

Cloud Tasks

Best practices and lessons learned

- cloud-tasks-devfest-milano-2023-slides.vercel.app ↗
- [jackdbd/cloud-tasks-devfest-milano-2023-slides](https://github.com/jackdbd/cloud-tasks-devfest-milano-2023-slides) 🌐



Giacomo Debidda

Freelance full stack developer / web performance consultant

I like TypeScript / Clojure / Zig

I also like:

-  - add me on [goodreads](#) ↗
-  - [surfskating](#) ↗ actually
-  - it's time for a rollerblading emoji 🎈

 [jackdbd](#)

 [jackdbd](#)

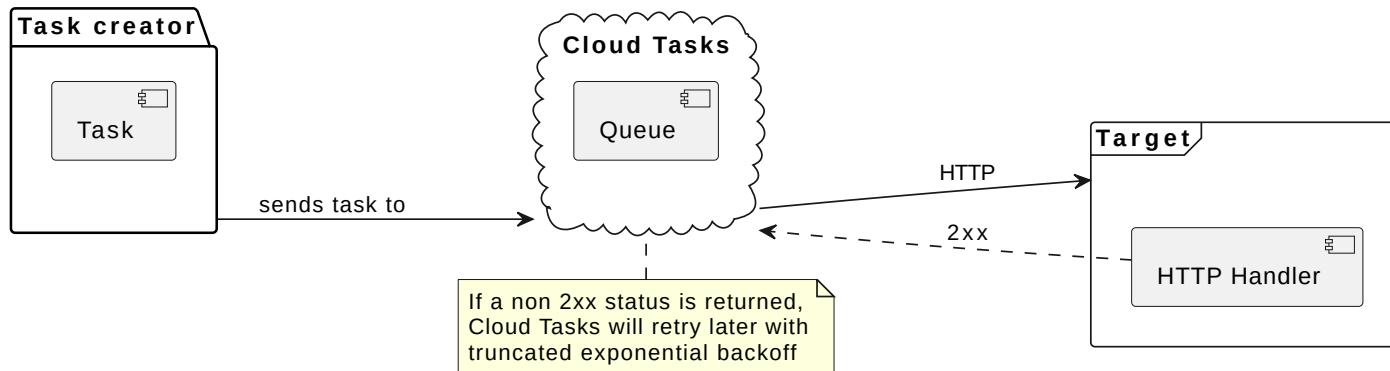
 [giacomodebidda.com](#)



What is Cloud Tasks?

GCP's **fully managed** service to manage **queues** of **background tasks**.

Background task implies that the user or requester **does not wait** for the task to finish.



Queues in theory

1. Infinitely long.
2. Open at both ends (enqueue, dequeue).
3. FIFO (First In, First Out).
4. Exactly-once semantics.

Queues in Cloud Tasks

1. Limited rate of processing (set by us).
2. We can only add tasks to a queue (enqueue), not remove them (dequeue). We can delete tasks by their ID though! Cloud Tasks automatically deletes old tasks (see `taskTtl` in v2beta3).
3. No task execution order is guaranteed. We can schedule tasks for later, configure retries, force specific tasks to run!
4. No exactly-once semantics is guaranteed.

Cloud Tasks aims for a strict "execute exactly once" semantic. ”
However, in situations where a design trade-off must be made between guaranteed execution and duplicate execution, the service errs on the side of guaranteed execution. As such, a non-zero number of duplicate executions do occur.

Source: Duplicate execution [🔗](#)

Cloud Tasks use cases

Buffering requests to comply with rate & concurrency limits from 3rd party APIs

APIs may impose rate and/or concurrency limits. If we exceed these limits, APIs respond with HTTP 429 Too Many Requests. We can configure a Cloud Tasks queue to comply with these limits, instead of doing it in our code.

Buffering requests to smooth out traffic spikes to our service

This is especially useful if our service is deployed on a single VM, or if we have an upper limit to the number of containers of a Cloud Run service.

Delegating slow operations to a dedicated service

APIs that send webhooks to an endpoint we control, typically require the endpoint to respond with an HTTP 2xx status code within a few seconds (e.g. Stripe API). If we want to perform a potentially slow operation (e.g. write to DB, send email, generate PDF), we can create a task, let Cloud Tasks dispatch it to a dedicated service, and respond to the webhook sender immediately.

Preserving requests in the context of a service outage

We need to send an email, but SendGrid is down. We create a task, let Cloud Tasks retry a bunch of times, and hopefully SendGrid will come back online.

How do we use Cloud Tasks?

Queues

Configuration for a Cloud Task queue:

- GCP region
- Rate limits
- Retries
- Logging
- `taskTtl`: maximum amount of time that a task is retained in this queue (from v2beta3)
- `tombstoneTtl`: amount of time the task tombstone is retained after a task is deleted or executed. The tombstone is used in task deduplication (from v2beta3)

Creating a queue

Using gcloud:

```
gcloud tasks queues create my-queue \
--location europe-west3 \
--max-dispatches-per-second 500 \
--max-attempts 100
```

Or using an IaC tool like pulumi:

```
import * as gcp from '@pulumi/gcp'

const enable_cloudtasks = new gcp.projects.Service(
  'enable-cloudtasks-api',
  { service: 'cloudtasks.googleapis.com' }
)

const queue = new gcp.cloudtasks.Queue('my-queue',
{
  location: 'europe-west3',
  name: 'my-queue',
  rateLimits: { maxDispatchesPerSecond: 500 },
  retryConfig: { maxAttempts: 100 },
  stackdriverLoggingConfig: { samplingRatio: 0.5 }
},
{ dependsOn: [enable_cloudtasks] }
)
```

Tasks

Object (e.g. Plain Old Javascript Object) that represents a **piece of work** to be performed by **some service**.

Explicit invocation: who creates the task retains full control of execution.

Task configuration for an HTTP task:

- URL of the endpoint (required)
- Unique name
- HTTP method
- HTTP request headers
- Request body
- Authentication (API Key, OIDC token, OAuth token)
- Schedule time
- Dispatch deadline (e.g. wait max 30 minutes for the worker to process the task)

Creating an HTTP task

```
const {CloudTasksClient} = require('@google-cloud/tasks')

const client = new CloudTasksClient()

const project = 'devfest-milano-2023'
const location = 'europe-west3'
const queue = 'my-queue'
const parent = client.queuePath(project, location, queue)

const url = 'https://some-api-endpoint'
const payload = { foo: "bar" }
const str = JSON.stringify(payload)
const serviceAccountEmail =
  `task-enqueuer-user@${project}.iam.gserviceaccount.com` 

const task = {
  httpRequest: {
    httpMethod: 'POST',
    url,
    oidcToken: { serviceAccountEmail, audience: url },
    headers: { 'Content-Type': 'application/json' },
    body: Buffer.from(str).toString('base64'),
  },
  name: 'my-task-id',
  // schedule the task 5 seconds from now
  scheduleTime: { seconds: Date.now() / 1000 + 5 },
}

const [response] = await client.createTask({parent, task})
```

IAM

Task creator

The service account attached to the service that creates tasks and sends them to Cloud Storage must have the IAM roles **Task Enqueuer** and **Service Account User**:

- `roles/cloudtasks.enqueuer`
- `roles/iam.serviceAccountUser`

Cloud Tasks

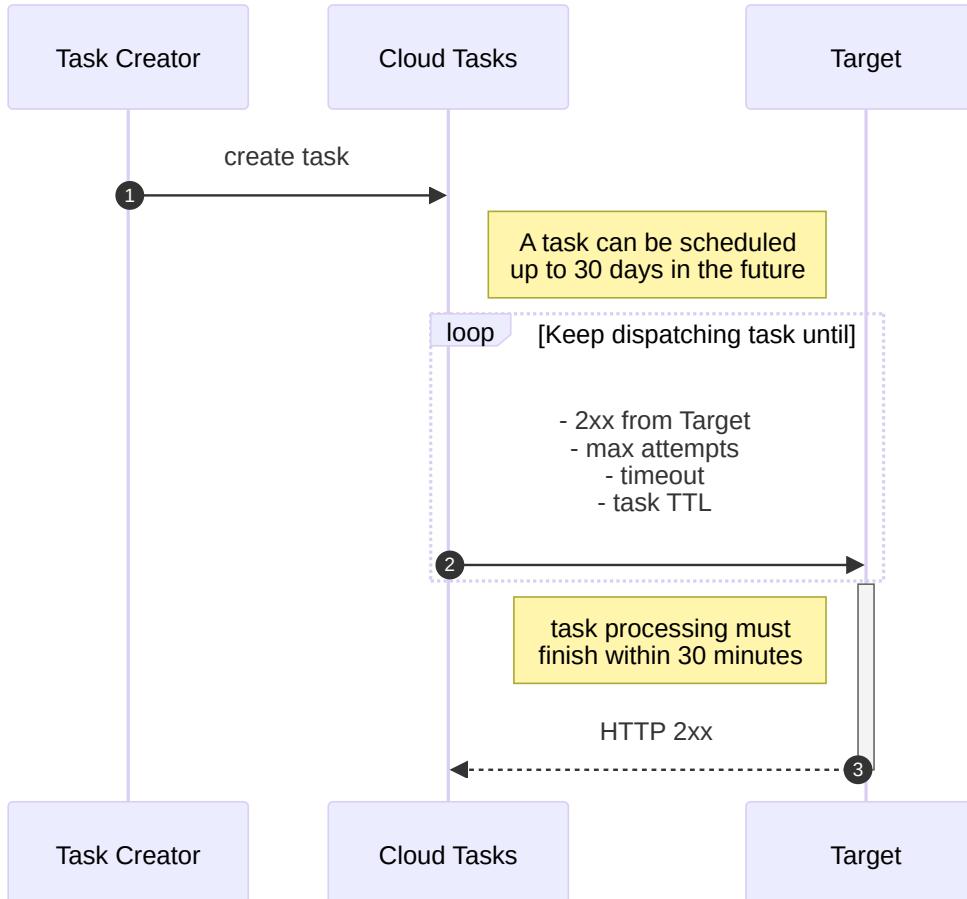
The service account for Cloud Tasks is managed by Google. Double check that this service account exists and that it has the IAM role **Cloud Tasks Service Agent**:

```
service-<project_number>@gcp-sa-cloudtasks.iam.gserviceaccount.com
```

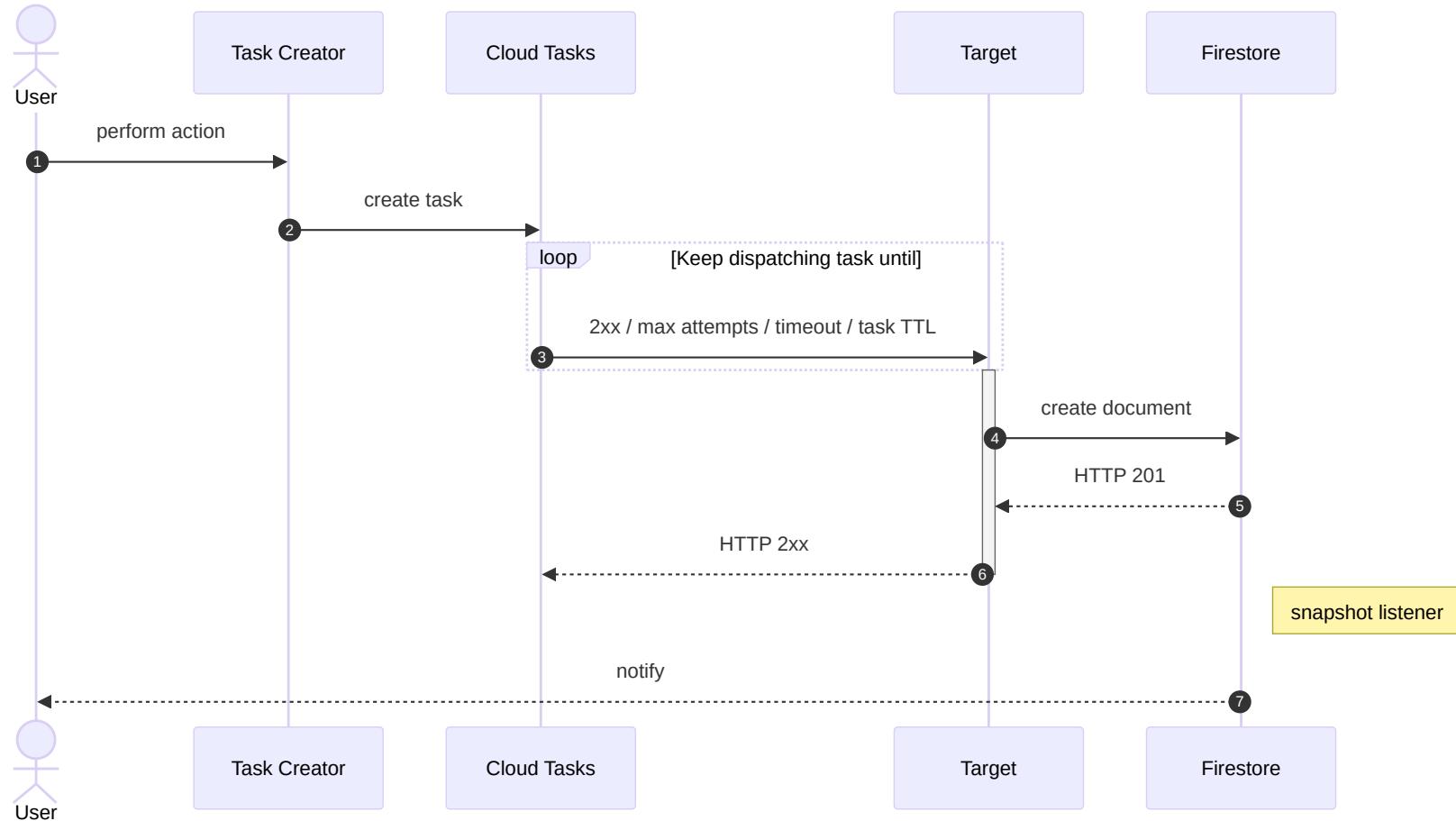
Task handler

The service account attached to the service that processes tasks must have the necessary IAM permissions to do its job. E.g. For a Cloud Run service that requires read/write access to Firestore we can assign the IAM role `roles/firestore.datastore.user`.

A basic example



A more advanced example



Notifications with SSE

We can implement a Cloud Run service that sends server-sent events (SSE) and receive those events on the frontend using an EventSource. Unfortunately, on serverless platforms SSE can be expensive and unreliable.

Pros:

- It's just HTTP, not a different protocol.
- Handles disconnect/reconnect and missed messages automatically.
- No need for special handshaking.

Cons:

- Unidirectional, server-to-client data stream.
- The Cloud Run service must be configured with CPU always allocated (i.e. never idle).
- Network proxies and other hardware/software can totally break SSE.
- When not used over HTTP/2, SSE suffers from a limitation to the maximum number of open connections, which can be especially painful when opening multiple tabs, as the limit is per browser and is set to a very low number (6).

SSE doesn't work well on Cloudflare Workers either.

Notifications with WebSockets

Cloud Run supports WebSockets, but it can be really expensive.

Pros:

- Bidirectional data stream.

Cons:

- It's not HTTP, it's a different protocol.
- The Cloud Run service must be configured with CPU always allocated.
- We need to handle request timeouts and client reconnects (or use a library that does it for us, like [Socket.IO](#) or [Sente](#) .

AWS has the IoT Device SDK , which allows MQTT over WSS (Websockets Secure). Google killed its IoT Core product.

A single Node.js server running Socket.IO could manage 55k connections.

Serverless-friendly notifications

Firestore

- Real-time updates available only in Firestore in Native mode
- Create a snapshot listener on the client
- Define security rules to restrict client-side access to the database

Firebase Cloud Messages (FCM)

- Push notifications via the browser Push API

Third-party services

- AWS SES, SendGrid, Mailgun, Postmark, etc (email)
- OneSignal, PushEngage, ntfy, etc (push notifications)
- Ably, PubNub, Pusher, etc (SSE, WebSockets)
- Twilio, AWS Pinpoint, etc (SMS)
- Telegram Bot API, Slack, WhatsApp API, etc

Cloud Tasks best practices

Make your tasks easy to identify

- Give your tasks a `name`. Use something self-explanatory. For example, `VERB-WHAT-UTC_TIMESTAMP`, like `scrape-example_com_homepage-1700834383`. Or maybe `VERB-WHAT-PERIOD`, like `generate-sales_report-2023-11`.*
- Add a custom HTTP header containing the identifier of the service that created that task. For example, `X-Task-Created-By` or `X-Task-Enqueued-By`.
- Keep in mind `tombstoneTtl` for task deduplication.

*Giving your name to task introduces a performance penalty. These costs can be magnified significantly if tasks are named sequentially, such as with timestamps.

Implement idempotent task handlers

Idempotency:

- is different from safety.
- means that multiple identical requests will have the same outcome.
- does not mean that the server has to respond in the same way on each request. If the constraint is violated, the server should return HTTP 409 Conflict.

We can implement an idempotent task handler using:

- a **constraint** from our business logic.
- **ETags** (only for existing resources, so only for PATCH requests, not for POST requests).
- an **idempotency key** (e.g. Stripe API).

Reference:

- HTTP methods: Idempotency and Safety
- Making POST and PATCH requests idempotent
- Pattern: Idempotent Consumer
- Handling Duplicate Messages (Idempotent Consumers) 

Least privilege

Follow the principle of least privilege.

Restrict queue management methods to a small set of people. Prefer using a Google Group instead of individual users.

```
gcloud projects add-iam-policy-binding PROJECT_ID \
--member group:EMAIL \
--role roles/cloudtasks.queueAdmin
```

If you have multiple queues in a project and want to limit access to individual queues, define IAM bindings at the queue level instead of the project level.

```
gcloud tasks queues add-iam-policy-binding QUEUE_NAME \
--location LOCATION \
--member serviceAccount:EMAIL \
--role roles/cloudtasks.enqueuer
```

Service perimeter

Set up a service perimeter using VPC Service Controls.

HTTP requests from a Cloud Tasks execution will be blocked when directed towards non-Cloud Functions endpoints, non-Cloud Run endpoints, and in general for any service outside of the perimeter you set up.

Service perimeters mitigate the risk of **data exfiltration**.

Test your system

Use an emulator?

No official emulator. There is [aertje/cloud-tasks-emulator](#) but I haven't tried it yet.

Integration tests

Test the integration between Cloud Tasks and your components (task creator and task handler).

Since Cloud Tasks is not a service under our control, these tests are **cross-system** integration tests and someone might argue they are not that useful.

System tests (e2e)

Test the system as a single piece, as an end user would use it.

Other tests: load testing, chaos testing

Stress test your system with an HTTP load generator like [Locust](#), [artillery](#) or [autocannon](#).

Simulate outages/issues of 3rd party services (e.g. SendGrid is down, Stripe takes too long to respond).

Cloud Tasks metrics

The Cloud Tasks API defines 4 metrics (sampled every 60 seconds):

1. `api/request_count` : count of Cloud Tasks API calls.
2. `queue/depth` : number of tasks in the queue.
3. `queue/task_attempt_count` : count of task attempts broken down by response code.
4. `queue/task_attempt_delays` : delay between each scheduled attempt time and actual attempt time.

We could also enable Cloud Tasks logging and monitor `AttemptDispatch` and `AttemptResponse`.

Task-specific metrics

If the HTTP target containing the task handler is a Cloud Run service, you could monitor:

- `container/billable_instance_time` : billable time aggregated across all container instances.
- `container/memory/utilizations` : memory utilization distribution across all container instances.

You can also define your own metrics:

- job duration: it depends on what services your job relies on (e.g. database, cache store, third-party API). The Ruby library `Cloudtasker` adds a `duration` field to each "Job done" log entry. This allows you to define a `log-based metric` and monitor it. See this article for an example.

Maybe also define SLIs, SLOs and error budgets.

Alert: queue overload

Queues or queue groups can become overloaded any time traffic increases suddenly. As a result, these queues can experience:

- Increased task creation latency
- Increased task creation error rate
- Reduced dispatch rate

To defend against this, Google recommends establishing controls in any situation where the create or dispatch rate of a queue or queue group can spike suddenly.

The 500/50/5 pattern

Google recommends a maximum of 500 operations per second to a cold queue or queue group, then increasing traffic by 50% every 5 minutes. In theory, you can grow to 740K operations per second after 90 minutes using this ramp up schedule.

Alert: target overload

Cloud Tasks can overload other services that you are using (e.g. Cloud Run, Firestore), and your network usage, if dispatches from a queue increase dramatically in a short period of time.

If a backlog of tasks has accumulated, then unpausing those queues can potentially overload these services.

The recommended defense is the same 500/50/5 pattern suggested for queue overload: if a queue dispatches more than 500 TPS, increase traffic triggered by a queue by no more than 50% every 5 minutes.

Cloud Tasks vs similar services

Cloud Tasks vs Cloud Pub/Sub (1/2)

TL;DR: push queue, explicit invocation VS pull queue, implicit invocation

Feature	Cloud Tasks	Cloud Pub/Sub
At least once delivery	✓	✓
Configurable retries	✓	✓
Task creation deduplication	✓	✗
Scheduled delivery	✓	✗
Ordered delivery	✗	✓ with ordering keys
Explicit rate controls	✓	✓ Pull subscriber clients can implement flow control
Pull via API	✗	✓
Batch insert	✗	✓
Multiple handlers/subscribers per message	✗	✓

Cloud Tasks vs Cloud Pub/Sub (2/2)

Feature	Cloud Tasks	Cloud Pub/Sub
Task/message retention	30 days	Up to 31 days
Max size of task/message	1 MB	10 MB
Max delivery rate	500 qps per queue	No upper limit
Geographic availability	Regional	Global
Maximum push handler/subscriber processing duration	30 minutes (HTTP) 10 minutes (App Engine Standard) automatic scaling 24 hours (App Engine Standard manual or basic scaling) 60 minutes (App Engine Flexible)	10 minutes for push operations
Number of queues/subscriptions per project	1,000/project, more available via quota increase request	10,000/project
Replay messages	✗	✓ with snapshots

Cloud Tasks

Trigger

A Cloud Tasks task runs immediately or according to its configured `scheduleTime`. You can schedule a task up to 30 days into the future.

Rates

A task can run up to the `maxDispatches` value of the queue it is dispatched from. The maximum value is 500 (see the queue's [rate limits config](#)).

Retries

Cloud Tasks adopts a truncated exponential backoff strategy to retry failed tasks (see the queue's [retry config](#)).

Cloud Scheduler

Trigger

A Cloud Scheduler job runs according to the `cron expression` you defined. You can schedule a job any time (?) into the future.

Rates

A job can run max once per second.

Retries

Cloud Scheduler adopts a truncated exponential backoff strategy to retry failed jobs.

Thanks!

cloud-tasks-devfest-milano-2023-slides.vercel.app 