Object generation numbers enable users to uniquely identify data resources and apply preconditions to guarantee atomicity of their multi-step transactions.

Even without Object Versioning (/storage/docs/object-versioning) enabled, all Cloud Storage objects have generation numbers and meta-generation numbers. The generation number changes each time the object is overwritten, and the meta-generation number changes each time the object's metadata is updated.

Since object meta-generation numbers reset to one for each new object generation, they are meaningful only when paired with a generation number.

Buckets also maintain a meta-generation number enabling users to uniquely identify a bucket metadata state. Since buckets have no payload data, and thus no generation numbers, their meta-generation numbers are meaningful on their own.

In parallel uploads (/storage/docs/composite-objects#uploads), you divide an object into multiple pieces, upload the pieces to a temporary location simultaneously, and `compose` the original object from these temporary pieces. If an independent process inadvertently uses the same name as one or more of the temporary pieces you've uploaded, then when you attempt to `compose` the object, incorrect components get used and the object becomes corrupted.

By using generation numbers, you prevent this corruption from happening. If you include the generation number of each uploaded piece when you make your `compose` request, either the `compose` occurs with the correct pieces or the request fails with a `404 Not Found` response.

Preconditions tell Cloud Storage to only perform a request if the generation or meta-generation number of the affected object meets your precondition criteria. These checks of the generation

and meta-generation numbers ensure that the object is in the expected state, allowing you to perform safe read-modify-write updates and conditional operations on objects.

When a `match` precondition uses a specific generation or meta-generation number, the Cloud Storage object to which the request applies must have the same generation/meta-generation number. If it does, the request succeeds. If it does not, the request fails and a `412 Precondition Failed` response is returned.

When a `match` precondition uses the value `0` instead of a generation number, the request only succeeds if there are no live objects in the Cloud Storage bucket with the name specified in the request. If there is such an object, the request fails and a `412 Precondition Failed` response is returned.

Preconditions are often used in mutating requests — uploads, deletes, copies, or metadata updates — to prevent race conditions (https://en.wikipedia.org/wiki/Race_condition#File_systems). Race conditions can arise when the same request is sent repeatedly or when independent processes interfere with each other. For example, multiple request retries after a network interruption (#retry-scenario), or users performing a read-modify-write operation on the same object (#rmw-scenario) can create race conditions.

In addition to preconditions that use generation and meta-generation numbers, there are also preconditions available that use ETags. XML API ETags for non-composite objects change only when content changes while ETags for composite objects (/storage/docs/composite-objects) and JSON API resources change whenever the content or metadata changes. For more information about ETags, see Hashes and ETags: Best Practices (/storage/docs/hashes-etags#_ETags).

Preconditions come at a performance and billing cost: for each mutating operation, you also issue a billable `GET` metadata request to determine the generation/meta-generation number of the object. As a performance consideration, preconditions can potentially double the network portion of the overall operation latency by adding an extra round trip, which may be an important factor in latency-sensitive operations. As a pricing consideration, the `GET` metadata request that allows you to use preconditions is billed at a rate of $0.004 per 10,000 operations.

Depending on your application, there are ways to avoid performance and billing costs associated with using preconditions, such as:

- Storing the generation and metageneration numbers of your objects locally so that you already know the correct numbers to use in your precondition.
- Using a naming scheme that avoids more than one mutation of the same object name so that you don't need to use preconditions.
- Having application knowledge of which objects are newly created, so you already know when to use the `if-generation-match:0` precondition.
- Remembering the results of `GET` calls performed prior to mutations.

In the XML API, generation and meta-generation numbers are exposed via the `x-goog-generation` (/storage/docs/xml-api/reference-headers#xgooggeneration) and `x-goog-metageneration` (/storage/docs/xml-api/reference-headers#xgoogmetageneration) response headers. These headers are returned in the response of a `HEAD` request (/storage/docs/xml-api/head-object) for an object.

See the HTTP Headers Reference (/storage/docs/xml-api/reference-headers) for a complete listing of precondition request headers that you can use in order to make the request conditional on the state of the requested object. For example, you can:

- Use the `x-goog-if-generation-match` (/storage/docs/xml-api/reference-headers#xgoogifgenerationmatch) precondition to execute a request only if the generation number in the precondition matches the generation number of the requested object. If you use `0` instead of a generation number, the request only succeeds if there is no live object in your bucket matching the object named in the request.
- Use the `x-goog-if-metageneration-match` (/storage/docs/xml-api/reference-headers#xgoogifmetagenerationmatch) preconditions to execute a request only if the meta-generation number in the header matches the meta-generation number of the requested object.
- Use the `If-Modified-Since` (/storage/docs/xml-api/reference-headers#ifmodifiedsince) precondition with `GET` or `HEAD` requests. These requests execute only if the time of creation for the most recent generation of the object — that is, the last modification of the object — occurred more recently than the time specified in the precondition.
- Use ETags and the `If-Match` (/storage/docs/xml-api/reference-headers#ifmatch) or `If-None-Match` (/storage/docs/xml-api/reference-headers#ifnonematch) preconditions with `GET` or `HEAD`

requests. These requests execute only if the requested object does or doesn't match the ETag specified in the precondition.

- Use multiple preconditions in the same request. For example, if you are using `x-goog-if-generation-match`, you can also use `x-goog-if-metageneration-match`.

In the JSON API, you can obtain the generation and meta-generation numbers via the `generation` and `metageneration` properties of a response containing an object (/storage/docs/json_api/v1/objects) or bucket (/storage/docs/json_api/v1/buckets) resource. An object or bucket resource is returned in the response body of a `GET` request for the object (/storage/docs/json_api/v1/objects/get) or for the bucket (/storage/docs/json_api/v1/buckets/get).

To use preconditions on requests, add the preconditions as query parameters to the end of the URL. For each precondition, add a query parameter with the same name as the precondition. For example, here is a request that uses `ifGenerationMatch`:
https://storage.googleapis.com/storage/v1/b/testgrid-triage-testing/o/test?ifGenerationMatch=1122334455

In this example, the API only performs this request if the object's generation is `1122334455`.

The JSON API also supports HTTP 1.1 ETags and the corresponding HTTP `If-Match` and `If-None-Match` headers for all resources, including buckets, objects, and ACLs. An ETag is returned as part of the response header whenever a resource is returned, as well as included in the resource itself.

Here are some examples of preconditions you can use in order to make the request conditional on the state of the requested object:

- Use the `ifGenerationMatch` and `ifGenerationNotMatch` preconditions to execute a request only if the generation number in the property does/doesn't match the generation number of the requested object. If you use `0` instead of a generation number in `ifGenerationMatch`, the request only succeeds if there is no live object in your bucket matching the object named in the request.
- Use the `ifMetagenerationMatch` and `ifMetagenerationNotMatch` preconditions to execute a request on a bucket or object. The requests succeeds only if the meta-generation number in the property does/doesn't match the meta-generation number of the requested bucket or object.

- Use ETags and the `If-Match` (/storage/docs/xml-api/reference-headers#ifmatch) or `If-None-Match` (/storage/docs/xml-api/reference-headers#ifnonematch) preconditions as headers in requests. These requests execute only if the requested object does or doesn't match the ETag specified in the precondition.
- Use multiple preconditions in the same request. For example, if you are using `ifGenerationMatch` while requesting an object, you can also use `ifMetagenerationMatch`.

Note that generation and meta-generation preconditions are not accepted for ACL operations; use the access-control entry resource ETag instead. This can be found inside each access-control entry resource, which is also accessible from the containing object or bucket resource.

A common pattern for updating bucket or object metadata involves reading the current state, applying modifications locally, and sending the modified metadata back to Cloud Storage for writing. This can be precarious if two or more independent processes attempt the sequence simultaneously.

Consider the following case: you want to add an ACL entry for a collaborator so they can access your bucket. At the same time, a co-worker wants to remove a separate collaborator who no longer needs access to the bucket.

To do this, both you and your co-worker read the same initial state for the bucket's metadata, and you each make your desired modification to the ACL entries of the bucket. When you write your modifications back to Cloud Storage, the metadata is updated correctly. Unfortunately, your changes are lost as soon as your co-worker uploads his modifications, because he had no way to take into account your update. As a result, the ACL entry for your collaborator is lost, she won't be able to access your bucket, and no one is aware of what happened (without going back and looking at the ACL entries).

You and your co-worker can prevent this race condition by adding an `if-metageneration-match` precondition to each of your write operations. In the precondition, you both use the

metageneration number of the bucket, which is part of the metadata you received in the initial read operation.

When your modifications add the ACL entry, the bucket's metageneration number changes. Now that preconditions are being used, when your co-worker attempts to write his version of the ACL entries, the metageneration number of the bucket does not match the number in the precondition, and he is informed of the failed update with a response code `412 Precondition Failed`. Having received this response code, your co-worker can react accordingly, such as by performing a new read-modify-write cycle using the updated metadata.

Cloud Storage is a distributed system. Because requests can fail due to network or service conditions, Google recommends that you retry failures (/storage/sla) with exponential backoff (/storage/docs/exponential-backoff). However, due to the nature of distributed systems, sometimes these retries can cause surprising behavior.

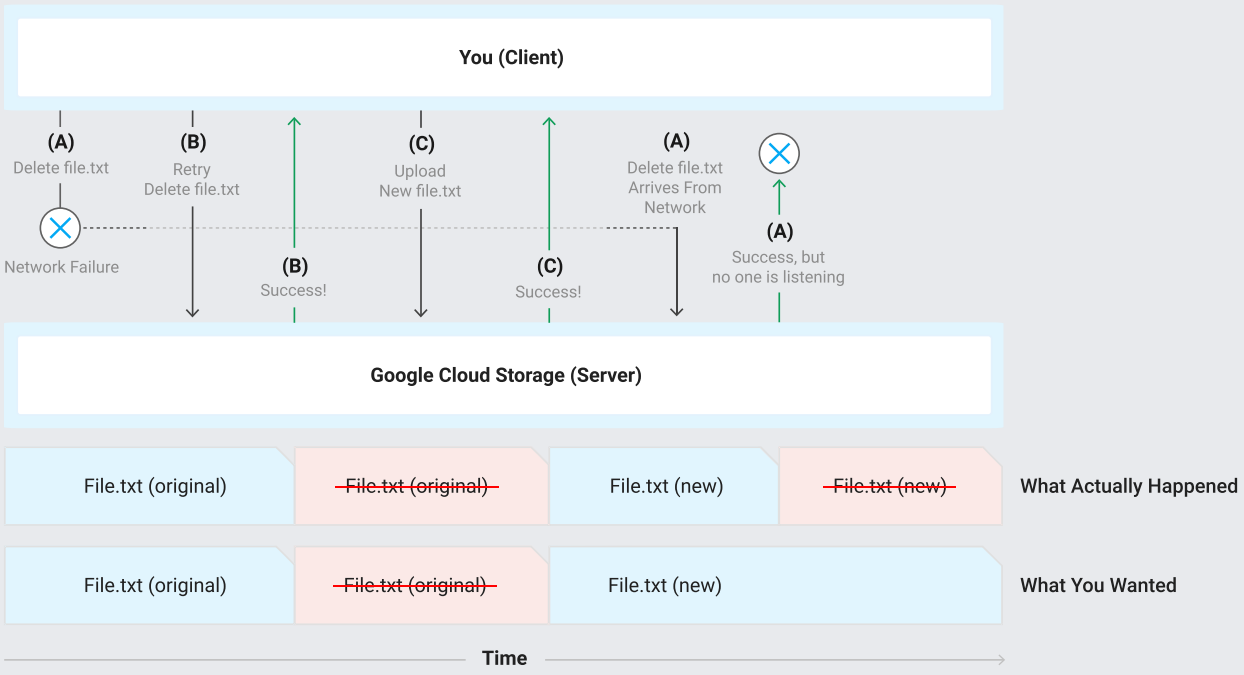
Consider the following case: you want to delete a file, `file.txt`, stored in Cloud Storage. Afterward, you want to add a new file with the same name to Cloud Storage.

To accomplish this, you issue a delete request to delete the object. However, a network condition — such as an intermediate router temporarily losing connectivity — prevents the request from reaching Cloud Storage, and you don't receive a response.

Because you didn't receive a response to the first request, you issue a second delete request for the object, which is successful, and you receive a response confirming the deletion. A minute later, you decide to upload a new `file.txt`, and your upload is successful.

A race condition arises if the router that lost connectivity subsequently regains it and sends your original, seemingly lost, delete request onward to Cloud Storage. When the request arrives at Cloud Storage, it succeeds because there is a new `file.txt` present. Cloud Storage sends a response that you do not receive because your client stopped listening for it. Not only does the new file get deleted, contrary to your intentions, but also you are not aware the second deletion occurred.

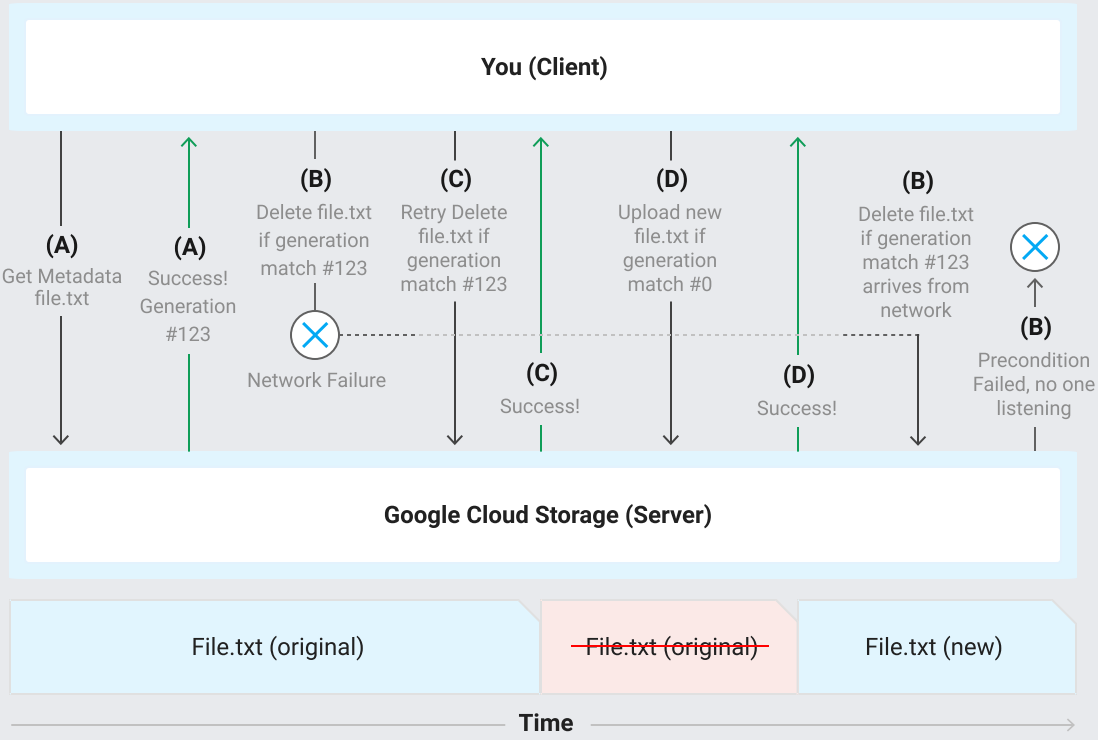
The following diagram shows what happened:



To prevent the above situation from occurring, you should begin by getting the metadata for `file.txt` in order to determine its current generation. You then send the delete request with an `if-generation-match` precondition that uses the generation number. Using the precondition ensures that *only* the object with that specific generation number gets deleted, regardless of when the delete request reaches Cloud Storage or how many times the delete request with the precondition is sent. With the `if-generation-match` precondition, any unintended attempts to mutate a different generation of `file.txt` fail with the response code `412 Precondition Failed`.

Since similar network interruptions could cause race conditions for the upload request that followed your delete request, you can avoid many of these with an `if-generation-match:0` applied to the upload request. Using this precondition ensures that retries of the upload don't accidentally write the object twice, because the precondition allows the request to succeed only if there aren't any current generations of the object.

With these preconditions in place, you protect your data from accidentally being lost when performing the delete and upload requests. This can be seen in the following diagram:



`if-generation-match:0` cannot prevent object creation from occurring twice if the first object is deleted, because the absence of the object is not uniquely identifiable. Consider the following case, in which no data is lost, but you end with a file that you didn't expect to:

1. You begin with a `GET` request for the metadata of `file.txt` to find its generation number. In the response, you find that `file.txt` does not exist.
2. Knowing this, you make a request to upload `file.txt` with the `if-generation-match:0` precondition, but the request times out when an intermediate router temporarily loses connectivity.
3. Having failed the first time, you retry your upload request, again with the `if-generation-match:0` precondition. This time the request succeeds.
4. Soon after, you send a request to delete `file.txt`, which succeeds.

5. If the router that lost connectivity now regains connectivity and sends your first upload request on to Cloud Storage, the precondition that went with the request is still a match, so `file.txt` is recreated. With or without the precondition, `file.txt` unexpectedly uploads a second time.