Best practices for Cloud Storage

Skip to main content

- Technology areas
  - Overview
  - Guides
  - Reference
  - Samples
  - Resources
- Cross-product tools
- Console

- Cloud Storage

- All resources
- Pricing
- Pricing examples
- Quotas & limits
- Release notes
- Frequently asked questions
- Best practices
- Public datasets
- Partners
- DMCA policy
- Service level agreement

Best practices for Cloud Storage Stay organized with collections Save and categorize content based on your preferences.

This page contains an index of best practices for Cloud Storage. You can use the information collected here as a quick reference of what to keep in mind when building an application that uses Cloud Storage.

If you are just starting out with Cloud Storage, this page may not be the best place to start, because it does not teach you the basics of how to use Cloud Storage. If you are a new user, we suggest that you start with Discover object storage with the Google Cloud console or Discover object storage with the Google Cloud CLI.

For best practices for media workloads, see Best practices for media workloads.

Naming

See Bucket naming and Object naming for name requirements and considerations.

Traffic

- Perform a back-of-the-envelope estimation of the amount of traffic that will be sent to Cloud Storage. Specifically, think about:
    - Operations per second. How many operations per second do you expect, for both buckets and objects, and for create, update, and delete operations.
    - Bandwidth. How much data will be sent, over what timeframe? Consider using a tool like Wolfram Alpha to avoid mistakes in your calculations.
    - Cache control. Specifying the `Cache-Control` metadata on publicly accessible objects will benefit read latency on hot or frequently accessed objects. See Viewing and Editing Metadata for instructions for setting object metadata, such as `Cache-Control`.
- Design your application to minimize spikes in traffic. If there are clients of your application doing updates, spread them out throughout the day.
- When designing applications for high request rates, be aware of rate limits for certain operations. Know the bandwidth limits for certain types of egress and follow the Request Rate and Access Distribution Guidelines. Be especially aware of autoscaling and the need to gradually ramp up request rates for the best performance.
- When handling errors:
    - Make sure your application uses a retry strategy in order to avoid problems due to large traffic bursts.
    - Retry using a new connection and possibly re-resolve the domain name. This helps avoid "server stickiness", where a retry attempts to go through the same path and hits the same unhealthy component that the initial request hit.
- If your application is latency sensitive, use hedged requests. Hedged requests allow you to retry faster and cut down on tail latency. They do this while not reducing your request deadline, which could cause requests

to time out prematurely. For more information, see The Tail at Scale.

- Understand the performance level customers expect from your application. This information helps you choose a storage option and region when creating new buckets. For example, consider colocating your compute resources with your Cloud Storage buckets for analytics applications.

Locations and data storage options

See the Storage class and Bucket location topics for guidance on how to best store your data.

ACLs and access control

- Cloud Storage requests refer to buckets and objects by their names. As a result, even though ACLs prevent unauthorized third parties from operating on buckets or objects, a third party can attempt requests with bucket or object names and determine their existence by observing the error responses. It can then be possible for information in bucket or object names to be leaked. If you are concerned about the privacy of your bucket or object names, you should take appropriate precautions, such as:
    - **Choosing bucket and object names that are difficult to guess.** For example, a bucket named `mybucket-gtbytul3` is random enough that unauthorized third parties cannot feasibly guess it or enumerate other bucket names from it.
    - **Avoiding use of sensitive information as part of bucket or object names.** For example, instead of naming your bucket `mysecretproject-prodbucket`, name it `somemeaninglesscodename-prod`. In some applications, you may want to keep sensitive metadata in custom Cloud Storage headers such as `x-goog-meta`, rather than encoding the metadata in object names.
- Using groups is preferable to explicitly listing large numbers of users. Not only does it scale better, it also provides a very efficient way to update the access control for a large number of objects all at once. Lastly, it's cheaper as you don't need to make a request per-object to change the ACLs.
- Review and follow access control best practices.
- The Cloud Storage access control system includes the ability to specify that objects are publicly readable. Make sure you intend for any objects you write with this permission to be public. Once "published", data on the Internet can be copied to many places, so it's effectively impossible to regain read control over an object written with this permission.
- The Cloud Storage access control system includes the ability to specify that buckets are publicly writable. While configuring a bucket this way can be convenient for various purposes, we recommend against using this permission - it can be abused for distributing illegal content, viruses, and other malware, and the bucket owner is legally and financially responsible for the content stored in their buckets.

  If you need to make content available securely to users who don't have user accounts, we recommend you use signed URLs. For example, with signed URLs you can provide a link to an object, and your

application's customers don't need to authenticate with Cloud Storage to access the object. When you create a signed URL you control the type (read, write, delete) and duration of access.

Data uploads

- If you use XMLHttpRequest (XHR) callbacks to get progress updates, do not close and re-open the connection if you detect that progress has stalled. Doing so creates a bad positive feedback loop during times of network congestion. When the network is congested, XHR callbacks can get backlogged behind the acknowledgement (ACK/NACK) activity from the upload stream, and closing and reopening the connection when this happens uses more network capacity at exactly the time when you can least afford it.
- For upload traffic, we recommend setting reasonably long timeouts. For a good end-user experience, you can set a client-side timer that updates the client status window with a message (e.g., "network congestion") when your application hasn't received an XHR callback for a long time. Don't just close the connection and try again when this happens.
- A convenient way to reduce the bandwidth needed for each request is to enable gzip compression. Although this requires additional CPU time to extract the results, the trade-off with network costs usually makes it very worthwhile.

  An object that was uploaded in gzip format can generally be served in gzip format as well. However, avoid uploading content that has both `content-encoding: gzip` and a `content-type` that is compressed, as this may lead to unexpected behavior.
- We recommend using resumable uploads, which allow you to resume transferring data even when a communication failure has interrupted the flow of data. You can also use XML API multipart uploads to upload parts of a file in parallel, which potentially reduces the time to complete the overall upload.

Deletion of data

See Delete objects for guidelines and considerations on deleting data. You can also use features for controlling data lifecycles to help protect your data from getting erroneously deleted by your application software or users.

Last updated 2026-04-03 UTC.
