PuigFlows

# Reintentos con Backoff Exponencial y Jitter

Cómo Reintentar sin Tumbar tu Sistema

## Retries with Exponential Backoff and Jitter

How to Retry Without Taking Down Your System

Guía técnica · Edición bilingüe (Español / English)

**puigflows.com** · Actualizado el 02/07/2026

**Español**

Casi todo sistema distribuido falla de forma intermitente: un timeout de red, un 503 momentáneo mientras un servicio se reinicia, un límite de tasa que se dispara en un pico de tráfico. Reintentar es la respuesta obvia, pero un reintento mal hecho convierte un problema pequeño en una caída total. Esta guía explica cómo reintentar bien: backoff exponencial para no insistir demasiado rápido, y jitter para que miles de clientes no vuelvan a la vez.

## Cuándo reintentar tiene sentido

Un reintento solo ayuda si el fallo es transitorio: algo que probablemente funcione si lo intentas de nuevo dentro de un momento. Un timeout de conexión, un 429 (Too Many Requests) o un 503 (Service Unavailable) son buenos candidatos. Un 400 (Bad Request) o un 401 (Unauthorized) no lo son: reintentar una petición mal formada solo desperdicia recursos y nunca tendrá éxito.

## El problema del reintento ingenuo

El primer instinto suele ser un bucle que reintenta de inmediato, o que espera un intervalo fijo. Ambos fallan bajo carga. Si un servicio se cae durante dos segundos, todos los clientes que fallaron reintentan al mismo tiempo y golpean al servicio justo cuando intenta recuperarse, y lo tumban otra vez. A esto se le llama «rebaño atronador» (thundering herd).

## Backoff exponencial

La idea del backoff exponencial es esperar cada vez más entre intentos: el primer reintento espera una base corta, el siguiente el doble, luego el cuádruple, y así sucesivamente, hasta un tope (cap). Esto da al servicio un tiempo de recuperación creciente y reduce la presión total.

La fórmula del retardo en el intento n es: delay = min(cap, base · 2^n). Con base = 0.5 s y cap = 30 s, los retardos serían 0.5, 1, 2, 4, 8, 16, 30, 30…

## Por qué el backoff por sí solo no basta: el jitter

El backoff exponencial separa los reintentos de un mismo cliente, pero no separa a clientes distintos entre sí. Si 1.000 clientes fallan en el mismo segundo, todos calcularán el mismo retardo y reintentarán sincronizados, en oleadas. El jitter rompe esa sincronía añadiendo aleatoriedad al retardo.

### Full Jitter

La estrategia más simple y, para la mayoría de los casos, la mejor opción por defecto. En lugar de esperar exactamente el retardo exponencial, esperas un valor aleatorio entre cero y ese retardo: delay = random(0, min(cap, base · 2^n)). Reparte los reintentos de forma uniforme en la ventana y minimiza la carga en el servidor.

### Equal Jitter

Un punto intermedio: se conserva la mitad del retardo exponencial y se aleatoriza la otra mitad. Evita esperas demasiado cortas, a costa de algo más de sincronía que el full jitter. Útil cuando te importa mantener un retardo mínimo entre intentos.

### Decorrelated Jitter

La variante más probada a gran escala. El retardo de cada intento se basa en el del intento anterior: sleep = min(cap, random(base, sleep_anterior · 3)). Crece de forma más agresiva y descorrelaciona aún más a los clientes. AWS la recomienda para escenarios de altísima concurrencia.

## Implementación en Python

Esta función reintenta una petición HTTP aplicando full jitter, distingue los errores que merece la pena reintentar y da prioridad a la cabecera Retry-After cuando el servidor la envía.

```
import random
import time
import requests

RETRYABLE_STATUS = {429, 500, 502, 503, 504}

def retry_request(method, url, *, max_retries=5, base=0.5, cap=30.0, **kwargs):
    attempt = 0
    while True:
        try:
            resp = requests.request(method, url, timeout=10, **kwargs)
        except (requests.ConnectionError, requests.Timeout):
            if attempt >= max_retries:
                raise
            delay = full_jitter(base, cap, attempt)
        else:
            if resp.status_code not in RETRYABLE_STATUS:
                return resp
            if attempt >= max_retries:
                resp.raise_for_status()
            delay = retry_after(resp) or full_jitter(base, cap, attempt)

        time.sleep(delay)
        attempt += 1

def full_jitter(base, cap, attempt):
    expo = min(cap, base * (2 ** attempt))
    return random.uniform(0, expo)

def retry_after(resp):
    value = resp.headers.get("Retry-After")
    if value is None:
        return None
    try:
        return float(value)  # segundos; la forma HTTP-date se omite por brevedad
    except ValueError:
        return None
```

Cambiar de estrategia es tan simple como sustituir full_jitter. La versión con decorrelated jitter conserva el último retardo entre intentos:

```
def decorrelated_jitter(base, cap, prev_sleep):
    # prev_sleep = base en el primer intento
    return min(cap, random.uniform(base, prev_sleep * 3))
```

## La misma lógica en Node.js / TypeScript

```
const RETRYABLE = new Set([429, 500, 502, 503, 504]);

interface RetryOptions {
  maxRetries?: number;
  base?: number; // ms
  cap?: number;  // ms
}

const sleep = (ms) => new Promise((r) => setTimeout(r, ms));

function fullJitter(base, cap, attempt) {
  const expo = Math.min(cap, base * 2 ** attempt);
  return Math.random() * expo;
}

export async function fetchWithRetry(url, init = {}, { maxRetries = 5, base = 500, cap = 30_000 } = {}) {
  let attempt = 0;
  while (true) {
    let res;
    try {
      res = await fetch(url, init);
    } catch (err) {
      if (attempt >= maxRetries) throw err; // error de red
      await sleep(fullJitter(base, cap, attempt));
      attempt++;
      continue;
    }

    if (!RETRYABLE.has(res.status)) return res;
    if (attempt >= maxRetries) return res; // agotado

    const retryAfter = Number(res.headers.get("retry-after"));
    const delay =
      Number.isFinite(retryAfter) && retryAfter > 0
        ? retryAfter * 1000
        : fullJitter(base, cap, attempt);

    await sleep(delay);
    attempt++;
  }
}
```

## Qué errores reintentar (y cuáles no)

Reintenta: errores de red (conexión rechazada, timeout, fallo temporal de DNS), 429 (Too Many Requests) y 5xx como 500, 502, 503 y 504.

No reintentes: 4xx de cliente como 400, 401, 403, 404 y 422. El problema está en tu petición, no en el servidor; reintentar nunca lo arreglará.

Caso especial: en escrituras no idempotentes, un 5xx o un timeout no garantiza que la operación no se haya aplicado. Reintentar a ciegas puede duplicar un pago o un pedido.

## Respeta la cabecera Retry-After

Cuando un servidor devuelve 429 o 503, a menudo incluye una cabecera Retry-After que indica cuántos segundos esperar. Si está presente, úsala en lugar de tu cálculo de backoff: el servidor sabe mejor que tú cuándo estará listo. El código anterior ya da prioridad a Retry-After sobre el jitter.

## Idempotencia: el requisito que no puedes saltarte

Reintentar solo es seguro si repetir la operación produce el mismo resultado que ejecutarla una vez. Las lecturas (GET) suelen serlo por naturaleza; las escrituras no: cobrar una tarjeta dos veces es un problema real. La solución estándar es enviar una clave de idempotencia (Idempotency-Key) que el servidor usa para deduplicar reintentos. Diseña tus endpoints de escritura como idempotentes antes de añadir reintentos automáticos.

## Pon límites: presupuesto de reintentos y deadlines

Número máximo de intentos: de tres a cinco suele bastar. Más allá rara vez ayuda y solo alarga el fallo.

Tope de retardo (cap): evita esperas absurdas de varios minutos entre intentos.

Presupuesto total / deadline: limita el tiempo total dedicado a una petición, no solo el número de intentos. Un usuario esperando una respuesta no tolerará 60 segundos de reintentos.

No reintentes en cada capa: si el cliente, el gateway y el servicio reintentan a la vez, los intentos se multiplican (3 × 3 × 3 = 27). Reintenta en un solo nivel.

## A gran escala: combínalo con otros patrones

Con muchos clientes concurrentes, el jitter no basta por sí solo. Combínalo con timeouts estrictos en cada petición, un circuit breaker que deje de enviar tráfico a un servicio claramente caído, y un token bucket que limite la tasa de reintentos del propio cliente. Juntos evitan que tu flota convierta una degradación leve en una caída completa.

## Errores comunes

Reintentar errores no transitorios (4xx) y desperdiciar recursos en peticiones que nunca tendrán éxito.

Backoff sin jitter: sincroniza a todos los clientes y perpetúa el rebaño atronador.

Ignorar Retry-After y machacar a un servidor que te pidió esperar.

Reintentar escrituras no idempotentes y duplicar efectos secundarios.

No poner tope ni deadline: un cliente atrapado en reintentos eternos retiene conexiones y memoria.

Reintentos anidados en varias capas que se multiplican en silencio.

## Resumen

Reintentar bien es un equilibrio: lo suficiente para sobrevivir a fallos transitorios, no tanto como para amplificar una caída. La receta probada es backoff exponencial con full jitter, reintentar solo errores transitorios, respetar Retry-After, exigir idempotencia en las escrituras y poner límites claros de intentos y tiempo. Es poco código y un retorno enorme en fiabilidad.

## Más allá de lo básico: la guía de producción

Hasta aquí tienes la receta esencial y suficiente para el 90 % de los casos. El resto de esta guía es la parte profunda: la matemática que explica por qué el jitter funciona, una simulación con la que puedes ver la diferencia con tus propios ojos, y los patrones que rodean a un buen reintento en un sistema real: timeouts y deadlines, idempotencia

de extremo a extremo, circuit breakers, presupuestos de reintentos, reintentos en bases de datos y colas, gRPC, peticiones «hedged», observabilidad y pruebas. Si construyes servicios que deben aguantar picos y fallos parciales, esto es lo que separa un reintento que salva el día de uno que provoca la caída.

## La matemática del backoff y por qué el jitter gana

Conviene entender qué optimiza cada estrategia. Con backoff exponencial puro, el retardo del intento `n` es determinista: `base · 2^n`. Dos clientes que fallan en el mismo milisegundo calcularán exactamente el mismo retardo y volverán a chocar. El problema no es el tamaño del retardo, sino su **varianza cero**: no hay dispersión.

El jitter introduce varianza a propósito. Con **full jitter**, el retardo es una variable aleatoria uniforme en `[0, base·2^n]`. Su valor esperado es la mitad del retardo determinista ( `base·2^n / 2` ), así que de media reintentas antes, pero repartido de forma uniforme por toda la ventana. Ese reparto es justo lo que rompe las oleadas: en lugar de mil peticiones en el mismo instante, tienes mil peticiones esparcidas por varios segundos.

La pregunta práctica es cuánto trabajo extra genera cada estrategia. Marc Brooker lo midió en el artículo de referencia de AWS con una simulación de clientes compitiendo por un recurso. Sus conclusiones, que siguen vigentes y guiando el diseño de los SDK de AWS, son claras: cualquier forma de jitter reduce drásticamente el número de llamadas frente a no usarlo; entre las variantes, **equal jitter** es la peor de las tres (hace algo más de trabajo que full jitter y tarda bastante más); y la elección real está entre **full jitter** (menos llamadas, algo más de tiempo total) y **decorrelated jitter** (algo más de llamadas, menos tiempo). Para la inmensa mayoría de los servicios, full jitter es el valor por defecto correcto.

La tabla siguiente muestra la ventana de espera de cada estrategia en los primeros intentos, con `base = 0.5 s` y `cap = 30 s` . Fíjate en que en el backoff puro el valor es fijo, mientras que en full jitter es un rango que empieza en cero.

- Intento 0 → puro: 0.5 s | full jitter: 0–0.5 s
- Intento 1 → puro: 1 s | full jitter: 0–1 s
- Intento 2 → puro: 2 s | full jitter: 0–2 s
- Intento 3 → puro: 4 s | full jitter: 0–4 s
- Intento 4 → puro: 8 s | full jitter: 0–8 s
- Intento 5 → puro: 16 s | full jitter: 0–16 s
- Intento 6+ → ambos topados a 30 s (con full jitter, 0–30 s)

## Simulación: compara las estrategias con números

La teoría se entiende mejor cuando la ejecutas. Este pequeño simulador modela `N` clientes que fallan a la vez contra un servicio con capacidad limitada por segundo y mide dos cosas: cuántos intentos totales hizo falta y cuánto tardó el último cliente en tener éxito. Ejecútalo cambiando la estrategia y verás por qué el jitter importa.

```
import random, heapq

def simulate(strategy, n_clients=1000, capacity_per_sec=200,
             base=0.5, cap=30.0, seed=1):
    """Devuelve (intentos_totales, tiempo_hasta_completar) para una estrategia."""
    rng = random.Random(seed)
    # Cada cliente empieza queriendo reintentar en t=0
    heap = [(0.0, i, 0) for i in range(n_clients)]  # (instante, cliente, intento)
    heapq.heapify(heap)
    done = set()
    attempts = 0
    # "tokens" de capacidad por segundo: cuántas peticiones acepta el server cada segundo
    served_in_bucket, current_bucket = 0, 0
    last_success_t = 0.0

    def next_delay(attempt, prev):
        expo = min(cap, base * (2 ** attempt))
        if strategy == "none":          # sin backoff: reintenta cada 1s fijo
            return 1.0
        if strategy == "backoff":       # exponencial sin jitter
            return expo
        if strategy == "full":          # full jitter
            return rng.uniform(0, expo)
        if strategy == "equal":         # equal jitter
            return expo / 2 + rng.uniform(0, expo / 2)
        if strategy == "decorrelated":  # decorrelated jitter
            return min(cap, rng.uniform(base, (prev or base) * 3))

    prev_delay = {}
    while heap:
        t, client, attempt = heapq.heappop(heap)
        if client in done:
            continue
        bucket = int(t)
        if bucket != current_bucket:
            current_bucket, served_in_bucket = bucket, 0
        attempts += 1
        if served_in_bucket < capacity_per_sec:
            served_in_bucket += 1         # el server acepta la petición -> éxito
            done.add(client)
            last_success_t = max(last_success_t, t)
        else:
            d = next_delay(attempt, prev_delay.get(client))
            prev_delay[client] = d
            heapq.heappush(heap, (t + d, client, attempt + 1))
    return attempts, round(last_success_t, 1)

for s in ["none", "backoff", "full", "equal", "decorrelated"]:
    total, t_end = simulate(s)
    print(f"{s:>13}: {total:>6} intentos, completado en {t_end:>5}s")
```

Al ejecutarlo verás el patrón que describe AWS: `none` y `backoff` (sin jitter) generan muchos más intentos porque los clientes siguen sincronizados y saturan el mismo segundo una y otra vez; las tres variantes con jitter reducen los intentos totales de forma notable. Es un modelo simplificado —no sustituye a una prueba de carga real— pero deja clara la intuición: el jitter convierte una avalancha en un goteo que el servidor puede absorber.

## Parsear Retry-After correctamente

La cabecera `Retry-After` tiene **dos formatos** según el estándar HTTP: un número entero de segundos ( `Retry-After: 120` ) o una fecha HTTP absoluta ( `Retry-After: Wed, 21 Oct 2026 07:28:00 GMT` ). Casi todo el mundo maneja el primero y olvida el segundo, lo que provoca que un valor de fecha se ignore en silencio. Además conviene **acotar** el valor: un servidor mal configurado podría pedirte esperar horas, y no quieres bloquear una petición de usuario tanto tiempo.

```
from datetime import datetime, timezone
from email.utils import parsedate_to_datetime

def parse_retry_after(value, *, max_wait=300.0):
    """Convierte una cabecera Retry-After en segundos, o None si no es válida.
    Acepta el formato de segundos y el de fecha HTTP. Acota a max_wait."""
    if not value:
        return None
    value = value.strip()
    # Formato 1: delta-seconds
    if value.isdigit():
        return min(float(value), max_wait)
    # Formato 2: HTTP-date
    try:
        when = parsedate_to_datetime(value)
        if when.tzinfo is None:
            when = when.replace(tzinfo=timezone.utc)
        delta = (when - datetime.now(timezone.utc)).total_seconds()
        return min(max(delta, 0.0), max_wait)
    except (TypeError, ValueError):
        return None
```

La misma lógica en TypeScript, lista para usar en el navegador o en Node:

```
function parseRetryAfter(value: string | null, maxWait = 300): number | null {
  if (!value) return null;
  const v = value.trim();
  // Formato de segundos
  if (/^\d+$/.test(v)) return Math.min(Number(v), maxWait);
  // Formato de fecha HTTP
  const when = Date.parse(v);
  if (Number.isNaN(when)) return null;
  const delta = (when - Date.now()) / 1000;
  return Math.min(Math.max(delta, 0), maxWait);
}
```

Regla de oro: si `Retry-After` está presente, **gana siempre** sobre tu cálculo de backoff. El servidor sabe mejor que tú cuándo estará listo. Tu backoff con jitter es solo el plan B para cuando no te dice nada.

## Timeouts y deadlines: el compañero obligatorio del reintento

Reintentar sin **timeouts** es peligroso: una petición que se queda colgada para siempre nunca llega a reintentarse, retiene una conexión y consume memoria. Y reintentar sin un **deadline global** es igual de malo: cinco intentos con backoff pueden sumar más de un minuto, y ningún usuario espera tanto. La regla es tener dos límites: un timeout **por intento** (cada petición individual) y un presupuesto **total** para toda la operación.

En Python con `httpx` , cada intento lleva su propio timeout y un reloj global corta el bucle:

```
import time, random, httpx

def request_with_deadline(client, method, url, *,
                          total_deadline=8.0, per_try_timeout=2.0,
                          base=0.3, cap=5.0, max_retries=5, **kwargs):
    start = time.monotonic()
    attempt = 0
    while True:
        remaining = total_deadline - (time.monotonic() - start)
        if remaining <= 0:
            raise TimeoutError("agotado el presupuesto total de la operación")
        timeout = min(per_try_timeout, remaining)
        try:
            resp = client.request(method, url, timeout=timeout, **kwargs)
            if resp.status_code not in {429, 500, 502, 503, 504}:
                return resp
        except (httpx.ConnectError, httpx.TimeoutException):
            if attempt >= max_retries:
                raise
        if attempt >= max_retries:
            return resp
        # No esperes más de lo que queda del presupuesto
        delay = min(random.uniform(0, min(cap, base * 2 ** attempt)),
                    max(0.0, total_deadline - (time.monotonic() - start)))
        time.sleep(delay)
        attempt += 1
```

En el navegador, `AbortController` cancela cada intento y el presupuesto restante se recalcula en cada vuelta:

```
async function withDeadline(url, init = {}, { totalDeadline = 8000, perTry = 2000 } = {}) {
  const start = Date.now();
  let attempt = 0;
  while (true) {
    const remaining = totalDeadline - (Date.now() - start);
    if (remaining <= 0) throw new Error("deadline agotado");
    const ctrl = new AbortController();
    const t = setTimeout(() => ctrl.abort(), Math.min(perTry, remaining));
    try {
      const res = await fetch(url, { ...init, signal: ctrl.signal });
      if (![429, 500, 502, 503, 504].includes(res.status)) return res;
    } catch (err) {
      if (attempt >= 5) throw err;
    } finally {
      clearTimeout(t);
    }
    const backoff = Math.random() * Math.min(5000, 300 * 2 ** attempt);
    await new Promise(r => setTimeout(r, Math.min(backoff, totalDeadline - (Date.now() - start))));
    attempt++;
  }
}
```

Un principio que se olvida a menudo: el deadline debe **propagarse** entre servicios. Si tu API recibe una petición con un presupuesto de 8 segundos y llama a un servicio interno, ese servicio no debería reintentar durante 30 segundos. Pasa el tiempo restante como una cabecera o un campo de contexto y que cada capa lo respete.

## Idempotencia de extremo a extremo con Idempotency-Key

Ya vimos que reintentar escrituras es peligroso porque un timeout no te dice si la operación llegó a aplicarse. La solución estándar de la industria —la que usan Stripe, PayPal y casi cualquier API de pagos— es la **clave de idempotencia**. El cliente genera un identificador único por operación y lo envía en la cabecera `Idempotency-Key`. El servidor guarda el resultado asociado a esa clave; si llega un reintento con la misma clave, devuelve el resultado almacenado en lugar de ejecutar la operación otra vez.

En el cliente basta con generar la clave una sola vez y reutilizarla en todos los reintentos de la misma operación lógica:

```
import uuid, requests

def create_charge(amount, currency, *, session):
    key = str(uuid.uuid4())           # una clave por operacion, NO por intento
    headers = {"Idempotency-Key": key}
    # retry_request es la funcion con backoff+jitter de la primera parte
    return retry_request("POST", "https://api.pagos.com/charges",
                         json={"amount": amount, "currency": currency},
                         headers=headers, session=session)
```

En el servidor, el patrón mínimo con FastAPI y una tabla de idempotencia. La clave está en usar una restricción de unicidad (o un `INSERT ... ON CONFLICT`) para que dos reintentos concurrentes no se ejecuten a la vez:

```
from fastapi import FastAPI, Header, HTTPException
import json, psycopg

app = FastAPI()

@app.post("/charges")
def create_charge(payload: dict, idempotency_key: str = Header(...)):
    with psycopg.connect("dbname=app") as conn, conn.cursor() as cur:
        # 1) Reserva la clave de forma atomica. Si ya existe, no inserta.
        cur.execute(
            """INSERT INTO idempotency_keys (key, status)
               VALUES (%s, 'in_progress')
               ON CONFLICT (key) DO NOTHING""",
            (idempotency_key,),
        )
        if cur.rowcount == 0:
            # La clave ya existia: devuelve el resultado guardado (o 409 si sigue en curso)
            cur.execute("SELECT status, response FROM idempotency_keys WHERE key = %s",
                        (idempotency_key,))
            status, response = cur.fetchone()
            if status == "in_progress":
                raise HTTPException(409, "operacion en curso, reintenta luego")
            return json.loads(response)

        # 2) Primera vez: ejecuta el efecto real (cobro) y guarda la respuesta
        result = charge_the_card(payload)   # tu logica de negocio
        cur.execute(
            "UPDATE idempotency_keys SET status='done', response=%s WHERE key=%s",
            (json.dumps(result), idempotency_key),
        )
        conn.commit()
        return result
```

Detalles que importan en producción: guarda también el **hash del cuerpo** de la petición junto a la clave y rechaza (con un 422) si llega la misma clave con un cuerpo distinto —eso indica un bug del cliente, no un reintento legítimo. Ponle una **caducidad** a las claves (24–72 horas suele bastar) para no acumularlas para siempre. Y decide qué hacer con las peticiones concurrentes: devolver 409 mientras la primera está en curso es lo más simple y seguro.

## Circuit breaker: deja de golpear a un servicio caído

El reintento asume que el fallo es puntual. Pero si un servicio lleva treinta segundos caído, seguir reintentando —aunque sea con backoff— solo desperdicia recursos y retrasa el fallo que el usuario acabará viendo igualmente. El **circuit breaker** (cortacircuitos) resuelve esto: cuando detecta demasiados fallos seguidos, «abre el circuito» y hace fallar las peticiones al instante, sin ni siquiera intentarlas, dando tiempo al servicio a recuperarse.

Tiene tres estados. En **closed** (cerrado) todo pasa con normalidad y se cuentan los fallos. Si superan un umbral, pasa a **open** (abierto): durante un tiempo de enfriamiento, toda petición falla de inmediato. Cumplido ese tiempo pasa a **half-open** (semiabierto): deja pasar una petición de prueba; si tiene éxito, vuelve a cerrado; si falla, vuelve a abierto. Una implementación compacta en Python:

```
import time

class CircuitBreaker:
    def __init__(self, fail_max=5, reset_timeout=30.0):
        self.fail_max = fail_max
        self.reset_timeout = reset_timeout
        self.failures = 0
        self.state = "closed"
        self.opened_at = 0.0

    def call(self, fn, *args, **kwargs):
        if self.state == "open":
            if time.monotonic() - self.opened_at >= self.reset_timeout:
                self.state = "half-open"      # toca probar de nuevo
            else:
                raise RuntimeError("circuito abierto: fallo inmediato")
        try:
            result = fn(*args, **kwargs)
        except Exception:
            self._on_failure()
            raise
        self._on_success()
        return result

    def _on_success(self):
        self.failures = 0
        self.state = "closed"

    def _on_failure(self):
        self.failures += 1
        if self.state == "half-open" or self.failures >= self.fail_max:
            self.state = "open"
            self.opened_at = time.monotonic()
```

Cómo se combinan reintento y cortacircuitos: el reintento vive **dentro** del estado cerrado (reintenta los fallos transitorios normales), y el cortacircuitos lo envuelve para cortar de raíz cuando el fallo deja de ser transitorio y pasa a ser una caída. Juntos evitan tanto el fallo por un hipo momentáneo como la tormenta de reintentos contra un servicio muerto.

## Presupuesto de reintentos: la lección de Google SRE

Aquí está el error más caro y menos evidente: los reintentos se **multiplican** entre capas. Si el cliente reintenta 3 veces, el gateway 3 veces y el servicio interno 3 veces, un solo fallo del fondo puede generar 3 × 3 × 3 = 27 peticiones. Bajo carga, esto convierte una degradación leve en un colapso total. Google lo describe en su libro de SRE y propone dos límites complementarios.

El primero es un **tope por petición**: hasta tres intentos y, si fallan, el error sube al llamador sin más. El segundo, más potente, es un **presupuesto de reintentos** por cliente: cada cliente lleva la cuenta de qué fracción de su tráfico son reintentos y solo reintenta mientras esa fracción esté por debajo de un umbral —Google usa el 10 %—. Con ese presupuesto, el crecimiento de tráfico durante un incidente se limita a apenas 1,1× en lugar de multiplicarse. Una implementación con «cubo» de presupuesto:

```
import threading, time

class RetryBudget:
    """Permite reintentar solo si los reintentos recientes son < ratio del trafico.
    Aproxima el patron de Google SRE (ratio del 10%)."""
    def __init__(self, ratio=0.1, ttl=10.0):
        self.ratio = ratio
        self.ttl = ttl
        self.requests = []   # timestamps de peticiones normales
        self.retries = []    # timestamps de reintentos
        self.lock = threading.Lock()

    def _prune(self, now):
        cutoff = now - self.ttl
        self.requests = [t for t in self.requests if t > cutoff]
        self.retries = [t for t in self.retries if t > cutoff]

    def record_request(self):
        with self.lock:
            self.requests.append(time.monotonic())

    def can_retry(self):
        with self.lock:
            now = time.monotonic()
            self._prune(now)
            allowed = self.ratio * max(len(self.requests), 1)
            if len(self.retries) < allowed:
                self.retries.append(now)
                return True
            return False
```

Antes de cada reintento, consultas `budget.can_retry()`. Si el servicio de fondo está caído, el ratio de reintentos se dispara enseguida y el presupuesto empieza a denegar, cortando la amplificación en el punto exacto donde nace. Regla práctica: **reintenta en una sola capa**, idealmente la más cercana al usuario, y deja que las demás propaguen el fallo.

## Reintentos en la base de datos: fallos de serialización y deadlocks

Los reintentos no son solo cosa de HTTP. Con los niveles de aislamiento **Repeatable Read** o **Serializable**, PostgreSQL puede abortar una transacción si detecta un conflicto de concurrencia que no puede resolver de forma segura. La documentación de PostgreSQL 18 es explícita: las aplicaciones que usan estos niveles **deben estar preparadas para reintentar** las transacciones que fallen por serialización. Los dos códigos de error a capturar son `40001` ( `serialization_failure` ) y `40P01` ( `deadlock_detected` ).

El patrón es un bucle de reintento alrededor de toda la transacción, con backoff y jitter, que hace rollback y vuelve a empezar. Nunca reintentes una única sentencia dentro de la misma transacción abortada: hay que rehacer la transacción completa.

```
import time, random, psycopg
from psycopg import errors

RETRYABLE_SQLSTATE = {"40001", "40P01"}   # serialization_failure, deadlock_detected

def run_tx(dsn, work, *, max_retries=5, base=0.05, cap=2.0):
    for attempt in range(max_retries + 1):
        try:
            with psycopg.connect(dsn) as conn:
                conn.isolation_level = psycopg.IsolationLevel.SERIALIZABLE
                with conn.cursor() as cur:
                    result = work(cur)   # tu logica: SELECT/UPDATE/INSERT...
                conn.commit()
                return result
        except errors.Error as e:
            if e.sqlstate in RETRYABLE_SQLSTATE and attempt < max_retries:
                time.sleep(random.uniform(0, min(cap, base * 2 ** attempt)))
                continue
            raise
```

Consejos que reducen la frecuencia de estos fallos: mantén las transacciones **cortas** (menos ventana de conflicto), accede a las filas en un **orden consistente** en todo el código (evita deadlocks circulares), y cuando sepas de antemano qué filas vas a modificar, considera `SELECT ... FOR UPDATE` para bloquearlas por adelantado: los conflictos se convierten en esperas en vez de en abortos.

## Reintentos en colas de mensajes

En sistemas basados en eventos, el reintento suele ser responsabilidad de la infraestructura de colas, no de tu código. Si tu consumidor procesa un mensaje y falla (o tarda demasiado), el broker lo vuelve a entregar. Los mecanismos clave son el **visibility timeout** (el tiempo que un mensaje queda oculto tras ser recibido; si no lo confirmas a tiempo, reaparece), el **contador de reintentos** y la **cola de mensajes muertos** o DLQ, donde acaban los mensajes que fallaron demasiadas veces para inspeccionarlos sin bloquear la cola principal.

La consecuencia más importante para tu código: como el mensaje puede entregarse **más de una vez**, tu consumidor debe ser **idempotente**. La forma habitual es registrar los identificadores de mensaje ya procesados y saltarse los repetidos:

```
def handle_message(msg, *, seen, process):
    if msg.id in seen:            # ya procesado en una entrega anterior
        msg.ack()
        return
    try:
        process(msg.body)         # efecto de negocio (idempotente a ser posible)
        seen.add(msg.id)          # persiste esto en un store durable, no en memoria
        msg.ack()                 # confirma: no se volvera a entregar
    except TransientError:
        msg.nack()                # devuelve a la cola; el broker aplicara backoff/DLQ
```

Configura un **máximo de reentregas** (por ejemplo 5) antes de mandar el mensaje a la DLQ. Un mensaje que falla infinitamente —un «mensaje veneno»— puede atascar toda la cola si no le pones ese límite. Y vigila la DLQ con una alerta: es tu señal de que algo se está rompiendo de forma sistemática.

## Reintentos en gRPC sin escribir código

Si usas gRPC, no hace falta que programes el bucle de reintentos a mano: el propio cliente lo hace si le pasas una **service config** con una política de reintentos. Defines los intentos máximos, el backoff y qué códigos de estado son reintentables, y gRPC aplica backoff exponencial con un jitter de ±20 % automáticamente.

```
{
  "methodConfig": [{
    "name": [{"service": "pagos.PagoService"}],
    "retryPolicy": {
      "maxAttempts": 4,
      "initialBackoff": "0.1s",
      "maxBackoff": "2s",
      "backoffMultiplier": 2,
      "retryableStatusCodes": ["UNAVAILABLE", "RESOURCE_EXHAUSTED"]
    }
  }]
}
```

`maxAttempts` incluye el intento original y debe ser mayor que 1. Elige con cuidado los `retryableStatusCodes`: `UNAVAILABLE` (servidor caído o reiniciando) es el candidato clásico; `RESOURCE_EXHAUSTED` suele equivaler a un 429. Nunca marques como reintentable un `INVALID_ARGUMENT` o un `UNAUTHENTICATED`. gRPC también ofrece una política de **hedging** como alternativa, que es justo el siguiente patrón.

## Peticiones «hedged»: adelántate a la cola larga

A veces el problema no es un fallo, sino la **latencia de cola**: el 99 % de las peticiones responde rápido, pero un 1 % se queda atascado y arruina tu p99. El patrón de peticiones **hedged** (o «backup requests»), descrito por Google en «The Tail at Scale», ataca esto: si una petición no ha respondido en, digamos, el percentil 95 de latencia, envías una **segunda petición en paralelo** a otra réplica y te quedas con la primera respuesta que llegue, cancelando la otra.

```
import asyncio

async def hedged_get(send, *, hedge_after=0.2):
    """Lanza una peticion; si tarda mas de hedge_after, lanza una copia y
    devuelve la primera que responda."""
    first = asyncio.create_task(send())
    done, pending = await asyncio.wait({first}, timeout=hedge_after)
    if first in done:
        return first.result()
    second = asyncio.create_task(send())     # peticion de respaldo
    done, pending = await asyncio.wait({first, second},
                                       return_when=asyncio.FIRST_COMPLETED)
    for t in pending:
        t.cancel()
    return next(iter(done)).result()
```

Dos advertencias imprescindibles: el hedging solo es seguro para operaciones **idempotentes** (lecturas o escrituras con clave de idempotencia), porque duplicas la petición a propósito; y debes **acotar** cuántas peticiones de respaldo permites (por ejemplo, un presupuesto del 5 % del tráfico), o en un incidente el hedging duplicará la carga justo cuando menos te conviene.

## No reinventes la rueda: librerías probadas

Todo lo anterior te da el modelo mental, pero en producción conviene apoyarse en librerías maduras y bien probadas en lugar de mantener tu propio bucle. Algunas referencias por ecosistema: en Python, `tenacity` (reintentos declarativos) y el adaptador `Retry` de `urllib3` / `requests`; en .NET, `Polly`; en Java, `Resilience4j`; en JavaScript, `p-retry` y `cockatiel` (que además trae circuit breaker). Un ejemplo con tenacity muestra lo conciso que queda:

```
from tenacity import (retry, stop_after_attempt, wait_random_exponential,
                      retry_if_exception_type)
import requests

@retry(
    stop=stop_after_attempt(5),
    wait=wait_random_exponential(multiplier=0.5, max=30),   # full jitter integrado
    retry=retry_if_exception_type((requests.ConnectionError, requests.Timeout)),
    reraise=True,
)
def fetch(url):
    resp = requests.get(url, timeout=10)
    resp.raise_for_status()
    return resp
```

Estas librerías ya resuelven detalles fáciles de equivocar: el jitter, la composición con circuit breakers, la observabilidad y los casos límite. Escribe tu propio bucle solo cuando necesites un comportamiento que la librería no cubra.

## Observabilidad: si no lo mides, no lo controlas

Los reintentos son silenciosos por diseño: absorben fallos para que el usuario no los vea. El peligro es que también ocultan que algo va mal hasta que es demasiado tarde. Por eso hay que instrumentarlos. Como mínimo, exporta estas métricas: número de intentos por operación, tasa de reintentos (reintentos / peticiones), tasa de éxito tras reintentar, número de operaciones que agotaron los intentos, y un histograma de los retardos aplicados.

- **Reintentos agotados**: si crece, un servicio del que dependes está degradado; debería disparar una alerta.
- **Tasa de reintentos** acercándose a tu presupuesto (p. ej. el 10 %): señal temprana de saturación, antes de que haya errores visibles.
- **Trazas distribuidas**: marca cada reintento como un span hijo con su número de intento, para ver en qué capa nacen los reintentos.
- **Logs con contexto**: incluye el número de intento y el motivo (código de estado o excepción) en cada reintento, nunca un log mudo.

Una alerta especialmente útil: dispara cuando la tasa de reintentos supere un umbral durante varios minutos. Suele avisarte de un problema aguas abajo **antes** de que tus usuarios empiecen a ver errores, porque los reintentos todavía los están tapando.

## Cómo probar tu lógica de reintentos

El código de reintentos es difícil de probar precisamente porque involucra tiempo y azar. La clave es hacerlo **determinista**: fija la semilla del generador aleatorio, controla el reloj y simula las respuestas del servidor. Con pytest y `responses` (o `httpx.MockTransport`) puedes verificar que reintentas lo que debes y no lo que no debes:

```
import responses, requests, pytest

@responses.activate
def test_reintenta_503_y_luego_tiene_exito(monkeypatch):
    monkeypatch.setattr("time.sleep", lambda s: None)   # no esperes de verdad
    responses.add(responses.GET, "https://api.test/x", status=503)
    responses.add(responses.GET, "https://api.test/x", status=503)
    responses.add(responses.GET, "https://api.test/x", json={"ok": True}, status=200)

    resp = retry_request("GET", "https://api.test/x", session=requests.Session())

    assert resp.status_code == 200
    assert len(responses.calls) == 3        # dos 503 + un exito

@responses.activate
def test_no_reintenta_400(monkeypatch):
    monkeypatch.setattr("time.sleep", lambda s: None)
    responses.add(responses.GET, "https://api.test/y", status=400)
    resp = retry_request("GET", "https://api.test/y", session=requests.Session())
    assert resp.status_code == 400
    assert len(responses.calls) == 1        # no debe reintentar un 4xx
```

Para pruebas de integración más realistas, herramientas como `toxiproxy` inyectan latencia, timeouts y cortes de red entre tu aplicación y sus dependencias, y te permiten comprobar que tu backoff, tus deadlines y tu circuit breaker se comportan como esperas bajo fallos de verdad. Probar los caminos de error es tan importante como probar el camino feliz: es justo cuando más te juegas.

## Elegir bien los parámetros: base, tope y número de intentos

Los valores por defecto que copias de un ejemplo rara vez son los correctos para tu caso. Los tres parámetros que gobiernan un reintento —la base, el tope y el número de intentos— dependen de qué estás protegiendo y de cuánta paciencia tiene quien espera la respuesta.

La **base** marca el primer retardo. Para llamadas interactivas, donde un humano espera al otro lado, empieza pequeña (100–300 ms): un primer reintento casi inmediato suele resolver un hipo momentáneo sin que el usuario lo note. Para trabajos en segundo plano, donde nadie mira la pantalla, puedes permitirte una base mayor (1–2 s) y ser más amable con el servicio de fondo.

El **tope** (cap) evita que el backoff exponencial crezca hasta esperas absurdas. Sin él, el octavo intento con base de 0,5 s esperaría más de dos minutos. Un tope de 20–30 s es razonable para procesos de fondo; para peticiones interactivas suele bastar con 2–5 s, porque más allá de eso es mejor fallar y dejar que el usuario reintente.

El **número de intentos** casi nunca debe pasar de cinco. La probabilidad de que un fallo transitorio persista tras tres o cuatro reintentos con backoff es baja; a partir de ahí solo alargas la agonía y retienes recursos. Y recuerda que el número de intentos y el deadline total son límites **independientes**: el que se cumpla primero manda. Un buen punto de partida para una API interactiva es base 200 ms, tope 3 s, máximo 4 intentos y deadline total de 8 s.

Una tabla mental útil para decidir:

- **Interactivo (usuario esperando)**: base 100–300 ms, tope 2–5 s, 3–4 intentos, deadline 5–10 s.
- **Trabajo de fondo / colas**: base 1–2 s, tope 20–60 s, 5+ intentos, deadline de minutos.
- **Arranque de servicios (esperar dependencias)**: base 0,5 s, tope 10 s, muchos intentos, deadline generoso.

## El reintento desde el frontend: la parte que ve el usuario

Casi todo lo anterior mira al servidor, pero el navegador también reintenta, y ahí las reglas cambian porque hay una persona mirando. Reintentar en silencio unas cuantas veces está bien para un fallo de red pasajero, pero si el problema persiste, esconderlo con reintentos infinitos solo genera una rueda girando eterna y un usuario frustrado. La buena práctica es combinar unos pocos reintentos automáticos con backoff y, si se agotan, dar el control a la persona.

- **Reintento automático breve**: 2–3 intentos con backoff corto para tapar cortes de red momentáneos, sin molestar al usuario.
- **Reintento manual explícito**: si fallan, muestra un mensaje claro («¿Sin conexión? Reintentar») con un botón. Devolver el control reduce la ansiedad y evita el bucle infinito.
- **Respeta el estado offline**: si `navigator.onLine` es falso, no machaques la red; espera al evento `online` y reintenta entonces.
- **UI optimista con reversión**: muestra el cambio de inmediato, reintenta en segundo plano y, si al final fracasa, revierte visualmente y avisa. Nunca dejes al usuario creyendo que algo se guardó cuando no fue así.

Un patrón concreto: escuchar la reconexión para reintentar las peticiones que quedaron en cola mientras no había red.

```
const pending = [];   // peticiones que fallaron por falta de red

async function sendResilient(task) {
  try {
    return await task();
  } catch (err) {
    if (!navigator.onLine) {
      pending.push(task);          // guardala hasta que vuelva la conexion
      return;
    }
    throw err;                     // otro tipo de error: que lo maneje el llamador
  }
}

window.addEventListener("online", async () => {
  const queued = pending.splice(0);      // vacia la cola
  for (const task of queued) {
    try { await task(); } catch { pending.push(task); }   // reencola si sigue fallando
  }
});
```

## Cuando el que reintenta eres tú: entrega de webhooks salientes

Hasta ahora tú eras el cliente que reintenta. Pero si tu sistema **envía** webhooks a terceros, los papeles se invierten: eres el servidor que debe reintentar la entrega cuando el receptor no responde. Los proveedores serios (Stripe, GitHub, Shopify) siguen todos el mismo patrón: reintentos con backoff exponencial durante un periodo largo —de horas a días— y, si nunca se logra, abandono con notificación.

Las decisiones clave al diseñar la entrega de webhooks salientes:

- **Programa, no bloquees**: no reintentes en el mismo hilo de la petición. Guarda el intento de entrega en una tabla o cola con un instante de «próximo intento» y deja que un worker lo recoja.
- **Backoff largo y creciente**: un esquema típico es reintentar tras 1 min, 5 min, 30 min, 2 h, 5 h... hasta cubrir 24–72 h. El receptor puede estar caído un buen rato.
- **Considera éxito solo un 2xx**: cualquier otra cosa (o un timeout) es un fallo que merece reintento.

- **Firma cada entrega** con HMAC y un timestamp para que el receptor pueda verificarla e implementar idempotencia por su lado.
- **Abandona con elegancia**: tras agotar la ventana, marca el endpoint como fallido, avisa al cliente (por email o panel) y ofrece reenvío manual.

El esqueleto de un worker de entrega, con la programación del siguiente intento calculada con backoff:

```
SCHEDULE = [60, 300, 1800, 7200, 18000, 43200, 86400]   # segundos entre intentos

def deliver_due_webhooks(store, http):
    for wh in store.due_now():                # entregas cuyo next_attempt <= ahora
        try:
            resp = http.post(wh.url, json=wh.payload,
                             headers=sign(wh.payload), timeout=5)
            ok = 200 <= resp.status_code < 300
        except Exception:
            ok = False
        if ok:
            store.mark_delivered(wh.id)
        elif wh.attempt < len(SCHEDULE):
            delay = SCHEDULE[wh.attempt]
            store.reschedule(wh.id, in_seconds=delay, attempt=wh.attempt + 1)
        else:
            store.mark_failed(wh.id)          # agotado: avisa al cliente
```

## Errores parciales: reintentar operaciones por lotes

Un caso que rompe los reintentos ingenuos es la operación **por lotes**: envías cien elementos en una sola petición y el servidor procesa noventa y ocho pero falla en dos. Si reintentas la petición entera, vuelves a procesar los noventa y ocho que ya funcionaron —con el consiguiente riesgo de duplicados— para arreglar solo dos. La solución es diseñar (y consumir) APIs por lotes que devuelvan un **resultado por elemento**, y reintentar únicamente los que fallaron con un error transitorio.

```
def batch_with_retry(send_batch, items, *, max_retries=4, base=0.5, cap=10.0):
    """send_batch(items) -> lista de resultados alineada con items:
    cada uno {'ok': bool, 'retryable': bool, 'value': ...}."""
    import random, time
    pending = list(items)
    results = {}
    for attempt in range(max_retries + 1):
        outcomes = send_batch(pending)
        retry_next = []
        for item, out in zip(pending, outcomes):
            if out["ok"]:
                results[id(item)] = out["value"]
            elif out["retryable"] and attempt < max_retries:
                retry_next.append(item)      # solo estos vuelven a intentarse
            else:
                results[id(item)] = out       # fallo definitivo
        if not retry_next:
            break
        time.sleep(random.uniform(0, min(cap, base * 2 ** attempt)))
        pending = retry_next
    return [results[id(it)] for it in items]
```

Este patrón —reintentar el subconjunto que falló, no el lote completo— reduce el trabajo desperdiciado y evita duplicados. Combínalo con idempotencia por elemento para que, incluso si un elemento se reintenta de más, el efecto sea seguro. Es la diferencia entre un reintento que cuesta dos operaciones y uno que cuesta cien.

## Caso práctico: reintentos contra APIs de modelos de IA

Si trabajas con modelos de lenguaje —OpenAI, Anthropic, o cualquier proveedor— los reintentos dejan de ser opcionales: estas APIs devuelven `429` con frecuencia cuando superas tus límites de tasa, y errores `500` / `529` (sobrecarga) en picos de demanda. Es uno de los sitios donde un buen backoff con jitter marca la diferencia entre una aplicación estable y una que se cae cada vez que hay tráfico.

Las particularidades de reintentar llamadas a LLMs:

- **Respeta los límites de tasa**: los `429` suelen venir con `Retry-After` o con cabeceras de tipo `x-ratelimit-reset`. Úsalas; el proveedor sabe exactamente cuándo se restablece tu cuota.
- **Distingue límites de tokens y de peticiones**: puedes tener cuota de peticiones pero agotar la de tokens. Si el error es por tokens, reintentar la misma petición gigante fallará igual; a veces la solución es trocear la entrada, no reintentar.
- **Cuida el coste**: cada reintento de una llamada que ya generó tokens (por ejemplo, si falla a mitad de un stream) puede volver a cobrarte. Limita los intentos y prefiere reanudar sobre reempezar cuando el proveedor lo permita.
- **Idempotencia**: muchas APIs de IA aceptan una cabecera de idempotencia para que un reintento tras un timeout no genere (ni cobre) dos veces la misma generación.

Un wrapper que reintenta llamadas a un LLM respetando `Retry-After` y aplicando full jitter como plan B:

```
import time, random, httpx

RETRYABLE = {429, 500, 502, 503, 529}   # 529 = "overloaded" en algunos proveedores

def call_llm(client, payload, *, max_retries=5, base=1.0, cap=60.0):
    for attempt in range(max_retries + 1):
        resp = client.post("/v1/messages", json=payload, timeout=60)
        if resp.status_code not in RETRYABLE:
            resp.raise_for_status()
            return resp.json()
        if attempt == max_retries:
            resp.raise_for_status()
        # El proveedor manda cuando reintentar; si no, full jitter
        ra = resp.headers.get("retry-after")
        delay = float(ra) if ra and ra.isdigit() else random.uniform(0, min(cap, base * 2 ** attempt))
        time.sleep(delay)
```

Para respuestas en **streaming** (Server-Sent Events), el reintento tiene un matiz: si la conexión se corta a mitad, reintentar desde cero repite tokens y coste. Lo ideal es reintentar solo el establecimiento de la conexión (antes de recibir el primer token) y, si el corte ocurre en mitad del stream, tratarlo como un fallo recuperable a nivel de aplicación —mostrar lo recibido y ofrecer «continuar»— en vez de relanzar la generación completa. Y como siempre: pon un tope de intentos y un deadline, porque un usuario esperando la respuesta de un modelo no tolera un minuto de reintentos silenciosos.

## Reintentar al arrancar: esperar a que las dependencias estén listas

Hay un escenario de reintentos que se olvida con frecuencia: el **arranque** de un servicio. En un despliegue con contenedores u orquestadores, tu aplicación puede iniciarse antes de que su base de datos, su caché o el servicio del que depende estén aceptando conexiones. Si tu proceso se rinde al primer fallo de conexión, entrará en un bucle de reinicios (el clásico `CrashLoopBackOff`) hasta que, por suerte, todo esté listo a la vez.

La solución es reintentar la conexión inicial con backoff, pero con parámetros distintos a los de una petición normal: aquí sí quieres **muchos intentos** y un deadline generoso, porque una dependencia que arranca puede tardar decenas de segundos. La idea es esperar activamente en vez de morir.

```
import time, random

def wait_for_dependency(check, *, timeout=60.0, base=0.5, cap=5.0):
    """Reintenta check() hasta que devuelva True o se agote el timeout global.
    check() lanza excepcion o devuelve False si la dependencia no esta lista."""
    start = time.monotonic()
    attempt = 0
    while True:
        try:
            if check():
                return True
        except Exception:
            pass
        if time.monotonic() - start >= timeout:
            raise TimeoutError("la dependencia no estuvo lista a tiempo")
        time.sleep(random.uniform(0, min(cap, base * 2 ** attempt)))
        attempt += 1

# Ejemplo: esperar a Postgres antes de arrancar la app
wait_for_dependency(lambda: db_ping(), timeout=90)
```

Aun así, reintentar en el arranque no sustituye a los **health checks**: expón un endpoint de « readiness » que sea verde solo cuando todas tus dependencias respondan, y deja que el orquestador no envíe tráfico hasta entonces. El reintento resuelve el arranque; el health check evita que recibas peticiones antes de estar listo.

## Reintento no es lo mismo que recuperación: ten un plan B

Un último matiz conceptual que ahorra muchos disgustos: el reintento asume que la operación **acabará funcionando** si insistes. Pero a veces no va a funcionar en la ventana de tiempo que tienes, y agotar los reintentos no es el final de la historia, sino el momento de activar un **fallback** (una alternativa degradada).

Ejemplos de degradación elegante cuando los reintentos se agotan: servir un valor cacheado aunque esté algo obsoleto en lugar de mostrar un error; usar un valor por defecto razonable cuando un servicio de personalización no responde; encolar la operación para procesarla más tarde en vez de fallar ante el usuario; o mostrar una versión reducida de la página que no dependa del servicio caído. La jerarquía mental es clara: primero intenta, luego reintenta con backoff, y cuando el presupuesto se agote, degrada con gracia. Un sistema resiliente no es el que nunca falla, sino el que sigue siendo útil aunque una de sus partes falle.

## Todo junto: una defensa por capas

Los patrones de esta guía no compiten entre sí; se **apilan**. El circuit breaker decide si merece la pena intentarlo siquiera; el reintento con backoff y jitter absorbe los fallos transitorios; el presupuesto de reintentos evita que la amplificación se descontrole; el deadline pone un techo al tiempo total; y el fallback entra en juego cuando, a pesar de todo, no se pudo. Verlos combinados en una sola función ayuda a entender cómo encajan.

```
import time, random

def resilient_call(fn, *, breaker, budget, cache_key=None, cache=None,
                   max_retries=3, base=0.2, cap=3.0, total_deadline=8.0):
    """Composes the patterns: circuit breaker + retry budget + backoff/jitter
    + deadline + graceful fallback. Returns the value or a degraded one."""
    start = time.monotonic()
    budget.record_request()
    attempt = 0
    while True:
        try:
            return breaker.call(fn)              # breaker fails fast if open
        except Exception:
            remaining = total_deadline - (time.monotonic() - start)
            can_more = (attempt < max_retries and remaining > 0
                        and budget.can_retry())    # respect the retry budget
            if not can_more:
                if cache is not None and cache_key in cache:
                    return cache[cache_key]        # graceful fallback: stale value
                raise                              # nothing left to do
            delay = min(random.uniform(0, min(cap, base * 2 ** attempt)), remaining)
            time.sleep(delay)
            attempt += 1
```

Léelo de fuera hacia dentro: primero el **breaker** (¿está el servicio lo bastante sano para intentarlo?), luego el **reintento** con su backoff, gobernado por el **presupuesto** y por el **deadline**, y por último el **fallback** a un valor cacheado cuando ya no queda margen. Cada capa tiene una única responsabilidad, y juntas convierten un fallo puntual de una dependencia en algo que, en el mejor caso, el usuario ni nota, y en el peor, degrada con elegancia en lugar de romperse.

No necesitas todas las capas desde el día uno. Empieza por el reintento con jitter y los límites; añade el circuit breaker cuando tengas dependencias que a veces se caen del todo; incorpora el presupuesto cuando operes a una escala en la que la amplificación sea un riesgo real; y diseña el fallback para los caminos donde un error visible sea inaceptable. La resiliencia se construye por capas, y cada una se justifica por un tipo de fallo concreto que has visto (o previsto) en producción.

## Checklist de producción

Antes de dar por buena tu estrategia de reintentos, repásala contra esta lista:

- Reintentas **solo** errores transitorios (errores de red, 429 y 5xx), nunca 4xx de cliente.
- Usas **backoff exponencial con full jitter** como opción por defecto.
- `Retry-After` tiene prioridad sobre tu backoff y parseas sus dos formatos (segundos y fecha).
- Todas las escrituras que reintentas son **idempotentes** (clave de idempotencia donde haga falta).
- Tienes un **timeout por intento** y un **deadline total**, y propagas el deadline entre servicios.
- Reintentas en una **sola capa** y tienes un **presupuesto de reintentos** para frenar la amplificación.
- Un **circuit breaker** corta cuando un servicio deja de responder de forma sostenida.
- Exportas **métricas** de reintentos y alertas cuando la tasa o los agotamientos suben.
- Tienes **pruebas** deterministas que cubren tanto lo que se reintenta como lo que no.

## Conclusión

Un buen reintento es de las inversiones con mejor retorno en fiabilidad: unas pocas líneas bien pensadas convierten fallos transitorios en invisibles para el usuario. Pero un mal reintento es de los errores más caros que existen, porque

amplifica los problemas justo cuando el sistema es más frágil. La diferencia está en los detalles que hemos recorrido: jitter para no sincronizar clientes, idempotencia para que repetir sea seguro, timeouts y deadlines para no colgarse, presupuestos y circuit breakers para no amplificar, y observabilidad para enterarte antes de que sea tarde. Empieza por la receta esencial —backoff exponencial con full jitter, solo errores transitorios, respetar `Retry-After` y exigir idempotencia— y ve añadiendo los patrones de producción a medida que tu escala lo pida. Tu yo del futuro, el que esté de guardia durante el próximo incidente, te lo agradecerá.

**English**

Almost every distributed system fails intermittently: a network timeout, a brief 503 while a service restarts, a rate limit that trips during a traffic spike. Retrying is the obvious answer, but a careless retry turns a small hiccup into a full outage. This guide covers how to retry well: exponential backoff so you do not hammer too fast, and jitter so thousands of clients do not all come back at once.

## When a retry actually makes sense

A retry only helps when the failure is transient: something that will probably succeed if you try again in a moment. A connection timeout, a 429 (Too Many Requests), or a 503 (Service Unavailable) are good candidates. A 400 (Bad Request) or a 401 (Unauthorized) are not: retrying a malformed request only wastes resources and will never succeed.

## The naive-retry problem

The first instinct is usually a loop that retries immediately, or that waits a fixed interval. Both fall apart under load. If a service is down for two seconds, every client that failed retries at the same time and hits the service exactly as it tries to recover, knocking it over again. This is called the «thundering herd».

## Exponential backoff

The idea behind exponential backoff is to wait progressively longer between attempts: the first retry waits a short base, the next waits double, then quadruple, and so on, up to a cap. This gives the service growing time to recover and reduces total pressure.

The delay for attempt n is: delay = min(cap, base · 2^n). With base = 0.5 s and cap = 30 s, the delays would be 0.5, 1, 2, 4, 8, 16, 30, 30…

## Why backoff alone is not enough: jitter

Exponential backoff spreads out the retries of a single client, but it does not separate different clients from each other. If 1,000 clients fail in the same second, they all compute the same delay and retry in lockstep, in waves. Jitter breaks that synchrony by adding randomness to the delay.

### Full Jitter

The simplest strategy and, for most cases, the best default. Instead of waiting exactly the exponential delay, you wait a random value between zero and that delay: delay = random(0, min(cap, base · 2^n)). It spreads retries uniformly across the window and minimizes server load.

### Equal Jitter

A middle ground: keep half of the exponential delay and randomize the other half. It avoids very short sleeps at the cost of slightly more synchrony than full jitter. Useful when you care about keeping a minimum delay between attempts.

### Decorrelated Jitter

The most battle-tested variant at scale. Each attempt's delay is based on the previous one: sleep = min(cap, random(base, prev_sleep · 3)). It grows more aggressively and decorrelates clients even further. AWS recommends it for very high concurrency scenarios.

## Implementation in Python

This function retries an HTTP request using full jitter, distinguishes the errors worth retrying, and prioritizes the Retry-After header when the server sends one.

```
import random
import time
import requests

RETRYABLE_STATUS = {429, 500, 502, 503, 504}

def retry_request(method, url, *, max_retries=5, base=0.5, cap=30.0, **kwargs):
    attempt = 0
    while True:
        try:
            resp = requests.request(method, url, timeout=10, **kwargs)
        except (requests.ConnectionError, requests.Timeout):
            if attempt >= max_retries:
                raise
            delay = full_jitter(base, cap, attempt)
        else:
            if resp.status_code not in RETRYABLE_STATUS:
                return resp
            if attempt >= max_retries:
                resp.raise_for_status()
            delay = retry_after(resp) or full_jitter(base, cap, attempt)

        time.sleep(delay)
        attempt += 1

def full_jitter(base, cap, attempt):
    expo = min(cap, base * (2 ** attempt))
    return random.uniform(0, expo)

def retry_after(resp):
    value = resp.headers.get("Retry-After")
    if value is None:
        return None
    try:
        return float(value)  # seconds; the HTTP-date form is omitted for brevity
    except ValueError:
        return None
```

Switching strategies is as simple as replacing full_jitter. The decorrelated-jitter version keeps the previous delay between attempts:

```
def decorrelated_jitter(base, cap, prev_sleep):
    # prev_sleep = base on the first attempt
    return min(cap, random.uniform(base, prev_sleep * 3))
```

## The same logic in Node.js / TypeScript

```
const RETRYABLE = new Set([429, 500, 502, 503, 504]);

interface RetryOptions {
  maxRetries?: number;
  base?: number; // ms
  cap?: number;  // ms
}

const sleep = (ms) => new Promise((r) => setTimeout(r, ms));

function fullJitter(base, cap, attempt) {
  const expo = Math.min(cap, base * 2 ** attempt);
  return Math.random() * expo;
}

export async function fetchWithRetry(url, init = {}, { maxRetries = 5, base = 500, cap = 30_000 } = {}) {
  let attempt = 0;
  while (true) {
    let res;
    try {
      res = await fetch(url, init);
    } catch (err) {
      if (attempt >= maxRetries) throw err; // network error
      await sleep(fullJitter(base, cap, attempt));
      attempt++;
      continue;
    }

    if (!RETRYABLE.has(res.status)) return res;
    if (attempt >= maxRetries) return res; // exhausted: let the caller inspect it

    const retryAfter = Number(res.headers.get("retry-after"));
    const delay =
      Number.isFinite(retryAfter) && retryAfter > 0
        ? retryAfter * 1000
        : fullJitter(base, cap, attempt);

    await sleep(delay);
    attempt++;
  }
}
```

## Which errors to retry (and which not to)

Retry: network errors (connection refused, timeout, temporary DNS failure), 429 (Too Many Requests), and 5xx such as 500, 502, 503, and 504.

Do not retry: client 4xx such as 400, 401, 403, 404, and 422. The problem is in your request, not the server; retrying will never fix it.

Special case: for non-idempotent writes, a 5xx or a timeout does not guarantee the operation was not applied. Blindly retrying can duplicate a payment or an order.

## Respect the Retry-After header

When a server returns 429 or 503, it often includes a Retry-After header telling you how many seconds to wait. If it is present, use it instead of your backoff calculation: the server knows better than you when it will be ready. The code above already prioritizes Retry-After over jitter.

## Idempotency: the requirement you cannot skip

Retrying is only safe if repeating the operation produces the same result as running it once. Reads (GET) usually are idempotent by nature; writes are not: charging a card twice is a real problem. The standard solution is to send an idempotency key (Idempotency-Key) that the server uses to deduplicate retries. Design your write endpoints to be idempotent before adding automatic retries.

## Set limits: retry budget and deadlines

Maximum attempts: three to five is usually enough. Beyond that rarely helps and only prolongs the failure.

Delay cap: avoid absurd waits of several minutes between attempts.

Total budget / deadline: limit the total time spent on a request, not just the number of attempts. A user waiting for a response will not tolerate 60 seconds of retries.

Do not retry at every layer: if the client, the gateway, and the service all retry at once, attempts multiply ($3 \times 3 \times 3 = 27$). Retry at a single level.

## At scale: combine it with other patterns

With many concurrent clients, jitter alone is not enough. Combine it with strict per-request timeouts, a circuit breaker that stops sending traffic to a clearly down service, and a token bucket that caps the client's own retry rate. Together they keep your fleet from turning a mild degradation into a full outage.

## Common pitfalls

Retrying non-transient errors (4xx), wasting resources on requests that will never succeed.

Backoff without jitter: it synchronizes all clients and perpetuates the thundering herd.

Ignoring Retry-After and hammering a server that asked you to wait.

Retrying non-idempotent writes and duplicating side effects.

No cap and no deadline: a client stuck in endless retries holds connections and memory.

Nested retries across several layers that silently multiply.

## Summary

Retrying well is a balance: enough to survive transient failures, not so much that you amplify an outage. The proven recipe is exponential backoff with full jitter, retry only transient errors, respect Retry-After, require idempotency on writes, and set clear limits on attempts and time. It is little code for a huge return in reliability.

## Beyond the basics: the production guide

So far you have the essential recipe, and it is enough for 90% of cases. The rest of this guide is the deep part: the math that explains why jitter works, a simulation you can run to see the difference for yourself, and the patterns that surround a good retry in a real system: timeouts and deadlines, end-to-end idempotency, circuit breakers, retry budgets, retries in databases and queues, gRPC, hedged requests, observability and testing. If you build services that

must survive spikes and partial failures, this is what separates a retry that saves the day from one that causes the outage.

## The math of backoff and why jitter wins

It helps to understand what each strategy optimizes. With pure exponential backoff, the delay for attempt `n` is deterministic: `base · 2^n`. Two clients that fail in the same millisecond will compute the exact same delay and collide again. The problem is not the size of the delay but its **zero variance**: there is no spread.

Jitter introduces variance on purpose. With **full jitter**, the delay is a uniform random variable in `[0, base·2^n]`. Its expected value is half the deterministic delay (`base·2^n / 2`), so on average you retry sooner, but spread evenly across the whole window. That spread is exactly what breaks the waves: instead of a thousand requests at the same instant, you get a thousand requests scattered over several seconds.

The practical question is how much extra work each strategy generates. Marc Brooker measured this in the reference AWS article with a simulation of clients competing for a resource. His conclusions, which still hold and still guide the design of the AWS SDKs, are clear: any form of jitter drastically reduces the number of calls compared to using none; among the variants, **equal jitter** is the worst of the three (it does slightly more work than full jitter and takes considerably longer); and the real choice is between **full jitter** (fewer calls, slightly more total time) and **decorrelated jitter** (slightly more calls, less time). For the vast majority of services, full jitter is the correct default.

The list below shows the wait window of each strategy in the first attempts, with `base = 0.5 s` and `cap = 30 s`. Notice how in pure backoff the value is fixed, while in full jitter it is a range starting at zero.

- Attempt 0 → pure: 0.5 s | full jitter: 0–0.5 s
- Attempt 1 → pure: 1 s | full jitter: 0–1 s
- Attempt 2 → pure: 2 s | full jitter: 0–2 s
- Attempt 3 → pure: 4 s | full jitter: 0–4 s
- Attempt 4 → pure: 8 s | full jitter: 0–8 s
- Attempt 5 → pure: 16 s | full jitter: 0–16 s
- Attempt 6+ → both capped at 30 s (with full jitter, 0–30 s)

## Simulation: compare the strategies with numbers

Theory sinks in when you run it. This small simulator models `N` clients that fail at once against a service with limited per-second capacity, and measures two things: how many total attempts it took and how long the last client took to succeed. Run it changing the strategy and you will see why jitter matters.

```
import random, heapq

def simulate(strategy, n_clients=1000, capacity_per_sec=200,
             base=0.5, cap=30.0, seed=1):
    """Returns (total_attempts, time_to_complete) for a strategy."""
    rng = random.Random(seed)
    # Every client starts wanting to retry at t=0
    heap = [(0.0, i, 0) for i in range(n_clients)]  # (time, client, attempt)
    heapq.heapify(heap)
    done = set()
    attempts = 0
    # per-second capacity: how many requests the server accepts each second
    served_in_bucket, current_bucket = 0, 0
    last_success_t = 0.0

    def next_delay(attempt, prev):
        expo = min(cap, base * (2 ** attempt))
        if strategy == "none":          # no backoff: retry every fixed 1s
            return 1.0
        if strategy == "backoff":       # exponential, no jitter
            return expo
        if strategy == "full":          # full jitter
            return rng.uniform(0, expo)
        if strategy == "equal":         # equal jitter
            return expo / 2 + rng.uniform(0, expo / 2)
        if strategy == "decorrelated":  # decorrelated jitter
            return min(cap, rng.uniform(base, (prev or base) * 3))

    prev_delay = {}
    while heap:
        t, client, attempt = heapq.heappop(heap)
        if client in done:
            continue
        bucket = int(t)
        if bucket != current_bucket:
            current_bucket, served_in_bucket = bucket, 0
        attempts += 1
        if served_in_bucket < capacity_per_sec:
            served_in_bucket += 1         # the server accepts the request -> success
            done.add(client)
            last_success_t = max(last_success_t, t)
        else:
            d = next_delay(attempt, prev_delay.get(client))
            prev_delay[client] = d
            heapq.heappush(heap, (t + d, client, attempt + 1))
    return attempts, round(last_success_t, 1)

for s in ["none", "backoff", "full", "equal", "decorrelated"]:
    total, t_end = simulate(s)
    print(f"{s:>13}: {total:>6} attempts, completed in {t_end:>5}s")
```

Running it, you will see the pattern AWS describes: `none` and `backoff` (no jitter) generate many more attempts because clients stay synchronized and saturate the same second over and over; the three jittered variants cut total attempts noticeably. It is a simplified model, no substitute for a real load test, but it makes the intuition clear: jitter turns an avalanche into a trickle the server can absorb.

## Parse Retry-After correctly

The `Retry-After` header has **two formats** per the HTTP standard: an integer number of seconds ( `Retry-After: 120` ) or an absolute HTTP date ( `Retry-After: Wed, 21 Oct 2026 07:28:00 GMT` ). Almost everyone handles the first and forgets the second, which causes a date value to be silently ignored. You should also **clamp** the value: a misconfigured server might ask you to wait hours, and you do not want to block a user request that long.

```
from datetime import datetime, timezone
from email.utils import parsedate_to_datetime

def parse_retry_after(value, *, max_wait=300.0):
    """Convert a Retry-After header into seconds, or None if invalid.
    Accepts the seconds format and the HTTP-date format. Clamps to max_wait."""
    if not value:
        return None
    value = value.strip()
    # Format 1: delta-seconds
    if value.isdigit():
        return min(float(value), max_wait)
    # Format 2: HTTP-date
    try:
        when = parsedate_to_datetime(value)
        if when.tzinfo is None:
            when = when.replace(tzinfo=timezone.utc)
        delta = (when - datetime.now(timezone.utc)).total_seconds()
        return min(max(delta, 0.0), max_wait)
    except (TypeError, ValueError):
        return None
```

The same logic in TypeScript, ready to use in the browser or in Node:

```
function parseRetryAfter(value: string | null, maxWait = 300): number | null {
  if (!value) return null;
  const v = value.trim();
  // Seconds format
  if (/^\d+$/.test(v)) return Math.min(Number(v), maxWait);
  // HTTP-date format
  const when = Date.parse(v);
  if (Number.isNaN(when)) return null;
  const delta = (when - Date.now()) / 1000;
  return Math.min(Math.max(delta, 0), maxWait);
}
```

Golden rule: if `Retry-After` is present, it **always wins** over your backoff calculation. The server knows better than you when it will be ready. Your jittered backoff is only the plan B for when it tells you nothing.

## Timeouts and deadlines: the retry's mandatory companion

Retrying without **timeouts** is dangerous: a request that hangs forever never gets retried, holds a connection, and consumes memory. And retrying without a **global deadline** is just as bad: five attempts with backoff can add up to more than a minute, and no user waits that long. The rule is to have two limits: a **per-attempt** timeout (each individual request) and a **total** budget for the whole operation.

In Python with `httpx` , each attempt carries its own timeout and a global clock cuts the loop:

```python
import time, random, httpx

def request_with_deadline(client, method, url, *,
                          total_deadline=8.0, per_try_timeout=2.0,
                          base=0.3, cap=5.0, max_retries=5, **kwargs):
    start = time.monotonic()
    attempt = 0
    while True:
        remaining = total_deadline - (time.monotonic() - start)
        if remaining <= 0:
            raise TimeoutError("total operation budget exhausted")
        timeout = min(per_try_timeout, remaining)
        try:
            resp = client.request(method, url, timeout=timeout, **kwargs)
            if resp.status_code not in {429, 500, 502, 503, 504}:
                return resp
        except (httpx.ConnectError, httpx.TimeoutException):
            if attempt >= max_retries:
                raise
        if attempt >= max_retries:
            return resp
        # Never wait longer than what remains of the budget
        delay = min(random.uniform(0, min(cap, base * 2 ** attempt)),
                    max(0.0, total_deadline - (time.monotonic() - start)))
        time.sleep(delay)
        attempt += 1
```

In the browser, `AbortController` cancels each attempt and the remaining budget is recomputed on every loop:

```javascript
async function withDeadline(url, init = {}, { totalDeadline = 8000, perTry = 2000 } = {}) {
  const start = Date.now();
  let attempt = 0;
  while (true) {
    const remaining = totalDeadline - (Date.now() - start);
    if (remaining <= 0) throw new Error("deadline exhausted");
    const ctrl = new AbortController();
    const t = setTimeout(() => ctrl.abort(), Math.min(perTry, remaining));
    try {
      const res = await fetch(url, { ...init, signal: ctrl.signal });
      if (![429, 500, 502, 503, 504].includes(res.status)) return res;
    } catch (err) {
      if (attempt >= 5) throw err;
    } finally {
      clearTimeout(t);
    }
    const backoff = Math.random() * Math.min(5000, 300 * 2 ** attempt);
    await new Promise(r => setTimeout(r, Math.min(backoff, totalDeadline - (Date.now() - start))));
    attempt++;
  }
}
```

A principle that is often forgotten: the deadline must be **propagated** across services. If your API receives a request with an 8-second budget and calls an internal service, that service should not retry for 30 seconds. Pass the remaining time as a header or context field and have every layer respect it.

## End-to-end idempotency with Idempotency-Key

We already saw that retrying writes is dangerous because a timeout does not tell you whether the operation was applied. The industry-standard solution, the one used by Stripe, PayPal, and almost any payments API, is the **idempotency key**. The client generates a unique identifier per operation and sends it in the `Idempotency-Key` header. The server stores the result keyed by that value; if a retry arrives with the same key, it returns the stored result instead of executing the operation again.

On the client you just generate the key once and reuse it across all retries of the same logical operation:

```
import uuid, requests

def create_charge(amount, currency, *, session):
    key = str(uuid.uuid4())            # one key per operation, NOT per attempt
    headers = {"Idempotency-Key": key}
    # retry_request is the backoff+jitter function from the first part
    return retry_request("POST", "https://api.payments.com/charges",
                         json={"amount": amount, "currency": currency},
                         headers=headers, session=session)
```

On the server, the minimal pattern with FastAPI and an idempotency table. The key is to use a uniqueness constraint (or an `INSERT ... ON CONFLICT`) so two concurrent retries do not run at the same time:

```
from fastapi import FastAPI, Header, HTTPException
import json, psycopg

app = FastAPI()

@app.post("/charges")
def create_charge(payload: dict, idempotency_key: str = Header(...)):
    with psycopg.connect("dbname=app") as conn, conn.cursor() as cur:
        # 1) Reserve the key atomically. If it already exists, do not insert.
        cur.execute(
            """INSERT INTO idempotency_keys (key, status)
               VALUES (%s, 'in_progress')
               ON CONFLICT (key) DO NOTHING""",
            (idempotency_key,),
        )
        if cur.rowcount == 0:
            # Key already existed: return the stored result (or 409 if still running)
            cur.execute("SELECT status, response FROM idempotency_keys WHERE key = %s",
                        (idempotency_key,))
            status, response = cur.fetchone()
            if status == "in_progress":
                raise HTTPException(409, "operation in progress, retry later")
            return json.loads(response)

        # 2) First time: run the real effect (charge) and store the response
        result = charge_the_card(payload)   # your business logic
        cur.execute(
            "UPDATE idempotency_keys SET status='done', response=%s WHERE key=%s",
            (json.dumps(result), idempotency_key),
        )
        conn.commit()
        return result
```

Details that matter in production: also store a **hash of the request body** alongside the key and reject (with a 422) if the same key arrives with a different body, that indicates a client bug, not a legitimate retry. Give keys an **expiry** (24–72 hours is usually enough) so you do not accumulate them forever. And decide what to do with concurrent requests: returning 409 while the first one is in progress is the simplest and safest.

## Circuit breaker: stop hammering a downed service

Retries assume the failure is momentary. But if a service has been down for thirty seconds, continuing to retry, even with backoff, just wastes resources and delays the failure the user will see anyway. The **circuit breaker** solves this: when it detects too many consecutive failures, it "opens the circuit" and fails requests instantly, without even attempting them, giving the service time to recover.

It has three states. In **closed** everything flows normally and failures are counted. If they exceed a threshold, it moves to **open**: for a cooldown period, every request fails immediately. Once that time elapses it moves to **half-open**: it lets a single probe request through; if it succeeds, it goes back to closed; if it fails, back to open. A compact implementation in Python:

```
import time

class CircuitBreaker:
    def __init__(self, fail_max=5, reset_timeout=30.0):
        self.fail_max = fail_max
        self.reset_timeout = reset_timeout
        self.failures = 0
        self.state = "closed"
        self.opened_at = 0.0

    def call(self, fn, *args, **kwargs):
        if self.state == "open":
            if time.monotonic() - self.opened_at >= self.reset_timeout:
                self.state = "half-open"      # time to probe again
            else:
                raise RuntimeError("circuit open: fail fast")
        try:
            result = fn(*args, **kwargs)
        except Exception:
            self._on_failure()
            raise
        self._on_success()
        return result

    def _on_success(self):
        self.failures = 0
        self.state = "closed"

    def _on_failure(self):
        self.failures += 1
        if self.state == "half-open" or self.failures >= self.fail_max:
            self.state = "open"
            self.opened_at = time.monotonic()
```

How retry and circuit breaker combine: the retry lives **inside** the closed state (it retries the normal transient failures), and the circuit breaker wraps it to cut things off at the root when the failure stops being transient and becomes an outage. Together they prevent both failing on a momentary hiccup and storming a dead service with retries.

## Retry budget: the Google SRE lesson

Here is the most expensive and least obvious mistake: retries **multiply** across layers. If the client retries 3 times, the gateway 3 times, and the internal service 3 times, a single deep failure can generate 3 × 3 × 3 = 27 requests. Under load, this turns a mild degradation into a full collapse. Google describes this in its SRE book and proposes two complementary limits.

The first is a **per-request cap**: up to three attempts, and if they fail, the error bubbles up to the caller. The second, more powerful one, is a per-client **retry budget**: each client tracks what fraction of its traffic is retries and only retries while that fraction stays below a threshold, Google uses 10%. With that budget, traffic growth during an incident is limited to about 1.1x instead of multiplying. An implementation with a budget "bucket":

```
import threading, time

class RetryBudget:
    """Allows a retry only if recent retries are < ratio of traffic.
    Approximates Google SRE's pattern (10% ratio)."""
    def __init__(self, ratio=0.1, ttl=10.0):
        self.ratio = ratio
        self.ttl = ttl
        self.requests = []   # timestamps of normal requests
        self.retries = []    # timestamps of retries
        self.lock = threading.Lock()

    def _prune(self, now):
        cutoff = now - self.ttl
        self.requests = [t for t in self.requests if t > cutoff]
        self.retries = [t for t in self.retries if t > cutoff]

    def record_request(self):
        with self.lock:
            self.requests.append(time.monotonic())

    def can_retry(self):
        with self.lock:
            now = time.monotonic()
            self._prune(now)
            allowed = self.ratio * max(len(self.requests), 1)
            if len(self.retries) < allowed:
                self.retries.append(now)
                return True
            return False
```

Before each retry, you check `budget.can_retry()`. If the backend is down, the retry ratio spikes quickly and the budget starts denying, cutting off amplification at the exact point where it is born. Practical rule: **retry in a single layer**, ideally the one closest to the user, and let the others propagate the failure.

## Retries in the database: serialization failures and deadlocks

Retries are not just an HTTP thing. With the **Repeatable Read** or **Serializable** isolation levels, PostgreSQL may abort a transaction if it detects a concurrency conflict it cannot resolve safely. The PostgreSQL 18 documentation is explicit: applications using these levels **must be prepared to retry** transactions that fail due to serialization errors. The two error codes to catch are `40001` ( `serialization_failure` ) and `40P01` ( `deadlock_detected` ).

The pattern is a retry loop around the whole transaction, with backoff and jitter, that rolls back and starts over. Never retry a single statement inside the same aborted transaction: the entire transaction must be redone.

```
import time, random, psycopg
from psycopg import errors

RETRYABLE_SQLSTATE = {"40001", "40P01"}   # serialization_failure, deadlock_detected

def run_tx(dsn, work, *, max_retries=5, base=0.05, cap=2.0):
    for attempt in range(max_retries + 1):
        try:
            with psycopg.connect(dsn) as conn:
                conn.isolation_level = psycopg.IsolationLevel.SERIALIZABLE
                with conn.cursor() as cur:
                    result = work(cur)   # your logic: SELECT/UPDATE/INSERT...
                conn.commit()
                return result
        except errors.Error as e:
            if e.sqlstate in RETRYABLE_SQLSTATE and attempt < max_retries:
                time.sleep(random.uniform(0, min(cap, base * 2 ** attempt)))
                continue
            raise
```

Tips that reduce how often these failures happen: keep transactions **short** (smaller conflict window), access rows in a **consistent order** throughout your code (avoid circular deadlocks), and when you know in advance which rows you will modify, consider `SELECT ... FOR UPDATE` to lock them upfront: conflicts turn into waits instead of aborts.

## Retries in message queues

In event-driven systems, retrying is usually the responsibility of the queue infrastructure, not your code. If your consumer processes a message and fails (or takes too long), the broker redelivers it. The key mechanisms are the **visibility timeout** (the time a message stays hidden after being received; if you do not acknowledge it in time, it reappears), the **retry counter**, and the **dead-letter queue** or DLQ, where messages that failed too many times end up so you can inspect them without blocking the main queue.

The most important consequence for your code: because a message can be delivered **more than once**, your consumer must be **idempotent**. The usual approach is to record the IDs of already-processed messages and skip repeats:

```
def handle_message(msg, *, seen, process):
    if msg.id in seen:            # already processed in a previous delivery
        msg.ack()
        return
    try:
        process(msg.body)         # business effect (idempotent if possible)
        seen.add(msg.id)          # persist this in a durable store, not in memory
        msg.ack()                 # acknowledge: it will not be redelivered
    except TransientError:
        msg.nack()                # return to the queue; the broker applies backoff/DLQ
```

Configure a **maximum number of redeliveries** (say 5) before sending the message to the DLQ. A message that fails forever, a "poison message", can clog the entire queue if you do not set that limit. And watch the DLQ with an alert: it is your signal that something is breaking systematically.

## Retries in gRPC without writing code

If you use gRPC, you do not need to hand-code the retry loop: the client itself does it if you pass a **service config** with a retry policy. You define the maximum attempts, the backoff, and which status codes are retryable, and gRPC applies exponential backoff with ±20% jitter automatically.

```
{
  "methodConfig": [{
    "name": [{"service": "payments.PaymentService"}],
    "retryPolicy": {
      "maxAttempts": 4,
      "initialBackoff": "0.1s",
      "maxBackoff": "2s",
      "backoffMultiplier": 2,
      "retryableStatusCodes": ["UNAVAILABLE", "RESOURCE_EXHAUSTED"]
    }
  }]
}
```

`maxAttempts` includes the original attempt and must be greater than 1. Choose the `retryableStatusCodes` carefully: `UNAVAILABLE` (server down or restarting) is the classic candidate; `RESOURCE_EXHAUSTED` usually maps to a 429. Never mark an `INVALID_ARGUMENT` or an `UNAUTHENTICATED` as retryable. gRPC also offers a **hedging** policy as an alternative, which is exactly the next pattern.

## Hedged requests: get ahead of the long tail

Sometimes the problem is not a failure but **tail latency**: 99% of requests respond fast, but 1% get stuck and ruin your p99. The **hedged** requests pattern (or "backup requests"), described by Google in "The Tail at Scale", attacks this: if a request has not responded within, say, the 95th latency percentile, you send a **second request in parallel** to another replica and keep whichever answer arrives first, canceling the other.

```
import asyncio

async def hedged_get(send, *, hedge_after=0.2):
    """Fire a request; if it takes longer than hedge_after, fire a copy and
    return the first to respond."""
    first = asyncio.create_task(send())
    done, pending = await asyncio.wait({first}, timeout=hedge_after)
    if first in done:
        return first.result()
    second = asyncio.create_task(send())     # backup request
    done, pending = await asyncio.wait({first, second},
                                       return_when=asyncio.FIRST_COMPLETED)
    for t in pending:
        t.cancel()
    return next(iter(done)).result()
```

Two essential caveats: hedging is only safe for **idempotent** operations (reads, or writes with an idempotency key), because you duplicate the request on purpose; and you must **cap** how many backup requests you allow (for example, a 5% budget of traffic), or during an incident hedging will double the load exactly when you least want it.

## Do not reinvent the wheel: battle-tested libraries

Everything above gives you the mental model, but in production you should lean on mature, well-tested libraries instead of maintaining your own loop. Some references by ecosystem: in Python, `tenacity` (declarative retries) and the `Retry` adapter of `urllib3` / `requests`; in .NET, `Polly`; in Java, `Resilience4j`; in JavaScript, `p-retry` and `cockatiel` (which also brings a circuit breaker). An example with tenacity shows how concise it gets:

```
from tenacity import (retry, stop_after_attempt, wait_random_exponential,
                      retry_if_exception_type)
import requests

@retry(
    stop=stop_after_attempt(5),
    wait=wait_random_exponential(multiplier=0.5, max=30),   # full jitter built in
    retry=retry_if_exception_type((requests.ConnectionError, requests.Timeout)),
    reraise=True,
)
def fetch(url):
    resp = requests.get(url, timeout=10)
    resp.raise_for_status()
    return resp
```

These libraries already handle the details that are easy to get wrong: the jitter, composition with circuit breakers, observability, and edge cases. Write your own loop only when you need behavior the library does not cover.

## Observability: if you do not measure it, you do not control it

Retries are silent by design: they absorb failures so the user does not see them. The danger is that they also hide that something is wrong until it is too late. That is why you must instrument them. At a minimum, export these metrics: number of attempts per operation, retry rate (retries / requests), success rate after retrying, number of operations that exhausted their attempts, and a histogram of the applied delays.

- **Exhausted retries**: if it grows, a service you depend on is degraded; it should trigger an alert.
- **Retry rate** approaching your budget (e.g. 10%): an early sign of saturation, before there are visible errors.
- **Distributed traces**: mark each retry as a child span with its attempt number, to see in which layer the retries are born.
- **Logs with context**: include the attempt number and the reason (status code or exception) in each retry, never a mute log.

An especially useful alert: fire when the retry rate exceeds a threshold for several minutes. It usually warns you of a downstream problem **before** your users start seeing errors, because the retries are still covering for them.

## How to test your retry logic

Retry code is hard to test precisely because it involves time and randomness. The key is to make it **deterministic**: seed the random generator, control the clock, and simulate the server responses. With pytest and `responses` (or `httpx.MockTransport`) you can verify that you retry what you should and not what you should not:

```
import responses, requests, pytest

@responses.activate
def test_retries_503_then_succeeds(monkeypatch):
    monkeypatch.setattr("time.sleep", lambda s: None)   # do not actually wait
    responses.add(responses.GET, "https://api.test/x", status=503)
    responses.add(responses.GET, "https://api.test/x", status=503)
    responses.add(responses.GET, "https://api.test/x", json={"ok": True}, status=200)

    resp = retry_request("GET", "https://api.test/x", session=requests.Session())

    assert resp.status_code == 200
    assert len(responses.calls) == 3        # two 503s + one success

@responses.activate
def test_does_not_retry_400(monkeypatch):
    monkeypatch.setattr("time.sleep", lambda s: None)
    responses.add(responses.GET, "https://api.test/y", status=400)
    resp = retry_request("GET", "https://api.test/y", session=requests.Session())
    assert resp.status_code == 400
    assert len(responses.calls) == 1        # must not retry a 4xx
```

For more realistic integration tests, tools like `toxiproxy` inject latency, timeouts, and network cuts between your application and its dependencies, and let you check that your backoff, deadlines, and circuit breaker behave as expected under real failures. Testing the error paths is as important as testing the happy path: it is exactly when you have the most at stake.

## Choosing parameters well: base, cap, and number of attempts

The defaults you copy from an example are rarely right for your case. The three parameters that govern a retry, the base, the cap, and the number of attempts, depend on what you are protecting and how patient whoever is waiting for the response is.

The **base** sets the first delay. For interactive calls, where a human waits on the other side, start small (100–300 ms): an almost immediate first retry usually clears a momentary hiccup without the user noticing. For background jobs, where nobody is watching the screen, you can afford a larger base (1–2 s) and be gentler on the backend.

The **cap** prevents exponential backoff from growing into absurd waits. Without it, the eighth attempt with a 0.5 s base would wait over two minutes. A cap of 20–30 s is reasonable for background processes; for interactive requests 2–5 s is usually enough, because beyond that it is better to fail and let the user retry.

The **number of attempts** should almost never exceed five. The probability that a transient failure persists after three or four backed-off retries is low; beyond that you only prolong the agony and hold resources. And remember that the number of attempts and the total deadline are **independent** limits: whichever hits first wins. A good starting point for an interactive API is base 200 ms, cap 3 s, max 4 attempts, and a total deadline of 8 s.

A handy mental table for deciding:

- **Interactive (user waiting)**: base 100–300 ms, cap 2–5 s, 3–4 attempts, deadline 5–10 s.
- **Background work / queues**: base 1–2 s, cap 20–60 s, 5+ attempts, deadline of minutes.
- **Service startup (waiting on dependencies)**: base 0.5 s, cap 10 s, many attempts, generous deadline.

## Retrying from the frontend: the part the user sees

Almost everything above looks at the server, but the browser retries too, and there the rules change because a person is watching. Silently retrying a few times is fine for a passing network glitch, but if the problem persists, hiding

it with infinite retries only produces an eternal spinner and a frustrated user. Good practice combines a few automatic backed-off retries and, if they run out, hands control back to the person.

- **Brief automatic retry**: 2–3 attempts with short backoff to cover momentary network cuts, without bothering the user.
- **Explicit manual retry**: if they fail, show a clear message ("Offline? Retry") with a button. Returning control reduces anxiety and avoids the infinite loop.
- **Respect the offline state**: if `navigator.onLine` is false, do not hammer the network; wait for the `online` event and retry then.
- **Optimistic UI with rollback**: show the change immediately, retry in the background, and if it ultimately fails, visually revert and warn. Never leave the user believing something was saved when it was not.

A concrete pattern: listen for reconnection to retry the requests that queued up while there was no network.

```
const pending = [];   // requests that failed for lack of network

async function sendResilient(task) {
  try {
    return await task();
  } catch (err) {
    if (!navigator.onLine) {
      pending.push(task);          // keep it until the connection is back
      return;
    }
    throw err;                     // other kind of error: let the caller handle it
  }
}

window.addEventListener("online", async () => {
  const queued = pending.splice(0);      // drain the queue
  for (const task of queued) {
    try { await task(); } catch { pending.push(task); }   // re-queue if still failing
  }
});
```

## When you are the one retrying: outbound webhook delivery

Until now you were the client doing the retrying. But if your system **sends** webhooks to third parties, the roles reverse: you are the server that must retry delivery when the receiver does not respond. The serious providers (Stripe, GitHub, Shopify) all follow the same pattern: exponential backoff retries over a long period, from hours to days, and if it never succeeds, giving up with a notification.

The key decisions when designing outbound webhook delivery:

- **Schedule, do not block**: do not retry on the same request thread. Store the delivery attempt in a table or queue with a "next attempt" time and let a worker pick it up.
- **Long, growing backoff**: a typical schedule retries after 1 min, 5 min, 30 min, 2 h, 5 h... covering 24–72 h. The receiver may be down for a good while.
- **Treat only a 2xx as success**: anything else (or a timeout) is a failure worth retrying.
- **Sign each delivery** with HMAC and a timestamp so the receiver can verify it and implement idempotency on its side.
- **Give up gracefully**: after exhausting the window, mark the endpoint as failed, notify the customer (by email or dashboard), and offer manual resend.

The skeleton of a delivery worker, with the next attempt scheduled via backoff:

```
SCHEDULE = [60, 300, 1800, 7200, 18000, 43200, 86400]   # seconds between attempts

def deliver_due_webhooks(store, http):
    for wh in store.due_now():               # deliveries whose next_attempt <= now
        try:
            resp = http.post(wh.url, json=wh.payload,
                             headers=sign(wh.payload), timeout=5)
            ok = 200 <= resp.status_code < 300
        except Exception:
            ok = False
        if ok:
            store.mark_delivered(wh.id)
        elif wh.attempt < len(SCHEDULE):
            delay = SCHEDULE[wh.attempt]
            store.reschedule(wh.id, in_seconds=delay, attempt=wh.attempt + 1)
        else:
            store.mark_failed(wh.id)          # exhausted: notify the customer
```

## Partial errors: retrying batch operations

A case that breaks naive retries is the **batch** operation: you send a hundred items in a single request and the server processes ninety-eight but fails on two. If you retry the whole request, you reprocess the ninety-eight that already worked, with the attendant risk of duplicates, just to fix two. The solution is to design (and consume) batch APIs that return a **per-item result** and retry only the ones that failed with a transient error.

```
def batch_with_retry(send_batch, items, *, max_retries=4, base=0.5, cap=10.0):
    """send_batch(items) -> list of results aligned with items:
    each one {'ok': bool, 'retryable': bool, 'value': ...}."""
    import random, time
    pending = list(items)
    results = {}
    for attempt in range(max_retries + 1):
        outcomes = send_batch(pending)
        retry_next = []
        for item, out in zip(pending, outcomes):
            if out["ok"]:
                results[id(item)] = out["value"]
            elif out["retryable"] and attempt < max_retries:
                retry_next.append(item)      # only these are attempted again
            else:
                results[id(item)] = out       # definitive failure
        if not retry_next:
            break
        time.sleep(random.uniform(0, min(cap, base * 2 ** attempt)))
        pending = retry_next
    return [results[id(it)] for it in items]
```

This pattern, retrying the subset that failed rather than the whole batch, reduces wasted work and avoids duplicates. Combine it with per-item idempotency so that, even if an item is retried one too many times, the effect is safe. It is the difference between a retry that costs two operations and one that costs a hundred.

## Case study: retrying against AI model APIs

If you work with language models, OpenAI, Anthropic, or any provider, retries stop being optional: these APIs return `429` frequently when you exceed your rate limits, and `500` / `529` (overloaded) errors during demand spikes. It is one of the places where good backoff with jitter makes the difference between a stable application and one that falls over every time there is traffic.

The particulars of retrying LLM calls:

- **Respect rate limits**: `429` s usually come with `Retry-After` or with headers like `x-ratelimit-reset`. Use them; the provider knows exactly when your quota resets.
- **Distinguish token and request limits**: you may have request quota but exhaust token quota. If the error is about tokens, retrying the same giant request will fail the same way; sometimes the fix is to split the input, not to retry.
- **Mind the cost**: every retry of a call that already generated tokens (for example, if it fails mid-stream) may bill you again. Cap the attempts and prefer resuming over restarting when the provider allows it.
- **Idempotency**: many AI APIs accept an idempotency header so a retry after a timeout does not generate (or charge for) the same generation twice.

A wrapper that retries LLM calls respecting `Retry-After` and applying full jitter as a plan B:

```
import time, random, httpx

RETRYABLE = {429, 500, 502, 503, 529}   # 529 = "overloaded" on some providers

def call_llm(client, payload, *, max_retries=5, base=1.0, cap=60.0):
    for attempt in range(max_retries + 1):
        resp = client.post("/v1/messages", json=payload, timeout=60)
        if resp.status_code not in RETRYABLE:
            resp.raise_for_status()
            return resp.json()
        if attempt == max_retries:
            resp.raise_for_status()
        # The provider tells you when to retry; if not, full jitter
        ra = resp.headers.get("retry-after")
        delay = float(ra) if ra and ra.isdigit() else random.uniform(0, min(cap, base * 2 ** attempt))
        time.sleep(delay)
```

For **streaming** responses (Server-Sent Events), retrying has a nuance: if the connection drops mid-stream, retrying from scratch repeats tokens and cost. Ideally you retry only the connection setup (before receiving the first token) and, if the cut happens mid-stream, treat it as a recoverable failure at the application level, show what was received and offer "continue", instead of relaunching the full generation. And as always: set a cap on attempts and a deadline, because a user waiting for a model's response will not tolerate a minute of silent retries.

## Retrying at startup: waiting for dependencies to be ready

There is a retry scenario that is frequently forgotten: a service's **startup**. In a deployment with containers or orchestrators, your application may start before its database, its cache, or the service it depends on are accepting connections. If your process gives up on the first connection failure, it will enter a restart loop (the classic `CrashLoopBackOff`) until, by luck, everything is ready at once.

The solution is to retry the initial connection with backoff, but with different parameters than a normal request: here you do want **many attempts** and a generous deadline, because a starting dependency can take tens of seconds. The idea is to wait actively instead of dying.

```
import time, random

def wait_for_dependency(check, *, timeout=60.0, base=0.5, cap=5.0):
    """Retry check() until it returns True or the global timeout expires.
    check() raises or returns False if the dependency is not ready."""
    start = time.monotonic()
    attempt = 0
    while True:
        try:
            if check():
                return True
        except Exception:
            pass
        if time.monotonic() - start >= timeout:
            raise TimeoutError("dependency was not ready in time")
        time.sleep(random.uniform(0, min(cap, base * 2 ** attempt)))
        attempt += 1

# Example: wait for Postgres before starting the app
wait_for_dependency(lambda: db_ping(), timeout=90)
```

Even so, retrying at startup does not replace **health checks**: expose a "readiness" endpoint that is green only when all your dependencies respond, and let the orchestrator hold traffic until then. The retry solves startup; the health check keeps you from receiving requests before you are ready.

## A retry is not a recovery: have a plan B

One last conceptual nuance that saves a lot of grief: a retry assumes the operation **will eventually work** if you insist. But sometimes it will not work within the time you have, and exhausting the retries is not the end of the story but the moment to activate a **fallback** (a degraded alternative).

Examples of graceful degradation when retries run out: serve a cached value even if slightly stale instead of showing an error; use a reasonable default when a personalization service does not respond; enqueue the operation to process later instead of failing in front of the user; or show a reduced version of the page that does not depend on the downed service. The mental hierarchy is clear: first attempt, then retry with backoff, and when the budget runs out, degrade gracefully. A resilient system is not one that never fails, but one that stays useful even when one of its parts does.

## Putting it all together: a layered defense

The patterns in this guide do not compete with each other; they **stack**. The circuit breaker decides whether it is even worth attempting; retry with backoff and jitter absorbs transient failures; the retry budget keeps amplification from running away; the deadline caps the total time; and the fallback steps in when, despite everything, it could not be done. Seeing them combined in a single function helps you understand how they fit.

```
import time, random

def resilient_call(fn, *, breaker, budget, cache_key=None, cache=None,
                   max_retries=3, base=0.2, cap=3.0, total_deadline=8.0):
    """Composes the patterns: circuit breaker + retry budget + backoff/jitter
    + deadline + graceful fallback. Returns the value or a degraded one."""
    start = time.monotonic()
    budget.record_request()
    attempt = 0
    while True:
        try:
            return breaker.call(fn)              # breaker fails fast if open
        except Exception:
            remaining = total_deadline - (time.monotonic() - start)
            can_more = (attempt < max_retries and remaining > 0
                        and budget.can_retry())    # respect the retry budget
            if not can_more:
                if cache is not None and cache_key in cache:
                    return cache[cache_key]        # graceful fallback: stale value
                raise                              # nothing left to do
            delay = min(random.uniform(0, min(cap, base * 2 ** attempt)), remaining)
            time.sleep(delay)
            attempt += 1
```

Read it from the outside in: first the **breaker** (is the service healthy enough to even try?), then the **retry** with its backoff, governed by the **budget** and the **deadline**, and finally the **fallback** to a cached value when there is no room left. Each layer has a single responsibility, and together they turn a momentary failure of a dependency into something the user, at best, does not even notice and, at worst, degrades gracefully instead of breaking.

You do not need every layer from day one. Start with retry plus jitter and the limits; add the circuit breaker when you have dependencies that sometimes go fully down; bring in the budget when you operate at a scale where amplification is a real risk; and design the fallback for the paths where a visible error is unacceptable. Resilience is built in layers, and each one is justified by a specific kind of failure you have seen (or foreseen) in production.

## Production checklist

Before signing off on your retry strategy, review it against this list:

- You retry **only** transient errors (network errors, 429, and 5xx), never client 4xx.
- You use **exponential backoff with full jitter** as the default.
- `Retry-After` takes priority over your backoff and you parse its two formats (seconds and date).
- Every write you retry is **idempotent** (idempotency key where needed).
- You have a **per-attempt timeout** and a **total deadline**, and you propagate the deadline across services.
- You retry in a **single layer** and have a **retry budget** to curb amplification.
- A **circuit breaker** cuts things off when a service stops responding for a sustained period.
- You export retry **metrics** and alert when the rate or the exhaustions rise.
- You have **deterministic tests** covering both what is retried and what is not.

## Conclusion

A good retry is one of the best returns on investment in reliability: a few well-thought-out lines turn transient failures into something the user never sees. But a bad retry is one of the most expensive mistakes there is, because it amplifies problems exactly when the system is most fragile. The difference lies in the details we have covered: jitter so as not to synchronize clients, idempotency so repeating is safe, timeouts and deadlines so nothing hangs, budgets

and circuit breakers so as not to amplify, and observability so you find out before it is too late. Start with the essential recipe, exponential backoff with full jitter, only transient errors, respect `Retry-After`, and require idempotency, and add the production patterns as your scale demands. Your future self, the one on call during the next incident, will thank you.