Cloud Storage is a key part of storing and working with Big Data on Google Cloud. Examples include:

- Loading data (/bigquery/loading-data-into-bigquery) into BigQuery.
- Using Dataproc (/dataproc/docs/), which automatically installs the HDFS-compatible Cloud Storage connector, enabling the use of Cloud Storage buckets in parallel with HDFS.
- Using a bucket to hold staging files and temporary data for Dataflow (/dataflow/docs/quickstarts) pipelines.

For Dataflow, a Cloud Storage bucket is required. For BigQuery and Dataproc, using a Cloud Storage bucket is optional but recommended.

gsutil is a command-line tool that enables you to work with Cloud Storage buckets and objects easily and robustly, in particular in big data scenarios. For example, with gsutil you can copy many files in parallel with a single command, copy large files efficiently, calculate checksums on your data, and measure performance from your local computer to Cloud Storage.

This page focuses on using gsutil for big data tasks. For a simpler introduction to gsutil, see Getting Started: Using the gsutil Tool (/storage/docs/getting-started-gsutil). Detailed documentation for all gsutil commands is available online (/storage/docs/gsutil) and in the built-in help you get from running `gsutil help`.

To get the most our of the examples shown on this page, you'll need:

- A Linux or Mac OS environment.
- Python (https://www.python.org/) version 2.7 installed.
- gsutil (/storage/docs/gsutil), which can be installed as part of the Google Cloud SDK (/sdk/docs/).
- gsutil configured (/storage/docs/gsutil_install#authenticate) to access protected data.

If you have a large number of files to upload you can use the `gsutil -m` option, to perform a parallel (multi-threaded/multi-processing) copy. To recursively copy subdirectories, use the `-R` flag of the `cp` command. For example, to copy files including subdirectories from a local directory named `top-level-dir` to a bucket, you can use:

You can use wildcards (/storage/docs/gsutil/addlhelp/WildcardNames) to match a specific set of names for an operation. For example, to copy only files that start with `image`:

You can remove files using the same wildcard:

In addition to copying local files to the cloud and vice versa, you can also copy in the cloud, for example:

gsutil automatically detects that you're moving multiple files and creates them in a new directory named `subdir2`.

If you want to synchronize a local directory with a bucket or vice versa, you can do that with the `gsutil rsync` (/storage/docs/gsutil/commands/rsync) command. For example, to make `gs://example-bucket` match the contents of the local directory `local-dir` you can use:

If you use the `rsync -d` flag, it signals gsutil to delete files at the destination (`gs://example-bucket` in the command above) that aren't present at the source (`local-dir`). You can also synchronize between two buckets.

In general, when working with big data, once your data is in the cloud it should stay there. Once your data is in Google's cloud, it's very fast to transfer it to other services like Compute Engine. Also, egress from buckets to Google Cloud services in the same location or a sub-location is free. For more information, see Network Pricing (/storage/pricing#network-pricing).

To copy a large local file to a bucket, use:

To copy a large file from an existing bucket (e.g., Cloud Storage public data (/storage/docs/public-datasets/)), use:

gsutil takes full advantage of Google Cloud Storage resumable upload and download features. For large files this is particularly important because the likelihood of a network failure at your ISP increases with the size of the data being transferred. By resuming an upload based on how many bytes the server actually received (/storage/docs/resumable-uploads), gsutil avoids unnecessarily resending bytes and ensures that the upload can eventually be completed. The same logic is applied for downloads based on the size of the local file.

If `gsutil cp` does not give you the performance you need when uploading large files, you can consider configuring composite uploads (#composite).

Typical big data tasks where you will want to configure a bucket include when you move data to a different storage class (/storage/docs/storage-classes), configure log access (/storage/docs/access-logs), configure object versioning (/storage/docs/object-versioning), or set up a lifecycle rule (/storage/docs/lifecycle).

You can list a bucket's configuration details with `gsutil ls -L -b` (/storage/docs/gsutil/commands/ls#listing-bucket-details):

In the output, notice the bucket configuration information, most of which is also configurable via gsutil:

- CORS (/storage/docs/cross-origin): controls Cross-Origin-Resource-Sharing settings for a bucket.
- Logging (/storage/docs/access-logs): allows you to log bucket usage.
- Website (/storage/docs/website-configuration): allows objects in the bucket to act as web pages or be used as static assets in a website.
- Versioning (/storage/docs/object-versioning): causes deletes on objects in the bucket to create noncurrent versions.
- Storage Class (/storage/docs/storage-classes): allows you to set the set storage class during bucket creation.
- Lifecycle (/storage/docs/lifecycle): allows periodic operations to run on the bucket - the most common is stale object deletion.

For example, suppose you only want to keep files in a particular bucket around for just one day, then you can set up the lifecycle rule for the bucket with:

Now, any objects in your bucket older than a day will automatically get deleted from this bucket. You can verify the configuration you just set with the `gsutil lifecycle` (/storage/docs/gsutil/commands/lifecycle) command (other configuration commands work in a similar fashion):

When working with big data, you will likely work on files collaboratively and you'll need to be able to give access to specific people or groups. Each object has an ACL that describes who can access it.

You can see a friendly view of an object's ACLs using the `gsutil acl` (/storage/docs/gsutil/commands/acl) command:

The entity that uploaded the object (in this case, your Google account as represented by the OAuth2 refresh token) automatically gets OWNER access to the object (see Project members and permissions (/storage/docs/projects#permissions)).

The rest of the ACL for the object is determined by the default object ACL on the bucket. This is a common point of confusion – the Bucket ACL controls access to the bucket (such as the ability to create and list objects), whereas the default object ACL controls the ACL that objects get upon creation; the two are not necessarily the same! For more information on the difference between the two, see Access Control (/storage/docs/access-control).

You can configure your bucket so that anyone with a Google account can list the files in your bucket. Note that this doesn't give them access to the data. So while users could see that `bigfile` exists in your bucket, they couldn't see its contents.

You can view the bucket's ACL with the `ls -Lb` command:

Now anyone who is authenticated with a Google account can list files in the bucket.

The following three sections cover the three common scenarios, sharing data publicly, with a group, and with a person.

For a world-readable bucket, you can configure:

For collaborators who are not members of your Google Cloud project, we recommend that you create a Google group (https://support.google.com/groups/answer/2464926?hl=en) and then add the Google group to the bucket. For example, for the gs-announce (https://groups.google.com/forum/#!forum/gs-announce) Google Group, you can configure:

For more information, see Using a Group to Control Access to Objects (/storage/docs/collaboration#group).

For many collaborators use a group. For one person, you can configure access as follows:

You can use the `gsutil du` command to display the total space used by all objects for a specified bucket. For example:

See the `gsutil du` (/storage/docs/gsutil/commands/du) command help for more options you can use, including how to return the size of all objects underneath a prefix.

You can also set up bucket logging where the total size of a bucket is automatically reported once a day. For more information, see Access Logs (/storage/docs/access-logs). If the number of objects in your bucket is large (e.g., hundreds of thousands or millions), this is a much more efficient way to track space usage. The `gsutil du` command calculates space usage by making bucket listing requests, which for large buckets can take a long time.

You can count the number of files in a bucket with:

You can clean a bucket quickly with the following command:

When performing copies, the `gsutil cp` and `gsutil rsync` commands validate that the checksum of the source file matches the checksum of the destination file. In the rare event that checksums do not match, gsutil will delete the invalid copy and print a warning message. For more information, see Checksum Validation (/storage/docs/gsutil/commands/cp#checksum-validation).

You can also use gsutil to get the checksum of a file in a bucket or calculate the checksum of a local object. For example, suppose you copy a Cloud Life Sciences public data (/life-sciences/docs/resources/public-datasets) file to your working bucket with:

Now, you can get the checksums of both the public bucket version of the file and your version of the file in your bucket to ensure they match:

Now, suppose your data is in a file at a local data center and you copied it into Cloud Storage. You can use `gsutil hash` (/storage/docs/gsutil/commands/hash) to get the checksum of your local file and then compare that with the checksum of the file you copied to a bucket. To get the checksum of a local file use:

For non-composite objects, running `gsutil ls -L` on an object in a bucket returns output like the following:

Running `gsutil hash` on a local file returns output like the following:

Both outputs have a CRC32c and MD5 value. There are no MD5 value for objects uploaded as composite objects as is the case when you configuring composite uploads (#composite) for gsutil.

You can improve the performance when uploading large files by configuring gsutil to upload each file by breaking it into parts, uploading the parts in parallel, and then using Cloud Storage's composite object feature to put the parts back together into a composite object. For more information, see Parallel Composite Uploads (/storage/docs/gsutil/commands/cp#parallel-composite-uploads). Parallel composite uploads are disabled by default.

n: Parallel composite uploads should not be used with objects you upload with a storage class of Nearline Storage, e Storage, or Archive Storage: doing so incurs early deletion charges for each composite piece.

Before configuring parallel composite uploads, you should be aware of the advantages and disadvantages. The primary advantage is that uploads for large files can be significantly faster if network and disk speed are not limiting factors. The disadvantages of parallel composite uploads are:

- When you (or your collaborators) use gsutil to download composite uploads (like those created using gsutil parallel composite uploads), we strongly recommend installing a compiled crcmod, as discussed in `gsutil help crcmod` (/storage/docs/gsutil/addlhelp/CRC32CandInstallingcrcmod). If you don't, you will see a message warning you that downloading composite objects without a compiled crcmod will run very slowly. Note that compiled crcmod is recommended for downloading, regardless of whether the parallel composite upload option is on or not.
- Composite objects in a bucket have no MD5 hash (#md5).

After you have configured and compiled crcmod following the steps in `gsutil help crcmod` (/storage/docs/gsutil/addlhelp/CRC32CandInstallingcrcmod), configure your `.boto` file so that parallel composite uploads are on by default. For more information, see gsutil config

(/storage/docs/gsutil/commands/config#additional-configuration-controllable-features), and in particular the `parallel_composite_upload_*` settings in the `GSUtil` section of the `.boto` file.

If you have enabled parallel composite uploads, and you upload a large file to your bucket, you'll notice that the file is uploaded in 50MB chunks instead of a single upload. If any failures occurred, for example, due to temporary network issues, you can re-run the copy with the no-clobber (`-n`) flag to transmit just the missing files.

Remember, that when checking the data integrity of a file in a bucket that was uploaded as a composite object, there is only a CRC32c hash value and no MD5 value.

If you configured parallel composite uploads (#composite), there can be temporary files left over from aborted uploads. If you do not wish to resume the upload, it's okay to delete these temporary files: