PUIGFLOWS

Circuit Breakers: Cómo Evitar Fallos en Cascada en Sistemas Distribuidos

Circuit Breakers: How to Prevent Cascading Failures in Distributed Systems

RECURSO PREMIUM

Guía · Dificultad: Intermedio · ~36 min de lectura por idioma
Edición bilingüe: Español + English · puigflows.com · Julio 2026

Por qué existen los circuit breakers

Cuando una dependencia empieza a fallar o a responder lento, cada petición que le envías retiene recursos de tu propio servicio: conexiones, workers, memoria. Si sigues llamándola como si nada, esas peticiones se acumulan esperando timeouts, tu pool de conexiones se agota y el fallo se propaga hacia arriba. Es el clásico fallo en cascada: una dependencia enferma acaba tumbando servicios que en principio estaban sanos.

Los reintentos, además, agravan el problema. Un servicio que recibe el doble o el triple de tráfico justo cuando intenta recuperarse rara vez lo consigue. El circuit breaker ataca esto de raíz: detecta que una dependencia está fallando de forma sostenida y corta las llamadas durante un tiempo, fallando rápido en local en lugar de esperar timeouts remotos. La dependencia gana aire para recuperarse y tú proteges tus propios recursos.

El nombre viene de los interruptores automáticos de la instalación eléctrica: cuando la corriente alcanza un nivel peligroso, el interruptor salta y corta el circuito antes de que algo se queme. Aquí es igual, pero con peticiones.

Los tres estados

Closed (cerrado). El estado normal: las peticiones pasan y el breaker registra éxitos y fallos. Si la proporción de fallos dentro de la ventana de observación supera el umbral configurado, el breaker se abre.

Open (abierto). Toda petición se rechaza inmediatamente con un error local, sin tocar la dependencia. Es la fase de "fallar rápido": en lugar de esperar 2 segundos a un timeout, respondes en microsegundos. Pasado el reset timeout, el breaker pasa a half-open.

Half-open (semiabierto). El breaker deja pasar un número limitado de peticiones de prueba (sondas). Si tienen éxito, considera que la dependencia se recuperó y vuelve a closed. Si alguna falla, vuelve a open y el temporizador se reinicia.

Half-open es el estado que más implementaciones caseras hacen mal. Si al expirar el timeout dejas pasar todo el tráfico embalsado de golpe, le lanzas una estampida (thundering herd) a un servicio convaleciente y lo vuelves a tirar. La clave es limitar cuántas sondas hay en vuelo simultáneamente.

Qué cuenta como fallo (y qué no)

Un error de tu cliente no siempre significa que la dependencia esté enferma. Como regla general:

Cuentan como fallo: timeouts, errores de conexión (DNS, connection refused, reset) y respuestas 5xx. Todos indican que el servicio no puede atender.

No cuentan: errores 4xx como 400, 401, 404 o 422. Son errores del cliente o de datos; el servicio está funcionando perfectamente al devolverlos. Si los cuentas, un bug en tu propio código puede abrir el circuito de una dependencia sana.

Caso especial, el 429: indica que estás por encima del límite de tasa. No abras el breaker por ello; respeta la cabecera Retry-After como vimos en el recurso de reintentos con backoff.

Detección: ventana deslizante y volumen mínimo

Hay dos formas habituales de medir fallos: por conteo (las últimas N peticiones) o por tiempo (las peticiones de los últimos N segundos). La ventana temporal suele comportarse mejor con tráfico irregular, y es la que usan Resilience4j y opossum por defecto.

Igual de importante es el volumen mínimo de peticiones: sin él, 1 fallo en 2 peticiones es un 50 % de errores y abriría el circuito con una muestra ridícula. Exige al menos 10-20 peticiones en la ventana antes de evaluar el umbral.

Implementación en Python

Esta implementación asíncrona usa ventana deslizante temporal, volumen mínimo y sondas limitadas en half-open. Es el núcleo completo del patrón en unas 70 líneas:

```
import asyncio
import time
from enum import Enum

class State(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitOpenError(Exception):
    """Rechazada sin llamar al servicio: el circuito está abierto."""

class CircuitBreaker:
    def __init__(self, failure_ratio=0.5, min_requests=10,
                 window_seconds=30.0, reset_timeout=30.0, half_open_max=3):
        self.failure_ratio = failure_ratio    # proporción de fallos que abre
        self.min_requests = min_requests      # volumen mínimo antes de evaluar
        self.window_seconds = window_seconds  # tamaño de la ventana deslizante
        self.reset_timeout = reset_timeout    # segundos en open antes de probar
        self.half_open_max = half_open_max    # sondas permitidas en half-open
        self._events = []                     # pares (timestamp, éxito)
        self._state = State.CLOSED
        self._opened_at = 0.0
        self._probes = 0
        self._probe_successes = 0
        self._lock = asyncio.Lock()

    async def call(self, func, *args, **kwargs):
        async with self._lock:
            if self._state is State.OPEN:
                if time.monotonic() - self._opened_at >= self.reset_timeout:
                    self._state = State.HALF_OPEN
                    self._probes = 0
                    self._probe_successes = 0
                else:
                    raise CircuitOpenError()
            if self._state is State.HALF_OPEN:
                if self._probes >= self.half_open_max:
                    raise CircuitOpenError()  # ya hay sondas en vuelo
                self._probes += 1
```

```
        try:
            result = await func(*args, **kwargs)
        except Exception:
            await self._record(ok=False)
            raise
        else:
            await self._record(ok=True)
            return result

    async def _record(self, ok):
        async with self._lock:
            now = time.monotonic()
            if self._state is State.HALF_OPEN:
                if not ok:
                    self._trip(now)  # la sonda falló: vuelve a open
                else:
                    self._probe_successes += 1
                    if self._probe_successes >= self.half_open_max:
                        self._state = State.CLOSED  # recuperado
                        self._events.clear()
                return
            self._events.append((now, ok))
            cutoff = now - self.window_seconds
            self._events = [e for e in self._events if e[0] >= cutoff]
            total = len(self._events)
            failures = sum(1 for _, exito in self._events if not exito)
            if total >= self.min_requests and failures / total >= self.failure_ratio:
                self._trip(now)

    def _trip(self, now):
        self._state = State.OPEN
        self._opened_at = now
        self._events.clear()
```

Y así se usa, con un fallback que devuelve datos degradados cuando el circuito está abierto:

```
breaker = CircuitBreaker(failure_ratio=0.5, min_requests=10, reset_timeout=30)

async def get_user(user_id):
    try:
        return await breaker.call(fetch_user_from_api, user_id)
    except CircuitOpenError:
        return get_cached_user(user_id)  # fallback: dato en caché, quizá viejo
```

Detalles que importan: se usa time.monotonic() y no time.time(), porque el reloj de pared puede saltar (NTP) y romper los temporizadores. El lock protege las transiciones de estado frente a corrutinas concurrentes. Y al abrir el circuito se limpia la ventana, para que los fallos viejos no reabran el circuito nada más cerrarse.

En TypeScript con opossum

En Node.js no hace falta escribirlo a mano: opossum es la librería de referencia y trae ventana deslizante, fallbacks y eventos de monitoreo:

```
import CircuitBreaker from "opossum";

async function fetchUser(id: string) {
  const res = await fetch("https://api.example.com/users/" + id, {
    signal: AbortSignal.timeout(2000),
  });
  if (!res.ok) throw new Error("HTTP " + res.status);
  return res.json();
}

const breaker = new CircuitBreaker(fetchUser, {
  timeout: 2500,                // cuenta como fallo si tarda más de 2.5 s
  errorThresholdPercentage: 50, // abre con un 50 % de fallos
  resetTimeout: 30000,          // 30 s en open antes de pasar a half-open
  volumeThreshold: 10,          // volumen mínimo antes de evaluar
  rollingCountTimeout: 30000,   // ventana deslizante de 30 s
});

breaker.fallback((id: string) => getCachedUser(id));

breaker.on("open", () => metrics.increment("user_api.breaker.open"));
breaker.on("halfOpen", () => log.warn("user_api breaker: half-open"));

const user = await breaker.fire("42");
```

Fíjate en que el timeout del breaker (2500 ms) es ligeramente mayor que el del propio fetch (2000 ms): así el error que dispara el breaker es el timeout real de la petición, con su propio mensaje, y no uno genérico del breaker. En el ecosistema Java, el equivalente moderno es Resilience4j, que sustituyó a Hystrix (en mantenimiento desde 2018).

Cómo elegir los parámetros

Umbral de fallos: 50 % es un buen punto de partida. Para dependencias críticas donde prefieres degradar pronto, baja hacia 20-30 %; para dependencias ruidosas, sube el umbral o amplía la ventana para evitar falsos positivos.

Reset timeout: 30-60 segundos es lo habitual. Demasiado corto y machacas a un servicio que aún no se recuperó; demasiado largo y sigues degradado sin necesidad.

Sondas en half-open: entre 1 y 5. Más sondas dan una señal de recuperación más fiable a cambio de algo más de riesgo.

Granularidad: un breaker por dependencia como mínimo, y por endpoint si un mismo servicio tiene rutas con fiabilidad muy distinta. Un breaker global que mezcla todas las dependencias no sirve: una API caída te cortaría el acceso a las sanas.

Fallbacks: degradar en lugar de romper

Abrir el circuito es la mitad del trabajo; la otra mitad es decidir qué recibe el usuario mientras tanto. Las opciones habituales, de mejor a peor experiencia: servir datos de caché aunque estén algo desactualizados, servir un valor por defecto razonable (recomendaciones genéricas en lugar de personalizadas), encolar la

operación para procesarla después, o devolver un error claro e inmediato. Incluso la última opción es mejor que sin breaker: el usuario ve el error en milisegundos en lugar de esperar un timeout de varios segundos.

Circuit breaker + timeouts + reintentos

Estos tres mecanismos no compiten: se apilan en capas, de dentro hacia fuera. El timeout va en la capa más interna y convierte la lentitud en un fallo detectable. Los reintentos con backoff y jitter envuelven al timeout y absorben fallos transitorios aislados. El circuit breaker envuelve todo lo anterior y corta el flujo cuando el fallo es sostenido. El orden importa: si pones los reintentos por fuera del breaker, cada reintento golpeará un circuito abierto y generará errores inútiles; y un fallo con sus reintentos agotados debe contar como un solo fallo para el breaker, no como tres.

Errores comunes

Contar los 4xx como fallos. Un bug en tu código que genera 400s abre el circuito de una dependencia perfectamente sana y conviertes un bug menor en una degradación total.

Half-open sin límite de sondas. Al expirar el timeout, todo el tráfico embalsado golpea de golpe al servicio convaleciente y lo vuelve a tirar. Es el fallo clásico de las implementaciones caseras.

Sin volumen mínimo. El primer fallo de la mañana, con tráfico bajo, abre el circuito con una muestra de dos peticiones.

Estado solo en memoria con muchas instancias. Cada réplica de tu servicio tiene su propio breaker y su propia visión: unas abren y otras no. Suele ser aceptable (cada instancia protege sus propios recursos), pero tenlo en cuenta al leer métricas; si necesitas una visión compartida, el estado puede ir en Redis, a cambio de más latencia y una nueva dependencia que también puede fallar.

Sin métricas ni alertas. Un breaker que se abre en silencio esconde el problema en lugar de exponerlo. El estado del circuito es de las mejores señales tempranas de incidente que existen.

Umbral demasiado sensible. Abrir el circuito por un blip de 3 fallos genera degradaciones innecesarias y entrena al equipo a ignorar las alertas.

Checklist para producción

- Un breaker por dependencia (o por endpoint crítico), nunca uno global.
- Solo timeouts, errores de conexión y 5xx disparan el breaker.
- Volumen mínimo de 10-20 peticiones y ventana deslizante de 30-60 s.
- Sondas limitadas en half-open (1-5 en vuelo).
- Fallback definido y probado para cada breaker: caché, valor por defecto o error claro.
- Métricas del estado del circuito y alerta cuando se abre.
- Capas en orden: timeout dentro, reintentos en medio, breaker fuera.
- Prueba el breaker en staging simulando la caída de la dependencia; un breaker sin probar es un breaker que no sabes si funciona.

Hasta aquí tienes el patrón completo y funcional: estados, detección, sondas, parámetros y capas. Lo que viene ahora es lo que separa una implementación de manual de una que sobrevive a producción real: bulkheads, estado distribuido, observabilidad, tests, el mesh y fallbacks que aguantan de verdad.

Bulkheads: que una dependencia no agote todo el pool

El circuit breaker decide *cuándo* dejar de llamar a una dependencia. El patrón bulkhead decide *cuánto* de tu servicio puede consumir una sola dependencia mientras todavía la estás llamando. El nombre viene del diseño naval: el casco de un barco se divide en compartimentos estancos para que una inundación en una sección no hunda el buque entero. En un servicio, los "compartimentos" son pools acotados de concurrencia: una dependencia lenta puede llenar su propio compartimento, pero no puede agotar los workers, conexiones o memoria que necesitan los demás caminos del código.

¿Por qué necesitas ambos? Porque el breaker solo se abre *después* de que los fallos se acumulen en la ventana. Durante esos 20 o 30 segundos de degradación antes de que salte, puede acumularse un número ilimitado de peticiones en vuelo contra una dependencia que responde en 8 segundos en lugar de 80 milisegundos. Cada una de esas peticiones retiene un worker. Con 200 workers y suficiente tráfico, tu servicio entero está atascado antes de que el breaker llegue a opinar. El bulkhead limita ese daño a un número fijo y conocido.

En Python asíncrono, el bulkhead más simple es un semáforo con un timeout de adquisición corto:

```
import asyncio

class BulkheadFullError(Exception):
    """Rechazada en local: demasiadas llamadas concurrentes a esta dependencia."""

class Bulkhead:
    def __init__(self, max_concurrent=20, acquire_timeout=0.05):
        self._sem = asyncio.Semaphore(max_concurrent)
        self._timeout = acquire_timeout

    async def call(self, func, *args, **kwargs):
        try:
            # No hagas cola indefinida: si el compartimento está lleno, falla rápido
            await asyncio.wait_for(self._sem.acquire(), self._timeout)
        except asyncio.TimeoutError:
            raise BulkheadFullError()
        try:
            return await func(*args, **kwargs)
        finally:
            self._sem.release()
```

Merece la pena detenerse en dos detalles. Primero, el `acquire_timeout` corto: un bulkhead que encola llamadas indefinidamente solo mueve el atasco de la dependencia al semáforo. Rechazar en 50 milisegundos conserva la propiedad de "fallar rápido" que hace funcionar a toda esta familia de patrones. Segundo, `BulkheadFullError` normalmente *no* debería contar como fallo de la dependencia para el breaker: la dependencia nunca vio la petición. Es una señal sobre tu propia capacidad, y merece su propia métrica.

En las capas, el bulkhead va por fuera del breaker: compruebas primero el breaker (barato, no retiene recursos), luego adquieres el hueco del bulkhead y después ejecutas la llamada envuelta en su timeout. Dimensiona cada

compartimento según el presupuesto real de latencia de la dependencia: si responde en 100 ms en p99 y le envías 100 peticiones por segundo, la Ley de Little dice que en régimen estable hay unas 10 peticiones en vuelo; un compartimento de 20-30 da margen cómodo y aun así limita un desastre a un número que tu servicio puede absorber.

Un registro de breakers: uno por dependencia sin boilerplate

La recomendación de "un breaker por dependencia, o por endpoint" choca rápido con las bases de código reales: nadie quiere cablear a mano cuarenta instancias. La respuesta estándar es un pequeño registro más un decorador, de modo que cada punto de llamada declara a qué circuito pertenece y la configuración vive en un solo sitio:

```
import functools

_REGISTRY: dict[str, CircuitBreaker] = {}

def get_breaker(name: str, **config) -> CircuitBreaker:
    if name not in _REGISTRY:
        _REGISTRY[name] = CircuitBreaker(**config)
    return _REGISTRY[name]

def with_breaker(name: str, **config):
    def decorator(func):
        breaker = get_breaker(name, **config)
        @functools.wraps(func)
        async def wrapper(*args, **kwargs):
            return await breaker.call(func, *args, **kwargs)
        return wrapper
    return decorator

@with_breaker("payments-api", failure_ratio=0.3, reset_timeout=60)
async def charge_card(payment):
    ...

@with_breaker("search-api")  # aquí los valores por defecto van bien
async def search_products(query):
    ...
```

El registro te da tres cosas además de ahorrar tecleo. Garantiza que todos los puntos de llamada que nombran `"payments-api"` comparten el mismo breaker; los duplicados accidentales con estado separado son un bug sutil que hace que los breakers parezcan poco fiables. Te da un único punto de iteración para la observabilidad: un exportador de métricas o un endpoint de salud puede recorrer `_REGISTRY` e informar de todos los circuitos del proceso. Y centraliza la configuración, que importa en cuanto empiezas a afinar: el circuito de pagos de arriba se abre con un 30 % de fallos y descansa un minuto entero porque un proveedor de pagos con problemas necesita un trato más suave que un índice de búsqueda.

El nombre merece un momento de reflexión. `"payments-api"` es un buen nombre de circuito; `"https://api.stripe.com/v1/charges?id=123"` no lo es, porque los breakers por URL fragmentan tu señal de fallo en muestras demasiado pequeñas para alcanzar nunca el volumen mínimo. Nombra los circuitos por el *dominio de fallo*: aquello que se cae como una unidad.

Compartir el estado del breaker con Redis

Todo lo anterior guarda el estado en la memoria del proceso, y para la mayoría de sistemas es la decisión correcta: cada instancia protege sus propios recursos, sin coordinación y sin latencia añadida. Pero hay dos situaciones que justifican una visión compartida. Primera, tráfico muy bajo por instancia: si cada una de tus 50 réplicas envía a una dependencia dos peticiones por minuto, ninguna instancia alcanza jamás el volumen mínimo, y sin embargo la flota en conjunto machaca a un servicio moribundo. Segunda, plataformas serverless, donde las instancias nacen y mueren demasiado rápido para acumular ventana alguna.

La trampa de los breakers distribuidos es la carrera: dos instancias leen "closed", ambas registran un fallo, ambas evalúan el umbral con datos obsoletos. La solución es la misma que en el rate limiting distribuido: mover el ciclo leer-evaluar-escribir a un script Lua atómico que Redis ejecuta como una unidad:

```
-- KEYS[1] = clave del circuito, p. ej. "cb:payments-api"
-- ARGV: now_ms, window_ms, min_requests, failure_ratio,
--        reset_timeout_ms, is_failure (0/1)
local state = redis.call('HGET', KEYS[1], 'state') or 'closed'
local opened_at = tonumber(redis.call('HGET', KEYS[1], 'opened_at') or 0)
local now = tonumber(ARGV[1])

if state == 'open' then
  if now - opened_at >= tonumber(ARGV[5]) then
    redis.call('HSET', KEYS[1], 'state', 'half_open', 'probes', 0)
    return 'half_open'
  end
  return 'open'
end

-- registra el resultado en un sorted set que actúa de ventana deslizante
local member = now .. ':' .. math.random(1000000) .. ':' .. ARGV[6]
redis.call('ZADD', KEYS[1] .. ':w', now, member)
redis.call('ZREMRANGEBYSCORE', KEYS[1] .. ':w', 0, now - tonumber(ARGV[2]))
redis.call('PEXPIRE', KEYS[1] .. ':w', ARGV[2])

local total = redis.call('ZCARD', KEYS[1] .. ':w')
if total >= tonumber(ARGV[3]) then
  local fails = 0
  for _, m in ipairs(redis.call('ZRANGE', KEYS[1] .. ':w', 0, -1)) do
    if string.sub(m, -1) == '1' then fails = fails + 1 end
  end
  if fails / total >= tonumber(ARGV[4]) then
    redis.call('HSET', KEYS[1], 'state', 'open', 'opened_at', now)
    redis.call('DEL', KEYS[1] .. ':w')
    return 'open'
  end
end
return 'closed'
```

En el lado de Python cargas el script una vez con `register_script` y lo llamas tras cada resultado; el valor devuelto te dice el estado actual del circuito. Pero lee la letra pequeña antes de adoptarlo. Cada llamada protegida cuesta ahora un viaje a Redis: un milisegundo aproximadamente que los breakers en memoria no pagan. Y has creado un modo de fallo nuevo: **si el propio Redis no responde, tu breaker debe fallar abierto** (permitir la llamada y recurrir al estado local), nunca fallar cerrado. Un mecanismo de resiliencia que se

convierte en punto único de fallo es un mal negocio. Un término medio pragmático que usan varias plataformas grandes: mantener las decisiones en local, pero publicar las transiciones de estado por pub/sub de Redis para que una instancia que abre su circuito les *sugiera* a las demás acortar sus ventanas.

Observabilidad: un breaker sin métricas es un breaker en el que no puedes confiar

El estado del circuito es una de las métricas con más señal que puede emitir tu servicio: dice "esta dependencia está lo bastante rota como para que dejáramos de llamarla", que normalmente va minutos por delante de los síntomas visibles para el usuario. Instrumenta tres cosas por circuito: el estado actual como gauge, las transiciones como contador, y las llamadas rechazadas frente a las permitidas como contadores.

```
from prometheus_client import Counter, Gauge

BREAKER_STATE = Gauge(
    "circuit_breaker_state",
    "0=closed, 1=half-open, 2=open",
    ["circuit"],
)
BREAKER_TRANSITIONS = Counter(
    "circuit_breaker_transitions_total",
    "Transiciones de estado",
    ["circuit", "from_state", "to_state"],
)
BREAKER_CALLS = Counter(
    "circuit_breaker_calls_total",
    "Llamadas vistas por el breaker",
    ["circuit", "outcome"],  # success | failure | rejected
)
```

Engancha esto en los puntos de transición del breaker (el método `_trip` y la resolución de half-open son las costuras naturales). Con opossum los eventos ya existen — `breaker.on("open", ...)`, `"halfOpen"`, `"close"`, `"reject"`, `"fallback"` — y mapearlos a contadores de `prom-client` son diez líneas.

Después escribe las alertas contra síntomas, no contra ruido. Que un circuito se abra una vez es un martes cualquiera; un circuito que se queda abierto es un incidente:

```
# PromQL — circuito atascado en open más de 5 minutos
min_over_time(circuit_breaker_state{circuit="payments-api"}[5m]) >= 2

# PromQL — aleteo (flapping): más de 4 aperturas en 15 minutos
increase(circuit_breaker_transitions_total{to_state="open"}[15m]) > 4
```

La alerta de flapping merece énfasis. Un breaker que se abre, pasa a half-open, se cierra y vuelve a abrirse inmediatamente te está diciendo que la dependencia está *parcialmente* degradada: lo bastante sana para pasar tres sondas, demasiado enferma para aguantar el tráfico completo. El flapping significa que tu número de sondas es demasiado bajo, tu reset timeout demasiado corto, o que la dependencia necesita un nivel de tráfico menor para recuperarse, lo cual es un argumento a favor de la recuperación gradual que se describe más abajo. En los dashboards, dibuja el gauge de estado como una banda de color bajo los paneles de latencia y tasa de

errores de la dependencia: el momento en que un humano puede ver "subieron los errores, la banda se puso roja, y la latencia de *nuestro* servicio se recuperó" es el momento en que el patrón se ha pagado solo.

Una línea de log estructurado por transición también merece la pena: incluye el nombre del circuito, el estado anterior y el nuevo, la proporción de fallos en el momento del disparo y el número de peticiones de la ventana. Cuando estés reconstruyendo un incidente a las 4 de la madrugada, "se abrió a las 03:41:07 con un 62 % de fallos sobre 34 peticiones" responde preguntas que un gauge no puede responder.

Cómo testear circuit breakers

Un breaker sin tests falla de una de dos maneras vergonzosas: nunca se abre (umbrales mal configurados, excepciones que no coinciden) o nunca se cierra (lógica de half-open rota). Ambas son invisibles hasta el incidente que se suponía que iba a justificar su existencia. El patrón, eso sí, es muy testeable, porque la lógica es una máquina de estados: la única dependencia incómoda es el tiempo, así que inyecta el reloj:

```
class CircuitBreaker:
    def __init__(self, ..., clock=time.monotonic):
        self._clock = clock  # inyectable para los tests
```

```
import pytest

class FakeClock:
    def __init__(self):
        self.now = 1000.0
    def __call__(self):
        return self.now
    def advance(self, seconds):
        self.now += seconds

async def failing(): raise ConnectionError()
async def ok(): return "ok"

@pytest.mark.asyncio
async def test_se_abre_al_superar_umbral_y_se_recupera():
    clock = FakeClock()
    br = CircuitBreaker(failure_ratio=0.5, min_requests=4,
                        reset_timeout=30, half_open_max=2, clock=clock)

    for _ in range(4):
        with pytest.raises(ConnectionError):
            await br.call(failing)

    # el circuito está abierto: rechaza sin tocar la dependencia
    with pytest.raises(CircuitOpenError):
        await br.call(ok)

    clock.advance(31)              # expira el reset timeout -> half-open
    assert await br.call(ok) == "ok"   # sonda 1
    assert await br.call(ok) == "ok"   # sonda 2 -> se cierra
    assert await br.call(ok) == "ok"   # vuelta a la normalidad

@pytest.mark.asyncio
async def test_half_open_limita_las_sondas():
```

```
    clock = FakeClock()
    br = CircuitBreaker(failure_ratio=0.5, min_requests=2,
                        reset_timeout=10, half_open_max=1, clock=clock)
    for _ in range(2):
        with pytest.raises(ConnectionError):
            await br.call(failing)
    clock.advance(11)

    started = asyncio.Event()
    async def slow_probe():
        started.set()
        await asyncio.sleep(0.05)
        return "ok"

    task = asyncio.create_task(br.call(slow_probe))
    await started.wait()
    with pytest.raises(CircuitOpenError):   # la segunda sonda se rechaza
        await br.call(ok)
    assert await task == "ok"
```

El segundo test es el que suspenden las implementaciones caseras: demuestra que mientras una sonda está en vuelo, el breaker no deja pasar a la estampida. Más allá de los tests unitarios, ensaya la situación real. Apunta staging a un proxy capaz de inyectar fallos — toxiproxy está construido exactamente para esto — y añade latencia o corta la conexión con la dependencia mientras miras tus dashboards. Estás verificando la cadena completa: el timeout salta, el breaker se abre, el fallback sirve, la alerta se dispara y el breaker se cierra cuando quitas el tóxico. Los equipos que hacen este simulacro cada trimestre suelen descubrir que su camino de fallback se ha podrido en silencio (una clave de caché caducada, un feature flag que nadie activó en staging) *antes* de que se lo descubra una caída real.

Circuit breaking en la infraestructura: Envoy e Istio

Si tus servicios corren detrás de Envoy — directamente o vía Istio — parte de este trabajo puede salir del código de aplicación. La *outlier detection* de Envoy observa los resultados por host y expulsa temporalmente los hosts malos del pool de balanceo; su configuración de *circuit breaking* limita las conexiones concurrentes y las peticiones pendientes por clúster, que en la práctica es un bulkhead en la capa del proxy.

En Istio ambas cosas se expresan en un `DestinationRule`:

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: payments
spec:
  host: payments.prod.svc.cluster.local
  trafficPolicy:
    connectionPool:            # el bulkhead
      tcp:
        maxConnections: 100
      http:
        http1MaxPendingRequests: 50
        maxRequestsPerConnection: 10
    outlierDetection:          # el breaker
      consecutive5xxErrors: 5
```

```
      interval: 10s           # cada cuánto se evalúan los hosts
      baseEjectionTime: 30s   # la expulsión crece con la reincidencia
      maxEjectionPercent: 50  # nunca expulsar más de la mitad del pool
```

`maxEjectionPercent` es el parámetro que la gente pasa por alto: existe para que un bug que hace que *todos* los hosts parezcan enfermos (por ejemplo, un canary del llamante enviando peticiones malformadas) no pueda expulsar el pool entero y convertir una degradación parcial en una total. Fíjate también en que la expulsión es por host: esto brilla cuando el problema es un pod malo en un deployment de veinte, algo que un breaker a nivel de aplicación no puede ver porque agrega todos los hosts detrás de un único hostname.

¿Sustituye entonces el mesh a los breakers de aplicación? No: operan sobre dominios de fallo distintos y se componen bien. El mesh ve hosts individuales y síntomas de nivel de red; puede expulsar un pod enfermo en segundos sin tocar código. La aplicación ve operaciones *lógicas* y contexto de negocio: sabe que un 200 con `"status": "DECLINED_UPSTREAM"` en el cuerpo es un fallo, sabe qué fallback servir, y puede proteger llamadas a APIs de terceros que viven completamente fuera del mesh. El reparto práctico: deja al mesh la expulsión por host y los límites de conexiones, y conserva los breakers de aplicación para las dependencias de terceros y allí donde exista un fallback con sentido.

El mismo patrón en otros stacks

Te encontrarás este patrón en todos los ecosistemas, con los mismos mandos bajo nombres distintos. Un repaso rápido a las implementaciones de referencia, para que las configuraciones anteriores se traduzcan bien:

Python — pybreaker. La implementación veterana del patrón del libro *Release It!* de Nygard, basada en fallos consecutivos, con listeners para instrumentación y estado compartido opcional respaldado en Redis ya incluido:

```
import pybreaker

db_breaker = pybreaker.CircuitBreaker(
    fail_max=5,           # fallos consecutivos, no una proporción
    reset_timeout=60,
    exclude=[KeyError],   # esto nunca cuenta como fallo de la dependencia
)

@db_breaker
def fetch_user(user_id):
    return db.query(...)
```

Java — Resilience4j. El sucesor de Hystrix y la implementación más rica del modelo de ventana deslizante que usa esta guía (las versiones mayores actuales corren sobre JVM modernas; Resilience4j 3 requiere Java 21). Sus tipos de ventana `COUNT_BASED` y `TIME_BASED` se corresponden exactamente con la discusión de conteo frente a tiempo de arriba, y `slowCallRateThreshold` añade algo que vale la pena robar: las llamadas que *tienen éxito pero van lentas* también pueden abrir el circuito.

```
CircuitBreakerConfig config = CircuitBreakerConfig.custom()
    .slidingWindowType(SlidingWindowType.TIME_BASED)
    .slidingWindowSize(30)                 // segundos
    .minimumNumberOfCalls(10)
    .failureRateThreshold(50)              // porcentaje
    .slowCallDurationThreshold(Duration.ofSeconds(2))
```

```
    .slowCallRateThreshold(80)             // 80% de llamadas lentas también lo dispara
    .waitDurationInOpenState(Duration.ofSeconds(30))
    .permittedNumberOfCallsInHalfOpenState(3)
    .build();
```

.NET — Polly v8. Polly reconstruyó su API alrededor de `ResiliencePipeline`s componibles, que codifican la regla de capas de antes: el orden en que añades estrategias *es* el orden de envoltura:

```
var pipeline = new ResiliencePipelineBuilder()
    .AddCircuitBreaker(new CircuitBreakerStrategyOptions
    {
        FailureRatio = 0.5,
        MinimumThroughput = 10,
        SamplingDuration = TimeSpan.FromSeconds(30),
        BreakDuration = TimeSpan.FromSeconds(30),
    })
    .AddRetry(new RetryStrategyOptions { MaxRetryAttempts = 3 })
    .AddTimeout(TimeSpan.FromSeconds(2))   // la capa más interna
    .Build();
```

Go — gobreaker. La implementación minimalista y muy extendida de Sony. Basada en contadores en lugar de ventana deslizante, con un callback `ReadyToTrip` donde expresas tu lógica de umbral explícitamente: más cercana en espíritu al Python artesanal de arriba que a Resilience4j.

La lección del repaso: sea cual sea la librería, siempre estás respondiendo las mismas cinco preguntas. ¿Qué cuenta como fallo? ¿Sobre qué ventana? ¿Con qué volumen mínimo? ¿Cuánto descanso en open? ¿Cuántas sondas para cerrar? Si puedes responderlas para tu dependencia, cualquiera de estas librerías es cuestión de veinte minutos.

Patrones avanzados cuando lo básico ya funciona

Recuperación gradual en lugar de un acantilado. El half-open estándar es binario: pasan tres sondas y, de golpe, vuelve el 100 % del tráfico. En servicios grandes ese acantilado puede volver a matar a una dependencia apenas recuperada: la firma del flapping descrita antes. El refinamiento es la rampa: cuando las sondas tienen éxito, admite el 10 % del tráfico, luego el 25 %, el 50 %, el 100 %, subiendo cada pocos segundos mientras los fallos se mantengan bajos, y volviendo a open si suben. Una puerta probabilística de pocas líneas (`random.random() < admit_fraction`, con las llamadas rechazadas yendo al fallback) te da la mayor parte del beneficio.

Umbrales de llamadas lentas. Una dependencia que sigue devolviendo 200s con 6 segundos de latencia puede hacerte más daño que una que falla limpiamente, porque nada "falla" y nada dispara el breaker; mientras tanto, todos los workers están aparcados. Contar las llamadas lentas-pero-exitosas hacia el umbral (como hace nativamente el `slowCallRateThreshold` de Resilience4j) cierra este hueco; si lo implementas tú, registra `(timestamp, ok, duración)` y evalúa ambas proporciones.

Propagación de deadlines. Los breakers protegen contra el fallo sostenido; los deadlines protegen cada petición individual. Si a una petición entrante le quedan 800 ms de presupuesto cuando llega a tu servicio, las llamadas que despliegues deben llevar timeouts que quepan dentro de ese presupuesto, no tus valores estáticos por defecto. Sin esto, una petición que el usuario ya abandonó sigue quemando capacidad a través de tres saltos

más. Propaga el deadline en una cabecera (gRPC lo hace de forma nativa) y recorta los timeouts por llamada al presupuesto restante.

Circuitos por tenant. Las APIs multi-tenant tienen un modo de fallo que un breaker único gestiona mal: la integración mal configurada de un tenant produce una tormenta de fallos y abre el circuito para todos. Indexar los breakers por `(dependencia, tenant)` — con un tope LRU en el registro para que la cardinalidad no explote — aísla el radio de la explosión al tenant que la causa.

Caso de estudio: el brownout del proveedor de pagos

Para aterrizar los números, aquí va un compuesto de un patrón de incidente que aparece en muchos postmortems, aplicado al stack de esta guía. Una API de e-commerce (12 réplicas, ~900 req/s) llama a un proveedor de pagos con un p99 de 220 ms. A las 14:02 el proveedor entra en brownout: el 40 % de las llamadas hacen timeout a los 2 segundos y el resto tiene éxito, pero lento.

Sin protecciones, la aritmética es cruel. Cada réplica mantiene normalmente ~2 llamadas de pago en vuelo; con las latencias del brownout eso se convierte en ~50, y los pools de workers (75 por réplica) se agotan en 90 segundos. Los health checks — servidos por los mismos workers — empiezan a fallar por timeout, el orquestador reinicia los pods "enfermos", la capacidad cae aún más, y a las 14:06 el *sitio entero* está caído, incluidos la navegación y el buscador, que jamás tocan pagos. La recuperación exige escalar a mano y aguantar una estampida de caché fría. Total: más de 40 minutos de caída completa por el fallo parcial del 40 % de una sola dependencia.

Con el stack de esta guía — timeout de 2 s, breaker al 50 % de fallos sobre 30 s con mínimo de 10 peticiones, bulkhead de 20 y fallback de respuesta cacheada — los timeouts convierten la lentitud en fallos contables de inmediato. Dentro de la primera ventana de 30 segundos el breaker de cada réplica salta (la proporción de fallos alcanza el 40-55 % según la mezcla de tráfico). El checkout pasa al fallback: los pedidos se aceptan, se encolan y se cobran de forma asíncrona cuando el proveedor se recupera, con un mensaje honesto de "la confirmación del pago te llegará por email". El bulkhead significa que, incluso durante la ventana de detección de 30 segundos, como máximo 20 workers por réplica estuvieron aparcados en pagos: la navegación ni se enteró. Los breakers pasan a half-open cada 30 s, fallan sus sondas en silencio hasta las 14:31, y entonces se cierran. Impacto total en el usuario: ~29 minutos de checkout degradado, cero minutos de caída, y el ingeniero de guardia lo vio pasar en un dashboard en lugar de pelearlo.

La diferencia entre esas dos cronologías no es ingenio — cada mecanismo individual son unas pocas decenas de líneas — sino que el fallo tenía un sitio diseñado al que ir.

Exponer el estado de los breakers en tu endpoint de salud

Por último, haz visibles los circuitos para los humanos y las máquinas que no leen Prometheus. Un endpoint de salud en FastAPI que recorre el registro:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/health/dependencies")
```

```
async def dependency_health():
    return {
        name: {
            "state": br.state.value,
            "window_requests": len(br._events),
        }
        for name, br in _REGISTRY.items()
    }
```

Dos advertencias desde la trinchera. *No* conectes esto a tu liveness probe: un pod con un breaker abierto hacia una API de terceros está degradado, no muerto, y reiniciarlo no arregla nada (el caso de estudio de arriba enseña adónde lleva ese camino). Si tienes que reflejarlo en el readiness, hazlo solo para dependencias que de verdad soporten todas las peticiones. Y trata el endpoint como una herramienta de incidentes: la primera pregunta ante un aviso de "el sitio va lento" es "¿qué circuitos están abiertos?", y esto la responde con un curl.

Dónde te deja todo esto

El breaker en sí nunca fue la parte difícil: son 70 líneas de máquina de estados. La ingeniería está en lo que lo rodea: qué cuenta como fallo, cómo anidan las capas de timeout, reintentos, breaker y bulkhead, qué sirve el fallback, cómo ves los cambios de estado y cómo demuestras que todo funciona antes de que producción lo demuestre por ti. Empieza con una dependencia — la que más te despierta de noche — dale el tratamiento completo de esta guía, y deja que el primer brownout silencioso defienda la causa del resto.

Más allá de HTTP: bases de datos, colas y gRPC

Hasta ahora todos los ejemplos eran HTTP, pero el patrón se aplica a cualquier dependencia que pueda fallar o ir lenta: lo único que cambia es qué aspecto tiene el "fallo".

Bases de datos. Aquí sé mucho más conservador. Una base de datos suele ser una dependencia dura: si está caída, rara vez existe un fallback, y un breaker abierto solo convierte un mensaje de error en otro. Donde los breakers *sí* se ganan el sueldo es con las réplicas: envuelve las lecturas a cada réplica de lectura en su propio circuito, y cuando el circuito de una réplica se abra, redirige su parte de lecturas a las réplicas restantes o a la primaria. Los criterios de fallo deben ser errores de conexión y timeouts de adquisición del pool — nunca errores de SQL, que son bugs tuyos, no enfermedad de la base de datos. Y fíjate en el solapamiento: un pool de conexiones bien configurado con un timeout de checkout corto ya se comporta como un bulkhead, que es una de las razones por las que los breakers de base de datos son menos comunes que los de HTTP.

Colas de mensajes. Los productores encajan de forma natural: si el broker deja de aceptar publicaciones, un breaker abierto te permite derramar los mensajes a un buffer local o a otra región en lugar de bloquear los hilos de las peticiones. Los consumidores normalmente *no* deberían ponerle un breaker a su propio broker — reducir el consumo es exactamente para lo que sirven las colas — pero las llamadas que el consumidor hace a *otros* servicios mientras procesa un mensaje sí deben protegerse: un consumidor machacando a un downstream muerto con el prefetch a tope es una de las formas más rápidas de llenar una dead-letter queue con miles de mensajes reintentables.

gRPC. La mecánica es idéntica, pero hay que mapear los códigos de estado. Cuenta `UNAVAILABLE`, `DEADLINE_EXCEEDED` y (normalmente) `RESOURCE_EXHAUSTED` como fallos; no cuentes `INVALID_ARGUMENT`,

`NOT_FOUND`, `PERMISSION_DENIED` ni `UNAUTHENTICATED`, que son la familia 4xx vestida de gRPC. Los interceptores del lado cliente son la costura natural: un interceptor puede consultar el registro y proteger todos los stubs del proceso, lo que gana por goleada a envolver los puntos de llamada uno a uno.

Conectar el breaker a tu cliente HTTP

Envolver cada punto de llamada a mano invita al día en que alguien se olvide. En httpx puedes empujar el breaker hasta la capa de transporte, de forma que cada petición que pasa por el cliente queda protegida automáticamente — incluidas las que se escriban cuando ya nadie esté pensando en resiliencia:

```
import httpx

class BreakerTransport(httpx.AsyncBaseTransport):
    """Aplica un circuit breaker a cada petición de este cliente."""

    def __init__(self, breaker: CircuitBreaker, inner=None):
        self._breaker = breaker
        self._inner = inner or httpx.AsyncHTTPTransport()

    async def handle_async_request(self, request):
        async def send():
            response = await self._inner.handle_async_request(request)
            if response.status_code >= 500:
                # convierte los 5xx en fallos sin consumir el cuerpo
                raise DependencyError(response.status_code)
            return response
        return await self._breaker.call(send)

payments_client = httpx.AsyncClient(
    base_url="https://api.payments.example.com",
    timeout=httpx.Timeout(2.0, connect=0.5),
    transport=BreakerTransport(get_breaker("payments-api")),
)
```

Dos cosas que observar. El transporte lanza una excepción con los 5xx para que el breaker pueda contarlos: para un cliente httpx normal, un 500 es un intercambio HTTP perfectamente exitoso, lo cual es técnicamente cierto y operativamente inútil. Y el timeout vive en el cliente, dentro del breaker, respetando el orden de capas: timeout en lo más interno, breaker por fuera. Si añades reintentos, van entre los dos — un transporte de reintentos envolviendo al transporte interno, envuelto a su vez por el breaker — y todo el stack queda declarado una sola vez, junto al cliente, en lugar de espolvoreado por la base de código.

La misma idea existe en casi todos los stacks: interceptores de axios o un wrapper de fetch en Node, `DelegatingHandler` en el `HttpClientFactory` de .NET (donde Polly se integra de forma nativa), interceptores de OkHttp en la JVM. Lo que importa es el principio: la configuración de resiliencia pertenece al *objeto cliente*, declarada una vez, no a cada llamada.

Elegir los criterios de fallo según el tipo de dependencia

"Timeouts, errores de conexión y 5xx" es el buen valor por defecto, pero las dependencias reales premian una segunda mirada. Una pequeña guía de campo:

APIs SaaS de terceros. Vigila las que devuelven 200 con un sobre de error en el cuerpo (`{"ok": false, "error": "internal_error"}`). Si el proveedor hace esto, tu predicado de fallo tiene que parsear el cuerpo, o tu breaker dormirá durante toda la caída. Los proveedores de pago son famosos por una trampa relacionada: un gateway timeout en un cobro *no* es un fallo limpio — el cobro puede haberse ejecutado — así que el breaker debe convivir con claves de idempotencia y el fallback no debe reintentar a ciegas jamás.

Servicios internos durante deploys. Los deploys progresivos producen ráfagas breves de resets de conexión y 503s que son perfectamente sanas. Si tu ventana y tu volumen mínimo están afinados demasiado justos, cada deploy de una dependencia te abre los circuitos: ruido que entrena a todo el mundo a ignorar las alertas de breaker. O bien amplía la ventana por encima de la duración típica de tus rollouts, o haz que tu plataforma marque las ventanas de deploy para poder suprimir las alertas (no el breaker en sí).

Cachés. Un breaker sobre Redis-como-caché debe saltar con errores de conexión e ir emparejado con un fallback que vaya directo a la fuente de verdad, más — crucial — un bulkhead en ese camino, porque la base de datos está a punto de recibir el tráfico que la caché absorbía. El breaker evita la peor versión de esto: que cada petición pague un timeout de 2 segundos de Redis *antes* de recurrir al fallback, lo que convierte una caída de caché en un incidente de latencia de todo el sitio.

APIs de LLM e inferencia. Las respuestas largas en streaming difuminan qué significa un timeout: la señal de salud que quieres es el tiempo hasta el primer token, no la duración total, que varía legítimamente de dos a noventa segundos según la longitud de la salida. Dispara con errores de conexión, 5xx, 529 ("overloaded", que usan algunos proveedores) y primeros tokens lentos; trata también como fallo un stream que muere a mitad. Los 429 de rate limit, como antes, pertenecen a la capa de reintentos con backoff, no al breaker.

Introducir breakers en una base de código existente

Meter resiliencia con calzador en un servicio que lleva años funcionando sin ella merece la misma cautela que cualquier cambio de comportamiento. Una secuencia de despliegue que evita incidentes autoinfligidos:

Etapa uno: modo sombra. Despliega el breaker con el disparo desactivado: registra resultados, calcula proporciones, emite métricas y no rechaza nada. Déjalo una o dos semanas con tráfico real. Estás aprendiendo las tasas de fallo base reales de tus dependencias, que siempre sorprenden: servicios que creías sólidos resultan vivir con un 2 % de errores permanente, y un umbral del 50 % que parecía conservador en realidad no se dispararía nunca.

```
class ShadowBreaker(CircuitBreaker):
    """Cuenta e informa, pero nunca rechaza. Para calibración."""
    async def call(self, func, *args, **kwargs):
        state_before = self._state
        try:
            return await super().call(func, *args, **kwargs)
        except CircuitOpenError:
            BREAKER_CALLS.labels(self.name, "would_reject").inc()
            return await func(*args, **kwargs)  # llama de todos modos
```

Etapa dos: calibrar con datos. Fija el umbral de cada circuito bien por encima de su base observada: una dependencia que vive al 2 % de errores aguanta un umbral del 30 %; una que sube al 15 % durante los batch nocturnos necesita un 50 % o una ventana más ancha. La métrica `would_reject` del modo sombra te dice

exactamente cuántas veces se *habría* abierto cada circuito; afina hasta que ese número coincida con los incidentes reales y con nada más.

Etapa tres: activar en una dependencia. Elige la que tenga un fallback de verdad bueno y un historial de despertar a la guardia. Activa el disparo detrás de un flag de configuración que puedas cambiar sin deploy: una variable de entorno leída al arrancar vale para empezar; un sistema de configuración dinámica es mejor porque además te da un interruptor de emergencia por circuito, que es la herramienta que querrás la primera vez que un breaker se porte mal en producción.

Etapa cuatro: expandir y codificar. Cuando dos o tres dependencias hayan sobrevivido a un incidente real detrás de breakers, mueve el patrón a tu librería cliente interna o a tu plantilla de servicios para que los servicios nuevos lo hereden por defecto. Los equipos que se quedan en "los servicios importantes tienen breakers" siguen expuestos por la cola larga; los que lo meten en la plantilla dejan de pensar en ello por completo — que es el objetivo.

Una nota organizativa: los umbrales de un breaker codifican suposiciones sobre los servicios de *otros equipos*. Cuando pongas un umbral del 30 % sobre la API de un equipo hermano, díselo — idealmente el umbral vive junto a un SLO que ambos equipos acordaron. La alternativa es el clásico incidente entre equipos en el que los breakers del servicio A se abrieron porque el servicio B consideraba que su propio 20 % de errores estaba "dentro del SLO", y ambos equipos tenían técnicamente razón.

Respuestas rápidas a preguntas comunes

¿Necesito un breaker si ya tengo reintentos y timeouts? Para una dependencia que falla poco y durante poco tiempo, reintentos y timeouts pueden bastar. El breaker se gana su sitio cuando el fallo es *sostenido*: es el único de los tres que deja de añadir carga a un servicio en apuros, y el único que convierte un timeout de varios segundos en un rechazo de microsegundos durante una caída.

¿El breaker vive en el llamante o en el llamado? En el llamante. Que el llamado se proteja de la sobrecarga es un patrón distinto (y también valioso): el load shedding. Se componen: tu breaker evita que malgastes tus propios recursos en una dependencia enferma; su load shedder evita que se derrita cuando la acribillen llamantes sin breakers.

¿Un breaker por servicio o uno por endpoint? Empieza por servicio. Divide por endpoint cuando observes rutas con modos de fallo de verdad independientes: un servicio de búsqueda cuyo endpoint `/suggest` se cae bajo carga mientras `/search` sigue sano justifica dos circuitos; dos rutas CRUD respaldadas por la misma base de datos, no.

¿Cuánto tiempo en open es "sano" en producción? Aproximadamente cero, casi siempre. Los circuitos deberían abrirse durante incidentes reales de la dependencia y esencialmente nunca en otro caso. Si tus dashboards muestran circuitos abriéndose cada semana sin incidente correspondiente, tus umbrales están demasiado justos o tu clasificación de 4xx/5xx está mal — y la fatiga de alertas acabará costándote el único circuito abierto que importaba.

¿Hay algún caso en el que lo correcto sea no poner breaker? Sí: dependencias duras sin fallback posible y sin fan-out que proteger. Si tu servicio es una API fina sobre una única base de datos y no puede hacer nada útil sin ella, un breaker sobre todo re-etiqueta errores. Invierte ese esfuerzo en el pool de conexiones, los timeouts y la capacidad.

Diseñar fallbacks que de verdad aguanten

La lista de fallbacks de antes — caché, valor por defecto, cola, error claro — se lee como si las opciones fueran intercambiables. En la práctica, cada una tiene detalles estructurales, y un fallback que nunca se diseñó para tráfico sostenido es la segunda cosa que se rompe durante un incidente.

Caché rancia, bien hecha. La versión ingenua cachea respuestas con un TTL y sirve lo que quede cuando el circuito se abre. El problema: los TTL se afinan para la frescura, así que a los cinco minutos de caída la caché está vacía y tu fallback muere en silencio. La versión robusta separa frescura de supervivencia: conserva las entradas mucho después de que caduquen, y deja que el fallback acepte explícitamente la ranciedad:

```
STALE_TTL = 86_400   # consérvala un día
FRESH_TTL = 300      # considérala fresca 5 minutos

async def get_profile(user_id):
    key = f"profile:{user_id}"
    try:
        data = await breaker.call(fetch_profile, user_id)
        await redis.set(key, serialize(data, at=now()), ex=STALE_TTL)
        return data
    except (CircuitOpenError, DependencyError):
        cached = await redis.get(key)
        if cached is None:
            raise                    # sin fallback disponible: error honesto
        entry = deserialize(cached)
        entry.meta["stale"] = now() - entry.at > FRESH_TTL
        return entry                 # quizá de hace horas — y etiquetada como tal
```

Fíjate en el flag `stale`: la API les dice a sus consumidores que el dato está degradado en lugar de mentir en silencio. Los frontends pueden pintar un aviso de "actualizado hace 3 h", y los consumidores batch pueden decidir que esa ranciedad es inaceptable para su caso. Degradado-pero-etiquetado gana a aparentemente-fresco-pero-erróneo en casi cualquier dominio; las excepciones — precios, inventario, decisiones de autorización — son exactamente los sitios donde el fallback correcto es un error, no una suposición.

Encolar y seguir, con su letra pequeña. El fallback del checkout del caso de estudio — aceptar el pedido, cobrar después — es el fallback más potente que existe, porque convierte una caída en mera latencia. Su precio se paga en diseño por adelantado. La operación tiene que ser genuinamente aplazable (un cobro lo es; una comprobación de permisos no). El mensaje al usuario tiene que cambiar: "pedido confirmado" pasa a ser "pedido recibido, la confirmación llegará después", y saltarse esa honestidad genera tickets de soporte y contracargos. La cola de aplazados necesita un límite de profundidad con una decisión sobre qué pasa cuando se llena: en algún punto "acepta todo y ya lo arreglaremos" se convierte en un pasivo, y el comportamiento correcto vuelve a degradar, a "checkout temporalmente no disponible". Y el camino de vaciado necesita sus propias protecciones: cuando el proveedor se recupera, una cola con cuarenta minutos de pedidos debe salir goteando por un rate limiter, no en estampida, o volverás a provocar exactamente el brownout que construyó la cola. Cada pieza de esto — el aplazamiento, el mensaje, el tope, el vaciado dosificado — tiene que existir *antes* del incidente, y por eso el "pues lo encolamos" dicho en mitad de una caída es un plan que no ha funcionado ni una sola vez.

Valores por defecto, con alcance corto. Servir recomendaciones genéricas cuando el servicio de personalización está caído es el fallback de manual, porque el radio de daño de equivocarse es casi cero. El

patrón generaliza solo hasta donde aguanta esa propiedad: un feature flag por defecto en "off" es seguro; un "aprobado" por defecto en un control antifraude es un evento de carrera profesional. Escribe el valor por defecto junto al breaker, en la revisión de código, con la pregunta "¿cuál es la peor petición a la que se le podría servir este defecto?" respondida en un comentario.

Una propiedad más que comparten los tres: **los fallbacks necesitan su propia observabilidad**. Cuenta las ejecuciones de fallback por circuito y alerta cuando se mantengan elevadas. Un servicio que lleva seis horas sirviendo caché rancia en silencio porque un breaker nunca se cerró es un incidente que nadie vio: las métricas estaban verdes, los usuarios recibían los datos de ayer, y la alerta que debió dispararse estaba en un dashboard que nadie construyó. El par de números que importa es simple: con qué frecuencia degradas y durante cuánto tiempo. Si cualquiera de los dos crece, la maquinaria de resiliencia ha dejado de ser una red de seguridad y ha empezado a ser una alfombra bajo la que barrer los fallos.

Arranques en frío, deploys y el estado del breaker

El estado del breaker vive en la memoria del proceso, lo que significa que cada deploy, reinicio y evento de autoescalado lo resetea. Esto tiene consecuencias que conviene conocer de antemano. Una instancia nueva arranca con todos los circuitos cerrados y la ventana vacía: si nace en mitad de la caída de una dependencia, quemará `min_requests` fallos reales antes de que su breaker se abra. Con autoescalado agresivo esto crea un bucle feo: la latencia sube (porque la dependencia va lenta), el autoescalador añade instancias, cada instancia nueva redescubre la caída desde cero y añade su propia ráfaga de peticiones condenadas a la dependencia moribunda. A escala de flota, el sistema "protegido" todavía puede entregar una estampida considerable, una instancia fría cada vez.

Tres mitigaciones, en orden creciente de esfuerzo. Primera, asegúrate de que el volumen mínimo no está sobredimensionado: una ventana que necesita 100 peticiones para evaluar deja a las instancias frías desprotegidas mucho más tiempo que una que necesita 10. Segunda, si usas el estado compartido en Redis de antes, las instancias nuevas heredan la visión de la flota de inmediato, lo que resuelve el problema de raíz y es el argumento más fuerte a favor del estado compartido en entornos con autoescalado. Tercera, una gracia de arranque: las instancias que nacen mientras el circuito de una hermana está abierto (visible vía el endpoint de salud o las pistas por pub/sub) pueden arrancar ese circuito en half-open en lugar de closed, enviando sondas en vez de tráfico completo hasta que la dependencia demuestre estar sana. Elijas lo que elijas, ensaya la combinación — haz un deploy durante una caída simulada de la dependencia en staging — porque las interacciones autoescalador-más-breaker son exactamente el tipo de comportamiento emergente que jamás aparece en los tests unitarios.

English Version

The complete guide in English follows.

Why circuit breakers exist

When a dependency starts failing or responding slowly, every request you send it ties up resources in your own service: connections, workers, memory. If you keep calling it as if nothing happened, those requests pile up waiting for timeouts, your connection pool runs dry, and the failure propagates upward. That is the classic cascading failure: one sick dependency ends up taking down services that were perfectly healthy.

Retries make it worse. A service that receives two or three times its normal traffic right when it is trying to recover rarely manages to. The circuit breaker attacks this at the root: it detects that a dependency is failing consistently and cuts off calls for a while, failing fast locally instead of waiting on remote timeouts. The dependency gets room to breathe and you protect your own resources.

The name comes from electrical circuit breakers: when the current reaches a dangerous level, the breaker trips and cuts the circuit before something burns. Same idea here, but with requests.

The three states

Closed. The normal state: requests flow through and the breaker records successes and failures. If the failure ratio within the observation window exceeds the configured threshold, the breaker opens.

Open. Every request is rejected immediately with a local error, without touching the dependency. This is the "fail fast" phase: instead of waiting 2 seconds for a timeout, you respond in microseconds. Once the reset timeout elapses, the breaker moves to half-open.

Half-open. The breaker lets through a limited number of trial requests (probes). If they succeed, it considers the dependency recovered and returns to closed. If any fails, it goes back to open and the timer restarts.

Half-open is the state most home-grown implementations get wrong. If you release all the backed-up traffic at once when the timeout expires, you unleash a thundering herd on a convalescing service and knock it right back down. The key is limiting how many probes are in flight at the same time.

What counts as a failure (and what does not)

An error in your client does not always mean the dependency is sick. As a general rule:

Count as failures: timeouts, connection errors (DNS, connection refused, reset) and 5xx responses. All of them mean the service cannot serve.

Do not count: 4xx errors like 400, 401, 404 or 422. Those are client or data errors; the service is working perfectly well when it returns them. If you count them, a bug in your own code can open the circuit of a healthy dependency.

Special case, 429: it means you are over the rate limit. Do not trip the breaker for it; honor the Retry-After header, as covered in the resource on retries with backoff.

Detection: sliding window and minimum volume

There are two common ways to measure failures: count-based (the last N requests) or time-based (the requests from the last N seconds). The time-based window usually behaves better with irregular traffic, and it is what Resilience4j and opossum use by default.

Just as important is the minimum request volume: without it, 1 failure out of 2 requests is a 50 % error rate and would open the circuit on a laughably small sample. Require at least 10-20 requests in the window before evaluating the threshold.

A Python implementation

This async implementation uses a time-based sliding window, minimum volume, and limited probes in half-open. It is the complete core of the pattern in about 70 lines:

```
import asyncio
import time
from enum import Enum

class State(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitOpenError(Exception):
    """Rejected without calling the service: the circuit is open."""

class CircuitBreaker:
    def __init__(self, failure_ratio=0.5, min_requests=10,
                 window_seconds=30.0, reset_timeout=30.0, half_open_max=3):
        self.failure_ratio = failure_ratio    # failure ratio that opens it
        self.min_requests = min_requests      # minimum volume before evaluating
        self.window_seconds = window_seconds  # sliding window size
        self.reset_timeout = reset_timeout    # seconds in open before probing
        self.half_open_max = half_open_max    # probes allowed in half-open
        self._events = []                     # (timestamp, success) pairs
        self._state = State.CLOSED
        self._opened_at = 0.0
        self._probes = 0
        self._probe_successes = 0
        self._lock = asyncio.Lock()

    async def call(self, func, *args, **kwargs):
        async with self._lock:
            if self._state is State.OPEN:
                if time.monotonic() - self._opened_at >= self.reset_timeout:
                    self._state = State.HALF_OPEN
                    self._probes = 0
                    self._probe_successes = 0
                else:
                    raise CircuitOpenError()
            if self._state is State.HALF_OPEN:
                if self._probes >= self.half_open_max:
                    raise CircuitOpenError()  # probes already in flight
                self._probes += 1
```

```
        try:
            result = await func(*args, **kwargs)
        except Exception:
            await self._record(ok=False)
            raise
        else:
            await self._record(ok=True)
            return result

    async def _record(self, ok):
        async with self._lock:
            now = time.monotonic()
            if self._state is State.HALF_OPEN:
                if not ok:
                    self._trip(now)  # probe failed: back to open
                else:
                    self._probe_successes += 1
                    if self._probe_successes >= self.half_open_max:
                        self._state = State.CLOSED  # recovered
                        self._events.clear()
                return
            self._events.append((now, ok))
            cutoff = now - self.window_seconds
            self._events = [e for e in self._events if e[0] >= cutoff]
            total = len(self._events)
            failures = sum(1 for _, success in self._events if not success)
            if total >= self.min_requests and failures / total >= self.failure_ratio:
                self._trip(now)

    def _trip(self, now):
        self._state = State.OPEN
        self._opened_at = now
        self._events.clear()
```

And this is how you use it, with a fallback that returns degraded data while the circuit is open:

```
breaker = CircuitBreaker(failure_ratio=0.5, min_requests=10, reset_timeout=30)

async def get_user(user_id):
    try:
        return await breaker.call(fetch_user_from_api, user_id)
    except CircuitOpenError:
        return get_cached_user(user_id)  # fallback: cached, possibly stale
```

Details that matter: it uses time.monotonic() rather than time.time(), because the wall clock can jump (NTP) and break the timers. The lock protects state transitions from concurrent coroutines. And opening the circuit clears the window, so old failures cannot reopen the circuit the moment it closes.

In TypeScript with opossum

In Node.js you do not need to write it by hand: opossum is the reference library and ships with a sliding window, fallbacks, and monitoring events:

```
import CircuitBreaker from "opossum";

async function fetchUser(id: string) {
  const res = await fetch("https://api.example.com/users/" + id, {
    signal: AbortSignal.timeout(2000),
  });
  if (!res.ok) throw new Error("HTTP " + res.status);
  return res.json();
}

const breaker = new CircuitBreaker(fetchUser, {
  timeout: 2500,                // counts as failure if it takes over 2.5 s
  errorThresholdPercentage: 50,  // opens at a 50 % failure rate
  resetTimeout: 30000,          // 30 s in open before moving to half-open
  volumeThreshold: 10,          // minimum volume before evaluating
  rollingCountTimeout: 30000,   // 30 s sliding window
});

breaker.fallback((id: string) => getCachedUser(id));

breaker.on("open", () => metrics.increment("user_api.breaker.open"));
breaker.on("halfOpen", () => log.warn("user_api breaker: half-open"));

const user = await breaker.fire("42");
```

Note that the breaker timeout (2500 ms) is slightly longer than the fetch's own timeout (2000 ms): that way the error that trips the breaker is the request's real timeout, with its own message, rather than a generic one from the breaker. In the Java ecosystem, the modern equivalent is Resilience4j, which replaced Hystrix (in maintenance mode since 2018).

How to choose the parameters

Failure threshold: 50 % is a good starting point. For critical dependencies where you would rather degrade early, go down to 20-30 %; for noisy dependencies, raise the threshold or widen the window to avoid false positives.

Reset timeout: 30-60 seconds is typical. Too short and you hammer a service that has not recovered yet; too long and you stay degraded for no reason.

Probes in half-open: between 1 and 5. More probes give a more reliable recovery signal in exchange for slightly more risk.

Granularity: one breaker per dependency at minimum, and per endpoint when a single service has routes with very different reliability. A global breaker that mixes all dependencies is useless: one downed API would cut off your access to the healthy ones.

Fallbacks: degrade instead of breaking

Opening the circuit is half the job; the other half is deciding what the user gets in the meantime. The usual options, from best to worst experience: serve cached data even if slightly stale, serve a sensible default (generic recommendations instead of personalized ones), queue the operation to process later, or return a clear,

immediate error. Even the last option beats having no breaker: the user sees the error in milliseconds instead of waiting through a multi-second timeout.

Circuit breaker + timeouts + retries

These three mechanisms do not compete: they stack in layers, from the inside out. The timeout sits in the innermost layer and turns slowness into a detectable failure. Retries with backoff and jitter wrap the timeout and absorb isolated transient failures. The circuit breaker wraps everything and cuts the flow when the failure is sustained. Order matters: if you put retries outside the breaker, every retry will slam into an open circuit and generate useless errors; and a failure with its retries exhausted should count as a single failure for the breaker, not three.

Common mistakes

Counting 4xx as failures. A bug in your code that produces 400s opens the circuit of a perfectly healthy dependency, turning a minor bug into a full degradation.

Half-open without a probe limit. When the timeout expires, all the backed-up traffic hits the convalescing service at once and knocks it down again. It is the classic failure of home-grown implementations.

No minimum volume. The first failure of the morning, under low traffic, opens the circuit on a two-request sample.

In-memory state with many instances. Each replica of your service has its own breaker and its own view: some open, some do not. This is usually acceptable (each instance protects its own resources), but keep it in mind when reading metrics; if you need a shared view, the state can live in Redis, at the cost of extra latency and a new dependency that can also fail.

No metrics or alerts. A breaker that opens silently hides the problem instead of exposing it. Circuit state is one of the best early incident signals there is.

An oversensitive threshold. Opening the circuit over a 3-failure blip causes unnecessary degradations and trains the team to ignore the alerts.

Production checklist

- One breaker per dependency (or per critical endpoint), never a global one.
- Only timeouts, connection errors and 5xx trip the breaker.
- Minimum volume of 10-20 requests and a 30-60 s sliding window.
- Limited probes in half-open (1-5 in flight).
- A fallback defined and tested for every breaker: cache, default value, or a clear error.
- Circuit-state metrics and an alert when it opens.
- Layers in order: timeout inside, retries in the middle, breaker outside.
- Test the breaker in staging by simulating the dependency going down; an untested breaker is a breaker you do not know works.

That is the complete, working pattern: states, detection, probes, parameters, and layering. What follows is what separates a textbook implementation from one that survives real production: bulkheads, distributed state, observability, testing, the mesh, and fallbacks that actually hold.

Bulkheads: don't let one dependency drain the whole pool

The circuit breaker decides *when* to stop calling a dependency. The bulkhead pattern decides *how much* of your service a single dependency is allowed to consume while it is still being called. The name comes from ship design: a hull is divided into watertight compartments so that flooding in one section does not sink the vessel. In a service, the "compartments" are bounded pools of concurrency — a slow dependency can fill its own compartment, but it cannot drain the workers, connections, or memory that other code paths need.

Why do you need both? Because a breaker only opens *after* failures accumulate in the window. During those 20 or 30 seconds of degradation before the breaker trips, an unbounded number of in-flight requests can pile up against a dependency that answers in 8 seconds instead of 80 milliseconds. Each of those requests holds a worker. With 200 workers and enough traffic, your whole service is wedged before the breaker ever gets a vote. The bulkhead caps that damage at a fixed, known number.

In async Python the simplest bulkhead is a semaphore with a short acquisition timeout:

```
import asyncio

class BulkheadFullError(Exception):
    """Rejected locally: too many concurrent calls to this dependency."""

class Bulkhead:
    def __init__(self, max_concurrent=20, acquire_timeout=0.05):
        self._sem = asyncio.Semaphore(max_concurrent)
        self._timeout = acquire_timeout

    async def call(self, func, *args, **kwargs):
        try:
            # Don't queue forever: if the compartment is full, fail fast
            await asyncio.wait_for(self._sem.acquire(), self._timeout)
        except asyncio.TimeoutError:
            raise BulkheadFullError()
        try:
            return await func(*args, **kwargs)
        finally:
            self._sem.release()
```

Two details are worth pausing on. First, the short `acquire_timeout`: a bulkhead that queues callers indefinitely just moves the pile-up from the dependency to the semaphore. Rejecting in 50 milliseconds keeps the "fail fast" property that makes this whole family of patterns work. Second, `BulkheadFullError` should usually *not* count as a dependency failure for the breaker — the dependency never saw the request. It is a signal about your own capacity, and worth tracking as its own metric.

Layered with the breaker, the bulkhead goes on the outside: check the breaker first (cheap, no resources held), then acquire the bulkhead slot, then run the timeout-wrapped call. Size each compartment from the dependency's real latency budget: if a dependency answers in 100 ms at p99 and you send it 100 requests per

second, Little's Law says about 10 requests are in flight in steady state — a compartment of 20-30 gives comfortable headroom while still capping a meltdown at a number your service can absorb.

A breaker registry: per-dependency breakers without boilerplate

The guidance "one breaker per dependency, or per endpoint" collides with real codebases fast: nobody wants to hand-wire forty breaker instances. The standard answer is a small registry plus a decorator, so call sites declare which circuit they belong to and configuration lives in one place:

```
import functools

_REGISTRY: dict[str, CircuitBreaker] = {}

def get_breaker(name: str, **config) -> CircuitBreaker:
    if name not in _REGISTRY:
        _REGISTRY[name] = CircuitBreaker(**config)
    return _REGISTRY[name]

def with_breaker(name: str, **config):
    def decorator(func):
        breaker = get_breaker(name, **config)
        @functools.wraps(func)
        async def wrapper(*args, **kwargs):
            return await breaker.call(func, *args, **kwargs)
        return wrapper
    return decorator

@with_breaker("payments-api", failure_ratio=0.3, reset_timeout=60)
async def charge_card(payment):
    ...

@with_breaker("search-api")  # defaults are fine here
async def search_products(query):
    ...
```

The registry buys you three things beyond less typing. It guarantees that every call site naming `"payments-api"` shares the same breaker — accidental duplicates with separate state are a subtle bug that makes breakers look flaky. It gives you one iteration point for observability: a metrics exporter or a health endpoint can walk `_REGISTRY` and report every circuit in the process. And it centralizes configuration, which matters once you start tuning: the payments circuit above opens at 30 % failures and rests for a full minute because a payment provider that is struggling needs gentler treatment than a search index.

Naming deserves a moment of thought. `"payments-api"` is a fine circuit name; `"https://api.stripe.com/v1/charges?id=123"` is not, because per-URL breakers fragment your failure signal into samples too small to ever cross the minimum volume. Name circuits after the *failure domain*: the thing that goes down as a unit.

Sharing breaker state with Redis

Everything so far keeps state in process memory, and for most systems that is the right call: each instance protects its own resources, no coordination needed, zero added latency. But two situations justify a shared view. First, very low per-instance traffic — if each of your 50 replicas sends a dependency two requests per minute, no single instance ever reaches minimum volume, yet the fleet collectively hammers a dying service. Second, serverless platforms, where instances are born and die too fast to accumulate a window at all.

The trap in distributed breakers is the race: two instances read "closed", both record a failure, both evaluate the threshold on stale data. The fix is the same as in distributed rate limiting — push the read-evaluate-write cycle into an atomic Lua script so Redis executes it as a unit:

```
-- KEYS[1] = circuit key, e.g. "cb:payments-api"
-- ARGV: now_ms, window_ms, min_requests, failure_ratio,
--        reset_timeout_ms, is_failure (0/1)
local state = redis.call('HGET', KEYS[1], 'state') or 'closed'
local opened_at = tonumber(redis.call('HGET', KEYS[1], 'opened_at') or 0)
local now = tonumber(ARGV[1])

if state == 'open' then
  if now - opened_at >= tonumber(ARGV[5]) then
    redis.call('HSET', KEYS[1], 'state', 'half_open', 'probes', 0)
    return 'half_open'
  end
  return 'open'
end

-- record the outcome in a sorted set acting as the sliding window
local member = now .. ':' .. math.random(1000000) .. ':' .. ARGV[6]
redis.call('ZADD', KEYS[1] .. ':w', now, member)
redis.call('ZREMRANGEBYSCORE', KEYS[1] .. ':w', 0, now - tonumber(ARGV[2]))
redis.call('PEXPIRE', KEYS[1] .. ':w', ARGV[2])

local total = redis.call('ZCARD', KEYS[1] .. ':w')
if total >= tonumber(ARGV[3]) then
  local fails = 0
  for _, m in ipairs(redis.call('ZRANGE', KEYS[1] .. ':w', 0, -1)) do
    if string.sub(m, -1) == '1' then fails = fails + 1 end
  end
  if fails / total >= tonumber(ARGV[4]) then
    redis.call('HSET', KEYS[1], 'state', 'open', 'opened_at', now)
    redis.call('DEL', KEYS[1] .. ':w')
    return 'open'
  end
end
return 'closed'
```

On the Python side you load the script once with `register_script` and call it after every outcome; the return value tells you the circuit's current state. But read the fine print before adopting this. Every guarded call now costs a Redis round trip — a millisecond or so that pure in-memory breakers do not pay. And you have created a new failure mode: **if Redis itself is unreachable, your breaker must fail open** (allow the call and fall back to local state), never fail closed. A resilience mechanism that becomes a single point of failure is a bad trade. A

pragmatic middle ground used by several large platforms: keep decisions local, but publish state transitions to Redis pub/sub so that one instance tripping *hints* the others to shorten their windows.

Observability: a breaker without metrics is a breaker you can't trust

The circuit state is one of the highest-signal metrics your service can emit: it says "this dependency is broken enough that we stopped calling it", which is usually minutes ahead of user-visible symptoms. Instrument three things per circuit: the current state as a gauge, transitions as a counter, and rejected-vs-allowed calls as counters.

```
from prometheus_client import Counter, Gauge

BREAKER_STATE = Gauge(
    "circuit_breaker_state",
    "0=closed, 1=half-open, 2=open",
    ["circuit"],
)
BREAKER_TRANSITIONS = Counter(
    "circuit_breaker_transitions_total",
    "State transitions",
    ["circuit", "from_state", "to_state"],
)
BREAKER_CALLS = Counter(
    "circuit_breaker_calls_total",
    "Calls seen by the breaker",
    ["circuit", "outcome"],  # success | failure | rejected
)
```

Hook these into the breaker's transition points (the `_trip` method and the half-open resolution are the natural seams). With opossum the events are already there — `breaker.on("open", ...)`, `"halfOpen"`, `"close"`, `"reject"`, `"fallback"` — and mapping them onto `prom-client` counters takes ten lines.

Then write the alerts against symptoms, not noise. A circuit opening once is Tuesday; a circuit that stays open is an incident:

```
# PromQL — circuit stuck open for more than 5 minutes
min_over_time(circuit_breaker_state{circuit="payments-api"}[5m]) >= 2

# PromQL — flapping: more than 4 open transitions in 15 minutes
increase(circuit_breaker_transitions_total{to_state="open"}[15m]) > 4
```

The flapping alert deserves emphasis. A breaker that opens, half-opens, closes, and immediately opens again is telling you the dependency is *partially* degraded — healthy enough to pass three probes, too sick to carry full traffic. Flapping means your probe count is too low, your reset timeout too short, or the dependency needs a lower traffic level to recover, which is an argument for the gradual-recovery ramp described below. On dashboards, plot the state gauge as a colored band under the dependency's latency and error-rate panels: the moment a human can see "errors rose, then the band went red, then latency of *our* service recovered" is the moment the pattern has paid for itself.

One structured log line per transition is also worth it — include the circuit name, the old and new state, the failure ratio at trip time, and the window's request count. When you are reconstructing an incident at 4 a.m., "opened at 03:41:07 with 62 % failures over 34 requests" answers questions that a gauge cannot.

Testing circuit breakers

An untested breaker fails in one of two embarrassing ways: it never opens (misconfigured thresholds, exceptions that don't match) or it never closes (broken half-open logic). Both are invisible until the outage that was supposed to prove its worth. The pattern is very testable, though, because the logic is a state machine — the only awkward dependency is time, so inject the clock:

```
class CircuitBreaker:
    def __init__(self, ..., clock=time.monotonic):
        self._clock = clock  # injectable for tests
```

```
import pytest

class FakeClock:
    def __init__(self):
        self.now = 1000.0
    def __call__(self):
        return self.now
    def advance(self, seconds):
        self.now += seconds

async def failing(): raise ConnectionError()
async def ok(): return "ok"

@pytest.mark.asyncio
async def test_opens_after_threshold_and_recovers():
    clock = FakeClock()
    br = CircuitBreaker(failure_ratio=0.5, min_requests=4,
                        reset_timeout=30, half_open_max=2, clock=clock)

    for _ in range(4):
        with pytest.raises(ConnectionError):
            await br.call(failing)

    # circuit is now open: calls rejected without touching the dependency
    with pytest.raises(CircuitOpenError):
        await br.call(ok)

    clock.advance(31)              # reset timeout elapses -> half-open
    assert await br.call(ok) == "ok"   # probe 1
    assert await br.call(ok) == "ok"   # probe 2 -> closes
    assert await br.call(ok) == "ok"   # back to normal

@pytest.mark.asyncio
async def test_half_open_limits_probes():
    clock = FakeClock()
    br = CircuitBreaker(failure_ratio=0.5, min_requests=2,
                        reset_timeout=10, half_open_max=1, clock=clock)
    for _ in range(2):
        with pytest.raises(ConnectionError):
```

```
            await br.call(failing)
    clock.advance(11)

    started = asyncio.Event()
    async def slow_probe():
        started.set()
        await asyncio.sleep(0.05)
        return "ok"

    task = asyncio.create_task(br.call(slow_probe))
    await started.wait()
    with pytest.raises(CircuitOpenError):   # second probe rejected
        await br.call(ok)
    assert await task == "ok"
```

The second test is the one home-grown implementations fail: it proves that while one probe is in flight, the breaker does not let the herd through. Beyond unit tests, rehearse the real thing. Point staging at a proxy that can inject failure — toxiproxy is built for exactly this — and add latency or cut the connection to the dependency while watching your dashboards. You are verifying the whole chain: timeout fires, breaker opens, fallback serves, alert triggers, breaker closes when the toxic is removed. Teams that run this drill quarterly tend to discover their fallback path has quietly rotted (an expired cache key, a feature flag nobody set in staging) *before* the outage does it for them.

Circuit breaking in the infrastructure: Envoy and Istio

If your services run behind Envoy — directly or via Istio — part of this job can move out of application code. Envoy's *outlier detection* watches per-host results and temporarily ejects bad hosts from the load-balancing pool; its *circuit breaking* config caps concurrent connections and pending requests per cluster, which is effectively a bulkhead at the proxy layer.

In Istio both are expressed in a `DestinationRule`:

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: payments
spec:
  host: payments.prod.svc.cluster.local
  trafficPolicy:
    connectionPool:            # the bulkhead
      tcp:
        maxConnections: 100
      http:
        http1MaxPendingRequests: 50
        maxRequestsPerConnection: 10
    outlierDetection:          # the breaker
      consecutive5xxErrors: 5
      interval: 10s            # how often hosts are evaluated
      baseEjectionTime: 30s    # ejection grows with repeat offenses
      maxEjectionPercent: 50   # never eject more than half the pool
```

`maxEjectionPercent` is the parameter people miss: it exists so that a bug that makes *every* host look unhealthy (say, a bad canary of the caller sending malformed requests) cannot eject the entire pool and turn a partial degradation into a total one. Note also that ejection is per-host — this shines when one bad pod in a deployment of twenty is the problem, something an application-level breaker cannot see because it aggregates all hosts behind one hostname.

So does the mesh replace application-level breakers? No — they operate on different failure domains and compose well. The mesh sees individual hosts and network-level symptoms; it can eject a sick pod in seconds without any code. The application sees *logical* operations and business context: it knows a 200 response with `"status": "DECLINED_UPSTREAM"` in the body is a failure, it knows which fallback to serve, and it can protect calls to third-party APIs that live outside the mesh entirely. The practical split: let the mesh handle host-level ejection and connection caps, keep application breakers for third-party dependencies and anywhere a meaningful fallback exists.

The same pattern in other stacks

You will meet this pattern in every ecosystem, with the same knobs under different names. A quick tour of the reference implementations, so the configs above translate:

Python — pybreaker. The long-standing implementation of the pattern from Nygard's *Release It!*, consecutive-failure based, with listeners for instrumentation and optional Redis-backed shared state built in:

```
import pybreaker

db_breaker = pybreaker.CircuitBreaker(
    fail_max=5,           # consecutive failures, not a ratio
    reset_timeout=60,
    exclude=[KeyError],   # never count these as dependency failures
)

@db_breaker
def fetch_user(user_id):
    return db.query(...)
```

Java — Resilience4j. The successor to Hystrix and the richest implementation of the sliding-window model this guide uses (current major versions run on modern JVMs; Resilience4j 3 requires Java 21). Its `COUNT_BASED` and `TIME_BASED` window types map exactly onto the count-vs-time discussion above, and `slowCallRateThreshold` adds something worth stealing: calls that *succeed but slowly* can also open the circuit.

```
CircuitBreakerConfig config = CircuitBreakerConfig.custom()
    .slidingWindowType(SlidingWindowType.TIME_BASED)
    .slidingWindowSize(30)                 // seconds
    .minimumNumberOfCalls(10)
    .failureRateThreshold(50)              // percent
    .slowCallDurationThreshold(Duration.ofSeconds(2))
    .slowCallRateThreshold(80)             // 80% slow calls also trips it
    .waitDurationInOpenState(Duration.ofSeconds(30))
```

```
    .permittedNumberOfCallsInHalfOpenState(3)
    .build();
```

.NET — Polly v8. Polly rebuilt its API around composable `ResiliencePipeline`s, which encode the layering rule from earlier — the order you add strategies *is* the wrapping order:

```
var pipeline = new ResiliencePipelineBuilder()
    .AddCircuitBreaker(new CircuitBreakerStrategyOptions
    {
        FailureRatio = 0.5,
        MinimumThroughput = 10,
        SamplingDuration = TimeSpan.FromSeconds(30),
        BreakDuration = TimeSpan.FromSeconds(30),
    })
    .AddRetry(new RetryStrategyOptions { MaxRetryAttempts = 3 })
    .AddTimeout(TimeSpan.FromSeconds(2))   // innermost
    .Build();
```

Go — gobreaker. Sony's minimal, widely used implementation. Counter-based rather than sliding-window, with a `ReadyToTrip` callback where you express your threshold logic explicitly — closer in spirit to the hand-rolled Python above than to Resilience4j.

The lesson from the tour: whatever the library, you are always answering the same five questions. What counts as failure? Over what window? With what minimum volume? How long to rest when open? How many probes to close? If you can answer those for your dependency, any of these libraries is twenty minutes of work.

Advanced patterns once the basics work

Gradual recovery instead of a cliff. Standard half-open is binary: three probes pass and boom, 100 % of traffic returns. For big services that cliff can re-kill a barely-recovered dependency — the flapping signature described earlier. The refinement is to ramp: after probes succeed, admit 10 % of traffic, then 25 %, 50 %, 100 %, stepping up every few seconds while failures stay low, and dropping back to open if they rise. A probabilistic gate a few lines long (`random.random() < admit_fraction`, with rejected calls going to the fallback) gets you most of the benefit.

Slow-call thresholds. A dependency that still returns 200s at 6-second latency can hurt you more than one that fails cleanly, because nothing "fails" and nothing trips — meanwhile every worker is parked. Counting slow-but-successful calls toward the threshold (as Resilience4j's `slowCallRateThreshold` does natively) closes this gap; if you roll your own, record `(timestamp, ok, duration)` and evaluate both ratios.

Deadline propagation. Breakers protect against sustained failure; deadlines protect each individual request. If an incoming request has 800 ms of budget left when it reaches your service, the calls you fan out should carry timeouts that fit inside that budget, not your static defaults. Without this, a request that has already been abandoned by the user keeps burning capacity through three more hops. Propagate the deadline in a header (gRPC does it natively) and clamp per-call timeouts to the remaining budget.

Per-tenant circuits. Multi-tenant APIs have a failure mode that a single breaker mishandles: one tenant's misconfigured integration produces a storm of failures and opens the circuit for everyone. Keying breakers by

`(dependency, tenant)` — with an LRU cap on the registry so cardinality cannot explode — isolates the blast radius to the tenant causing it.

Case study: the payment provider brownout

To make the numbers concrete, here is a composite of an incident pattern that shows up in many postmortems, applied to the stack in this guide. An e-commerce API (12 replicas, ~900 req/s) calls a payment provider with a p99 of 220 ms. At 14:02 the provider enters a brownout: 40 % of calls time out at the 2-second mark, the rest succeed slowly.

Without protections, the arithmetic is grim. Each replica normally holds ~2 in-flight payment calls; at brownout latencies that becomes ~50, and worker pools (75 per replica) are exhausted within 90 seconds. Health checks — served by the same workers — start timing out, the orchestrator restarts "unhealthy" pods, capacity drops further, and by 14:06 the *entire site* is down, including browsing and search, which never touch payments. Recovery requires manually scaling and a cold-cache stampede. Total: 40+ minutes of full outage from a 40 % partial failure of one dependency.

With the stack from this guide — 2 s timeout, breaker at 50 % failures over 30 s with 10-request minimum, bulkhead of 20, cached-response fallback: the timeouts convert slowness into countable failures immediately. Within the first 30-second window each replica's breaker trips (the failure ratio hits ~40-55 % depending on traffic mix). Checkout switches to the fallback: orders are accepted, queued, and charged asynchronously when the provider recovers, with an honest "payment confirmation will arrive by email" message. The bulkhead means that even during the 30-second detection window, at most 20 workers per replica were ever parked on payments — browsing never noticed. Breakers half-open every 30 s, fail their probes quietly until 14:31, then close. Total user impact: ~29 minutes of degraded checkout, zero minutes of outage, and the on-call engineer watched it happen on a dashboard instead of fighting it.

The delta between those two timelines is not cleverness — every individual mechanism is a few dozen lines — it is that the failure had somewhere designed to go.

Exposing breaker state in your health endpoint

Finally, make the circuits visible to humans and machines that are not scraping Prometheus. A FastAPI health endpoint that walks the registry:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/health/dependencies")
async def dependency_health():
    return {
        name: {
            "state": br.state.value,
            "window_requests": len(br._events),
        }
        for name, br in _REGISTRY.items()
    }
```

Two warnings from the field. Do *not* wire this into your liveness probe — a pod with an open breaker to a third-party API is degraded, not dead, and restarting it fixes nothing (the case study above shows where that road leads). If you must reflect it in readiness, do so only for dependencies that are truly load-bearing for every request. And treat the endpoint as an incident tool: the first question in a "site is slow" page is "which circuits are open?", and this answers it in one curl.

Where this leaves you

The breaker itself was never the hard part — it is 70 lines of state machine. The engineering is in the surroundings: what counts as failure, how the timeout, retry, breaker, and bulkhead layers nest, what the fallback serves, how you see state changes, and how you prove all of it works before production does the proving for you. Start with one dependency — the one that pages you most — give it the full treatment from this guide, and let the first quiet brownout make the case for the rest.

Beyond HTTP: databases, queues and gRPC

Everything so far used HTTP examples, but the pattern applies to any dependency that can fail or slow down — the only thing that changes is what "failure" looks like.

Databases. Be much more conservative here. A database is usually a hard dependency: if it is down, a fallback rarely exists, and an open breaker just converts one error message into another. Where breakers *do* earn their keep is with replicas: wrap reads to each read-replica in its own circuit, and when a replica's circuit opens, route its share of reads to the remaining replicas or to the primary. Failure criteria should be connection errors and acquisition timeouts from the pool — never SQL errors, which are your bugs, not the database's sickness. And note the overlap: a well-configured connection pool with a short checkout timeout already behaves like a bulkhead, which is one reason database breakers are less common than HTTP ones.

Message queues. Producers are a natural fit: if the broker stops accepting publishes, an open breaker lets you spill messages to a local buffer or an alternate region instead of blocking request threads. Consumers usually should *not* breaker their own broker — backing off consumption is what queues are for — but calls the consumer makes to *other* services while processing a message absolutely should be protected: a consumer hammering a dead downstream at full prefetch is one of the fastest ways to fill a dead-letter queue with thousands of retryable messages.

gRPC. The mechanics are identical, but the status codes need mapping. Count `UNAVAILABLE`, `DEADLINE_EXCEEDED` and (usually) `RESOURCE_EXHAUSTED` as failures; do not count `INVALID_ARGUMENT`, `NOT_FOUND`, `PERMISSION_DENIED` or `UNAUTHENTICATED`, which are the 4xx family in gRPC clothing. Client-side interceptors are the natural seam — one interceptor can consult the registry and protect every stub in the process, which beats wrapping call sites one by one.

Wiring a breaker into your HTTP client

Wrapping every call site by hand invites the day someone forgets. In httpx you can push the breaker down into the transport layer, so every request through the client is protected automatically — including ones written after you have stopped thinking about resilience:

```
import httpx

class BreakerTransport(httpx.AsyncBaseTransport):
    """Applies a circuit breaker to every request through this client."""

    def __init__(self, breaker: CircuitBreaker, inner=None):
        self._breaker = breaker
        self._inner = inner or httpx.AsyncHTTPTransport()

    async def handle_async_request(self, request):
        async def send():
            response = await self._inner.handle_async_request(request)
            if response.status_code >= 500:
                # surface 5xx as failures without consuming the body
                raise DependencyError(response.status_code)
            return response
        return await self._breaker.call(send)

payments_client = httpx.AsyncClient(
    base_url="https://api.payments.example.com",
    timeout=httpx.Timeout(2.0, connect=0.5),
    transport=BreakerTransport(get_breaker("payments-api")),
)
```

Two things to notice. The transport raises on 5xx so the breaker can count it — a plain httpx client considers a 500 a perfectly successful HTTP exchange, which is technically true and operationally useless. And the timeout lives on the client, inside the breaker, respecting the layering order: timeout innermost, breaker outside it. If you add retries, they go between the two — a retrying transport wrapping the inner transport, wrapped in turn by the breaker — and the whole stack is declared once, next to the client, instead of being sprinkled across the codebase.

The same idea exists in most stacks: axios interceptors or a fetch wrapper in Node, `DelegatingHandler` in .NET's `HttpClientFactory` (where Polly integrates natively), OkHttp interceptors on the JVM. The principle is what matters: resilience configuration belongs on the *client object*, declared once, not on each call.

Choosing failure criteria by dependency type

"Timeouts, connection errors and 5xx" is the right default, but real dependencies reward a second look. A short field guide:

Third-party SaaS APIs. Watch for the ones that return 200 with an error envelope in the body (`{"ok": false, "error": "internal_error"}`). If the provider does this, your failure predicate must parse the body, or your breaker will sleep through the outage. Payment providers are notorious for a related trap: a gateway timeout on a charge is *not* a clean failure — the charge may have gone through — so the breaker must coexist with idempotency keys, and the fallback must never blind-retry.

Internal services during deploys. Rolling deploys produce brief bursts of connection resets and 503s that are entirely healthy. If your window and minimum volume are tuned too tight, every deploy of a dependency opens your circuits — noise that trains everyone to ignore breaker alerts. Either widen the window past your typical rollout duration or have your platform mark deploy windows so alerts (not the breaker itself) can be suppressed.

Caches. A breaker on Redis-as-cache should trip on connection errors and be paired with a fallback that goes straight to the source of truth, plus — critically — a bulkhead on that path, because the database is about to receive the traffic the cache was absorbing. The breaker prevents the worst version of this: every request paying a 2-second Redis timeout *before* falling back, which turns a cache outage into a sitewide latency incident.

LLM and inference APIs. Long, streaming responses blur what a timeout means: time-to-first-token is the health signal you want, not total duration, which legitimately varies from two to ninety seconds with output length. Trip on connect errors, 5xx, 529 ("overloaded" — some providers use it) and slow first tokens; treat a stream that dies midway as a failure too. Rate-limit 429s, as before, belong to the retry-with-backoff layer, not the breaker.

Rolling breakers into an existing codebase

Retrofitting resilience into a service that has run for years without it deserves the same caution as any behavior change. A rollout sequence that avoids self-inflicted incidents:

Stage one: shadow mode. Deploy the breaker with tripping disabled — it records outcomes, computes ratios, emits metrics, and rejects nothing. Run it for one or two weeks of real traffic. You are learning your dependencies' actual baseline failure rates, which are always a surprise: services you thought solid turn out to run at 2 % errors permanently, and a threshold of 50 % that seemed conservative would in fact never fire.

```
class ShadowBreaker(CircuitBreaker):
    """Counts and reports, but never rejects. For calibration."""
    async def call(self, func, *args, **kwargs):
        state_before = self._state
        try:
            return await super().call(func, *args, **kwargs)
        except CircuitOpenError:
            BREAKER_CALLS.labels(self.name, "would_reject").inc()
            return await func(*args, **kwargs)  # call anyway
```

Stage two: calibrate from data. Set each circuit's threshold well above its observed baseline — a dependency living at 2 % errors can take a 30 % threshold; one that spikes to 15 % during nightly batch jobs needs 50 % or a wider window. The `would_reject` metric from shadow mode tells you exactly how often each circuit *would* have opened; tune until that number matches real incidents and nothing else.

Stage three: enforce on one dependency. Pick the one with a genuinely good fallback and a history of causing pages. Enable enforcement behind a config flag you can flip without a deploy — an environment variable read at startup is fine to begin with; a dynamic config system is better because it also gives you a per-circuit kill switch, which is the tool you will want the first time a breaker misbehaves in production.

Stage four: expand and codify. Once two or three dependencies have survived a real incident behind breakers, move the pattern into your internal client library or service template so new services inherit it by default. Teams that stop at "the important services have breakers" stay exposed through the long tail; teams that put it in the template stop thinking about it entirely — which is the goal.

One organizational note: breaker thresholds encode assumptions about *other teams'* services. When you set a 30 % threshold on a sister team's API, tell them — ideally the threshold lives next to an SLO both teams agreed

on. The alternative is the classic cross-team incident where service A's breakers opened because service B considered its own 20 % error rate "within SLO", and both teams were technically right.

Quick answers to common questions

Do I need a breaker if I already have retries and timeouts? For a dependency that fails rarely and briefly, retries and timeouts may be enough. The breaker earns its place when failure is *sustained*: it is the only one of the three that stops adding load to a struggling service, and the only one that converts a multi-second timeout into a microsecond rejection during an outage.

Should the breaker live in the caller or the callee? The caller. The callee protecting itself from overload is a different (also valuable) pattern — load shedding. They compose: your breaker keeps you from wasting your own resources on a sick dependency; its load shedder keeps it from melting when callers without breakers pile on.

One breaker for a service, or one per endpoint? Start per-service. Split per-endpoint when you observe routes with genuinely independent failure modes — a search service whose `/suggest` endpoint falls over under load while `/search` stays healthy justifies two circuits; two CRUD routes backed by the same database do not.

What is a healthy amount of open time in production? Approximately zero, almost always. Circuits should open during real dependency incidents and essentially never otherwise. If your dashboards show circuits opening weekly with no corresponding incident, your thresholds are too tight or your 4xx/5xx classification is wrong — and the alert fatigue will eventually cost you the one open circuit that mattered.

Is there any case where no breaker is the right call? Yes: hard dependencies with no possible fallback and no fan-out to protect. If your service is a thin API over one database and cannot do anything meaningful without it, a breaker mostly relabels errors. Spend the effort on the connection pool, timeouts, and capacity instead.

Designing fallbacks that actually hold up

The fallback list from earlier — cache, default value, queue, clear error — reads as if the options were interchangeable. In practice each one has load-bearing details, and a fallback that was never designed for sustained traffic is the second thing that breaks during an incident.

Stale-from-cache, done properly. The naive version caches responses with a TTL and serves whatever is left when the circuit opens. The problem: TTLs are tuned for freshness, so five minutes into an outage the cache is empty and your fallback quietly dies. The robust version separates freshness from survival — keep entries long after they go stale, and let the fallback explicitly accept staleness:

```
STALE_TTL = 86_400   # keep for a day
FRESH_TTL = 300      # consider fresh for 5 minutes

async def get_profile(user_id):
    key = f"profile:{user_id}"
    try:
        data = await breaker.call(fetch_profile, user_id)
        await redis.set(key, serialize(data, at=now()), ex=STALE_TTL)
```

```
        return data
    except (CircuitOpenError, DependencyError):
        cached = await redis.get(key)
        if cached is None:
            raise                       # no fallback available: honest error
        entry = deserialize(cached)
        entry.meta["stale"] = now() - entry.at > FRESH_TTL
        return entry                    # possibly hours old — and labeled as such
```

Note the `stale` flag: the API tells its consumers the data is degraded instead of silently lying. Frontends can render a "last updated 3 h ago" banner, and batch consumers can decide staleness is unacceptable for their use. Degraded-but-labeled beats fresh-looking-but-wrong in almost every domain; the exceptions — pricing, inventory, authorization decisions — are exactly the places where the right fallback is an error, not a guess.

Queue-and-continue, with its fine print. The case study's checkout fallback — accept the order, charge later — is the most powerful fallback there is, because it converts an outage into mere latency. Its price is paid in design work up front. The operation must be genuinely deferrable (a charge is; an auth check is not). The user messaging must change: "order confirmed" becomes "order received, confirmation to follow", and skipping that honesty generates support tickets and chargebacks. The deferred queue needs a depth limit with a decision for what happens when it fills — at some point "accept everything, sort it out later" becomes a liability, and the correct behavior degrades again, to "checkout temporarily unavailable". And the drain path needs its own protections: when the provider recovers, a queue of forty minutes of orders must trickle out through a rate limiter, not stampede out, or you will re-trigger the very brownout that built the queue. Every piece of this — the deferral, the messaging, the cap, the paced drain — has to exist *before* the incident, which is why "we'll just queue it" said during an outage is a plan that has never once worked.

Default values, scoped tightly. Serving generic recommendations when the personalization service is down is a textbook fallback because the blast radius of being wrong is near zero. The pattern generalizes only as far as that property holds: a default feature-flag value of "off" is safe; a default "approved" from a fraud check is a career event. Write the default next to the breaker, in code review, with the question "what is the worst request this default could be served to?" answered in a comment.

One more property all three share: **fallbacks need their own observability**. Count fallback executions per circuit and alert on sustained elevation. A service that has been quietly serving stale cache for six hours because a breaker never closed is an incident nobody noticed — the metrics were green, the users were getting yesterday's data, and the alert that should have fired was on a dashboard nobody built. The pair of numbers that matters is simple: how often you degrade, and for how long. If either grows, the resilience machinery has stopped being a safety net and started being a rug to sweep failures under.

Cold starts, deploys, and breaker state

Breaker state lives in process memory, which means every deploy, restart, and autoscale event resets it. This has consequences worth knowing in advance. A new instance starts with all circuits closed and an empty window — if it boots into the middle of a dependency outage, it will burn through `min_requests` real failures before its breaker opens. With aggressive autoscaling this creates a nasty loop: latency rises (because the dependency is slow), the autoscaler adds instances, each new instance re-discovers the outage from scratch and

adds its own volley of doomed requests to the dying dependency. Fleet-wide, the "protected" system can still deliver a substantial thundering herd, one cold instance at a time.

Three mitigations, in increasing order of effort. First, make sure minimum volume is not oversized — a window needing 100 requests to evaluate keeps cold instances unprotected for far longer than one needing 10. Second, if you run the Redis-shared state from earlier, new instances inherit the fleet's view immediately, which solves the problem outright and is the strongest argument for shared state in autoscaling environments. Third, a startup grace: instances that boot while a sibling's circuit is open (visible via the health endpoint or pub/sub hints) can start that circuit in half-open rather than closed, sending probes instead of full traffic until the dependency proves healthy. Whichever you choose, rehearse the combination — deploy during a simulated dependency outage in staging — because autoscaler-plus-breaker interactions are exactly the kind of emergent behavior that never shows up in unit tests.