
* request is successful if response HTTP status code is 2XX.

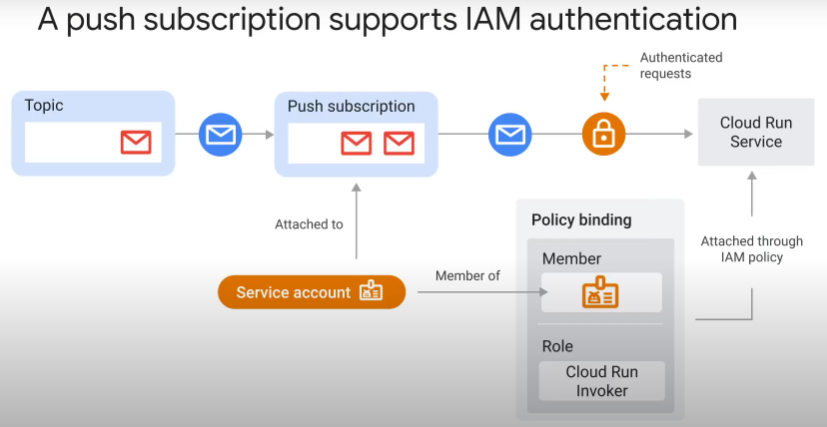
Retries:

* message retention duration configured in subscription
  + option to enable retention of acknowledged messages
  + max of 7 days (default)
  + will only resend if message within retention duration
* retries only if acknowledgement deadline expires, or negative acknowledgement received from subscriber
  + max acknowledgement deadline is 10 min
* if retry policy triggered:
  + immediate
  + or with exponential backoff delay
    - PubSub waits 10s before resending a message
      * if still fails, increases delay before retrying again
      * until max of 10-min interval between retries
* if message older than age specified by max retention duration => discarded

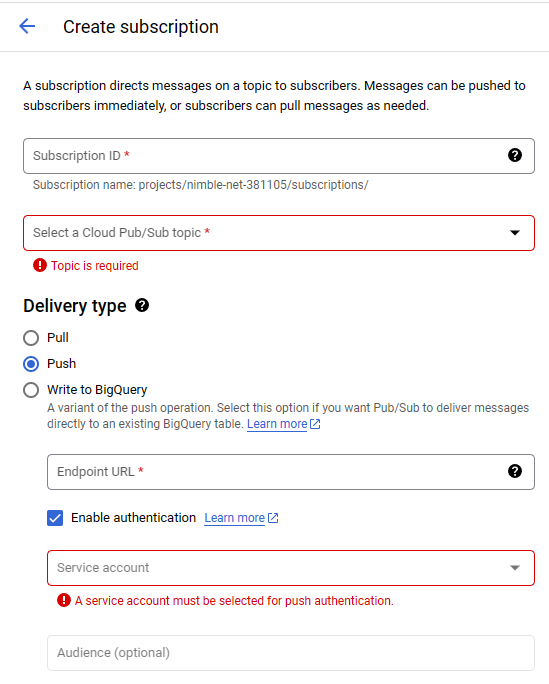
Dead letter topic / queue:

* if messages are undeliverable after configured message retention duration, they will be discarded
* To save undeliverable messages, republish to a specified Dead Letter topic
* Steps:
  + create Dead Letter topic
  + configure or update subscription to enable Dead Lettering
  + can configure max delivery attempts
  + can reduce message retention duration > reduce message storage fees

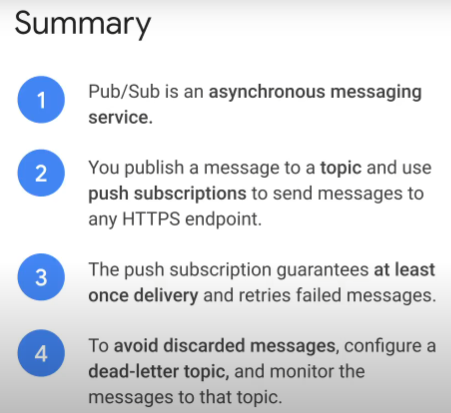
Use **IAM** to restrict access to messages from specified push subscriptions:

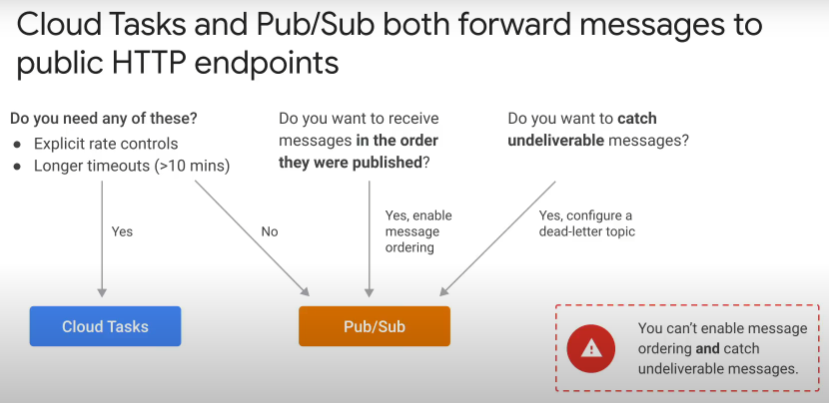


* only want messages from specific push subscriptions
* use IAM to restrict access to Subscriber (Cloud Run service)
* steps:
  1. create Service Account.
  + Enter a Service Account ID and grant “Cloud Run Invoker” role (policy binding).
  1. In subscription,
  + select PubSub topic
  + specify PUSH delivery type
  + enter endpoint URL of Cloud Run service
  + select to enable authentication
  + select Service Account created with “Cloud Run Invoker” role



* PubSub automatically adds a Google-signed Open ID Connect (OIDC) token to header of request from this topic.
* IAM will reject all other requests



* Cloud Task conceptually supports same workflow as PubSub
* also forward messages to any public HTTP endpoint
* Similarities:
  + guarantees delivery of a message at least once
  + supports IAM authentication
* Differences > the reasons when to use Cloud Tasks
  + https://cloud.google.com/tasks/docs/comp-pub-sub
  1. explicit control of rate of requests; scheduled delivery.
     + can specify max task dispatches per second and max number of concurrent tasks
     + useful when sending requests to external endpoint that limits requests per second, or another resource with capacity constraint
     + PubSub adjusts request rate based on error rate, but no guarantee
  2. longer duration before timeout to acknowledge a message
     + max 30 min vs 10 min for PubSub
     + useful when want to complete longer-running task that cannot be split into multiple parallel tasks
* Differences > the reasons when to use PubSub.

PubSub supports

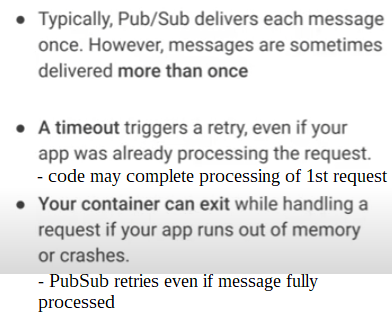
* 1. message ordering. Cloud Tasks don’t.
  + requests delivered in order they are published
  + however, option increases latency for message delivery
  1. Dead Lettering. Cloud Tasks don’t.
     + forward undeliverable messages to a dead letter topic
  2. pull via API
  3. multiple handlers/subscribers per message
  4. global availability vs regional for Cloud Tasks
* PubSub can do regional isolation by locating Publishers, Subscribers, topic and subscription in 1 region (data will not leave region)
* Cannot enable message ordering and dead lettering at same time for PubSub.

#### Handling Retries

Focus:

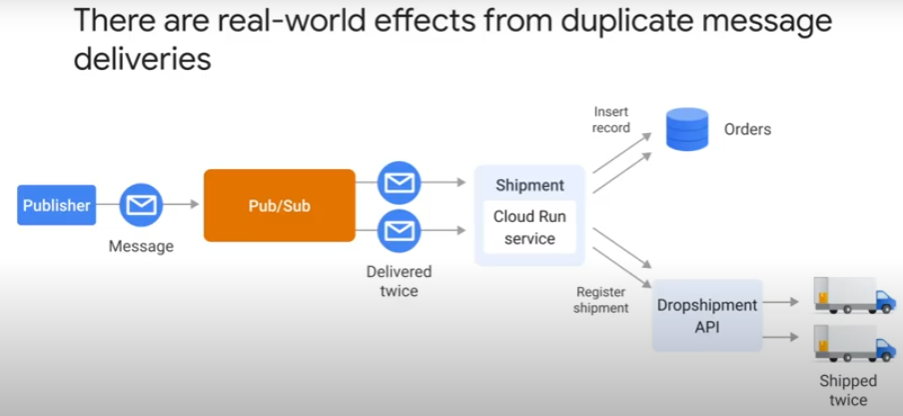
* retries
* duplicate deliveries

Application code must be designed to handle same message delivered more than once:



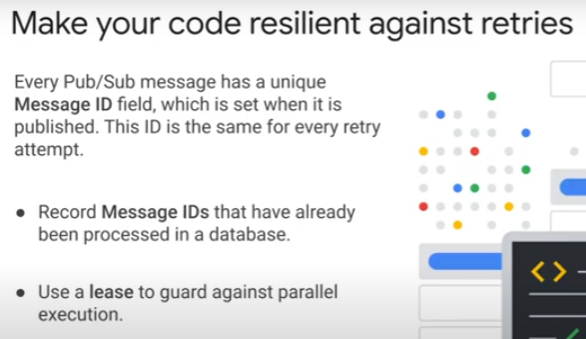
* applies to PubSub and Cloud Tasks
* because PubSub has distributed architecture

Real-world consequences:



* e,g, Cloud Run service for shipment receives messages (orders) twice > ships 2 X

How to make code resilient against retries:



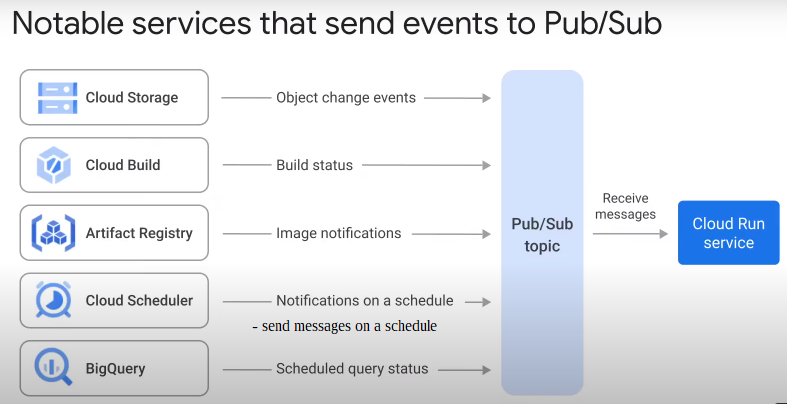
* small chance of retry while message being processed
* solution: use unique message ID to record state of message processing in a database
  + e.g. “in progress” with a timestamp of when the lease expires, “completed”, “failed”
  + check against database for every message received
    - reject “in progress” and “completed” requests
  + use lease to handle “in progress” messages exclusively
    - a subscriber can extend deadline to acknowledge message being processed with a call to modify the Ack deadline.
    - https://cloud.google.com/pubsub/docs/lease-management
  + should prevent most instances of duplicate processing

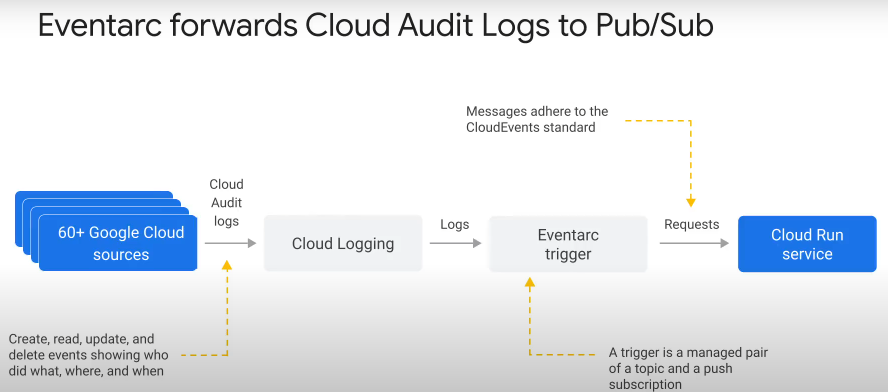
#### Receiving Events

* From other Google Cloud services, ? 3rd party services
* any process that can access PubSub API can publish messages to a topic

Many Google services integrated with PubSub:

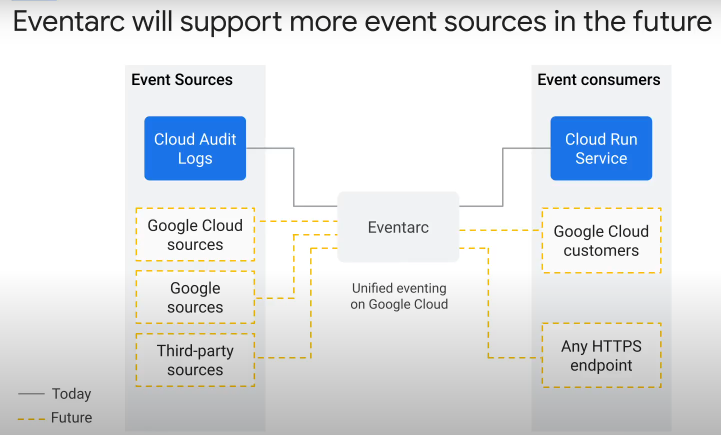
* these services push (POST) events
* attach push subscription to an existing topic



Eventarc asynchronously delivers events:

* event-driven architecture – flow of events between decoupled microservices
* event providers (sources):
  1. Google Cloud services
  2. Cloud Audit Logs
  + the create, read, update, delete (CRUD) events for cloud resources
  + register who did what, where and when.
  + may not have the event you need to build your integration – need to check.
    - For example, for Cloud Build, Cloud Audit Log will only show a Create Build event. Will not show status of build, i.e. success or failure.
  1. 3rd party sources
     + e.g. SaaS > e.g. Datadog, Check Point CloudGuard, PagerDuty
* an event is a discrete state change
  + CloudEvents format
  + https://cloud.google.com/eventarc/docs/cloudevents
* create a response to an event by creating a **trigger**.
  + Event occurs no matter there is a target destination or not
  + a trigger is a managed pair of a topic and a push subscription
    - note: any subscription created will persist even when there is no activity and does not expire.
  + specify filters to capture specific events
* Event destinations:
  + Cloud Run
  + Cloud Functions (2nd gen)
  + GKE
  + Workflows
* <https://cloud.google.com/eventarc/docs/overview>

Eventarc to be hub for events:



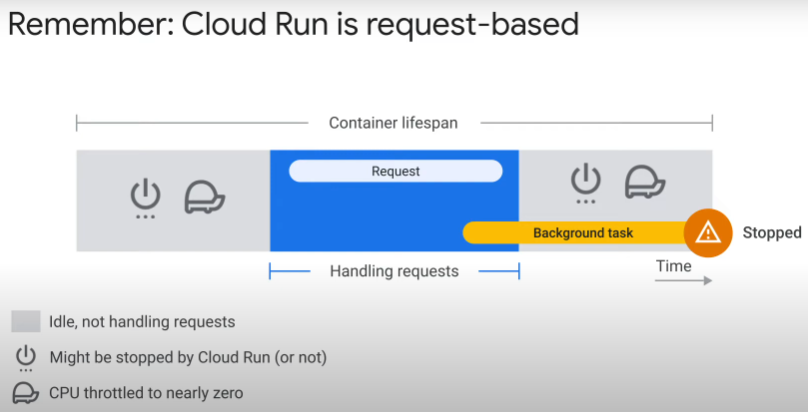
* in future, can use Eventarc to receive events from
  + more Google Cloud services, e.g. Firestore, Big Query.
  + Other Google services, e.g. Gmail, Google Meet
  + more 3rd party sources

#### Background Tasks

* Use secondary PubSub to run background tasks

Recap:

* with Cloud Run, application code should finish all processing before returning request to user
* if a Cloud Run container is not handling requests for > 100ms,
  + it transitions to idle state
    - CPU is trottled to near zero
    - network requests to 3rd party APIs will likely fail
    - an idle container may be shut down at any time

Plausible scenarios to run background tasks:

* user submits file > need to process file in background without blocking request
* application creates a new record > need to update search index in background
* request is start of a workflow with multiple steps and takes time to complete

Run background tasks using PubSub / Cloud Tasks:

* don’t run background thread in Cloud Run container
* instead, publish message to PubSub
* push subscription from PubSub will forward message to secondary, or same, Cloud Run service
* second Cloud Run service executes the task
* Benefits:
  + more reliable architecture
  + second Cloud Run service performing task can scale independently of first Cloud Run service handling requests
  + errors handling (codes) with retries (subscription)

