Cloud Storage is a highly scalable service that uses auto-scaling technology to achieve very high request rates. This page lays out guidelines for optimizing the scaling and performance that Cloud Storage provides.

Cloud Storage is a multi-tenant service, meaning that users share the same set of underlying resources. In order to make the best use of these shared resources, buckets have an initial IO capacity of around 1000 object write requests per second (including uploading, updating, and deleting objects) and 5000 object read requests per second. These initial read and write rates average to 2.5PB written and 13PB read in a month for 1MB objects. As the request rate for a given bucket grows, Cloud Storage automatically increases the IO capacity for that bucket by distributing the request load across multiple servers.

As a bucket approaches its IO capacity limit, Cloud Storage typically takes on the order of minutes to detect and accordingly redistribute the load across more servers. Consequently, if the request rate on your bucket increases faster than Cloud Storage can perform this redistribution, you may run into temporary limits, specifically higher latency and error rates. Ramping up the request rate gradually for your buckets, as described below (#ramp-up), avoids such latency and errors.

Cloud Storage supports consistent object listing (/storage/docs/consistency), which enables users to run data processing workflows easily against Cloud Storage. In order to provide consistent object listing, Cloud Storage maintains an index of object keys for each bucket. This index is stored in lexicographical order and is updated whenever objects are written to or deleted from a bucket. Adding and deleting objects whose keys all exist in a small range of the index naturally increases the chances of contention.

Cloud Storage detects such contention and automatically redistributes the load on the affected index range across multiple servers. Similar to scaling a bucket's IO capacity, when accessing a new range of the index, such as when writing objects under a new prefix, you should ramp up the request rate gradually, as described below (#ramp-up). Not doing so may result in temporarily higher latency and error rates.

The following sections provide best practices on how to ramp up the request rate, choose object keys, and distribute requests in order to avoid temporary limits on your bucket.

To ensure that Cloud Storage auto-scaling always provides the best performance, you should ramp up your request rate gradually for any bucket that hasn't had a high request rate in several weeks or that has a new range of object keys. If your request rate is less than 1000 write requests per second or 5000 read requests per second, then no ramp-up is needed. If your request rate is expected to go over these thresholds, you should start with a request rate below or near the thresholds and then double the request rate no faster than every 20 minutes.

We recommend that you run performance and scalability tests to ensure that this guideline works for your specific us
rior to ramping up your traffic in production.

If you run into any issues such as increased latency or error rates, pause your ramp-up or reduce the request rate temporarily in order to give Cloud Storage more time to scale your bucket. You should use exponential backoff (/storage/docs/exponential-backoff) to retry your requests when receiving errors with 5xx or 429 response codes from Cloud Storage.

Auto-scaling of an index range can be slowed when using sequential names, such as object keys based on a sequence of numbers or timestamp. This occurs because requests are constantly shifting to a new index range, making redistributing the load harder and less effective.

In order to maintain a high request rate, avoid using sequential names. Using completely random object names will give you the best load distribution. If you want to use sequential numbers or timestamps as part of your object names, introduce randomness to the object names by adding a hash value before the sequence number or timestamp.

For example, if the original object names you want to use are:

You can compute the MD5 hash of the original object name and add the first 6 characters of the hash as a prefix to the object name. The new object names become:

Note that the random string doesn’t necessarily need to be at the beginning of the object name. Adding a random string after a common prefix still allows auto-scaling to work, but the effect is limited to that prefix, with no consideration of the rest of the bucket.

For example:

The above naming allows for efficient auto-scaling of objects in `images/animals` and `images/landscape,` but not `images/clouds`.

As mentioned above, using a random string after a common prefix only helps with auto-scaling under that prefix. Once the requests shift to a new prefix, you may no longer benefit from the previous auto-scaling effects. This is especially a problem when the prefixes follow a sequential pattern.

For example, if you write files under a new timestamp-based prefix every hour:

Although auto-scaling helps to increase the write rate under a prefix over time, the write rate resets at the beginning of each hour. This results in a sub-optimal write rate and periodic increases in latency and error rate. If you need to write to different prefixes over time, to avoid this problem, make sure the new prefixes are evenly distributed across the entire key range.

Sometimes you'll want to perform a bulk upload or deletion of data in Cloud Storage. In both cases, you may not have control over the object names. Nevertheless, you can control the order in which the objects are uploaded or deleted to achieve the highest write or deletion rate possible.

To do so, you should distribute the uploads or deletes across multiple prefixes. For example, if you have many folders and many files under each folder to upload, a good strategy is to upload from multiple folders in parallel and randomly choose which folders and files are uploaded. Doing so allows the system to distribute the load more evenly across the entire key range, which allows you to achieve a high request rate after the initial ramp-up.