

PuigFlows

Timeouts y Deadlines en Producción: Presupuestos de Tiempo, Propagación entre Servicios y Reintentos que No Multiplican la Carga

Timeouts and Deadlines in Production: Time Budgets, Propagation Across
Services and Retries That Don't Multiply Load

Guia / Guide - Edicion ampliada / Expanded edition - 10 October 2026

Josue Puig - puigflows.com

Contenido / Contents: Espanol (primera parte) - English (second part).

Timeouts y Deadlines en Producción: Presupuestos de Tiempo, Propagación entre Servicios y Reintentos que No Multiplican la Carga

Casi todos los incidentes en cascada que he visto empiezan igual: una dependencia se pone lenta (no cae, se pone lenta), las peticiones se quedan esperando, los hilos o las conexiones se agotan y, en cuestión de minutos, un servicio que funcionaba perfectamente deja de responder porque estaba esperando a otro. La causa raíz casi siempre es la misma: nadie decidió cuánto tiempo estaba dispuesto a esperar. Esta guía trata de tomar esa decisión de forma deliberada, repartirla entre capas y propagarla de punta a punta.

1. Por qué un timeout ausente es un fallo de diseño

Una petición sin límite de tiempo retiene recursos (una conexión, un hilo, una corrutina, memoria, una conexión a la base de datos) durante un tiempo indefinido. La ley de Little lo resume: el número de peticiones simultáneas en el sistema es igual a la tasa de llegada multiplicada por el tiempo que cada una tarda en resolverse. Si tu servicio recibe 200 peticiones por segundo y una dependencia pasa de responder en 50 ms a no responder, en cinco segundos tienes 1.000 peticiones colgadas. Con un pool de 100 conexiones, ya no atiendes a nadie, ni siquiera a las rutas que no usan esa dependencia.

Lo traicionero es que muchos clientes HTTP no tienen límite por defecto. La librería requests de Python espera indefinidamente si no pasas `timeout=`. `httpx` usa 5 segundos por defecto, que ya es algo, pero ese valor se aplica a cada fase, no al total. En Go, `http.Client{}` sin configurar no tiene `timeout`. Empieza por auditar eso: un grep de las llamadas salientes sin `timeout` explícito suele ser el trabajo más rentable del trimestre.

2. Anatomía de un timeout: no son uno, son cuatro

Cuando dices «timeout de 5 segundos» estás ocultando varias fases distintas, cada una con su propio modo de fallo:

- Pool: el tiempo que esperas para obtener una conexión libre del pool. Si es largo, estás encolando trabajo dentro de tu propio proceso.
- Connect: establecer la conexión TCP (y TLS). Un host inalcanzable puede tardar decenas de segundos en fallar si no lo acotas; en una red sana debería ser cuestión de milisegundos.
- Write: enviar la petición.
- Read: esperar datos del servidor. Ojo: es el tiempo máximo entre dos lecturas del socket, no el tiempo total de la respuesta.

Este último detalle es la trampa clásica. Un servidor que envía un byte cada 4 segundos nunca dispara un `read timeout` de 5 segundos, pero puede mantenerte conectado durante horas. Por eso necesitas, además, un límite de tiempo total. En `httpx` se configura así:

```
import asyncio
import httpx

# Timeouts por fase: connect corto, read moderado, pool muy corto
TIMEOUTS = httpx.Timeout(connect=0.5, read=2.0, write=2.0, pool=0.2)

async def fetch_profile(client: httpx.AsyncClient, user_id: str) -> dict:
    # asyncio.timeout (Python 3.11+) impone el límite TOTAL de la operación,
    # incluida la espera del pool, el connect y todas las lecturas.
    async with asyncio.timeout(3.0):
        resp = await client.get(
```

```

        f"https://profiles.internal/users/{user_id}",
        timeout=TIMEOUTS,
    )
    resp.raise_for_status()
    return resp.json()

client = httpx.AsyncClient(
    limits=httpx.Limits(max_connections=100, max_keepalive_connections=20),
)

```

Cada capa cumple un papel: los timeouts por fase detectan fallos específicos rápido (un connect que tarda 500 ms en una red interna ya es una señal), y el `asyncio.timeout` externo garantiza que, pase lo que pase, la operación completa termina. Si el límite total salta, se lanza `TimeoutError` y la corrutina interna se cancela, liberando la conexión.

3. Timeout frente a deadline

Un timeout es una duración relativa («espera como máximo 2 segundos»). Un deadline es un instante absoluto («esta petición deja de tener sentido a las 12:00:03.500»). La diferencia importa en cuanto hay más de un salto. Imagina la cadena gateway -> pedidos -> inventario -> base de datos, cada uno con un timeout de 3 segundos. Si inventario tarda 2,8 segundos en responder a pedidos, pedidos responde al gateway a los 3,1 s, y el gateway ya se ha rendido a los 3,0 s: el cliente recibió un error, pero pedidos e inventario han trabajado en vano durante todo ese tiempo. Peor aún: si el cliente reintenta, la carga se multiplica justo cuando el sistema está peor.

La solución es propagar el tiempo restante. El gateway fija un presupuesto total (por ejemplo, 3 s), y cada servicio pasa al siguiente lo que le queda, restando lo ya consumido. Cuando el presupuesto se agota, todos dejan de trabajar. gRPC hace esto de forma nativa con la cabecera `grpc-timeout`; en HTTP tienes que hacerlo tú, con una cabecera propia. Conviene propagar tiempo restante relativo (en milisegundos) en lugar de un instante absoluto, porque así no dependes de que los relojes de las máquinas estén sincronizados.

4. Un presupuesto de tiempo propagable en Python

Este módulo guarda el deadline en una `ContextVar` (que se propaga correctamente entre corrutinas) usando el reloj monótonico, y ofrece una función para calcular cuánto puedes gastar en la siguiente llamada:

```

# deadline.py
import contextvars
import time

HEADER = "x-request-timeout-ms"

_deadline: contextvars.ContextVar[float | None] = contextvars.ContextVar(
    "deadline", default=None
)

class DeadlineExceeded(Exception):
    """El presupuesto de tiempo de la petición se ha agotado."""

def start(header_value: str | None, default_s: float, max_s: float) -> float:
    """Fija el deadline a partir de la cabecera entrante (si es válida)."""
    budget = default_s
    if header_value:
        try:
            budget = float(header_value) / 1000.0
        except ValueError:
            pass
    budget = max(0.0, min(budget, max_s)) # nunca confíes ciegamente en el cliente
    _deadline.set(time.monotonic() + budget)
    return budget

```

```

def remaining() -> float:
    d = _deadline.get()
    if d is None:
        raise RuntimeError("deadline no inicializado")
    return d - time.monotonic()

def budget(reserve: float = 0.05, cap: float | None = None) -> float:
    """Segundos que puedes gastar en la próxima llamada, dejando un margen
    (reserve) para procesar la respuesta y devolverla."""
    left = remaining() - reserve
    if left <= 0:
        raise DeadlineExceeded()
    return min(left, cap) if cap is not None else left

```

Y así se engancha a FastAPI. El middleware lee la cabecera, fija el deadline y envuelve toda la petición en `asyncio.timeout`, de modo que el handler se cancela si se agota el presupuesto:

```

import asyncio
import httpx
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
import deadline

app = FastAPI()
client = httpx.AsyncClient()

@app.middleware("http")
async def deadline_middleware(request: Request, call_next):
    total = deadline.start(
        request.headers.get(deadline.HEADER), default_s=3.0, max_s=10.0
    )
    try:
        async with asyncio.timeout(total):
            return await call_next(request)
    except (TimeoutError, deadline.DeadlineExceeded):
        return JSONResponse({"error": "deadline_exceeded"}, status_code=504)

async def call_downstream(url: str) -> httpx.Response:
    t = deadline.budget(reserve=0.05, cap=2.0) # nunca más de 2 s por llamada
    return await client.get(
        url,
        timeout=httpx.Timeout(t, connect=min(t, 0.5)),
        # propagamos lo que queda: el siguiente servicio hace la misma cuenta
        headers={deadline.HEADER: str(int(t * 1000))},
    )

@app.get("/orders/{order_id}")
async def get_order(order_id: str):
    inv = await call_downstream(f"http://inventory/items?order={order_id}")
    return {"order": order_id, "inventory": inv.json()}

```

Hay un detalle de diseño en `default_s` y `max_s`: si no llega cabecera usas un valor por defecto, y si llega una absurdamente grande la recortas. Un cliente (o un bug) que envíe un presupuesto de una hora no debe poder retener tus recursos durante una hora.

5. Reintentos dentro del presupuesto

Los reintentos y los deadlines se complementan, pero solo si los reintentos respetan el presupuesto. Un bucle de tres intentos de 2 segundos con una petición que solo dispone de 3 segundos es un error de diseño: el tercer intento nunca tendrá tiempo de completarse. La regla: cada intento recibe `min(tiempo_por_intento, tiempo_restante)`, y si queda menos que el tiempo mínimo razonable para una llamada, no se intenta.

```

import asyncio
import random
import httpx
import deadline

```

```

RETRYABLE = (httpx.TimeoutException, httpx.ConnectError)

async def get_with_retry(client, url, attempts=3, per_try=1.0, min_useful=0.1):
    last_exc = None
    for i in range(attempts):
        try:
            t = deadline.budget(reserve=0.05, cap=per_try)
        except deadline.DeadlineExceeded:
            break
        if t < min_useful:           # no merece la pena lanzar una llamada condenada
            break
        try:
            return await client.get(
                url, timeout=t, headers={deadline.HEADER: str(int(t * 1000))}
            )
        except RETRYABLE as exc:
            last_exc = exc
            # backoff exponencial con jitter completo, acotado por lo que queda
            sleep = random.uniform(0, 0.05 * 2**i)
            await asyncio.sleep(min(sleep, max(deadline.remaining() - 0.1, 0)))
    raise last_exc or deadline.DeadlineExceeded()

```

Dos advertencias importantes. Primero, solo reintentes operaciones idempotentes (o con clave de idempotencia): un timeout en una escritura no significa que haya fallado, significa que no sabes si se aplicó. Segundo, en un sistema con varias capas, si cada capa reintenta 3 veces, una petición puede convertirse en $3 \times 3 \times 3 = 27$ llamadas al nivel más bajo. Reintenta en una sola capa (normalmente la más cercana a la dependencia) y usa un presupuesto de reintentos global (por ejemplo, un máximo del 10 % de las llamadas pueden ser reintentos).

6. Go: contexto y deadlines nativos

Go lo hace especialmente bien porque `context.Context` lleva el deadline y casi toda la biblioteca estándar lo respeta. Esto es un cliente configurado con todas las fases acotadas y un handler que deriva su contexto del de la petición entrante:

```

package main

import (
    "context"
    "errors"
    "net"
    "net/http"
    "time"
)

var client = &http.Client{
    // Timeout total como red de seguridad (incluye leer el cuerpo)
    Timeout: 5 * time.Second,
    Transport: &http.Transport{
        DialContext: (&net.Dialer{
            Timeout: 500 * time.Millisecond, // connect
            KeepAlive: 30 * time.Second,
        }).DialContext,
        TLSHandshakeTimeout: 500 * time.Millisecond,
        ResponseHeaderTimeout: 2 * time.Second, // hasta recibir las cabeceras
        MaxIdleConnsPerHost: 20,
        IdleConnTimeout: 90 * time.Second,
    },
}

func ordersHandler(w http.ResponseWriter, r *http.Request) {
    // Deriva del contexto de la petición: si el cliente se va, se cancela todo.
    ctx, cancel := context.WithTimeout(r.Context(), 800*time.Millisecond)
    defer cancel()

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "http://inventory/items", nil)
    if err != nil {
        http.Error(w, "bad request", http.StatusInternalServerError)
    }
}

```

```

        return
    }
    resp, err := client.Do(req)
    if err != nil {
        if errors.Is(err, context.DeadlineExceeded) {
            http.Error(w, "upstream timeout", http.StatusGatewayTimeout)
            return
        }
        http.Error(w, "upstream error", http.StatusBadGateway)
        return
    }
    defer resp.Body.Close()
    w.WriteHeader(resp.StatusCode)
}

func main() {
    srv := &http.Server{
        Addr:           ":8080",
        Handler:        http.HandlerFunc/ordersHandler),
        ReadHeaderTimeout: 2 * time.Second, // protege de clientes lentos (slowloris)
        ReadTimeout:      5 * time.Second,
        WriteTimeout:      10 * time.Second,
        IdleTimeout:       60 * time.Second,
    }
    _ = srv.ListenAndServe()
}

```

Con gRPC es todavía más cómodo: el deadline del contexto viaja en la cabecera `grpc-timeout` y el servidor lo recibe como el deadline de su propio contexto. Si ese contexto se usa en las llamadas salientes, la propagación es automática. En Python se pasa con `stub.GetItem(request, timeout=0.5)`, y en el servidor puedes consultar `context.time_remaining()`.

7. Repartir el presupuesto por capas

La regla para dimensionar es de fuera hacia dentro: el timeout de cada capa debe ser estrictamente menor que el de la capa que la contiene, con margen para el procesamiento intermedio. Un esquema razonable para una petición de usuario con objetivo de 3 segundos:

- Navegador o cliente móvil: 10 s (lo que el usuario tolera antes de abandonar).
- Balanceador o API gateway: 4 s.
- Servicio de aplicación: presupuesto de 3 s, con llamadas a dependencias limitadas a 1–1,5 s cada una.
- Base de datos: `statement_timeout` de 1 s en rutas de lectura interactivas.

¿Cómo elegir los números? Mide la latencia real de cada dependencia y fija el timeout cerca de su p99 o p99,9 más un margen, no a ojo. Un timeout por debajo del p99 provoca fallos artificiales en el 1 % de las peticiones; uno 20 veces por encima del p99 equivale a no tenerlo cuando algo se rompe. En un proxy como Envoy, el equivalente sería:

```

route_config:
  virtual_hosts:
    - name: orders
      domains: ["*"]
      routes:
        - match: { prefix: "/orders" }
          route:
            cluster: orders_service
            timeout: 3s                # deadline total de la ruta
            retry_policy:
              retry_on: connect-failure,refused-stream,reset
              num_retries: 1
              per_try_timeout: 1.5s    # cada intento, acotado

```

8. Base de datos y balanceadores

La base de datos es otra dependencia que debe respetar tu presupuesto. PostgreSQL ofrece límites propios que deberías fijar explícitamente por rol o por transacción:

```
-- Valores por rol de la aplicación (se aplican a cada sesión nueva)
ALTER ROLE app_web SET statement_timeout = '2s';
ALTER ROLE app_web SET lock_timeout = '500ms';
ALTER ROLE app_web SET idle_in_transaction_session_timeout = '10s';

-- Ajuste local a una transacción concreta (p. ej., una consulta de informe)
BEGIN;
SET LOCAL statement_timeout = '30s';
SELECT /* informe pesado */ count(*) FROM events;
COMMIT;
```

`statement_timeout` aborta consultas largas, `lock_timeout` evita esperar indefinidamente un bloqueo (la causa típica de colas de peticiones detrás de una migración) e `idle_in_transaction_session_timeout` mata sesiones que abren una transacción y se olvidan de cerrarla, reteniendo bloqueos y bloqueando el vacuum. Desde código, con `psycopg3`, ajusta el valor con `SELECT set_config('statement_timeout', %s, true)` usando el tiempo restante del presupuesto: así la base de datos nunca trabaja más allá de lo que tu petición puede esperar.

Con los balanceadores, el problema habitual es el desajuste de timeouts de inactividad. Un ALB de AWS cierra por defecto las conexiones inactivas a los 60 segundos. Si tu servidor cierra las conexiones keep-alive antes que el balanceador (por ejemplo, el `keepAliveTimeout` de 5 s por defecto de Node.js), el balanceador intentará reutilizar una conexión que acaba de cerrarse y devolverá 502 de forma intermitente. La regla: el keep-alive del servidor de aplicación debe ser mayor que el idle timeout del balanceador que tiene delante.

9. Errores habituales

- Confundir timeout con cancelación. Que el cliente deje de esperar no detiene el trabajo en el servidor. Si el handler no comprueba el contexto o el deadline, seguirá consumiendo CPU y conexiones de base de datos para nadie. Comprueba `remaining()` antes de cada paso caro y cancela cuando se agota.
- El mismo timeout en todas las capas. Si todos esperan 3 s, cuando el nivel inferior tarda 3 s todos fallan a la vez y ninguno puede devolver una respuesta degradada con elegancia. Cada capa interior debe agotarse antes.
- Timeouts en escrituras sin idempotencia. Tras un timeout no sabes si la operación se aplicó. Usa claves de idempotencia y consulta el estado antes de reintentar.
- Tragarse la cancelación. En `asyncio`, un `except Exception` demasiado amplio o un `except asyncio.CancelledError`: `pass` impide que el timeout funcione. `CancelledError` hereda de `BaseException` a propósito; relánzala siempre tras limpiar.
- Timeouts demasiado generosos «por si acaso». Un timeout de 60 s en una llamada que normalmente tarda 50 ms protege de nada: bajo fallo, el sistema se saturará mucho antes de que salte.
- Olvidar el timeout del pool. Si esperar una conexión libre no tiene límite, el fallo de una dependencia se convierte en una cola infinita dentro de tu propio proceso.
- Confiar en un deadline entrante sin límite superior. Siempre recorta el valor recibido a un máximo razonable.

10. Observabilidad

Un timeout silencioso es casi tan malo como no tenerlo. Registra, como mínimo, estas métricas: número de timeouts por dependencia y por fase (`connect`, `read`, `pool`, `deadline total`), el tiempo restante de

presupuesto al entrar en cada servicio (un histograma; si la mediana ya es baja, algo aguas arriba consume demasiado) y la proporción de peticiones que terminan en 504. Una alerta útil es la que vigila el ratio de timeouts, no solo su valor absoluto: pasar del 0,1 % al 2 % suele ser la primera señal visible de una dependencia degradada, mucho antes de que se vea la saturación.

11. Dimensionar con números: la ley de Little aplicada a pools y timeouts

Hasta aquí hemos hablado de timeouts como si fueran una cuestión de criterio. En realidad se pueden calcular, y el cálculo es lo que separa un valor razonable de uno que «parece bien». La ley de Little dice que el número medio de peticiones en vuelo es $L = \lambda \times W$, donde λ es la tasa de llegada y W el tiempo medio que cada petición permanece en el sistema. Cuando la dependencia está sana, W es su latencia normal. Cuando se cuelga, W pasa a ser, como mucho, tu timeout. Y ese «como mucho» es precisamente la razón de que el timeout exista: es el techo de W , y por tanto el techo de L .

Pongamos números. Tu servicio recibe $\lambda = 200$ peticiones por segundo y cada una llama a un servicio de perfiles que responde en 50 ms. En régimen normal hay $200 \times 0,05 = 10$ llamadas concurrentes: un pool de 100 conexiones va sobradísimo. Ahora el servicio de perfiles se cuelga. Si tu timeout es de 3 segundos, el número de llamadas en vuelo tiende a $200 \times 3 = 600$, seis veces el pool, y lo alcanza (100 conexiones) en medio segundo. Si el timeout es de 500 ms, el máximo es $200 \times 0,5 = 100$: justo el tamaño del pool, así que al menos las rutas que no usan esa dependencia siguen respirando, siempre que el pool de perfiles esté aislado del resto.

De ahí sale una regla de dimensionado muy útil: el timeout de una llamada debe cumplir $\lambda \times \text{timeout} \leq \text{tamaño del pool dedicado}$, o bien debes aceptar explícitamente que, en caso de fallo, parte de la carga se rechazará en la puerta (load shedding). No hay una tercera opción: o limitas el tiempo, o limitas la concurrencia, o te saturas. Este pequeño script lo calcula para tus propias cifras y te dice cuánto tarda en saturarse el pool:

```
# little.py - cuánto aguanta un pool cuando una dependencia se cuelga
from dataclasses import dataclass

@dataclass
class Dependency:
    name: str
    rate_rps: float          # lambda: peticiones/s que la usan
    p50_s: float             # latencia sana
    p999_s: float            # latencia sana en la cola
    timeout_s: float         # el timeout que has configurado
    pool_size: int           # conexiones o permisos dedicados

def analyze(d: Dependency) -> dict:
    healthy_l = d.rate_rps * d.p50_s          # L en régimen normal
    tail_l = d.rate_rps * d.p999_s            # L con la cola de latencia
    worst_l = d.rate_rps * d.timeout_s        # L si la dependencia se cuelga
    secs_to_exhaust = d.pool_size / d.rate_rps # tiempo hasta agotar el pool
    return {
        "dependencia": d.name,
        "L_sano": round(healthy_l, 1),
        "L_cola": round(tail_l, 1),
        "L_peor_caso": round(worst_l, 1),
        "pool": d.pool_size,
        "se_satura": worst_l > d.pool_size,
        "segundos_hasta_saturar": round(secs_to_exhaust, 2),
        # pool mínimo razonable: cubrir la cola con un 50 % de margen
        "pool_recomendado": int(tail_l * 1.5) + 1,
        # timeout máximo que no desborda el pool
        "timeout_max_sin_saturar": round(d.pool_size / d.rate_rps, 3),
    }

if __name__ == "__main__":
    perfiles = Dependency("perfiles", 200, 0.05, 0.40, timeout_s=3.0, pool_size=100)
    for k, v in analyze(perfiles).items():
```

```
print(f"{k:28} {v}")
```

Con esas cifras el script te dirá que el peor caso son 600 llamadas concurrentes, que el pool se agota en 0,5 segundos y que el timeout máximo que no desborda un pool de 100 es de 0,5 s. Fíjate en que el resultado no es «pon el timeout en 500 ms» a ciegas: el p99,9 sano de esta dependencia es de 400 ms, así que 500 ms deja un margen muy estrecho. Esa tensión es la información valiosa. O amplías el pool (con cuidado: cada conexión tiene coste en el otro extremo), o reduces la tasa con una caché, o aceptas rechazar peticiones cuando el pool se llene en lugar de encolarlas. Lo que no puedes hacer es ignorarla.

12. Cancelación estructurada en Python: `asyncio.timeout_at`, `TaskGroup` y `ExceptionGroup`

En el apartado 2 usamos `asyncio.timeout()` con una duración relativa. Para presupuestos propagados es más natural su hermano `asyncio.timeout_at()`, que recibe un instante absoluto del reloj del bucle de eventos (`loop.time()`). Es exactamente la semántica de un deadline: puedes calcularlo una vez a la entrada de la petición y usarlo en cualquier punto de la cadena sin ir restando a mano. Además, el objeto devuelto por `timeout()` admite `reschedule()` para mover el límite mientras la operación está en curso, útil cuando quieres conceder tiempo extra a una operación crítica.

Lo que casi nadie tiene claro es cómo se materializa el timeout. Cuando vence, `asyncio` no «mata» nada: cancela la tarea que está dentro del bloque lanzando `CancelledError` en el siguiente `await`, y al salir del bloque `async with` lo convierte en `TimeoutError`. Esto tiene dos consecuencias prácticas. Una: el código dentro del bloque solo se interrumpe en puntos de `await`, así que una función síncrona de dos segundos (un `json.loads` gigante, una expresión regular patológica, un driver bloqueante) no se puede interrumpir y el timeout llegará tarde. Dos: si dentro del bloque capturas `CancelledError` y no la relanzas, el timeout deja de funcionar en silencio.

Para varias llamadas en paralelo, `TaskGroup` (Python 3.11+) es la herramienta correcta frente a `gather`: si una tarea falla, cancela a las demás, y todas las excepciones se agrupan en un `ExceptionGroup` que se maneja con `except*`. Combinado con un timeout externo obtienes un contrato limpio: o todo termina dentro del presupuesto, o todo se cancela y no queda ninguna tarea huérfana consumiendo recursos.

```
import asyncio
import httpx

async def fetch(client: httpx.AsyncClient, url: str) -> dict:
    resp = await client.get(url)
    resp.raise_for_status()
    return resp.json()

async def build_order_view(client: httpx.AsyncClient, order_id: str, deadline_at: float) -> dict:
    """Consulta tres servicios en paralelo bajo UN mismo deadline absoluto.
    deadline_at está expresado en el reloj del bucle (loop.time())."""
    try:
        async with asyncio.timeout_at(deadline_at):
            async with asyncio.TaskGroup() as tg:
                order = tg.create_task(fetch(client, f"http://orders/{order_id}"))
                items = tg.create_task(fetch(client, f"http://inventory/items?order={order_id}"))
                price = tg.create_task(fetch(client, f"http://pricing/quote?order={order_id}"))
            return {"order": order.result(), "items": items.result(), "price": price.result()}
    except* httpx.HTTPStatusError as eg:
        # una dependencia respondió con error: aquí decides si degradas o propagas
        statuses = [e.response.status_code for e in eg.exceptions]
        raise RuntimeError(f"dependencias con error: {statuses}") from eg
    # TimeoutError NO se captura aquí: sube al middleware, que responde 504

# uso: el deadline se fija una vez al entrar la petición
async def handler(order_id: str):
    loop = asyncio.get_running_loop()
    deadline_at = loop.time() + 2.5
    async with httpx.AsyncClient() as client:
        return await build_order_view(client, order_id, deadline_at)
```

Un matiz sobre `asyncio.shield()`: sirve para proteger una operación que no debe cancelarse a medias (por ejemplo, confirmar una transacción), pero cuidado, porque «protegerla» significa que seguirá ejecutándose después de que tu petición haya expirado. Úsala solo para trabajo corto y cuyo resultado importe incluso si nadie lo espera, y mide siempre cuántas tareas protegidas quedan vivas. Un `shield` sin límite propio es un `timeout` que has apagado con buenos modales.

13. Fan-out con degradación: dependencias críticas y opcionales

Un `TaskGroup` puro es un contrato de «todo o nada», y eso es demasiado rígido para muchas páginas reales. En una ficha de producto, el precio y la disponibilidad son críticos; las recomendaciones, las reseñas y el contador de «gente viendo esto» no lo son. Si las recomendaciones tardan 2 segundos y el usuario habría aceptado una página sin ellas, fallar toda la petición es peor que servirla incompleta. La solución es clasificar cada dependencia y darle un sub-presupuesto propio, siempre recortado por lo que queda del presupuesto global.

```
import asyncio
import logging
from typing import Awaitable, TypeVar

log = logging.getLogger("fanout")
T = TypeVar("T")

async def optional(name: str, coro: Awaitable[T], timeout_s: float, fallback: T) -> T:
    """Ejecuta una dependencia OPCIONAL: si agota su sub-presupuesto o falla,
    devuelve el fallback en lugar de tumbar la petición."""
    try:
        async with asyncio.timeout(timeout_s):
            return await coro
    except TimeoutError:
        log.warning("optional_timeout", extra={"dep": name, "timeout_s": timeout_s})
        return fallback
    except Exception:
        log.exception("optional_failed", extra={"dep": name})
        return fallback

async def product_page(client, product_id: str, remaining_s: float) -> dict:
    # las críticas comparten el presupuesto; las opcionales reciben una fracción
    reco_budget = min(0.4, remaining_s * 0.25)
    reviews_budget = min(0.3, remaining_s * 0.20)
    async with asyncio.TaskGroup() as tg:
        price = tg.create_task(fetch_json(client, f"http://pricing/{product_id}"))
        stock = tg.create_task(fetch_json(client, f"http://stock/{product_id}"))
        reco = tg.create_task(optional("reco", fetch_json(client, f"http://reco/{product_id}"),
        reco_budget, fallback=[]))
        reviews = tg.create_task(optional("reviews", fetch_json(client,
        f"http://reviews/{product_id}"), reviews_budget, fallback=None))
    return {
        "price": price.result(),
        "stock": stock.result(),
        "recommendations": reco.result(), # lista vacía si no llegó a tiempo
        "reviews": reviews.result(),     # None: el frontend oculta el bloque
    }
```

Hay una decisión de producto escondida aquí que conviene hacer explícita con el equipo: qué ve el usuario cuando falta algo. Una lista vacía de recomendaciones que el frontend no pinta es una degradación elegante; un spinner eterno no lo es. Y registra cada fallback como una métrica (un contador por dependencia), porque un sistema que degrada en silencio durante semanas es un sistema que ha dejado de tener esa funcionalidad sin que nadie se entere.

14. Peticiones de cobertura (hedging) con presupuesto

Los percentiles altos de latencia casi nunca vienen de una dependencia rota, sino de una réplica lenta puntual: un GC, un vecino ruidoso, un disco con un mal momento. Si tienes varias réplicas, una técnica descrita en el artículo «The Tail at Scale» de Google es la petición de cobertura: lanzas la petición a una

réplica; si no ha respondido pasado un umbral (típicamente el p95 de esa llamada), lanzas una segunda a otra réplica, te quedas con la primera respuesta que llegue y cancelas la otra. El coste es un pequeño porcentaje extra de carga (si el umbral es el p95, aproximadamente un 5 %) a cambio de recortar de forma notable la cola de latencia.

Tres condiciones para que no sea contraproducente. Primero, solo sobre operaciones idempotentes y sin efectos secundarios: lecturas, no escrituras. Segundo, con un presupuesto global de cobertura: si la dependencia está realmente degradada, todas las peticiones superarán el umbral y duplicarías la carga justo cuando menos puede soportarla. Tercero, respetando el deadline: la petición de cobertura solo se lanza si queda tiempo útil.

```
import asyncio
import random

class HedgeBudget:
    """Permite como máximo un porcentaje de peticiones de cobertura (ventana simple)."""
    def __init__(self, max_ratio: float = 0.05):
        self.max_ratio = max_ratio
        self.total = 0
        self.hedged = 0

    def allow(self) -> bool:
        self.total += 1
        if self.total > 1000:
            self.total //= 2
            self.hedged //= 2
        if self.hedged / max(self.total, 1) < self.max_ratio:
            self.hedged += 1
            return True
        return False

budget = HedgeBudget(0.05)

async def hedged_get(client, urls: list[str], hedge_after_s: float, remaining_s: float):
    """Pide a la primera réplica; si tarda más que hedge_after_s y hay presupuesto y tiempo,
    lanza una segunda. Devuelve la primera respuesta y cancela la otra."""
    first = asyncio.create_task(client.get(urls[0]))
    tasks = {first}
    try:
        done, _ = await asyncio.wait({first}, timeout=hedge_after_s)
        if not done and remaining_s > hedge_after_s * 2 and budget.allow():
            tasks.add(asyncio.create_task(client.get(random.choice(urls[1:]))))
        done, _ = await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED, timeout=remaining_s)
        if not done:
            raise TimeoutError("ninguna réplica respondió a tiempo")
        return next(iter(done)).result()
    finally:
        for t in tasks:
            if not t.done():
                t.cancel()
                # no dejes la perdedora consumiendo recursos
```

Mide el efecto antes de generalizarlo: compara el p99 con y sin cobertura sobre tráfico real y vigila las métricas de «peticiones de cobertura lanzadas» y «peticiones de cobertura ganadoras». Si la ganadora casi nunca es la segunda, el umbral está demasiado bajo y solo estás gastando carga.

15. Clientes HTTP en otros lenguajes: lo que cada uno hace por defecto

El apartado 1 mencionó requests, httpx y Go, pero el problema no es de un lenguaje, es de cada cliente concreto. Conviene conocer los valores por defecto de los que usas, porque son la configuración que estás ejecutando hoy aunque nadie la haya escrito. Como orientación (comprueba siempre la versión que tienes instalada): requests y urllib3 no imponen ningún timeout; httpx impone 5 segundos por fase; aiohttp usa un timeout total de 5 minutos, que en la práctica equivale a no tenerlo para una llamada interactiva; el cliente de Go estándar no tiene ninguno; y fetch en Node (undici) espera hasta 300 segundos por las cabeceras y

por el cuerpo. En todos los casos, el valor por defecto es demasiado generoso para un servicio que atiende peticiones de usuarios.

En Node y en el navegador, la pieza clave es `AbortSignal`. `AbortSignal.timeout(ms)` crea una señal que se cancela sola, y `AbortSignal.any([...])` combina varias, de modo que puedes cancelar la llamada tanto si vence el presupuesto como si el cliente original cierra la conexión. Este es el equivalente del middleware de `deadline` en TypeScript:

```
// deadline.ts - presupuesto de tiempo propagable en Node 20+
import { AsyncLocalStorage } from "node:async_hooks";

const HEADER = "x-request-timeout-ms";
const store = new AsyncLocalStorage<{ deadlineAt: number; signal: AbortSignal }>();

export function withDeadline<T>(  
  incoming: string | undefined,  
  clientGone: AbortSignal,  
  fn: () => Promise<T>,  
  { defaultMs = 3000, maxMs = 10000 } = {}  
) : Promise<T> {  
  let budget = Number(incoming);  
  if (!Number.isFinite(budget) || budget <= 0) budget = defaultMs;  
  budget = Math.min(budget, maxMs); // nunca confíes ciegamente en el cliente  
  const signal = AbortSignal.any([AbortSignal.timeout(budget), clientGone]);  
  return store.run({ deadlineAt: performance.now() + budget, signal }, fn);  
}  
  
export function remainingMs(): number {  
  const s = store.getStore();  
  if (!s) throw new Error("deadline no inicializado");  
  return s.deadlineAt - performance.now();  
}  
  
export async function callDownstream(url: string, capMs = 2000): Promise<Response> {  
  const s = store.getStore()!;  
  const t = Math.min(remainingMs() - 50, capMs); // 50 ms de reserva  
  if (t <= 0) throw new DOMException("deadline agotado", "TimeoutError");  
  return fetch(url, {  
    // se cancela si vence el sub-presupuesto, el presupuesto global o el cliente se va  
    signal: AbortSignal.any([s.signal, AbortSignal.timeout(t)]),  
    headers: { [HEADER]: String(Math.floor(t)) },  
  });  
}
```

Dos detalles. Usa `performance.now()` (reloj monotónico) y no `Date.now()`, por la misma razón que usamos `time.monotonic()` en Python: un ajuste de NTP en mitad de la petición no debe alargar ni acortar tu presupuesto. Y captura el error correctamente: una señal cancelada por `timeout` lanza una excepción con `name === "TimeoutError"`, mientras que una cancelación manual lanza `"AbortError"`. Distinguir las te permite responder 504 en el primer caso y simplemente abandonar en el segundo (el cliente ya no está para leer la respuesta).

En Java, `HttpClient` separa `connectTimeout` (del cliente) de `timeout` (por petición, que cubre hasta recibir las cabeceras de la respuesta, no la lectura completa del cuerpo). Para un límite total sobre el cuerpo necesitas envolver la llamada asíncrona con `orTimeout()` sobre el `CompletableFuture`. La lección se repite en todos los lenguajes: averigua qué fases cubre exactamente cada ajuste y añade un límite total propio si el cliente no lo ofrece.

16. gRPC: deadlines de verdad, y qué significan sus códigos

gRPC es el único protocolo mainstream que trae los deadlines incorporados en el contrato, y aun así hay dos trampas. La primera: por defecto no hay deadline. Si el cliente no lo fija, la llamada puede esperar indefinidamente, y esa es la configuración que traen todos los clientes generados. La segunda: el deadline solo se propaga si el servidor usa el contexto entrante en sus llamadas salientes. Si en el handler creas un contexto nuevo (`context.Background()` en Go) rompes la cadena sin darte cuenta, y la siguiente capa

vuelve a no tener límite.

```
package main

import (
    "context"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

// Interceptor de servidor: rechaza al instante lo que ya no tiene tiempo útil
// y exige que toda llamada entrante traiga un deadline.
func deadlineGuard(minUseful time.Duration, maxBudget time.Duration) grpc.UnaryServerInterceptor {
    return func(ctx context.Context, req any, info *grpc.UnaryServerInfo,
        handler grpc.UnaryHandler) (any, error) {

        dl, ok := ctx.Deadline()
        if !ok {
            // sin deadline: aplicamos uno por defecto en lugar de esperar para siempre
            var cancel context.CancelFunc
            ctx, cancel = context.WithTimeout(ctx, maxBudget)
            defer cancel()
        } else if time.Until(dl) < minUseful {
            // llega con menos tiempo del que cuesta hacer el trabajo: no lo empieces
            return nil, status.Error(codes.DeadlineExceeded, "presupuesto insuficiente")
        }
        return handler(ctx, req)
    }
}

func inventoryCall(ctx context.Context, c InventoryClient, id string) (*Item, error) {
    // IMPORTANTE: derivamos del ctx entrante; así el deadline del cliente original
    // sigue vigente y se acorta con cada salto.
    ctx, cancel := context.WithTimeout(ctx, 800*time.Millisecond)
    defer cancel()
    return c.GetItem(ctx, &GetItemRequest{Id: id})
}
```

Sobre los códigos: DEADLINE_EXCEEDED significa que venció el tiempo (del cliente o de algún eslabón), y CANCELLED que alguien canceló explícitamente, normalmente porque el cliente se desconectó. No son intercambiables en tus métricas: un pico de DEADLINE_EXCEEDED apunta a lentitud en una dependencia; un pico de CANCELLED apunta a clientes que se impacientan (o a un balanceador con un timeout menor que el tuyo). Si los mezclas en un solo contador de «errores», pierdes la mitad de la señal.

También puedes fijar timeouts y reintentos por método en la configuración de servicio, sin tocar el código. El timeout de ahí es un deadline total por llamada, y la política de reintentos opera dentro de él:

```
{
  "methodConfig": [{
    "name": [{ "service": "inventory.Inventory", "method": "GetItem" }],
    "timeout": "1.5s",
    "retryPolicy": {
      "maxAttempts": 3,
      "initialBackoff": "0.05s",
      "maxBackoff": "0.3s",
      "backoffMultiplier": 2,
      "retryableStatusCodes": ["UNAVAILABLE"]
    }
  }]
}
```

Observa que solo se reintenta UNAVAILABLE y no DEADLINE_EXCEEDED: reintentar tras agotar el deadline total no tiene sentido porque ya no queda presupuesto, y reintentar ante un timeout por intento multiplicaría la carga sobre un servicio lento. Es la misma regla del apartado 5, expresada como configuración.

17. Servidores de aplicación: lo que mata a la petición, y lo que no

Los timeouts del servidor protegen en la otra dirección: de los clientes lentos o maliciosos. Y tienen una peculiaridad que sorprende: muchos timeouts de servidor no cancelan el trabajo, solo cortan la conexión. En Go, `http.TimeoutHandler` devuelve un 503 al cliente cuando vence, pero la goroutine del handler sigue ejecutándose hasta terminar a menos que respete `r.Context()`. El timeout de un worker de gunicorn (`--timeout`, 30 s por defecto) no es un límite de petición: mide cuánto lleva el worker sin dar señales de vida al proceso maestro y, cuando vence, mata el worker entero, con todas las peticiones que estuviera atendiendo en ese momento.

```
# gunicorn.conf.py - un servidor ASGI con límites coherentes
bind = "0.0.0.0:8000"
workers = 4
worker_class = "uvicorn.workers.UvicornWorker"

# El worker se reinicia si no responde al maestro en este tiempo. NO es un límite por petición:
# ponlo por encima de tu deadline máximo para no matar peticiones válidas, pero no a 0.
timeout = 30

# Tiempo para que los workers terminen sus peticiones tras SIGTERM (debe ser menor que
# terminationGracePeriodSeconds de Kubernetes, o el kubelet enviará SIGKILL antes).
graceful_timeout = 20

# Keep-alive del servidor: MAYOR que el idle timeout del balanceador (ALB: 60 s)
keepalive = 65

# Reciclar workers para acotar fugas lentas
max_requests = 5000
max_requests_jitter = 500

// Servidor Go: límites por fase y un handler que respeta el contexto
mux := http.NewServeMux()
mux.HandleFunc("/report", func(w http.ResponseWriter, r *http.Request) {
    // Un handler legítimamente largo amplía SOLO su propio límite de escritura.
    rc := http.NewResponseController(w)
    _ = rc.SetWriteDeadline(time.Now().Add(60 * time.Second))
    // ... y aun así debe respetar r.Context() para cancelarse si el cliente se va.
    generateReport(r.Context(), w)
})

srv := &http.Server{
    Addr:      ":8080",
    Handler:   http.TimeoutHandler(mux, 5*time.Second, "timeout"), // red de seguridad global
    ReadHeaderTimeout: 2 * time.Second,
    ReadTimeout: 5 * time.Second,
    WriteTimeout: 10 * time.Second,
    IdleTimeout: 75 * time.Second, // > idle timeout del balanceador
}
```

Fíjate en la jerarquía: los límites globales (`WriteTimeout`, `TimeoutHandler`) son la red de seguridad, y los endpoints que necesitan más tiempo lo piden de forma explícita y acotada con `ResponseController` (Go 1.20+) en lugar de relajar el límite de todo el servidor. En Node pasa algo equivalente: `server.keepAliveTimeout` (5 s por defecto) debe ser mayor que el `idle timeout` del balanceador, y `server.headersTimeout` debe ser mayor que `keepAliveTimeout`; si no, verás el 502 intermitente descrito en el apartado 8.

18. Proxies y balanceadores: el mapa completo de timeouts en la ruta de una petición

Entre el navegador y tu código suele haber de dos a cuatro intermediarios (CDN, balanceador, ingress, sidecar), cada uno con sus propios temporizadores y valores por defecto. Un error de configuración en uno cualquiera produce síntomas que parecen de otra capa: 502 a intervalos regulares, 504 a los 60 segundos exactos, conexiones cortadas a mitad de un streaming. Antes de depurar el código, escribe la lista de

temporizadores de la ruta completa. La tabla mental que uso:

- nginx: proxy_connect_timeout, proxy_send_timeout y proxy_read_timeout valen 60 s por defecto. proxy_read_timeout es entre dos lecturas, no total. Cuidado con proxy_next_upstream: por defecto reintenta en otro servidor ante un timeout, lo que puede multiplicar peticiones no idempotentes; acótalo con proxy_next_upstream_tries y proxy_next_upstream_timeout.
- Envoy: el timeout de ruta (route.timeout) es de 15 s por defecto y 0s lo desactiva; incluye todos los reintentos, mientras que retry_policy.per_try_timeout acota cada intento. Además hay un stream_idle_timeout (5 minutos por defecto) que corta conexiones sin tráfico, relevante para SSE y websockets.
- Istio: se expresa en el VirtualService (timeout y retries.perTryTimeout). Un detalle importante: por defecto Istio reintenta dos veces ciertos fallos, así que revisa los reintentos implícitos antes de añadir los tuyos.
- AWS ALB: idle timeout de 60 s por defecto (configurable). Si tu aplicación tarda más sin enviar bytes, el balanceador devuelve 504.
- ingress-nginx (Kubernetes): se ajusta con las anotaciones proxy-connect-timeout, proxy-read-timeout y proxy-send-timeout, y heredan el comportamiento de nginx.

```
# Istio: timeout total de ruta y reintentos acotados dentro de él
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: orders
spec:
  hosts: ["orders"]
  http:
    - route:
        - destination: { host: orders }
      timeout: 3s          # deadline total, incluidos los reintentos
      retries:
        attempts: 2
        perTryTimeout: 1.2s    # 2 x 1.2 s = 2.4 s < 3 s: caben dentro del total
        retryOn: connect-failure,refused-stream,unavailable
```

La regla invariante de toda la cadena, en una línea: cada capa interior debe agotarse antes que la exterior, y los reintentos de una capa deben caber dentro de su propio timeout total. Si perTryTimeout × attempts supera el timeout de la ruta, el último intento nunca termina y has configurado trabajo que se descarta siempre. Revisa esa multiplicación en cada ruta; es un error silencioso y muy habitual.

19. Colas y trabajos en segundo plano: el timeout que se esconde en el broker

Los workers de colas tienen una versión propia del mismo problema, y es especialmente cruel porque falla de forma duplicando trabajo en lugar de perderlo. En SQS, si el procesamiento de un mensaje tarda más que el visibility timeout (30 s por defecto), el mensaje vuelve a ser visible y otro worker lo recoge mientras el primero sigue trabajando: ahora dos procesos hacen lo mismo. En Kafka, si el consumidor tarda más de max.poll.interval.ms (5 minutos por defecto) entre dos llamadas a poll(), el broker lo da por muerto y dispara un rebalanceo. En Celery con Redis como broker, el visibility_timeout (1 hora por defecto) hace que una tarea larga con acks_late se reentregue si no termina a tiempo.

La regla es la misma, vista desde la cola: el timeout del broker debe ser mayor que el tiempo máximo de procesamiento, y el procesamiento debe tener un límite propio menor que el del broker. Cuando el trabajo puede ser legítimamente largo, la solución es un latido (heartbeat) que va extendiendo la visibilidad mientras haya progreso real, con un tope absoluto:

```

import threading
import time
import boto3

sqs = boto3.client("sqs")
QUEUE_URL = "https://sqs.eu-west-1.amazonaws.com/123456789012/jobs"

VISIBILITY_S = 60          # visibilidad inicial y extensión por latido
MAX_JOB_S = 600            # tope absoluto: pasado esto, el trabajo se aborta

def heartbeat(receipt: str, stop: threading.Event, started: float) -> None:
    """Extiende la visibilidad mientras el trabajo siga vivo, sin superar el tope."""
    while not stop.wait(VISIBILITY_S / 3):
        if time.monotonic() - started > MAX_JOB_S:
            return          # dejamos que el mensaje reaparezca y vaya a la DLQ
        sqs.change_message_visibility(
            QueueUrl=QUEUE_URL, ReceiptHandle=receipt, VisibilityTimeout=VISIBILITY_S
        )

def process(msg: dict) -> None:
    stop, started = threading.Event(), time.monotonic()
    t = threading.Thread(target=heartbeat, args=(msg["ReceiptHandle"], stop, started), daemon=True)
    t.start()
    try:
        run_job(msg["Body"], timeout_s=MAX_JOB_S)    # el trabajo tiene SU PROPIO límite
        sqs.delete_message(QueueUrl=QUEUE_URL, ReceiptHandle=msg["ReceiptHandle"])
    finally:
        stop.set()

```

Y como los reintentos de cola son at-least-once, el trabajo debe ser idempotente: cualquier mensaje puede procesarse dos veces aunque tu configuración sea perfecta. Un timeout bien dimensionado reduce los duplicados, no los elimina. Por último, acuérdate de Kubernetes: si un pod recibe SIGTERM mientras procesa un trabajo largo, tiene terminationGracePeriodSeconds (30 s por defecto) antes del SIGKILL. Un trabajo de 5 minutos necesita un periodo de gracia mayor o, mejor, que el worker devuelva el mensaje a la cola al recibir la señal.

20. Bases de datos y cachés: que el presupuesto llegue hasta el último eslabón

En el apartado 8 fijamos límites por rol. Aquí vamos un paso más allá: que la base de datos conozca el presupuesto de esta petición concreta. Tres límites de PostgreSQL trabajan juntos: `statement_timeout` (por consulta), `lock_timeout` (espera de bloqueos) y, desde PostgreSQL 17, `transaction_timeout`, que limita la duración total de una transacción y cubre el hueco que dejaba `idle_in_transaction_session_timeout`: una transacción que no está inactiva pero encadena consultas durante minutos. En la práctica, `statement_timeout` no basta si tu transacción ejecuta cincuenta consultas de 100 ms cada una.

El patrón útil es fijar los límites con `SET LOCAL` al comienzo de cada transacción, derivados del presupuesto restante. Con `psycopg 3` y el módulo `deadline` del apartado 4:

```

import psycopg
from psycopg_pool import AsyncConnectionPool
import deadline

# El pool tiene SU PROPIO timeout de espera: pedir una conexión no puede bloquear sin límite.
pool = AsyncConnectionPool(
    "postgresql://app_web@db/orders",
    min_size=5, max_size=20,
    timeout=0.2,          # espera máxima por una conexión libre
    max_lifetime=1800,
    kwargs={"connect_timeout": 2},
)

async def run_in_budget(query: str, params: tuple, cap_s: float = 1.5):
    t = deadline.budget(reserve=0.05, cap=cap_s)    # segundos disponibles AHORA
    ms = max(int(t * 1000), 1)
    async with pool.connection() as conn:

```

```

async with conn.transaction():
    # set_config(..., true) equivale a SET LOCAL: se revierte al terminar la transacción
    await conn.execute("SELECT set_config('statement_timeout', %s, true)", (str(ms),))
    await conn.execute("SELECT set_config('lock_timeout', %s, true)", (str(min(ms, 300)),))
    cur = await conn.execute(query, params)
    return await cur.fetchall()

```

Dos observaciones. El timeout del pool (0,2 s) debe ser corto: si no consigues una conexión en 200 ms, el pool está saturado y esperar más solo empeora la cola; responde 503 enseguida. Y si pasas por PgBouncer, ten presente que tiene sus propios temporizadores (query_timeout, query_wait_timeout, server_idle_timeout), y que los ajustes SET de sesión no sobreviven al pooling en modo transaction: por eso conviene usar la forma SET LOCAL / set_config(..., true) dentro de la transacción, que sí es segura.

Con Redis y otras cachés el principio se repite: configura socket_connect_timeout y socket_timeout en el cliente (en redis-py ambos son None por defecto, es decir, sin límite) con valores de decenas de milisegundos. Una caché que tarda 2 segundos en responder no está ayudando, está lastrando; es preferible tratar el timeout como un fallo de caché y consultar la fuente, o directamente degradar, antes que dejar a la petición esperando a lo que debía ser un atajo.

21. Rechazar pronto: control de admisión con el deadline

Un timeout limita cuánto esperas después de aceptar el trabajo. El control de admisión decide, antes de empezar, si merece la pena aceptarlo. Es la otra mitad de la defensa, y la que evita que un pico convierta tu cola interna en el cuello de botella. Dos reglas sencillas: (1) no empieces trabajo que ya no puede terminar: si al llegar la petición quedan menos milisegundos de los que cuesta, rechaza al instante; (2) no encoles sin límite: un semáforo con una espera máxima muy corta convierte «saturación» en un 503 rápido en lugar de en una latencia creciente.

```

import asyncio
from fastapi import Request
from fastapi.responses import JSONResponse
import deadline

MIN_USEFUL_S = 0.15                # menos que esto no da para un viaje a la base de datos
inflight = asyncio.Semaphore(200)  # concurrencia máxima del servicio
QUEUE_WAIT_S = 0.05               # cuánto tolerar esperando un hueco

@app.middleware("http")
async def admission_control(request: Request, call_next):
    total = deadline.start(request.headers.get(deadline.HEADER), default_s=3.0, max_s=10.0)
    if total < MIN_USEFUL_S:
        return JSONResponse({"error": "insufficient_budget"}, status_code=504)
    try:
        async with asyncio.timeout(QUEUE_WAIT_S):
            await inflight.acquire()
    except TimeoutError:
        # saturados: fallar rápido y decirle al cliente cuándo volver
        return JSONResponse({"error": "overloaded"}, status_code=503, headers={"Retry-After": "1"})
    try:
        async with asyncio.timeout(total):
            return await call_next(request)
    except (TimeoutError, deadline.DeadlineExceeded):
        return JSONResponse({"error": "deadline_exceeded"}, status_code=504)
    finally:
        inflight.release()

```

Fíjate en que 503 y 504 significan cosas distintas y tus métricas deben separarlos: 503 es «no te he aceptado, vuelve a intentarlo enseguida» (y el cliente puede reintentar a otra réplica); 504 es «te acepté pero no llegué a tiempo». Si los mezclas, un incidente de capacidad y un incidente de latencia de una dependencia parecerán el mismo.

22. Cómo elegir los números: un método con datos

La pregunta que más me hacen es «¿qué timeout pongo?». La respuesta honesta es «el que salga de medir», y el método es corto. Primero, mide la latencia de cada llamada saliente con histogramas por dependencia y por operación (no una media: la media esconde exactamente lo que buscas). Segundo, toma el p99,9 de los últimos treinta días y súmale un margen del 25–50 %: ese es tu timeout por intento. Tercero, comprueba la suma: los intentos y las dependencias en serie deben caber en el presupuesto de la petición; si no caben, algo se recorta (menos reintentos, llamadas en paralelo, una caché) y no el timeout de más arriba. Cuarto, revisa trimestralmente: la latencia cambia con el tráfico y con los despliegues.

```
# p99.9 de la latencia de las llamadas salientes, por dependencia, últimos 30 días
histogram_quantile(
    0.999,
    sum by (le, dependency) (
        rate(http_client_request_duration_seconds_bucket[30d])
    )
)

# Ratio de timeouts por dependencia y fase (la alerta que importa)
sum by (dependency, phase) (rate(client_timeouts_total[5m]))
/
sum by (dependency) (rate(http_client_requests_total[5m]))
```

Un ejemplo de reparto para una petición de usuario con un objetivo de 3 segundos, útil como plantilla de partida:

- Presupuesto total de la petición: 3.000 ms.
- Reserva para serialización, middleware y respuesta: 100 ms.
- Autenticación (caché local con fallback al servicio): 150 ms máximo.
- Llamada crítica A (pedidos), p99,9 = 600 ms: timeout por intento de 800 ms, un único reintento solo si queda tiempo.
- Llamada crítica B (inventario), p99,9 = 400 ms: timeout de 550 ms, en paralelo con A (no suma).
- Opcionales (recomendaciones, reseñas): 300 ms cada una, en paralelo, con fallback.
- Base de datos: statement_timeout derivado del presupuesto restante, nunca por encima de 1.500 ms.

Si sumas el camino crítico en serie (autenticación + A con un reintento + base de datos) obtienes $150 + 800 + 800 + 1.500 = 3.250$ ms: no cabe. Ese es exactamente el tipo de hallazgo que quieres tener en un documento de diseño y no a las 3 de la madrugada. La salida es decidir qué recortar: quitar el reintento de A, bajar el tope de la base de datos o paralelizar la autenticación con A.

23. Probar los timeouts: inyectar latencia y comprobar que fallas rápido

Un timeout que nunca se ha disparado en un entorno de pruebas es una suposición. Toxiproxy permite colocar un proxy TCP entre tu servicio y la dependencia e inyectar latencia o cortes desde un test. El objetivo no es comprobar que el sistema «aguanta», sino que falla dentro del presupuesto: que responde 504 en menos de, digamos, 3,3 segundos aunque la dependencia tarde 30.

```
import time
import httpx
import pytest

TOXI = "http://localhost:8474"          # API de Toxiproxy
APP = "http://localhost:8000"

@pytest.fixture
def slow_inventory():
    """Añade 5 s de latencia a TODAS las respuestas del servicio de inventario."""
    toxic = {"name": "slow", "type": "latency", "stream": "downstream",
```

```

        "attributes": {"latency": 5000, "jitter": 0}}
    httpx.post(f"{TOXI}/proxies/inventory/toxics", json=toxic).raise_for_status()
    yield
    httpx.delete(f"{TOXI}/proxies/inventory/toxics/slow")

def test_request_fails_within_budget(slow_inventory):
    t0 = time.monotonic()
    r = httpx.get(f"{APP}/orders/42", headers={"x-request-timeout-ms": "2000"}, timeout=10)
    elapsed = time.monotonic() - t0
    assert r.status_code == 504                # falla de forma controlada...
    assert elapsed < 2.3                      # ...y DENTRO del presupuesto (margen de 300 ms)

def test_optional_dependency_degrades_not_fails(slow_reco):
    r = httpx.get(f"{APP}/products/7", timeout=5)
    assert r.status_code == 200
    assert r.json()["recommendations"] == []    # degradó en lugar de fallar

```

Añade a la batería un test de sobrecarga (dispara más concurrencia de la que aguanta el servicio y comprueba que responde 503 rápido en lugar de degradar a todos) y un test de cancelación (el cliente se desconecta y verificas, por métricas o por conexiones activas en la base de datos, que el trabajo en el servidor se detiene). Estos tres tipos de test cubren las tres promesas de este artículo: fallar dentro del presupuesto, rechazar pronto y no trabajar para nadie.

24. Un caso práctico (ilustrativo): el checkout que se cayó por un servicio de segundo orden

Este caso es una composición ilustrativa, con cifras redondas, de patrones que se repiten en muchos incidentes reales. Una tienda tiene un servicio de checkout que, además de cobrar, llama a un servicio de «puntos de fidelidad» para mostrar el saldo. Los puntos son un adorno. El cliente HTTP del checkout no tiene timeout (el valor por defecto del cliente es infinito) y comparte un pool de 50 conexiones con la llamada al proveedor de pagos.

- T+0 min: el servicio de puntos empieza a responder en 8 segundos por una consulta lenta tras un despliegue. No devuelve errores.
- T+1 min: con 40 peticiones por segundo, la ley de Little da $40 \times 8 = 320$ llamadas en vuelo hacia puntos. Las 50 conexiones del pool compartido se agotan en poco más de un segundo.
- T+2 min: la llamada al proveedor de pagos, que está perfectamente sana, no consigue conexión. El checkout deja de cobrar. Los reintentos del cliente multiplican las peticiones por 3.
- T+9 min: se identifica el origen mirando la latencia por dependencia (el dashboard de histogramas por dependencia no existía y hubo que mirar trazas a mano).
- Corrección: timeout de 300 ms para puntos con fallback «saldo no disponible»; pool aislado para pagos; un presupuesto total de 4 s propagado desde el gateway; reintentos solo en la capa del cliente de pagos y solo con clave de idempotencia; alerta sobre el ratio de timeouts por dependencia.
- Resultado de un incidente equivalente seis semanas después: puntos vuelve a degradarse, el checkout muestra «saldo no disponible» durante 11 minutos y los cobros siguen funcionando con la misma latencia. Nadie recibe una alerta de madrugada.

Observa qué no arregló el incidente: no hubo que arreglar el servicio de puntos para que el checkout sobreviviera. Esa es la esencia de todo este artículo. La resiliencia no consiste en que las dependencias no fallen, sino en que tu servicio tenga decidido, de antemano y con números, qué hace cuando lo hacen.

25. Presupuesto de reintentos: cómo evitar la amplificación en cascada

En el apartado 5 vimos que tres capas con tres reintentos cada una producen 27 llamadas. La defensa más eficaz no es contar reintentos por petición, sino limitar el porcentaje global de tráfico que puede ser

reintento. La idea, popularizada por los clientes de Finagle y gRPC, es un cubo de fichas (token bucket) por dependencia: cada petición exitosa deposita una fracción de ficha, y cada reintento gasta una entera. En régimen sano el cubo está lleno y los reintentos fluyen; cuando la dependencia degrada, el cubo se vacía en segundos y los reintentos se detienen solos, justo cuando más daño harían.

```
import threading

class RetryBudget:
    """Cubo de fichas: cada éxito aporta 'ratio' fichas; cada reintento cuesta 1.
    Con ratio=0.1, los reintentos no pueden superar ~10 % del tráfico exitoso."""
    def __init__(self, ratio: float = 0.1, max_tokens: float = 20.0, initial: float = 10.0):
        self.ratio = ratio
        self.max_tokens = max_tokens
        self.tokens = initial
        self._lock = threading.Lock()

    def on_success(self) -> None:
        with self._lock:
            self.tokens = min(self.max_tokens, self.tokens + self.ratio)

    def try_acquire_retry(self) -> bool:
        with self._lock:
            if self.tokens >= 1.0:
                self.tokens -= 1.0
                return True
            return False

retry_budget = RetryBudget(ratio=0.1)

async def call_with_budget(client, url):
    resp = await client.get(url) # el primer intento NUNCA consume ficha
    if resp.status_code < 500:
        retry_budget.on_success()
        return resp
    if retry_budget.try_acquire_retry(): # el reintento sí
        resp = await client.get(url)
        if resp.status_code < 500:
            retry_budget.on_success()
        return resp
```

Combina los dos límites: el deadline impide reintentar cuando no queda tiempo, y el presupuesto de reintentos impide reintentar cuando el sistema entero está mal. Cada uno cubre un fallo del otro. Y expón como métrica el nivel del cubo: un cubo que permanece vacío durante minutos es una señal limpia de que la dependencia lleva ese tiempo degradada.

26. Observabilidad del presupuesto: trazas con el tiempo restante

En el apartado 10 propusimos métricas agregadas. Para depurar casos concretos, lo más valioso es que cada span de una traza lleve el tiempo restante de presupuesto al empezar y al terminar. Con eso una traza de un 504 se lee en segundos: ves el eslabón donde el presupuesto se desplomó y si fue por una llamada lenta, por un reintento o por una espera en el pool. Con OpenTelemetry basta un par de atributos:

```
from opentelemetry import trace
import deadline

tracer = trace.get_tracer("orders")

async def call_downstream_traced(client, name: str, url: str, cap_s: float = 2.0):
    with tracer.start_as_current_span(f"call {name}") as span:
        span.set_attribute("budget.remaining_ms.start", int(deadline.remaining() * 1000))
        t = deadline.budget(reserve=0.05, cap=cap_s)
        span.set_attribute("budget.allocated_ms", int(t * 1000))
        try:
            return await client.get(url, timeout=t)
        except Exception as exc:
            # distingue por tipo: TimeoutException, ConnectError, PoolTimeout...
            span.set_attribute("error.kind", type(exc).__name__)
            raise
```

```
finally:  
    span.set_attribute("budget.remaining_ms.end", int(deadline.remaining() * 1000))
```

Con estos atributos puedes responder preguntas que antes requerían reconstruirlo todo a mano: ¿qué porcentaje de peticiones llega a la base de datos con menos de 200 ms de presupuesto?, ¿qué dependencia consume más presupuesto en el percentil 99?, ¿cuántas llamadas se lanzan con un presupuesto tan bajo que estaban condenadas? Esa última cifra es una medida directa del desperdicio, y reducirla es una de las mejoras de capacidad más baratas que existen.

27. Preguntas frecuentes

¿Un timeout global no es suficiente? No: sin límites por fase no distingues entre un connect lento (problema de red) y un read lento (problema de la dependencia), y sin límites por llamada una sola dependencia lenta puede consumir todo el presupuesto de la petición dejando sin tiempo a las demás.

¿Debo propagar el deadline hacia servicios de terceros? No como cabecera propia: no la entenderán y filtrarías información interna. Usa el sub-presupuesto como timeout de tu cliente hacia ellos y que el resto lo gestione tu código.

¿Y si mis operaciones son legítimamente largas? Conviértelas en asíncronas: acepta la petición, devuelve 202 con un identificador y deja que el cliente consulte el estado o reciba un webhook. Un timeout alto en una petición síncrona es una deuda, no una solución.

¿Qué hago con las escrituras que dan timeout? Asume que el resultado es desconocido. Consulta el estado por la clave de idempotencia antes de decidir, o devuelve al usuario un estado «pendiente de confirmar» honesto en lugar de un error engañoso.

¿Cuántos reintentos son razonables? Uno, en una sola capa, con jitter y dentro del presupuesto. Si necesitas más, el problema suele ser la fiabilidad de la dependencia, no la política de reintentos.

Glosario

- **Deadline:** instante absoluto a partir del cual un trabajo deja de tener valor. Se propaga como tiempo restante.
- **Presupuesto de tiempo:** lo que queda hasta el deadline; cada llamada gasta una parte y pasa el resto.
- **Ley de Little:** $L = \lambda \times W$; relaciona concurrencia, tasa de llegada y tiempo de permanencia.
- **Hedging:** lanzar una segunda petición a otra réplica tras un umbral para recortar la cola de latencia.
- **Control de admisión:** decidir al llegar si el trabajo se acepta, para no encolar lo que no puede terminar a tiempo.
- **Bulkhead:** aislar recursos (pools, semáforos) por dependencia para que el fallo de una no agote los de otra.
- **Idempotencia:** propiedad de una operación que puede repetirse sin cambiar el resultado más allá de la primera vez.

Checklist de producción

- Toda llamada saliente tiene timeout explícito, por fase y total.
- El tiempo restante se propaga entre servicios (cabecera o gRPC) y se recorta a un máximo.
- Los timeouts decrecen de fuera hacia dentro y están basados en percentiles medidos.
- Los reintentos respetan el presupuesto, ocurren en una sola capa y solo sobre operaciones idempotentes.
- El trabajo se cancela cuando el presupuesto se agota o el cliente se desconecta.

- `statement_timeout`, `lock_timeout` e `idle_in_transaction_session_timeout` configurados por rol.
- El keep-alive de la aplicación supera el idle timeout del balanceador.
- Métricas y alertas por dependencia y por tipo de timeout.
- Pruebas de fallo (latencia inyectada con Toxiproxy o similar) que verifican que el sistema responde 504 en lugar de colgarse.

Conclusión

Un timeout es una promesa que haces a tus usuarios y a tus propios servicios: «no te haré esperar más de esto». Un deadline propagado convierte esa promesa en algo que todo el sistema respeta, en lugar de que cada componente improvise la suya. Empieza por lo barato (auditar las llamadas sin timeout, fijar los de la base de datos y el balanceador) y avanza hacia la propagación de presupuestos. Cuando la próxima dependencia se ponga lenta, tu sistema fallará rápido, de forma acotada y visible, en lugar de agotarse en silencio.

Timeouts and Deadlines in Production: Time Budgets, Propagation Across Services and Retries That Don't Multiply Load

Almost every cascading incident I have seen starts the same way: a dependency gets slow (it doesn't go down, it gets slow), requests pile up waiting, threads or connections run out, and within minutes a service that was perfectly healthy stops responding because it was waiting on another one. The root cause is nearly always the same: nobody decided how long they were willing to wait. This guide is about making that decision deliberately, splitting it across layers, and propagating it end to end.

1. Why a missing timeout is a design flaw

A request with no time limit holds on to resources (a connection, a thread, a coroutine, memory, a database connection) for an unbounded time. Little's law sums it up: the number of requests in flight equals the arrival rate multiplied by the time each one takes to resolve. If your service receives 200 requests per second and a dependency goes from answering in 50 ms to not answering at all, within five seconds you have 1,000 hung requests. With a pool of 100 connections, you are no longer serving anyone, not even the routes that never touch that dependency.

The treacherous part is that many HTTP clients have no limit by default. Python's requests library waits forever unless you pass `timeout=`. `httpx` defaults to 5 seconds, which is something, but that value applies to each phase, not to the total. In Go, an unconfigured `http.Client{}` has no timeout. Start by auditing that: a grep for outbound calls without an explicit timeout is often the highest-value task of the quarter.

2. Anatomy of a timeout: it's not one, it's four

When you say "a 5-second timeout" you are hiding several distinct phases, each with its own failure mode:

- Pool: how long you wait to get a free connection from the pool. If it is long, you are queueing work inside your own process.
- Connect: establishing the TCP (and TLS) connection. An unreachable host can take tens of seconds to fail if you don't bound it; on a healthy network it should take milliseconds.
- Write: sending the request.
- Read: waiting for data from the server. Careful: this is the maximum time between two socket reads, not the total response time.

That last detail is the classic trap. A server that sends one byte every 4 seconds never triggers a 5-second read timeout, yet can keep you connected for hours. That is why you also need a total time limit. In `httpx` it looks like this:

```
import asyncio
import httpx

# Per-phase timeouts: short connect, moderate read, very short pool wait
TIMEOUTS = httpx.Timeout(connect=0.5, read=2.0, write=2.0, pool=0.2)

async def fetch_profile(client: httpx.AsyncClient, user_id: str) -> dict:
    # asyncio.timeout (Python 3.11+) enforces the TOTAL limit of the operation,
    # including the pool wait, the connect and every read.
    async with asyncio.timeout(3.0):
        resp = await client.get(
            f"https://profiles.internal/users/{user_id}",
            timeout=TIMEOUTS,
        )
```

```

        resp.raise_for_status()
        return resp.json()

client = httpx.AsyncClient(
    limits=httpx.Limits(max_connections=100, max_keepalive_connections=20),
)

```

Each layer has a role: per-phase timeouts detect specific failures quickly (a connect that takes 500 ms on an internal network is already a signal), and the outer `asyncio.timeout` guarantees that, whatever happens, the whole operation ends. When the total limit fires, `TimeoutError` is raised and the inner coroutine is cancelled, which releases the connection.

3. Timeout versus deadline

A timeout is a relative duration ("wait at most 2 seconds"). A deadline is an absolute instant ("this request stops making sense at 12:00:03.500"). The difference matters as soon as there is more than one hop. Picture the chain gateway -> orders -> inventory -> database, each with a 3-second timeout. If inventory takes 2.8 seconds to answer orders, orders answers the gateway at 3.1 s, and the gateway gave up at 3.0 s: the client got an error, yet orders and inventory worked in vain the whole time. Worse, if the client retries, load multiplies exactly when the system is at its weakest.

The fix is to propagate the remaining time. The gateway sets a total budget (say, 3 s), and each service hands the next one whatever is left, minus what it has already spent. When the budget runs out, everyone stops working. gRPC does this natively through the `grpc-timeout` header; over HTTP you have to do it yourself, with a custom header. It is better to propagate relative remaining time (in milliseconds) rather than an absolute instant, because that way you don't depend on machine clocks being in sync.

4. A propagatable time budget in Python

This module stores the deadline in a `ContextVar` (which propagates correctly across coroutines) using the monotonic clock, and offers a function to compute how much you can spend on the next call:

```

# deadline.py
import contextvars
import time

HEADER = "x-request-timeout-ms"

_deadline: contextvars.ContextVar[float | None] = contextvars.ContextVar(
    "deadline", default=None
)

class DeadlineExceeded(Exception):
    """The request's time budget has run out."""

def start(header_value: str | None, default_s: float, max_s: float) -> float:
    """Set the deadline from the incoming header (if valid)."""
    budget = default_s
    if header_value:
        try:
            budget = float(header_value) / 1000.0
        except ValueError:
            pass
    budget = max(0.0, min(budget, max_s)) # never blindly trust the caller
    _deadline.set(time.monotonic() + budget)
    return budget

def remaining() -> float:
    d = _deadline.get()
    if d is None:
        raise RuntimeError("deadline not initialised")
    return d - time.monotonic()

def budget(reserve: float = 0.05, cap: float | None = None) -> float:

```

```

"""Seconds you may spend on the next call, leaving a margin (reserve)
to process the response and send it back."""
left = remaining() - reserve
if left <= 0:
    raise DeadlineExceeded()
return min(left, cap) if cap is not None else left

```

And this is how it hooks into FastAPI. The middleware reads the header, sets the deadline and wraps the whole request in `asyncio.timeout`, so the handler is cancelled if the budget runs out:

```

import asyncio
import httpx
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
import deadline

app = FastAPI()
client = httpx.AsyncClient()

@app.middleware("http")
async def deadline_middleware(request: Request, call_next):
    total = deadline.start(
        request.headers.get(deadline.HEADER), default_s=3.0, max_s=10.0
    )
    try:
        async with asyncio.timeout(total):
            return await call_next(request)
    except (TimeoutError, deadline.DeadlineExceeded):
        return JSONResponse({"error": "deadline_exceeded"}, status_code=504)

async def call_downstream(url: str) -> httpx.Response:
    t = deadline.budget(reserve=0.05, cap=2.0) # never more than 2 s per call
    return await client.get(
        url,
        timeout=httpx.Timeout(t, connect=min(t, 0.5)),
        # propagate what is left: the next service does the same arithmetic
        headers={deadline.HEADER: str(int(t * 1000))},
    )

@app.get("/orders/{order_id}")
async def get_order(order_id: str):
    inv = await call_downstream(f"http://inventory/items?order={order_id}")
    return {"order": order_id, "inventory": inv.json()}

```

There is a design detail in `default_s` and `max_s`: if no header arrives you use a default, and if an absurdly large one arrives you clamp it. A client (or a bug) sending a one-hour budget must not be able to hold your resources for an hour.

5. Retries inside the budget

Retries and deadlines complement each other, but only if retries respect the budget. A loop of three 2-second attempts for a request that only has 3 seconds is a design error: the third attempt will never have time to complete. The rule: each attempt gets `min(per_try_time, remaining_time)`, and if less than a reasonable minimum remains for a call, you don't try.

```

import asyncio
import random
import httpx
import deadline

RETRYABLE = (httpx.TimeoutException, httpx.ConnectError)

async def get_with_retry(client, url, attempts=3, per_try=1.0, min_useful=0.1):
    last_exc = None
    for i in range(attempts):
        try:
            t = deadline.budget(reserve=0.05, cap=per_try)
        except deadline.DeadlineExceeded:

```

```

        break
    if t < min_useful:          # don't launch a call that is doomed
        break
    try:
        return await client.get(
            url, timeout=t, headers={deadline.HEADER: str(int(t * 1000))}
        )
    except RETRYABLE as exc:
        last_exc = exc
        # exponential backoff with full jitter, bounded by what is left
        sleep = random.uniform(0, 0.05 * 2**i)
        await asyncio.sleep(min(sleep, max(deadline.remaining() - 0.1, 0)))
    raise last_exc or deadline.DeadlineExceeded()

```

Two important warnings. First, only retry idempotent operations (or ones with an idempotency key): a timeout on a write does not mean it failed, it means you don't know whether it was applied. Second, in a system with several layers, if each layer retries 3 times, a single request can become $3 \times 3 \times 3 = 27$ calls at the lowest level. Retry in a single layer (usually the one closest to the dependency) and use a global retry budget (for example, at most 10% of calls may be retries).

6. Go: native contexts and deadlines

Go does this particularly well because `context.Context` carries the deadline and almost the whole standard library honours it. Below is a client with every phase bounded and a handler that derives its context from the incoming request's:

```

package main

import (
    "context"
    "errors"
    "net"
    "net/http"
    "time"
)

var client = &http.Client{
    // Total timeout as a safety net (includes reading the body)
    Timeout: 5 * time.Second,
    Transport: &http.Transport{
        DialContext: (&net.Dialer{
            Timeout: 500 * time.Millisecond, // connect
            KeepAlive: 30 * time.Second,
        }).DialContext,
        TLSHandshakeTimeout: 500 * time.Millisecond,
        ResponseHeaderTimeout: 2 * time.Second, // until headers arrive
        MaxIdleConnsPerHost: 20,
        IdleConnTimeout: 90 * time.Second,
    },
}

func ordersHandler(w http.ResponseWriter, r *http.Request) {
    // Derive from the request context: if the client leaves, everything is cancelled.
    ctx, cancel := context.WithTimeout(r.Context(), 800*time.Millisecond)
    defer cancel()

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "http://inventory/items", nil)
    if err != nil {
        http.Error(w, "bad request", http.StatusInternalServerError)
        return
    }
    resp, err := client.Do(req)
    if err != nil {
        if errors.Is(err, context.DeadlineExceeded) {
            http.Error(w, "upstream timeout", http.StatusGatewayTimeout)
            return
        }
        http.Error(w, "upstream error", http.StatusBadGateway)
        return
    }
}

```

```

    }
    defer resp.Body.Close()
    w.WriteHeader(resp.StatusCode)
}

func main() {
    srv := &http.Server{
        Addr:           ":8080",
        Handler:        http.HandlerFunc/ordersHandler),
        ReadHeaderTimeout: 2 * time.Second, // protects against slow clients (slowloris)
        ReadTimeout:      5 * time.Second,
        WriteTimeout:      10 * time.Second,
        IdleTimeout:       60 * time.Second,
    }
    _ = srv.ListenAndServe()
}

```

With gRPC it is even more convenient: the context deadline travels in the `grpc-timeout` header and the server receives it as the deadline of its own context. If that context is used for outgoing calls, propagation is automatic. In Python you pass it with `stub.GetItem(request, timeout=0.5)`, and on the server you can query `context.time_remaining()`.

7. Splitting the budget across layers

The sizing rule works from the outside in: each layer's timeout must be strictly smaller than the one containing it, with room for intermediate processing. A reasonable scheme for a user request with a 3-second target:

- Browser or mobile client: 10 s (what the user tolerates before giving up).
- Load balancer or API gateway: 4 s.
- Application service: a 3 s budget, with calls to dependencies limited to 1–1.5 s each.
- Database: `statement_timeout` of 1 s on interactive read paths.

How do you choose the numbers? Measure each dependency's real latency and set the timeout near its p99 or p99.9 plus a margin, not by gut feeling. A timeout below the p99 causes artificial failures in 1% of requests; one 20 times above the p99 is the same as having none when something breaks. In a proxy such as Envoy, the equivalent would be:

```

route_config:
  virtual_hosts:
    - name: orders
      domains: ["*"]
      routes:
        - match: { prefix: "/orders" }
          route:
            cluster: orders_service
            timeout: 3s                # total deadline for the route
            retry_policy:
              retry_on: connect-failure,refused-stream,reset
              num_retries: 1
              per_try_timeout: 1.5s    # each attempt, bounded

```

8. Database and load balancers

The database is another dependency that must respect your budget. PostgreSQL offers its own limits that you should set explicitly per role or per transaction:

```

-- Per-role defaults for the application (applied to every new session)
ALTER ROLE app_web SET statement_timeout = '2s';
ALTER ROLE app_web SET lock_timeout = '500ms';
ALTER ROLE app_web SET idle_in_transaction_session_timeout = '10s';

```

```
-- Local override for one specific transaction (e.g., a report query)
BEGIN;
SET LOCAL statement_timeout = '30s';
SELECT /* heavy report */ count(*) FROM events;
COMMIT;
```

`statement_timeout` aborts long queries, `lock_timeout` prevents waiting forever on a lock (the typical cause of request queues piling up behind a migration), and `idle_in_transaction_session_timeout` kills sessions that open a transaction and forget to close it, holding locks and blocking vacuum. From code, with `psycopg3`, set the value with `SELECT set_config('statement_timeout', %s, true)` using the budget's remaining time: that way the database never works beyond what your request can wait for.

With load balancers, the usual problem is mismatched idle timeouts. An AWS ALB closes idle connections after 60 seconds by default. If your server closes keep-alive connections before the balancer does (for instance, Node.js's default `keepAliveTimeout` of 5 s), the balancer will try to reuse a connection that has just been closed and intermittently return 502s. The rule: the application server's keep-alive must be greater than the idle timeout of the balancer in front of it.

9. Common mistakes

- Confusing timeout with cancellation. A client that stops waiting does not stop the work on the server. If the handler doesn't check the context or the deadline, it keeps burning CPU and database connections for nobody. Check `remaining()` before every expensive step and cancel when it runs out.
- The same timeout on every layer. If everyone waits 3 s, when the lowest tier takes 3 s they all fail at once and none can return a gracefully degraded response. Each inner layer must give up earlier.
- Timeouts on writes without idempotency. After a timeout you don't know whether the operation was applied. Use idempotency keys and check state before retrying.
- Swallowing cancellation. In `asyncio`, an overly broad `except Exception` or an `except asyncio.CancelledError`: `pass` stops the timeout from working. `CancelledError` deliberately inherits from `BaseException`; always re-raise it after cleanup.
- Timeouts that are too generous "just in case". A 60 s timeout on a call that normally takes 50 ms protects nothing: under failure, the system will saturate long before it fires.
- Forgetting the pool timeout. If waiting for a free connection has no limit, one dependency failing turns into an infinite queue inside your own process.
- Trusting an incoming deadline with no upper bound. Always clamp the received value to a reasonable maximum.

10. Observability

A silent timeout is almost as bad as having none. Record at least these metrics: the number of timeouts per dependency and per phase (connect, read, pool, total deadline), the remaining budget when entering each service (a histogram; if the median is already low, something upstream is consuming too much), and the share of requests ending in a 504. A useful alert watches the timeout ratio, not just its absolute value: going from 0.1% to 2% is usually the first visible sign of a degraded dependency, well before saturation shows up.

11. Sizing with numbers: Little's law applied to pools and timeouts

So far we have talked about timeouts as if they were a matter of judgement. In fact they can be calculated, and the calculation is what separates a reasonable value from one that merely "looks fine". Little's law says the average number of requests in flight is $L = \lambda \times W$, where λ is the arrival rate and W is the average time each request stays in the system. When the dependency is healthy, W is its normal latency. When it hangs, W becomes, at most, your timeout. And that "at most" is precisely why the timeout exists: it

is the ceiling of W , and therefore the ceiling of L .

Let us put numbers on it. Your service receives $\lambda = 200$ requests per second and each one calls a profiles service that answers in 50 ms. In normal operation there are $200 \times 0.05 = 10$ concurrent calls: a pool of 100 connections is more than enough. Now the profiles service hangs. If your timeout is 3 seconds, the number of calls in flight tends towards $200 \times 3 = 600$, six times the pool, and it reaches 100 connections in half a second. If the timeout is 500 ms, the maximum is $200 \times 0.5 = 100$: exactly the pool size, so at least the routes that do not use that dependency keep breathing, provided the profiles pool is isolated from the rest.

A very useful sizing rule follows: a call's timeout must satisfy $\lambda \times \text{timeout} \leq \text{size of its dedicated pool}$, or you must explicitly accept that, during a failure, part of the load will be rejected at the door (load shedding). There is no third option: either you limit time, or you limit concurrency, or you saturate. This small script computes it for your own figures and tells you how long the pool takes to saturate:

```
# little.py - how much a pool can take when a dependency hangs
from dataclasses import dataclass

@dataclass
class Dependency:
    name: str
    rate_rps: float          # lambda: requests/s that use it
    p50_s: float             # healthy latency
    p999_s: float            # healthy latency in the tail
    timeout_s: float         # the timeout you have configured
    pool_size: int           # dedicated connections or permits

def analyze(d: Dependency) -> dict:
    healthy_l = d.rate_rps * d.p50_s                # L in normal operation
    tail_l = d.rate_rps * d.p999_s                  # L with the latency tail
    worst_l = d.rate_rps * d.timeout_s               # L if the dependency hangs
    secs_to_exhaust = d.pool_size / d.rate_rps       # time until the pool is exhausted
    return {
        "dependency": d.name,
        "L_healthy": round(healthy_l, 1),
        "L_tail": round(tail_l, 1),
        "L_worst_case": round(worst_l, 1),
        "pool": d.pool_size,
        "saturates": worst_l > d.pool_size,
        "seconds_to_saturate": round(secs_to_exhaust, 2),
        # minimum reasonable pool: cover the tail with a 50 % margin
        "recommended_pool": int(tail_l * 1.5) + 1,
        # maximum timeout that does not overflow the pool
        "max_timeout_without_saturating": round(d.pool_size / d.rate_rps, 3),
    }

if __name__ == "__main__":
    profiles = Dependency("profiles", 200, 0.05, 0.40, timeout_s=3.0, pool_size=100)
    for k, v in analyze(profiles).items():
        print(f"{k:32} {v}")
```

With those figures the script will tell you the worst case is 600 concurrent calls, that the pool is exhausted in 0.5 seconds, and that the maximum timeout that does not overflow a pool of 100 is 0.5 s. Notice that the result is not "blindly set the timeout to 500 ms": this dependency's healthy p99.9 is 400 ms, so 500 ms leaves a very narrow margin. That tension is the valuable information. Either you enlarge the pool (carefully: every connection costs something at the other end), or you reduce the rate with a cache, or you accept rejecting requests when the pool fills instead of queueing them. What you cannot do is ignore it.

12. Structured cancellation in Python: `asyncio.timeout_at`, `TaskGroup` and `ExceptionGroup`

In section 2 we used `asyncio.timeout()` with a relative duration. For propagated budgets its sibling `asyncio.timeout_at()` is more natural: it takes an absolute instant on the event loop's clock (`loop.time()`).

That is exactly deadline semantics: you can compute it once when the request arrives and use it anywhere in the chain without subtracting by hand. In addition, the object returned by `timeout()` supports `reschedule()` to move the limit while the operation is in progress, useful when you want to grant extra time to a critical operation.

What almost nobody has clear is how the timeout materialises. When it expires, `asyncio` does not "kill" anything: it cancels the task inside the block by raising `CancelledError` at the next `await`, and on leaving the `async with block` it converts that into `TimeoutError`. This has two practical consequences. One: the code inside the block is only interrupted at `await` points, so a two-second synchronous function (a giant `json.loads`, a pathological regular expression, a blocking driver) cannot be interrupted and the timeout will arrive late. Two: if inside the block you catch `CancelledError` and do not re-raise it, the timeout silently stops working.

For several calls in parallel, `TaskGroup` (Python 3.11+) is the right tool compared with `gather`: if one task fails it cancels the others, and all the exceptions are grouped into an `ExceptionGroup` handled with `except*`. Combined with an outer timeout you get a clean contract: either everything finishes within the budget, or everything is cancelled and no orphan task is left consuming resources.

```
import asyncio
import httpx

async def fetch(client: httpx.AsyncClient, url: str) -> dict:
    resp = await client.get(url)
    resp.raise_for_status()
    return resp.json()

async def build_order_view(client: httpx.AsyncClient, order_id: str, deadline_at: float) -> dict:
    """Query three services in parallel under ONE absolute deadline.
    deadline_at is expressed on the event loop clock (loop.time())."""
    try:
        async with asyncio.timeout_at(deadline_at):
            async with asyncio.TaskGroup() as tg:
                order = tg.create_task(fetch(client, f"http://orders/{order_id}"))
                items = tg.create_task(fetch(client, f"http://inventory/items?order={order_id}"))
                price = tg.create_task(fetch(client, f"http://pricing/quote?order={order_id}"))
            return {"order": order.result(), "items": items.result(), "price": price.result()}
    except* httpx.HTTPStatusError as eg:
        # a dependency answered with an error: here you decide whether to degrade or propagate
        statuses = [e.response.status_code for e in eg.exceptions]
        raise RuntimeError(f"dependencies with errors: {statuses}") from eg
    # TimeoutError is NOT caught here: it bubbles up to the middleware, which answers 504

# usage: the deadline is set once when the request arrives
async def handler(order_id: str):
    loop = asyncio.get_running_loop()
    deadline_at = loop.time() + 2.5
    async with httpx.AsyncClient() as client:
        return await build_order_view(client, order_id, deadline_at)
```

A nuance about `asyncio.shield()`: it protects an operation that must not be cancelled halfway (for example, committing a transaction), but beware, because "protecting it" means it will keep running after your request has expired. Use it only for short work whose result matters even if nobody is waiting, and always measure how many protected tasks are left alive. A shield with no limit of its own is a timeout you have switched off politely.

13. Fan-out with degradation: critical and optional dependencies

A pure `TaskGroup` is an "all or nothing" contract, and that is too rigid for many real pages. On a product page, price and availability are critical; recommendations, reviews and the "people viewing this" counter are not. If recommendations take 2 seconds and the user would have accepted a page without them, failing the whole request is worse than serving it incomplete. The solution is to classify each dependency and give it its own sub-budget, always clamped by what remains of the global budget.

```

import asyncio
import logging
from typing import Awaitable, TypeVar

log = logging.getLogger("fanout")
T = TypeVar("T")

async def optional(name: str, coro: Awaitable[T], timeout_s: float, fallback: T) -> T:
    """Run an OPTIONAL dependency: if it exhausts its sub-budget or fails,
    return the fallback instead of bringing the request down."""
    try:
        async with asyncio.timeout(timeout_s):
            return await coro
    except TimeoutError:
        log.warning("optional_timeout", extra={"dep": name, "timeout_s": timeout_s})
        return fallback
    except Exception:
        log.exception("optional_failed", extra={"dep": name})
        return fallback

async def product_page(client, product_id: str, remaining_s: float) -> dict:
    # critical calls share the budget; optional ones get a fraction
    reco_budget = min(0.4, remaining_s * 0.25)
    reviews_budget = min(0.3, remaining_s * 0.20)
    async with asyncio.TaskGroup() as tg:
        price = tg.create_task(fetch_json(client, f"http://pricing/{product_id}"))
        stock = tg.create_task(fetch_json(client, f"http://stock/{product_id}"))
        reco = tg.create_task(optional("reco", fetch_json(client, f"http://reco/{product_id}"),
        reco_budget, fallback=[]))
        reviews = tg.create_task(optional("reviews", fetch_json(client,
        f"http://reviews/{product_id}"), reviews_budget, fallback=None))
    return {
        "price": price.result(),
        "stock": stock.result(),
        "recommendations": reco.result(),      # empty list if it did not arrive in time
        "reviews": reviews.result(),          # None: the frontend hides the block
    }

```

There is a product decision hidden here that is worth making explicit with the team: what does the user see when something is missing. An empty recommendations list that the frontend does not render is graceful degradation; an endless spinner is not. And record every fallback as a metric (a counter per dependency), because a system that degrades silently for weeks is a system that has lost that feature without anyone noticing.

14. Hedged requests with a budget

High latency percentiles almost never come from a broken dependency, but from a single slow replica: a GC pause, a noisy neighbour, a disk having a bad moment. If you have several replicas, a technique described in Google's "The Tail at Scale" paper is the hedged request: you send the request to one replica; if it has not answered after a threshold (typically that call's p95), you send a second one to another replica, keep whichever answer arrives first and cancel the other. The cost is a small percentage of extra load (if the threshold is the p95, roughly 5%) in exchange for noticeably cutting the latency tail.

Three conditions keep it from backfiring. First, only on idempotent, side-effect-free operations: reads, not writes. Second, with a global hedging budget: if the dependency is genuinely degraded, every request will exceed the threshold and you would double the load exactly when it can least take it. Third, respecting the deadline: the hedge is only sent if useful time remains.

```

import asyncio
import random

class HedgeBudget:
    """Allows at most a percentage of hedged requests (simple window)."""
    def __init__(self, max_ratio: float = 0.05):
        self.max_ratio = max_ratio
        self.total = 0

```

```

        self.hedged = 0

    def allow(self) -> bool:
        self.total += 1
        if self.total > 1000:                # crude decay so history does not accumulate
            self.total //= 2
            self.hedged //= 2
        if self.hedged / max(self.total, 1) < self.max_ratio:
            self.hedged += 1
            return True
        return False

budget = HedgeBudget(0.05)

async def hedged_get(client, urls: list[str], hedge_after_s: float, remaining_s: float):
    """Ask the first replica; if it takes longer than hedge_after_s and there is budget and time,
    send a second one. Return the first answer and cancel the other."""
    first = asyncio.create_task(client.get(urls[0]))
    tasks = {first}
    try:
        done, _ = await asyncio.wait({first}, timeout=hedge_after_s)
        if not done and remaining_s > hedge_after_s * 2 and budget.allow():
            tasks.add(asyncio.create_task(client.get(random.choice(urls[1:]))))
        done, _ = await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED, timeout=remaining_s)
        if not done:
            raise TimeoutError("no replica answered in time")
        return next(iter(done)).result()
    finally:
        for t in tasks:
            if not t.done():
                t.cancel()                # do not leave the loser consuming resources

```

Measure the effect before rolling it out widely: compare the p99 with and without hedging on real traffic and watch the "hedged requests sent" and "hedged requests won" metrics. If the hedge almost never wins, the threshold is too low and you are only burning load.

15. HTTP clients in other languages: what each one does by default

Section 1 mentioned requests, httpx and Go, but the problem is not a language's, it belongs to each specific client. You should know the defaults of the ones you use, because they are the configuration you are running today even if nobody wrote it down. As a guide (always check the version you have installed): requests and urllib3 impose no timeout at all; httpx imposes 5 seconds per phase; aiohttp uses a 5-minute total timeout, which for an interactive call is practically the same as having none; the standard Go client has none; and fetch in Node (undici) waits up to 300 seconds for headers and for the body. In every case the default is far too generous for a service answering user requests.

In Node and in the browser, the key piece is AbortSignal. AbortSignal.timeout(ms) creates a signal that cancels itself, and AbortSignal.any([...]) combines several, so you can cancel the call both when the budget expires and when the original client closes the connection. This is the TypeScript equivalent of the deadline middleware:

```

// deadline.ts - propagatable time budget on Node 20+
import { AsyncLocalStorage } from "node:async_hooks";

const HEADER = "x-request-timeout-ms";
const store = new AsyncLocalStorage<{ deadlineAt: number; signal: AbortSignal }>();

export function withDeadline<T>(<
    incoming: string | undefined,
    clientGone: AbortSignal,
    fn: () => Promise<T>,
    { defaultMs = 3000, maxMs = 10000 } = {}
): Promise<T> {
    let budget = Number(incoming);
    if (!Number.isFinite(budget) || budget <= 0) budget = defaultMs;
    budget = Math.min(budget, maxMs); // never blindly trust the caller
    const signal = AbortSignal.any([AbortSignal.timeout(budget), clientGone]);

```

```

    return store.run({ deadlineAt: performance.now() + budget, signal }, fn);
}

export function remainingMs(): number {
  const s = store.getStore();
  if (!s) throw new Error("deadline not initialised");
  return s.deadlineAt - performance.now();
}

export async function callDownstream(url: string, capMs = 2000): Promise<Response> {
  const s = store.getStore()!;
  const t = Math.min(remainingMs() - 50, capMs); // 50 ms reserve
  if (t <= 0) throw new DOMException("deadline exhausted", "TimeoutError");
  return fetch(url, {
    // cancelled if the sub-budget expires, the global budget expires, or the client leaves
    signal: AbortSignal.any([s.signal, AbortSignal.timeout(t)]),
    headers: { [HEADER]: String(Math.floor(t)) },
  });
}

```

Two details. Use `performance.now()` (a monotonic clock) rather than `Date.now()`, for the same reason we used `time.monotonic()` in Python: an NTP adjustment in the middle of the request must not lengthen or shorten your budget. And catch the error correctly: a signal cancelled by timeout throws an exception with name `=== "TimeoutError"`, while a manual cancellation throws `"AbortError"`. Telling them apart lets you answer 504 in the first case and simply give up in the second (the client is no longer there to read the response).

In Java, `HttpClient` separates `connectTimeout` (on the client) from `timeout` (per request, which covers up to receiving the response headers, not reading the whole body). For a total limit on the body you need to wrap the asynchronous call with `orTimeout()` on the `CompletableFuture`. The lesson repeats across languages: find out exactly which phases each setting covers and add your own total limit if the client does not offer one.

16. gRPC: real deadlines, and what its codes mean

gRPC is the only mainstream protocol that has deadlines built into its contract, and even so there are two traps. The first: by default there is no deadline. If the client does not set one, the call can wait indefinitely, and that is the configuration every generated client ships with. The second: the deadline only propagates if the server uses the incoming context in its outgoing calls. If in the handler you create a fresh context (`context.Background()` in Go) you break the chain without noticing, and the next layer is back to having no limit.

```

package main

import (
    "context"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

// Server interceptor: instantly rejects what no longer has useful time
// and requires every incoming call to carry a deadline.
func deadlineGuard(minUseful time.Duration, maxBudget time.Duration) grpc.UnaryServerInterceptor {
    return func(ctx context.Context, req any, info *grpc.UnaryServerInfo,
        handler grpc.UnaryHandler) (any, error) {

        dl, ok := ctx.Deadline()
        if !ok {
            // no deadline: apply a default one instead of waiting forever
            var cancel context.CancelFunc
            ctx, cancel = context.WithTimeout(ctx, maxBudget)
            defer cancel()

```

```

    } else if time.Until(dl) < minUseful {
        // arrives with less time than the work costs: do not start it
        return nil, status.Error(codes.DeadlineExceeded, "insufficient budget")
    }
    return handler(ctx, req)
}

func inventoryCall(ctx context.Context, c InventoryClient, id string) (*Item, error) {
    // IMPORTANT: derive from the incoming ctx; that way the original client's deadline
    // stays in force and shrinks at every hop.
    ctx, cancel := context.WithTimeout(ctx, 800*time.Millisecond)
    defer cancel()
    return c.GetItem(ctx, &GetItemRequest{Id: id})
}

```

About the codes: DEADLINE_EXCEEDED means time ran out (the client's or some link's), and CANCELLED means someone explicitly cancelled, usually because the client disconnected. They are not interchangeable in your metrics: a spike of DEADLINE_EXCEEDED points to slowness in a dependency; a spike of CANCELLED points to impatient clients (or to a load balancer with a shorter timeout than yours). If you mix them in a single "errors" counter, you lose half the signal.

You can also set timeouts and retries per method in the service config, without touching code. The timeout there is a total deadline per call, and the retry policy operates inside it:

```

{
  "methodConfig": [{
    "name": [{ "service": "inventory.Inventory", "method": "GetItem" }],
    "timeout": "1.5s",
    "retryPolicy": {
      "maxAttempts": 3,
      "initialBackoff": "0.05s",
      "maxBackoff": "0.3s",
      "backoffMultiplier": 2,
      "retryableStatusCodes": ["UNAVAILABLE"]
    }
  ]}
}

```

Note that only UNAVAILABLE is retried and not DEADLINE_EXCEEDED: retrying after the total deadline has been exhausted makes no sense because no budget is left, and retrying on a per-attempt timeout would multiply the load on a slow service. It is the same rule as in section 5, expressed as configuration.

17. Application servers: what kills the request, and what does not

Server timeouts protect in the other direction: from slow or malicious clients. And they have a peculiarity that surprises people: many server timeouts do not cancel the work, they only cut the connection. In Go, `http.TimeoutHandler` returns a 503 to the client when it fires, but the handler goroutine keeps running until it finishes unless it respects `r.Context()`. A gunicorn worker's timeout (--timeout, 30 s by default) is not a request limit: it measures how long the worker has gone without signs of life to the master process and, when it fires, kills the whole worker, along with every request it was serving at that moment.

```

# gunicorn.conf.py - an ASGI server with coherent limits
bind = "0.0.0.0:8000"
workers = 4
worker_class = "uvicorn.workers.UvicornWorker"

# The worker is restarted if it does not answer the master within this time. It is NOT a per-request
# limit: set it above your maximum deadline so valid requests are not killed, but not to 0.
timeout = 30

# Time for workers to finish their requests after SIGTERM (must be lower than Kubernetes'
# terminationGracePeriodSeconds, or the kubelet will send SIGKILL first).
graceful_timeout = 20

```

```
# Server keep-alive: GREATER than the load balancer's idle timeout (ALB: 60 s)
keepalive = 65

# Recycle workers to bound slow leaks
max_requests = 5000
max_requests_jitter = 500

// Go server: per-phase limits and a handler that respects the context
mux := http.NewServeMux()
mux.HandleFunc("/report", func(w http.ResponseWriter, r *http.Request) {
    // A legitimately long handler extends ONLY its own write limit.
    rc := http.NewResponseController(w)
    _ = rc.SetWriteDeadline(time.Now().Add(60 * time.Second))
    // ... and still must respect r.Context() to cancel itself if the client leaves.
    generateReport(r.Context(), w)
})

srv := &http.Server{
    Addr:           ":8080",
    Handler:        http.TimeoutHandler(mux, 5*time.Second, "timeout"), // global safety net
    ReadHeaderTimeout: 2 * time.Second,
    ReadTimeout:    5 * time.Second,
    WriteTimeout:   10 * time.Second,
    IdleTimeout:    75 * time.Second, // > the load balancer's idle timeout
}
```

Notice the hierarchy: the global limits (`WriteTimeout`, `TimeoutHandler`) are the safety net, and endpoints that need more time ask for it explicitly and in a bounded way with `ResponseController` (Go 1.20+) instead of relaxing the limit for the whole server. Something equivalent happens in Node: `server.keepAliveTimeout` (5 s by default) must be greater than the load balancer's idle timeout, and `server.headersTimeout` must be greater than `keepAliveTimeout`; otherwise you will see the intermittent 502 described in section 8.

18. Proxies and load balancers: the full map of timeouts on a request's path

Between the browser and your code there are usually two to four intermediaries (CDN, load balancer, ingress, sidecar), each with its own timers and defaults. A misconfiguration in any one of them produces symptoms that look like they belong to another layer: 502s at regular intervals, 504s at exactly 60 seconds, connections cut halfway through a stream. Before debugging the code, write down the list of timers on the whole path. The mental table I use:

- **nginx**: `proxy_connect_timeout`, `proxy_send_timeout` and `proxy_read_timeout` default to 60 s. `proxy_read_timeout` is between two reads, not a total. Beware of `proxy_next_upstream`: by default it retries on another server after a timeout, which can multiply non-idempotent requests; bound it with `proxy_next_upstream_tries` and `proxy_next_upstream_timeout`.
- **Envoy**: the route timeout (`route.timeout`) is 15 s by default and 0s disables it; it includes all retries, whereas `retry_policy.per_try_timeout` bounds each attempt. There is also a `stream_idle_timeout` (5 minutes by default) that cuts connections without traffic, relevant for SSE and websockets.
- **Istio**: expressed in the `VirtualService` (`timeout` and `retries.perTryTimeout`). An important detail: by default Istio retries certain failures twice, so check the implicit retries before adding your own.
- **AWS ALB**: idle timeout of 60 s by default (configurable). If your application takes longer without sending bytes, the balancer returns 504.
- **ingress-nginx** (Kubernetes): tuned with the `proxy-connect-timeout`, `proxy-read-timeout` and `proxy-send-timeout` annotations, and inherits nginx behaviour.

```
# Istio: total route timeout and bounded retries inside it
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
```

```

name: orders
spec:
  hosts: ["orders"]
  http:
    - route:
        - destination: { host: orders }
      timeout: 3s          # total deadline, retries included
      retries:
        attempts: 2
        perTryTimeout: 1.2s    # 2 x 1.2 s = 2.4 s < 3 s: they fit inside the total
        retryOn: connect-failure,refused-stream,unavailable

```

The invariant rule for the whole chain, in one line: every inner layer must give up before the outer one, and a layer's retries must fit inside its own total timeout. If $\text{perTryTimeout} \times \text{attempts}$ exceeds the route timeout, the last attempt never finishes and you have configured work that is always discarded. Check that multiplication on every route; it is a silent and very common mistake.

19. Queues and background jobs: the timeout hiding in the broker

Queue workers have their own version of the same problem, and it is especially cruel because it fails by duplicating work instead of losing it. In SQS, if processing a message takes longer than the visibility timeout (30 s by default), the message becomes visible again and another worker picks it up while the first is still working: now two processes are doing the same thing. In Kafka, if the consumer takes longer than `max.poll.interval.ms` (5 minutes by default) between two `poll()` calls, the broker declares it dead and triggers a rebalance. In Celery with Redis as the broker, the `visibility_timeout` (1 hour by default) makes a long task with `acks_late` be redelivered if it does not finish in time.

The rule is the same, seen from the queue: the broker's timeout must be greater than the maximum processing time, and processing must have its own limit lower than the broker's. When the work can legitimately be long, the solution is a heartbeat that keeps extending visibility while there is real progress, with an absolute cap:

```

import threading
import time
import boto3

sqs = boto3.client("sqs")
QUEUE_URL = "https://sqs.eu-west-1.amazonaws.com/123456789012/jobs"

VISIBILITY_S = 60          # initial visibility and extension per heartbeat
MAX_JOB_S = 600            # absolute cap: beyond this, the job is aborted

def heartbeat(receipt: str, stop: threading.Event, started: float) -> None:
    """Extend visibility while the job is alive, without exceeding the cap."""
    while not stop.wait(VISIBILITY_S / 3):
        if time.monotonic() - started > MAX_JOB_S:
            return          # let the message reappear and go to the DLQ
        sqs.change_message_visibility(
            QueueUrl=QUEUE_URL, ReceiptHandle=receipt, VisibilityTimeout=VISIBILITY_S
        )

def process(msg: dict) -> None:
    stop, started = threading.Event(), time.monotonic()
    t = threading.Thread(target=heartbeat, args=(msg["ReceiptHandle"], stop, started), daemon=True)
    t.start()
    try:
        run_job(msg["Body"], timeout_s=MAX_JOB_S)    # the job has ITS OWN limit
        sqs.delete_message(QueueUrl=QUEUE_URL, ReceiptHandle=msg["ReceiptHandle"])
    finally:
        stop.set()

```

And since queue retries are at-least-once, the work must be idempotent: any message can be processed twice even if your configuration is perfect. A well-sized timeout reduces duplicates, it does not eliminate them. Finally, remember Kubernetes: if a pod receives `SIGTERM` while processing a long job, it has

terminationGracePeriodSeconds (30 s by default) before SIGKILL. A 5-minute job needs a longer grace period or, better, for the worker to return the message to the queue when it receives the signal.

20. Databases and caches: getting the budget to the last link

In section 8 we set per-role limits. Here we go one step further: making the database aware of this specific request's budget. Three PostgreSQL limits work together: `statement_timeout` (per query), `lock_timeout` (lock waits) and, since PostgreSQL 17, `transaction_timeout`, which limits a transaction's total duration and covers the gap left by `idle_in_transaction_session_timeout`: a transaction that is not idle but chains queries for minutes. In practice, `statement_timeout` is not enough if your transaction runs fifty queries of 100 ms each.

The useful pattern is to set the limits with `SET LOCAL` at the start of each transaction, derived from the remaining budget. With `psycpg 3` and the `deadline` module from section 4:

```
import psycpg
from psycpg_pool import AsyncConnectionPool
import deadline

# The pool has ITS OWN wait timeout: asking for a connection cannot block without limit.
pool = AsyncConnectionPool(
    "postgresql://app_web@db/orders",
    min_size=5, max_size=20,
    timeout=0.2,           # maximum wait for a free connection
    max_lifetime=1800,
    kwargs={"connect_timeout": 2},
)

async def run_in_budget(query: str, params: tuple, cap_s: float = 1.5):
    t = deadline.budget(reserve=0.05, cap=cap_s)           # seconds available NOW
    ms = max(int(t * 1000), 1)
    async with pool.connection() as conn:
        async with conn.transaction():
            # set_config(..., true) is equivalent to SET LOCAL: reverted when the transaction ends
            await conn.execute("SELECT set_config('statement_timeout', %s, true)", (str(ms),))
            await conn.execute("SELECT set_config('lock_timeout', %s, true)", (str(min(ms, 300)),))
            cur = await conn.execute(query, params)
            return await cur.fetchall()
```

Two observations. The pool timeout (0.2 s) should be short: if you cannot get a connection in 200 ms, the pool is saturated and waiting longer only makes the queue worse; answer 503 right away. And if you go through PgBouncer, keep in mind it has its own timers (`query_timeout`, `query_wait_timeout`, `server_idle_timeout`), and that session-level `SET` commands do not survive pooling in transaction mode: that is why the `SET LOCAL` / `set_config(..., true)` form inside the transaction, which is safe, is the one to use.

With Redis and other caches the principle repeats: configure `socket_connect_timeout` and `socket_timeout` on the client (in `redis-py` both are `None` by default, that is, unlimited) with values of tens of milliseconds. A cache that takes 2 seconds to answer is not helping, it is dragging you down; it is better to treat the timeout as a cache miss and query the source, or simply degrade, than to leave the request waiting on what was meant to be a shortcut.

21. Rejecting early: admission control with the deadline

A timeout limits how long you wait after accepting the work. Admission control decides, before starting, whether it is worth accepting. It is the other half of the defence, and the one that stops a spike from turning your internal queue into the bottleneck. Two simple rules: (1) do not start work that can no longer finish: if when the request arrives fewer milliseconds remain than the work costs, reject immediately; (2) do not queue without limit: a semaphore with a very short maximum wait turns "saturation" into a quick 503 instead of growing latency.

```

import asyncio
from fastapi import Request
from fastapi.responses import JSONResponse
import deadline

MIN_USEFUL_S = 0.15 # less than this is not enough for a database round trip
inflight = asyncio.Semaphore(200) # service's maximum concurrency
QUEUE_WAIT_S = 0.05 # how long to tolerate waiting for a slot

@app.middleware("http")
async def admission_control(request: Request, call_next):
    total = deadline.start(request.headers.get(deadline.HEADER), default_s=3.0, max_s=10.0)
    if total < MIN_USEFUL_S:
        return JSONResponse({"error": "insufficient_budget", status_code=504})
    try:
        async with asyncio.timeout(QUEUE_WAIT_S):
            await inflight.acquire()
    except TimeoutError:
        # saturated: fail fast and tell the client when to come back
        return JSONResponse({"error": "overloaded", status_code=503, headers={"Retry-After": "1"}})
    try:
        async with asyncio.timeout(total):
            return await call_next(request)
    except (TimeoutError, deadline.DeadlineExceeded):
        return JSONResponse({"error": "deadline_exceeded", status_code=504})
    finally:
        inflight.release()

```

Notice that 503 and 504 mean different things and your metrics must keep them apart: 503 is "I did not accept you, try again right away" (and the client may retry on another replica); 504 is "I accepted you but did not make it in time". If you mix them, a capacity incident and a dependency-latency incident will look the same.

22. How to choose the numbers: a data-driven method

The question I am asked most is "what timeout should I set?". The honest answer is "whatever comes out of measuring", and the method is short. First, measure the latency of every outbound call with histograms per dependency and per operation (not an average: the average hides exactly what you are looking for). Second, take the p99.9 over the last thirty days and add a 25–50% margin: that is your per-attempt timeout. Third, check the sum: serial attempts and dependencies must fit in the request budget; if they do not, something gets trimmed (fewer retries, parallel calls, a cache) and not the timeout further up. Fourth, review quarterly: latency changes with traffic and with deployments.

```

# p99.9 of outbound call latency, per dependency, last 30 days
histogram_quantile(
    0.999,
    sum by (le, dependency) (
        rate(http_client_request_duration_seconds_bucket[30d])
    )
)

# Timeout ratio per dependency and phase (the alert that matters)
sum by (dependency, phase) (rate(client_timeouts_total[5m]))
/
sum by (dependency) (rate(http_client_requests_total[5m]))

```

An example allocation for a user request with a 3-second target, useful as a starting template:

- Total request budget: 3,000 ms.
- Reserve for serialisation, middleware and the response: 100 ms.
- Authentication (local cache with fallback to the service): 150 ms maximum.
- Critical call A (orders), p99.9 = 600 ms: per-attempt timeout of 800 ms, a single retry only if time remains.

- Critical call B (inventory), p99.9 = 400 ms: 550 ms timeout, in parallel with A (does not add up).
- Optional calls (recommendations, reviews): 300 ms each, in parallel, with fallback.
- Database: statement_timeout derived from the remaining budget, never above 1,500 ms.

If you add up the serial critical path (authentication + A with one retry + database) you get 150 + 800 + 800 + 1,500 = 3,250 ms: it does not fit. That is exactly the kind of finding you want in a design document and not at 3 a.m. The way out is deciding what to trim: drop A's retry, lower the database cap, or run authentication in parallel with A.

23. Testing timeouts: inject latency and check that you fail fast

A timeout that has never fired in a test environment is an assumption. Toxiproxy lets you put a TCP proxy between your service and the dependency and inject latency or cuts from a test. The goal is not to check that the system "holds up", but that it fails within the budget: that it answers 504 in less than, say, 3.3 seconds even if the dependency takes 30.

```
import time
import httpx
import pytest

TOXI = "http://localhost:8474"          # Toxiproxy API
APP = "http://localhost:8000"

@pytest.fixture
def slow_inventory():
    """Add 5 s of latency to ALL responses from the inventory service."""
    toxic = {"name": "slow", "type": "latency", "stream": "downstream",
            "attributes": {"latency": 5000, "jitter": 0}}
    httpx.post(f"{TOXI}/proxies/inventory/toxics", json=toxic).raise_for_status()
    yield
    httpx.delete(f"{TOXI}/proxies/inventory/toxics/slow")

def test_request_fails_within_budget(slow_inventory):
    t0 = time.monotonic()
    r = httpx.get(f"{APP}/orders/42", headers={"x-request-timeout-ms": "2000"}, timeout=10)
    elapsed = time.monotonic() - t0
    assert r.status_code == 504                # fails in a controlled way...
    assert elapsed < 2.3                      # ...and WITHIN the budget (300 ms margin)

def test_optional_dependency_degrades_not_fails(slow_reco):
    r = httpx.get(f"{APP}/products/7", timeout=5)
    assert r.status_code == 200
    assert r.json()["recommendations"] == []    # it degraded instead of failing
```

Add to the suite an overload test (fire more concurrency than the service can take and check that it answers 503 quickly instead of degrading everyone) and a cancellation test (the client disconnects and you verify, through metrics or active database connections, that the work on the server stops). These three kinds of test cover the three promises of this article: fail within the budget, reject early, and do not work for anyone.

24. A worked case (illustrative): the checkout that went down because of a second-tier service

This case is an illustrative composite, with round figures, of patterns that repeat across many real incidents. A shop has a checkout service that, besides taking payment, calls a "loyalty points" service to display the balance. Points are decoration. The checkout's HTTP client has no timeout (the client's default is infinite) and shares a 50-connection pool with the call to the payment provider.

- T+0 min: the points service starts answering in 8 seconds because of a slow query after a deployment. It returns no errors.
- T+1 min: at 40 requests per second, Little's law gives $40 \times 8 = 320$ calls in flight towards points. The 50

connections of the shared pool are exhausted in just over a second.

- T+2 min: the call to the payment provider, which is perfectly healthy, cannot get a connection. Checkout stops charging. The client's retries multiply requests by 3.
- T+9 min: the source is identified by looking at latency per dependency (the per-dependency histogram dashboard did not exist and traces had to be inspected by hand).
- Fix: a 300 ms timeout for points with a "balance unavailable" fallback; an isolated pool for payments; a 4 s total budget propagated from the gateway; retries only in the payment client layer and only with an idempotency key; an alert on the timeout ratio per dependency.
- Result of an equivalent incident six weeks later: points degrades again, checkout shows "balance unavailable" for 11 minutes and charges keep working with the same latency. Nobody gets a night-time alert.

Notice what the incident did not require: nobody had to fix the points service for the checkout to survive. That is the essence of this whole article. Resilience does not mean dependencies never fail; it means your service has decided in advance, with numbers, what it does when they do.

25. Retry budgets: how to avoid cascading amplification

In section 5 we saw that three layers with three retries each produce 27 calls. The most effective defence is not counting retries per request, but limiting the global percentage of traffic that may be retries. The idea, popularised by the Finagle and gRPC clients, is a token bucket per dependency: each successful request deposits a fraction of a token, and each retry spends a whole one. In healthy operation the bucket is full and retries flow; when the dependency degrades, the bucket empties within seconds and retries stop by themselves, exactly when they would do the most damage.

```
import threading

class RetryBudget:
    """Token bucket: every success adds 'ratio' tokens; every retry costs 1.
    With ratio=0.1, retries cannot exceed ~10% of successful traffic."""
    def __init__(self, ratio: float = 0.1, max_tokens: float = 20.0, initial: float = 10.0):
        self.ratio = ratio
        self.max_tokens = max_tokens
        self.tokens = initial
        self._lock = threading.Lock()

    def on_success(self) -> None:
        with self._lock:
            self.tokens = min(self.max_tokens, self.tokens + self.ratio)

    def try_acquire_retry(self) -> bool:
        with self._lock:
            if self.tokens >= 1.0:
                self.tokens -= 1.0
                return True
            return False

retry_budget = RetryBudget(ratio=0.1)

async def call_with_budget(client, url):
    resp = await client.get(url)          # the first attempt NEVER consumes a token
    if resp.status_code < 500:
        retry_budget.on_success()
        return resp
    if retry_budget.try_acquire_retry():   # the retry does
        resp = await client.get(url)
        if resp.status_code < 500:
            retry_budget.on_success()
        return resp
```

Combine the two limits: the deadline prevents retrying when no time is left, and the retry budget prevents retrying when the whole system is unwell. Each covers a failure of the other. And expose the bucket level

as a metric: a bucket that stays empty for minutes is a clean signal that the dependency has been degraded for that long.

26. Observing the budget: traces with the remaining time

In section 10 we proposed aggregate metrics. For debugging specific cases, the most valuable thing is for every span in a trace to carry the remaining budget at its start and at its end. With that, the trace of a 504 reads in seconds: you see the link where the budget collapsed and whether it was due to a slow call, a retry or a wait on the pool. With OpenTelemetry a couple of attributes is enough:

```
from opentelemetry import trace
import deadline

tracer = trace.get_tracer("orders")

async def call_downstream_traced(client, name: str, url: str, cap_s: float = 2.0):
    with tracer.start_as_current_span(f"call {name}") as span:
        span.set_attribute("budget.remaining_ms.start", int(deadline.remaining() * 1000))
        t = deadline.budget(reserve=0.05, cap=cap_s)
        span.set_attribute("budget.allocated_ms", int(t * 1000))
        try:
            return await client.get(url, timeout=t)
        except Exception as exc:
            # distinguish by type: TimeoutException, ConnectError, PoolTimeout...
            span.set_attribute("error.kind", type(exc).__name__)
            raise
        finally:
            span.set_attribute("budget.remaining_ms.end", int(deadline.remaining() * 1000))
```

With these attributes you can answer questions that used to require reconstructing everything by hand: what percentage of requests reach the database with less than 200 ms of budget? Which dependency consumes the most budget at the 99th percentile? How many calls are launched with a budget so low that they were doomed? That last figure is a direct measure of waste, and reducing it is one of the cheapest capacity improvements there is.

27. Frequently asked questions

Is a global timeout not enough? No: without per-phase limits you cannot tell a slow connect (a network problem) from a slow read (the dependency's problem), and without per-call limits a single slow dependency can consume the request's whole budget, leaving the others no time.

Should I propagate the deadline to third-party services? Not as a custom header: they will not understand it and you would leak internal information. Use the sub-budget as the timeout of your client towards them and let your own code handle the rest.

What if my operations are legitimately long? Make them asynchronous: accept the request, return 202 with an identifier and let the client poll for status or receive a webhook. A high timeout on a synchronous request is debt, not a solution.

What do I do with writes that time out? Assume the outcome is unknown. Check the state through the idempotency key before deciding, or give the user an honest "pending confirmation" state instead of a misleading error.

How many retries are reasonable? One, in a single layer, with jitter and within the budget. If you need more, the problem is usually the dependency's reliability, not the retry policy.

Glossary

- **Deadline:** the absolute instant after which a piece of work has no value. It is propagated as remaining time.

- Time budget: what is left until the deadline; every call spends a part and passes on the rest.
- Little's law: $L = \lambda \times W$; it relates concurrency, arrival rate and time in the system.
- Hedging: sending a second request to another replica after a threshold to cut the latency tail.
- Admission control: deciding on arrival whether the work is accepted, so as not to queue what cannot finish in time.
- Bulkhead: isolating resources (pools, semaphores) per dependency so one's failure does not exhaust another's.
- Idempotency: the property of an operation that can be repeated without changing the result beyond the first time.

Production checklist

- Every outbound call has an explicit timeout, per phase and in total.
- Remaining time is propagated between services (header or gRPC) and clamped to a maximum.
- Timeouts shrink from the outside in and are based on measured percentiles.
- Retries respect the budget, happen in a single layer and only on idempotent operations.
- Work is cancelled when the budget runs out or the client disconnects.
- `statement_timeout`, `lock_timeout` and `idle_in_transaction_session_timeout` set per role.
- The application's keep-alive exceeds the load balancer's idle timeout.
- Metrics and alerts per dependency and per timeout type.
- Failure tests (latency injected with Toxiproxy or similar) that verify the system answers 504 instead of hanging.

Conclusion

A timeout is a promise you make to your users and to your own services: "I won't make you wait longer than this." A propagated deadline turns that promise into something the whole system honours, instead of each component improvising its own. Start with the cheap wins (auditing calls with no timeout, setting database and load balancer limits) and work towards budget propagation. The next time a dependency slows down, your system will fail fast, in a bounded and visible way, instead of quietly running itself dry.