Engineer

Exponential backoff

⌛ a retry pause that multiplies after every failure

What it is

Exponential backoff is a rule for how long to wait before trying again. After each failed attempt, the wait is multiplied by a fixed factor, most often two, until it reaches a ceiling: one second, then two, four, eight and so on. The rule lives in the client, but it exists for the sake of whatever the client is calling. A server, a network path or a database that fails under too much load does not recover sooner for being asked again at once. It recovers when its callers ease off, and a wait that grows with every failure makes them ease off more the longer the trouble lasts.

The oldest and most widely deployed example is in TCP. RFC 6298, the Standards Track document from June 2011 that specifies how TCP computes its retransmission timeout, sets that timeout to one second before the first round-trip time has been measured, and says that whenever the computed value falls below one second it should be rounded up. Each time the timer expires and a segment has to be sent again, the host must double the timeout. An upper bound is allowed, as long as it is at least 60 seconds. When a fresh round-trip measurement arrives, the timeout is computed from the measurements again, which can bring it straight back down after a series of doublings. The algorithm goes back to Van Jacobson’s 1988 paper Congestion Avoidance and Control, and the RFC’s security section states what is at stake: the Internet relies on a correct retransmission timer to keep the network stable and to avoid congestion collapse.

In application code the same idea comes with a handful of settings: an initial wait, a multiplier, a maximum wait at which the growth stops, and a limit on the number of attempts or on the total time spent retrying. Written out, the wait after the nth failure is the smaller of the cap and base × multiplier to the power of n, counting from zero. The AWS analysis discussed below calls this capped

exponential backoff. The cap matters more than it looks. Doubling a one-second wait ten times gives 1,024 seconds, more than 17 minutes, which is longer than most callers will wait for anything.

The last setting is jitter, a random element in each wait. The gRPC connection backoff protocol, which describes how gRPC clients reconnect after a failed connection, starts at one second, multiplies by 1.6, stops growing at 120 seconds and applies a jitter of 0.2, so each wait is moved up or down at random by as much as 20%. It also gives every connection attempt at least 20 seconds to complete, and it resets the backoff to its initial value once the server has accepted a connection, which it takes to be the moment the SETTINGS frame arrives. The protocol allows other implementations, for example on a mobile phone that should wake up as rarely as possible, but requires implementations to spread simultaneous retries apart and to avoid a substantially higher attempt rate.

Use the concept in a decision

Put “What is exponential backoff?” to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

A retry is extra load, and it arrives at the worst moment: exactly when the thing being called is failing. If a service slows down because it is overloaded, and every caller responds by trying again at once, the load rises as capacity falls. The Google Cloud Storage documentation lists retrying without backoff as an anti-pattern for that reason, since immediate or very short retries can lead to cascading failures, in which one failure triggers others. Backoff turns the retry from an amplifier into a brake: the longer the trouble lasts, the less each client adds to it.

Backoff on its own is not enough, because clients that fail together also wait together. In Exponential Backoff And Jitter, published by AWS in March 2015, Marc Brooker simulated many clients using optimistic concurrency control to update the same row in a remote database. With every client competing in every round, the total work grew with the square of the number of clients. Capped

exponential backoff helped only a little. The calls still came in clusters, with quiet gaps in between, so contention hardly fell. Adding jitter, by drawing each wait at random between zero and the capped exponential value, removed the gaps, left an approximately constant rate of calls, and in the case with 100 contending clients cut the number of calls by more than half. Of the variants he compared, backoff without jitter was, in Brooker's words, "the clear loser". He was also explicit about the limit: none of the approaches changes the fact that the work grows with the square of the contention. Jitter reduces the waste; it does not remove the conflict. In a May 2023 update, the post notes that most AWS SDKs now include exponential backoff and jitter in their standard and adaptive retry modes.

Backoff decides when to try again, not whether it is safe to. The Cloud Storage documentation puts two questions first. One is whether the response points to a transient problem: HTTP 408, 429 and 5xx status codes, socket timeouts and TCP disconnects. The other is whether the request is idempotent. A list operation can be repeated freely. Creating a Pub/Sub notification cannot, because each successful call creates a new notification. Between the two sit conditionally idempotent operations, which become safe to retry when they carry a precondition, such as an upload that succeeds only if the object's generation number still matches the one the client expects.

Sometimes the server knows better than the client's formula. The Retry-After header field, defined in RFC 9110, tells the client how long it ought to wait before a follow-up request, either as a date or as a number of seconds, and with a 503 Service Unavailable response it says how long the service expects to be unavailable. The 429 Too Many Requests status from RFC 6585, which signals rate limiting, may carry the same field. A client with a backoff loop can use that value instead of its own computed wait, since it comes from the side that can see its own load.

What it looks like in practice

A retry loop with capped exponential backoff and full jitter fits in a few lines of Python:

```
import random
import time

class TransientError(Exception):
    """A timeout, a 408, 429 or 5xx response, or a dropped connection."""
```

```
def call_with_backoff(op, attempts=5, base=1.0, cap=30.0):
    for attempt in range(attempts):
        try:
            return op()
        except TransientError:
            if attempt == attempts - 1:
                raise
            ceiling = min(cap, base * 2**attempt)
            time.sleep(random.uniform(0, ceiling))
```

With these values, five attempts are separated by four waits whose ceilings are 1, 2, 4 and 8 seconds. The loop therefore sleeps for at most 15 seconds before it gives up, and for about 7.5 seconds on average, since each wait is drawn evenly from zero to its ceiling. Only `TransientError` is retried; an authorization failure or a malformed request raises at once. And the loop repeats whatever `op` does, so `op` has to be safe to repeat: a read, or a write that the server can recognise as a repeat.

Client libraries ship their own versions of this loop, and their defaults differ, even for the same service. For Cloud Storage, the Python client starts at one second, doubles each wait up to 60 seconds and gives up after a default deadline of 120 seconds. The C++ client also starts at one second and doubles, but it waits up to five minutes between attempts and keeps retrying for up to 15 minutes, more than seven times as long. It also retries operations that are never idempotent, unless the program opts into a strict idempotency policy. The gcloud command line uses what Google calls truncated binary exponential backoff, with up to 32 retries by default. Two programs that call the same API with the same failure can therefore report it after two minutes or after a quarter of an hour, depending on the language they are written in.

The waiting does not have to block a thread. In background work, a job that fails can go back on a queue with a later due time, and the backoff becomes a property of the job rather than of a sleeping process. That is the natural place for it whenever nobody is waiting on the answer.

In email with failover and delivery logging, from 2025, backoff sits on the asynchronous side of the system. Mail goes out through three providers, with SendGrid as the primary and SMTP2GO and Azure Communication Services behind it, and failover between them is automatic. The send path itself goes through a client with a 30-second timeout, and the asynchronous delivery path has retries and backoff, so that a short hiccup does not cost a message. The page

sums up the result: when one provider has a bad afternoon, mail takes a degraded path rather than suffering an outage. It also names the price, which is three integrations to keep working instead of one.

Where it goes wrong

Every client waits the same amount. Without jitter, clients that failed together retry together, and the server sees the same spike again after one, two, four and eight seconds. Brooker's simulation found that plain exponential backoff barely reduced contention for this reason. Add jitter: in his comparison, Full Jitter and Decorrelated Jitter both cut client work and server load substantially, and either is a sound default.

The wait has no ceiling, or the retries have no end. An uncapped doubling passes 17 minutes after ten failures, and a loop without a limit keeps a request alive long after anyone is waiting for its answer. Google lists ignoring retry limits as an anti-pattern: retrying indefinitely can exhaust the application's resources or keep sending requests to a service that will not recover without intervention. Set a maximum wait and a total deadline that suit the caller, short where latency matters and longer for batch work.

Permanent errors are retried. An authorization failure or an invalid request fails the same way on every attempt. Retrying it only delays the error, and without a limit the client can end up in an infinite loop, as the Cloud Storage documentation warns. Sort errors into transient and permanent before the loop, retry only the transient ones, such as 408, 429, 5xx and dropped connections, and log the rest with their cause.

A non-idempotent operation is retried blindly. If a request timed out after the server had already acted on it, a retry repeats the action, which Google's list of anti-patterns describes as unintended overwrites or deletions of data. Retry only idempotent operations, make writes conditional where the API allows it, and otherwise make sure the receiving side can detect a duplicate.

Retries are stacked in several layers. If the application makes three attempts and the client library under it makes three attempts for each of those attempts, one failure becomes nine attempts, which is Google's own example, and every further layer that retries on its own multiplies the count again. The

documentation adds that so many retries can lead to request throttling and higher latency without improving the success rate. Retry in one layer, usually the library, and turn retries off or down in the others.

The backoff is never reset, or nobody sees it. A client that keeps its last, long wait after a recovery sits out the maximum at the next small failure; gRPC resets to the initial backoff as soon as a connection is accepted, and TCP recomputes its timer from fresh measurements. Reset after a success. And because retries hide trouble from users, they also hide it from the team. Count them in monitoring and raise an alert when the retry rate climbs, because a dependency that needs three attempts per call is already failing.

Further reading

- RFC 6298: Computing TCP's Retransmission Timer — the Standards Track rules for TCP's retransmission timeout, including the doubling each time the timer expires.
- gRPC Connection Backoff Protocol — the reconnection algorithm, its five parameters with their values, and when the backoff is reset.
- Exponential Backoff And Jitter — Marc Brooker's 2015 simulation of contending clients, comparing no jitter with Full, Equal and Decorrelated Jitter.
- Retry strategy — what Cloud Storage considers safe to retry, the defaults of each client library and five retry anti-patterns.
- RFC 9110: HTTP Semantics — the Retry-After field, with which a server tells a client how long to wait before its next request.



Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is exponential backoff?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/exponential-backoff/