PUIGFLOWS

Cache-Aside en Producción: TTLs, Invalidación y Cómo Evitar el Cache Stampede

Cache-Aside in Production: TTLs, Invalidation, and How to Prevent Cache Stampedes

Guía · Intermedio · ~36 min de lectura por idioma / ~36 min read per language
Edición bilingüe — Español & English · July 2026

VERSIÓN EN ESPAÑOL

Por qué una sola clave puede tumbar tu base de datos

Tu endpoint más visitado sirve 5.000 peticiones por segundo desde Redis sin despeinarse. A las 14:03, la clave expira. Durante los 800 ms que tarda la consulta en recalcularse, **todas** las peticiones fallan la caché y van directas a PostgreSQL: 4.000 consultas idénticas ejecutándose a la vez, el pool de conexiones saturado, latencias por las nubes y, en el peor caso, una caída en cascada. Eso es un *cache stampede* (también llamado *thundering herd*), y es de los incidentes más comunes —y más evitables— en sistemas con caché.

Esta guía cubre el patrón cache-aside bien hecho y las tres defensas contra la estampida que usan los sistemas en producción: lock distribuido, single-flight y expiración temprana probabilística.

El patrón cache-aside en 30 segundos

En cache-aside, la aplicación es responsable de leer y poblar la caché. Se lee la clave; si falla, se consulta la base de datos y se escribe el resultado con un TTL:

```
import json
import redis

r = redis.Redis(decode_responses=True)
TTL = 300  # 5 minutos

def get_product(product_id: int) -> dict:
    key = f"product:{product_id}"
    cached = r.get(key)
    if cached is not None:
        return json.loads(cached)

    product = fetch_product_from_db(product_id)  # consulta costosa
    r.set(key, json.dumps(product), ex=TTL)
    return product
```

Este código es correcto para tráfico moderado. El problema aparece justo cuando la caché más falta hace: bajo carga alta, en el instante en que la clave expira.

Defensa 1: lock distribuido con Redis

La idea: cuando la clave falta, solo un proceso obtiene un lock y recalcula; el resto espera a que la caché se repueble. El lock se adquiere con `SET NX EX`, que es atómico, y se libera con un script Lua que comprueba que el lock sigue siendo tuyo:

```
import time
import uuid

LOCK_TTL = 10        # segundos: mayor que el p99 de la consulta
WAIT_INTERVAL = 0.05 # 50 ms entre comprobaciones

RELEASE_SCRIPT = """
if redis.call("get", KEYS[1]) == ARGV[1] then
    return redis.call("del", KEYS[1])
end
return 0
"""
release = r.register_script(RELEASE_SCRIPT)

def get_product_with_lock(product_id: int) -> dict:
    key = f"product:{product_id}"
    lock_key = f"lock:{key}"

    cached = r.get(key)
    if cached is not None:
        return json.loads(cached)

    token = str(uuid.uuid4())
    # SET NX EX es atómico: nunca uses SETNX + EXPIRE por separado
    if r.set(lock_key, token, nx=True, ex=LOCK_TTL):
        try:
            product = fetch_product_from_db(product_id)
            r.set(key, json.dumps(product), ex=TTL)
            return product
        finally:
            release(keys=[lock_key], args=[token])

    # No conseguimos el lock: otro proceso está recalculando.
    # Esperamos con límite en vez de golpear la base de datos.
    deadline = time.monotonic() + LOCK_TTL
    while time.monotonic() < deadline:
        time.sleep(WAIT_INTERVAL)
        cached = r.get(key)
        if cached is not None:
            return json.loads(cached)
```

```
    # Último recurso: una consulta directa es mejor que un error
    return fetch_product_from_db(product_id)
```

Tres detalles que marcan la diferencia: el token único evita borrar un lock ajeno si tu proceso se pasa del `LOCK_TTL`; la espera tiene un límite (nunca esperes un lock indefinidamente); y el fallback final degrada a una consulta directa en lugar de devolver un error al usuario.

Defensa 2: single-flight dentro del proceso

El lock distribuido coordina entre instancias, pero dentro de una misma instancia con 500 peticiones concurrentes seguirías haciendo 500 GET a Redis y 500 intentos de lock. El patrón single-flight deduplica el trabajo en memoria: las peticiones concurrentes por la misma clave comparten una única promesa:

```
const inFlight = new Map<string, Promise<unknown>>();

export function singleFlight<T>(key: string, fn: () => Promise<T>): Promise<T> {
  const existing = inFlight.get(key);
  if (existing) return existing as Promise<T>;

  const p = fn().finally(() => inFlight.delete(key));
  inFlight.set(key, p);
  return p;
}

// Uso: 500 peticiones concurrentes -> una sola consulta
const product = await singleFlight(`product:${id}`, async () => {
  const cached = await redis.get(key);
  if (cached) return JSON.parse(cached);
  const fresh = await db.products.findById(id);
  await redis.set(key, JSON.stringify(fresh), "EX", 300);
  return fresh;
});
```

En Go este patrón viene de serie en `golang.org/x/sync/singleflight`; en Python se consigue igual con un diccionario de `asyncio.Task`. Es tan barato que deberías usarlo siempre, incluso combinado con las otras defensas: single-flight elimina la contención local y el lock distribuido coordina entre instancias.

Defensa 3: expiración temprana probabilística (XFetch)

Las dos defensas anteriores reaccionan cuando la clave ya expiró. XFetch (del paper *Optimal Probabilistic Cache Stampede Prevention*, de Vattani, Chierichetti y Lowenstein) evita que expire: cada lectura decide, con una probabilidad que crece al acercarse la expiración, si le toca refrescar la clave de forma anticipada. El

resultado es que una clave caliente se renueva antes de expirar, sin coordinar nada y sin que se sincronicen los refrescos:

```
import math
import random
import time

BETA = 1.0  # > 1.0 = refresco más agresivo

def get_with_early_refresh(key: str, ttl: int, recompute) -> dict:
    raw = r.get(key)
    if raw is not None:
        value = json.loads(raw)
        delta = value["_delta"]    # cuánto tardó el último recálculo
        expiry = value["_expiry"]  # timestamp de expiración lógica

        # La probabilidad de refrescar crece cerca de la expiración;
        # cuanto más cara es la consulta (delta), antes se refresca.
        if time.time() - delta * BETA * math.log(random.random()) < expiry:
            return value["data"]

    start = time.time()
    data = recompute()
    delta = time.time() - start
    value = {"data": data, "_delta": delta, "_expiry": time.time() + ttl}
    r.set(key, json.dumps(value), ex=ttl + 60)  # margen sobre el TTL lógico
    return data
```

Fíjate en que el TTL real de Redis lleva un margen extra: la expiración "lógica" se controla en el valor, de modo que aunque a una petición le toque refrescar, el resto sigue sirviendo el dato anterior mientras tanto. XFetch brilla en claves calientes con tráfico constante; para claves poco leídas no aporta nada, porque nadie las lee cerca de su expiración.

TTL con jitter: evita expiraciones sincronizadas

Si tras un despliegue o una carga masiva escribes mil claves con el mismo TTL, mil claves expirarán en el mismo segundo cinco minutos después. La solución es trivial y casi nadie la aplica:

```
ttl = 300 + random.randint(0, 60)  # 300 s + jitter aleatorio
r.set(key, payload, ex=ttl)
```

Invalidación: borra, no actualices

Cuando el dato cambia, la tentación es actualizar la caché desde el write path. No lo hagas: dos escrituras concurrentes pueden dejar en caché un valor obsoleto de forma permanente (la escritura "vieja" gana la carrera contra la "nueva"). Borrar es más seguro, porque el peor caso es un miss:

```
def update_product(product_id: int, changes: dict) -> None:
    save_to_db(product_id, changes)
    r.delete(f"product:{product_id}")  # el próximo lector recalcula
```

Si necesitas invalidar familias enteras de claves (por ejemplo, todos los listados que incluyen un producto), usa claves versionadas: guarda un contador `version:products`, inclúyelo en las claves derivadas (`listing:v42:page:3`) e incremétalo al escribir. Las claves antiguas quedan huérfanas y expiran solas por TTL.

Errores comunes

SETNX y EXPIRE por separado. Si el proceso muere entre ambas llamadas, el lock queda sin TTL y bloquea la clave para siempre. `SET key value NX EX ttl` es una sola operación atómica.

Liberar el lock con DEL a secas. Si tu recomputación tardó más que el `LOCK_TTL`, el lock ya es de otro proceso y se lo estarías borrando. Compara el token en un script Lua.

Esperar el lock sin límite. Una espera infinita convierte un incidente de base de datos en un incidente de toda la aplicación.

No cachear resultados vacíos. Si "no existe" no se cachea, cada petición por un ID inexistente va a la base de datos. Cachea el negativo con un TTL corto (10-60 s).

TTLs idénticos tras cargas masivas. Sin jitter, provocas estampidas periódicas y perfectamente sincronizadas.

Actualizar la caché al escribir. Las carreras entre escrituras concurrentes dejan datos obsoletos sin fecha de caducidad. Borra y deja que el read path recalcule.

Qué defensa usar

Single-flight: siempre. Es local, gratis y sin efectos secundarios.

Lock distribuido: cuando la recomputación es cara y tienes varias instancias. Es la defensa más robusta para misses reales.

XFetch: para claves calientes de lectura constante donde una ligera anticipación del refresco es aceptable. Elimina el miss antes de que ocurra.

Jitter en TTLs: siempre que escribas muchas claves a la vez.

En un sistema real, las defensas se apilan: jitter en todos los TTLs, single-flight en cada instancia, lock distribuido para el miss frío y, para el puñado de claves verdaderamente calientes, XFetch. Con esas cuatro

piezas, la expiración de una clave pasa de ser un evento de riesgo a un detalle de implementación que nadie nota.

Cache-aside frente a read-through, write-through y write-behind

Cache-aside no es la única forma de organizar una caché, y conviene saber qué estás descartando al elegirla. En **read-through**, la propia capa de caché sabe cargar los datos: la aplicación pide la clave y, si no está, es la caché (o una librería que la envuelve) quien consulta la base de datos. En **write-through**, cada escritura pasa por la caché y se propaga de forma síncrona a la base de datos, de modo que la caché nunca está desactualizada. En **write-behind** (o write-back), la escritura se apunta en la caché y se vuelca a la base de datos de forma asíncrona, en lotes.

¿Por qué entonces cache-aside domina en producción? Por tres razones prácticas. La primera es que **tolera fallos de la caché sin perder datos**: como la base de datos es siempre la fuente de verdad y las escrituras van directas a ella, un Redis caído degrada la latencia pero no corrompe nada. Con write-behind, en cambio, un Redis caído puede llevarse por delante escrituras que aún no se habían volcado. La segunda es que **solo cachea lo que se lee**: write-through paga el coste de escribir en caché datos que quizá nadie consulte jamás, algo especialmente doloroso en tablas con mucha escritura y poca lectura. La tercera es el **control**: al vivir en el código de la aplicación, puedes aplicar TTLs distintos por tipo de dato, cachear resultados agregados que no se corresponden con ninguna fila concreta, y decidir caso por caso qué se cachea y qué no.

El precio es conocido: la ventana de inconsistencia entre la escritura en base de datos y la invalidación, los misses fríos y toda la lógica extra que estás leyendo en esta guía. Como regla rápida: usa cache-aside por defecto; considera read-through si tu framework te lo da hecho (por ejemplo, Spring Cache o una capa DAO con anotaciones) y no necesitas TTLs a medida; reserva write-through para datos que se leen inmediatamente después de escribirse; y trata write-behind como una optimización agresiva que exige colas duraderas y tolerancia a perder alguna escritura.

Stale-while-revalidate: sirve lo viejo mientras recalculas

XFetch decide *probabilísticamente* cuándo refrescar. Hay una alternativa más simple de razonar y muy usada: separar la frescura lógica de la expiración real, el patrón **stale-while-revalidate** (SWR), también llamado soft TTL. Cada entrada guarda dos marcas de tiempo: un TTL blando (cuándo el dato pasa a considerarse "rancio") y un TTL duro (cuándo deja de ser servible). Entre ambos, la petición devuelve el dato rancio *inmediatamente* y dispara un refresco en segundo plano. El usuario nunca espera; la base de datos recibe como mucho un recálculo por ventana de frescura.

```
import asyncio
import json
import time

SOFT_TTL = 300   # a los 5 min el dato pasa a "rancio"
HARD_TTL = 3600  # a la hora deja de servirse
```

```
_refreshing: dict[str, asyncio.Task] = {}  # single-flight local

async def get_swr(key: str, recompute) -> dict:
    raw = await r.get(key)
    now = time.time()

    if raw is not None:
        entry = json.loads(raw)
        if now < entry["fresh_until"]:
            return entry["data"]              # fresco: servir y listo
        # rancio pero servible: servir ya y refrescar en background
        if key not in _refreshing:
            _refreshing[key] = asyncio.create_task(
                _refresh(key, recompute)
            )
        return entry["data"]

    # miss duro: no queda más remedio que bloquear
    return await _refresh(key, recompute)

async def _refresh(key: str, recompute) -> dict:
    try:
        data = await recompute()
        entry = {"data": data, "fresh_until": time.time() + SOFT_TTL}
        await r.set(key, json.dumps(entry), ex=HARD_TTL)
        return data
    finally:
        _refreshing.pop(key, None)
```

Fíjate en la relación entre ambos TTLs: el duro debe ser bastante mayor que el blando (aquí 12×), porque es tu colchón ante incidentes. Si la base de datos se cae durante 40 minutos, todas las claves cuyo TTL duro aún no venció siguen sirviéndose rancias, y tu web sigue en pie mientras el equipo de guardia arregla el problema de verdad. Ese comportamiento —degradar a datos viejos en lugar de a errores— es la diferencia entre un susto y un incidente con página de estado.

¿Cuándo SWR y cuándo XFetch? SWR es más fácil de explicar, de testear y de depurar, y da una garantía clara: nadie espera mientras haya *algo* servible. XFetch reparte mejor los refrescos cuando tienes muchísimas claves calientes, porque no necesita el diccionario de tareas en vuelo. En la práctica, SWR con single-flight cubre el 95 % de los casos y es lo que implementan librerías como `stale-while-revalidate-cache` en Node o el modo SWR de los CDNs (la cabecera HTTP `Cache-Control: stale-while-revalidate` es exactamente este patrón, estandarizado en el RFC 5861).

Cuando Redis se cae: fail-open, timeouts y un circuit breaker alrededor de la caché

Todo lo anterior asume que Redis responde. La pregunta incómoda es: ¿qué hace tu endpoint cuando Redis tarda 5 segundos en contestar o rechaza conexiones? Si la respuesta es "propaga la excepción", tu caché —una optimización— se ha convertido en un punto único de fallo. La regla de oro: **una caché rota nunca debe romper la petición**. A esto se le llama *fail-open*: si Redis falla, te comportas como si fuera un miss y vas a la base de datos.

```
import logging
from redis.exceptions import RedisError

logger = logging.getLogger(__name__)

# Timeouts agresivos: la caché debe responder en milisegundos.
r = redis.Redis(
    decode_responses=True,
    socket_connect_timeout=0.1,   # 100 ms para conectar
    socket_timeout=0.15,          # 150 ms por operación
    retry_on_timeout=False,       # el retry aquí lo hace TU código
)

def cache_get(key: str) -> str | None:
    try:
        return r.get(key)
    except RedisError:
        logger.warning("cache_get failed, failing open", exc_info=True)
        return None   # se trata como un miss

def cache_set(key: str, value: str, ttl: int) -> None:
    try:
        r.set(key, value, ex=ttl)
    except RedisError:
        logger.warning("cache_set failed, skipping", exc_info=True)
        # no propagar: la respuesta ya se calculó, solo no se cachea
```

Los timeouts merecen atención: los valores por defecto de la mayoría de clientes (segundos, o infinito) son absurdos para una caché. Si Redis normalmente responde en menos de 1 ms, esperar más de 150 ms ya es señal de problema grave, y bloquear el worker medio segundo por operación multiplica el daño. Con timeouts cortos y fail-open, un Redis degradado le cuesta a cada petición 150 ms extra como máximo, no una cascada de workers bloqueados.

Hay una trampa en el fail-open puro: si Redis se cae del todo, el 100 % de tu tráfico pasa a golpear la base de datos de repente —la estampida definitiva—. Por eso conviene envolver la caché en un **circuit breaker**: tras N

fallos consecutivos, deja de intentar hablar con Redis durante un tiempo (evitando pagar el timeout en cada petición) y, si tu base de datos no aguanta el tráfico sin caché, activa un modo de degradación explícito (servir respuestas simplificadas, activar rate limiting agresivo, o servir desde una caché local de emergencia). Lo importante es haber decidido *antes* del incidente cuál es el plan B, y haberlo probado apagando Redis en staging. Si nunca has hecho ese experimento, tu plan B es una hipótesis.

```
import time

class CacheBreaker:
    """Circuit breaker mínimo alrededor de la caché."""
    def __init__(self, failure_threshold: int = 5, cooldown: float = 30.0):
        self.failures = 0
        self.threshold = failure_threshold
        self.cooldown = cooldown
        self.opened_at: float | None = None

    def available(self) -> bool:
        if self.opened_at is None:
            return True
        if time.monotonic() - self.opened_at >= self.cooldown:
            self.opened_at = None      # medio abierto: probar de nuevo
            self.failures = 0
            return True
        return False                   # abierto: ni lo intentes

    def record_success(self) -> None:
        self.failures = 0

    def record_failure(self) -> None:
        self.failures += 1
        if self.failures >= self.threshold:
            self.opened_at = time.monotonic()

breaker = CacheBreaker()

def cache_get_cb(key: str) -> str | None:
    if not breaker.available():
        return None                    # circuito abierto: miss directo
    try:
        value = r.get(key)
        breaker.record_success()
        return value
    except RedisError:
```

```
        breaker.record_failure()
        return None
```

Caché de dos niveles: memoria local + Redis

Redis es rápido, pero sigue siendo un salto de red: 0,5–1 ms en el mejor caso, más la serialización. Para claves muy calientes (configuración, feature flags, el catálogo de categorías) merece la pena un primer nivel **en la memoria del propio proceso**, con Redis como segundo nivel. Una lectura L1 cuesta nanosegundos y elimina por completo el tráfico de red para los datos más pedidos.

```
# pip install cachetools
from cachetools import TTLCache

l1 = TTLCache(maxsize=10_000, ttl=30)   # 30 s: corto a propósito

def get_two_tier(key: str, recompute) -> dict:
    # Nivel 1: memoria del proceso (nanosegundos)
    if key in l1:
        return l1[key]

    # Nivel 2: Redis (menos de un milisegundo)
    cached = cache_get(key)
    if cached is not None:
        value = json.loads(cached)
        l1[key] = value
        return value

    # Nivel 3: base de datos
    value = recompute()
    cache_set(key, json.dumps(value), ttl=300)
    l1[key] = value
    return value
```

El compromiso está en la invalidación: cuando el dato cambia, borras la clave de Redis, pero cada instancia conserva su copia L1 hasta 30 segundos. Por eso el TTL del L1 debe ser corto: es tu cota máxima de inconsistencia entre instancias. Si necesitas invalidación L1 inmediata, tienes dos opciones. La artesanal: publicar los borrados en un canal pub/sub de Redis al que todas las instancias están suscritas, y que cada una borre su entrada local al recibirlo. La nativa: el **client-side caching asistido por servidor** que Redis ofrece desde la versión 6 con `CLIENT TRACKING` sobre el protocolo RESP3: el servidor recuerda qué claves ha leído cada conexión y le envía un mensaje de invalidación cuando alguien las modifica.

```
# Client-side caching con redis-py (requiere RESP3, Redis >= 7.4)
# El cliente mantiene una caché local que Redis invalida solo.
# redis-py lo expone como CacheConfig en las versiones recientes:
from redis.cache import CacheConfig

r_tracked = redis.Redis(
    protocol=3,
    decode_responses=True,
    cache_config=CacheConfig(max_size=5_000),
)

value = r_tracked.get("config:flags")   # 1ª vez: viaja a Redis
value = r_tracked.get("config:flags")   # 2ª vez: se sirve local
# Si otro cliente hace SET config:flags, Redis envía la
# invalidación por esta misma conexión y la copia local se borra.
```

Dos advertencias antes de usarlo: el tracking por defecto consume memoria en el servidor (la tabla de invalidación crece con las claves rastreadas; existe el modo broadcasting con prefijos, que no gasta memoria pero invalida más de la cuenta), y los servicios gestionados no siempre lo soportan bien tras sus proxies. Pruébalo con tu proveedor concreto antes de apostar por él; el pub/sub artesanal sigue siendo la opción más portable.

Consistencia avanzada: la carrera que "borra, no actualices" no cubre

La sección de invalidación recomendaba borrar la clave tras escribir en la base de datos. Eso elimina la carrera entre dos escrituras, pero queda una segunda carrera más sutil, entre una lectura y una escritura concurrentes:

1. La petición A (lectura) hace miss y consulta la base de datos: obtiene el valor *viejo*.
2. La petición B (escritura) actualiza la base de datos y borra la clave de la caché.
3. La petición A —que estaba pausada por el scheduler, un GC, o simple mala suerte— despierta y escribe en la caché el valor viejo que leyó en el paso 1.

Resultado: la caché guarda un dato obsoleto *sin fecha de caducidad natural*, porque la invalidación de B llegó antes que el `SET` de A. Es una carrera poco frecuente (requiere que una lectura se solape exactamente con una escritura y además se retrase), y por eso mucha gente vive años sin verla… hasta que el dato cacheado es un saldo o un precio.

Hay tres mitigaciones, de menos a más esfuerzo. La primera: **TTLs razonables**. Si toda clave expira en minutos, el daño de esta carrera está acotado; esta es la defensa real de la mayoría de sistemas. La segunda: el **doble borrado con retardo** (delayed double delete), popularizado por los equipos que operan caché a gran escala: borra la clave, escribe en la base de datos y programa un segundo borrado unos cientos de milisegundos después, que barre el valor rancio que cualquier lectura solapada haya podido escribir:

```
async def update_product_ddd(product_id: int, changes: dict) -> None:
    key = f"product:{product_id}"
    await r.delete(key)              # 1er borrado
    await save_to_db(product_id, changes)
    await asyncio.sleep(0.5)         # > p99 de (consulta + set) de una lectura
    await r.delete(key)              # 2º borrado: barre la carrera

# En producción, el sleep + delete se hace fuera de la petición:
# encola {"key": key, "delete_at": now + 0.5} en una cola o usa
# un worker con retardo, para no alargar la escritura del usuario.
```

La tercera, para quien no puede tolerar ni esa ventana: **invalidación por CDC** (change data capture). En lugar de que la aplicación borre claves, un proceso lee el log de replicación de la base de datos (Debezium sobre el WAL de PostgreSQL o el binlog de MySQL) y traduce cada cambio confirmado en borrados de caché. Como la invalidación deriva del log, es imposible olvidarse de invalidar desde algún code path (el clásico script de datos que actualiza filas sin pasar por el servicio), y el orden respeta el de los commits. Facebook describió esta arquitectura en su paper de memcache y es la norma en sistemas donde la caché protege datos críticos. El coste: infraestructura extra y una latencia de invalidación de decenas o cientos de milisegundos, así que la ventana no desaparece, solo se vuelve sistemática y monitorizable.

Hot keys: cuando el problema no es la expiración sino la popularidad

Una **hot key** es una clave que concentra una fracción desproporcionada del tráfico: la home, el producto viral, la configuración global que todas las peticiones leen. Las hot keys causan dos problemas propios. En un clúster de Redis, cada clave vive en un solo shard, así que una clave con 200.000 lecturas por segundo satura un nodo mientras los demás bostezan —añadir shards no ayuda en absoluto—. Y cuando una hot key expira, la estampida es proporcional a su popularidad.

Para **detectarlas** tienes tres herramientas. Con la política de evicción LFU activada, `redis-cli --hotkeys` recorre el keyspace y te enseña las claves con mayor frecuencia de acceso. `MONITOR` en una ventana corta (segundos, nunca en producción sostenida: cuesta hasta la mitad del rendimiento) permite muestrear qué claves aparecen más. Y lo más sostenible: instrumentar tu propia capa de caché para registrar los accesos por clave en un histograma o un top-K aproximado (un sketch tipo count-min) que exportes a tus métricas.

Para **mitigarlas**, tres técnicas que se combinan. La primera ya la vimos: caché local L1, que absorbe la inmensa mayoría de lecturas de la clave caliente en el propio proceso. La segunda: **réplicas de lectura**: dirigir los GET a réplicas del shard reparte el tráfico de lectura, al precio de leer datos con el retardo de replicación. La tercera, para el caso extremo: **partir la clave**. En vez de una clave `trending:home`, escribe N copias (`trending:home:0` … `trending:home:7`) y que cada lector elija una al azar. Multiplicas por N la memoria y las escrituras, pero divides por N el tráfico del shard más cargado:

```
import random

N_SPLITS = 8

def get_hot(key: str) -> str | None:
    return r.get(f"{key}:{random.randrange(N_SPLITS)}")

def set_hot(key: str, value: str, ttl: int) -> None:
    pipe = r.pipeline()
    for i in range(N_SPLITS):
        # jitter por copia: que no expiren las 8 a la vez
        pipe.set(f"{key}:{i}", value, ex=ttl + random.randint(0, 30))
    pipe.execute()
```

Memoria y evicción: la caché también se llena

Una caché sin límite de memoria es una bomba de relojería: Redis crecerá hasta que el sistema operativo mate el proceso. En producción, configura siempre `maxmemory` (deja margen sobre la RAM de la máquina: el copy-on-write de los snapshots RDB puede duplicar temporalmente el consumo) y una política de evicción explícita:

```
# redis.conf para un nodo dedicado a caché
maxmemory 6gb
maxmemory-policy allkeys-lfu
maxmemory-samples 10        # más muestras = evicción más precisa
lfu-decay-time 1            # el contador de frecuencia decae por minuto
```

Para una caché pura, las políticas razonables son dos: `allkeys-lru` (expulsa lo menos recientemente usado) y `allkeys-lfu` (expulsa lo menos *frecuentemente* usado, disponible desde Redis 4). LRU es el valor seguro por defecto; LFU gana cuando el patrón de acceso está sesgado —unas pocas claves muy calientes y una cola larga de claves tibias— porque un escaneo puntual de claves frías no desaloja a las estrellas. Evita las variantes `volatile-*` en una caché: solo consideran claves con TTL y, si todas lo tienen, se comportan igual pero con más trabajo; y si un día escribes claves sin TTL, Redis empezará a devolver errores de memoria en lugar de evictar.

Vigila dos métricas de `INFO stats`: `evicted_keys` (si crece de forma sostenida, tu caché es demasiado pequeña para tu working set, y estás pagando misses que creías tener cacheados) y `mem_fragmentation_ratio`. Y una higiene simple que ahorra gigabytes: no serialices campos que nunca lees en el read path (blobs, descripciones largas, campos de auditoría). Una entrada de caché no es una fila de base de datos; es la *vista* que tu endpoint necesita.

Serialización: JSON está bien, hasta que no lo está

JSON es el punto de partida correcto: legible, universal, depurable con `redis-cli GET`. Pero a partir de cierto volumen, el tamaño del payload importa dos veces: en la RAM de Redis (multiplicado por el número de claves) y en la red (multiplicado por las lecturas por segundo). Dos palancas baratas:

```
import gzip, json

def serialize(value: dict) -> bytes:
    raw = json.dumps(value, separators=(",", ":")).encode()
    # comprimir solo si compensa: gzip tiene coste de CPU
    if len(raw) > 1024:
        return b"\x1f" + gzip.compress(raw, compresslevel=1)
    return b"\x00" + raw

def deserialize(blob: bytes) -> dict:
    if blob[0:1] == b"\x1f":
        return json.loads(gzip.decompress(blob[1:]))
    return json.loads(blob[1:])
```

El byte de prefijo distingue entradas comprimidas de las que no lo están, lo que permite migrar gradualmente y comprimir solo payloads grandes (comprimir 200 bytes es contraproducente). Con documentos JSON típicos, gzip a nivel 1 reduce el tamaño 5–10× por un coste de CPU de microsegundos. Si además el volumen de operaciones es alto, considera **msgpack** en lugar de JSON: serialización binaria 2–4× más compacta y más rápida de parsear, al precio de perder la legibilidad directa en `redis-cli`. La combinación msgpack + compresión condicional es el estándar de facto en cachés de alto tráfico.

Lecturas por lotes: MGET y pipelines

Un endpoint que renderiza una lista de 30 productos no debe hacer 30 GET secuenciales: son 30 round-trips, y el tiempo de red domina por completo. Redis ofrece dos herramientas. `MGET` lee N claves en una sola operación; el **pipeline** agrupa operaciones heterogéneas en un solo viaje:

```
def get_products_batch(ids: list[int]) -> dict[int, dict]:
    keys = [f"product:{i}" for i in ids]
    values = r.mget(keys)             # 1 round-trip para N claves

    found, missing = {}, []
    for pid, raw in zip(ids, values):
        if raw is not None:
            found[pid] = json.loads(raw)
        else:
            missing.append(pid)
```

```
    if missing:
        # una sola consulta con WHERE id IN (...), nunca un bucle
        rows = fetch_products_from_db(missing)
        pipe = r.pipeline()
        for pid, row in rows.items():
            pipe.set(f"product:{pid}", json.dumps(row),
                     ex=300 + random.randint(0, 60))
        pipe.execute()                # 1 round-trip para los sets
        found.update(rows)

    return found
```

La estructura es siempre la misma: un MGET, repartir hits y misses, *una* consulta por lotes a la base de datos para los misses y un pipeline para reescribirlos. Un detalle para clústeres: MGET exige que todas las claves caigan en el mismo slot; si usas Redis Cluster, agrupa con hash tags (`product:{catalog}:123`) o deja que el cliente parta el MGET por slot (los buenos clientes lo hacen solos).

Observabilidad: si no mides el hit ratio, no tienes caché, tienes fe

Tres números cuentan la historia completa de una caché: el **hit ratio** (qué fracción de lecturas se sirve de caché), la **latencia de recomputación** (cuánto cuesta cada miss) y las **estampidas evitadas** (cuántas veces las defensas hicieron su trabajo). Instruméntalos en tu capa de caché, no confíes solo en las métricas del servidor Redis —el hit ratio global de Redis mezcla todos tus tipos de datos y esconde el que va mal—:

```
from prometheus_client import Counter, Histogram

CACHE_OPS = Counter(
    "cache_ops_total",
    "Operaciones de caché por resultado",
    ["cache_name", "result"],   # result: hit | miss | stale | error
)
RECOMPUTE_SECONDS = Histogram(
    "cache_recompute_seconds",
    "Duración de la recomputación tras un miss",
    ["cache_name"],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1, 2.5, 5],
)
STAMPEDE_WAITS = Counter(
    "cache_stampede_waits_total",
    "Peticiones que esperaron el lock en vez de recomputar",
    ["cache_name"],
)
```

```
def get_product_observed(product_id: int) -> dict:
    key = f"product:{product_id}"
    cached = cache_get(key)
    if cached is not None:
        CACHE_OPS.labels("products", "hit").inc()
        return json.loads(cached)

    CACHE_OPS.labels("products", "miss").inc()
    with RECOMPUTE_SECONDS.labels("products").time():
        product = fetch_product_from_db(product_id)
    cache_set(key, json.dumps(product), ttl=300)
    return product
```

Con eso, el hit ratio por tipo de dato es una consulta PromQL: `sum(rate(cache_ops_total{result="hit"}[5m])) by (cache_name) / sum(rate(cache_ops_total{result=~"hit|miss"}[5m])) by (cache_name)`. Alerta sobre *cambios*, no sobre valores absolutos: un hit ratio del 60 % puede ser excelente para un tipo de dato y desastroso para otro; lo que siempre es mala señal es que caiga 20 puntos tras un despliegue (¿alguien cambió el formato de las claves y ahora todo es miss?) o que `cache_ops_total{result="error"}` deje de ser cero. Añade también un panel con `evicted_keys` y la memoria usada del propio Redis: un hit ratio que se degrada lentamente mientras las evicciones suben es la firma inconfundible de una caché que se quedó pequeña.

Cache warming: no salgas a producción con la caché vacía

Cada despliegue con caché local, cada failover de Redis y cada flush accidental te deja con una caché fría: el primer minuto de tráfico golpea la base de datos con la tasa de misses al máximo. Si tu base de datos aguanta el tráfico sin caché, no necesitas nada especial (y deberías saberlo con certeza, no suponerlo). Si no lo aguanta, precalienta: antes de enrutar tráfico a la versión nueva, un script recorre las N claves más populares del día anterior (esa lista sale de tus métricas o de un `ZSET` de popularidad) y las recomputa de forma controlada, con concurrencia limitada:

```
async def warm_cache(top_keys: list[str], concurrency: int = 8) -> None:
    sem = asyncio.Semaphore(concurrency)   # no estampides tu propia BD

    async def warm_one(key: str) -> None:
        async with sem:
            if await r.exists(key):
                return                     # ya caliente
            entity_id = int(key.split(":")[1])
            data = await fetch_product_from_db_async(entity_id)
            await r.set(key, json.dumps(data),
                        ex=300 + random.randint(0, 60))
```

```
    await asyncio.gather(*(warm_one(k) for k in top_keys))
```

El semáforo es la parte importante: un warmer sin límite de concurrencia es, literalmente, una estampida autoinfligida. Y no intentes precalentarlo todo: con el 5 % de claves más populares suele bastar para cubrir la mayoría del tráfico; la cola larga se calienta sola.

Diseño de claves: nombres, versiones de esquema y multi-tenancy

El diseño de las claves parece un detalle cosmético hasta que provoca tu primer incidente. Tres decisiones que conviene tomar el primer día y no improvisar después.

Convención de nombres. Adopta un formato jerárquico con separador fijo y documéntalo: `app:tipo:identificador`, por ejemplo `shop:product:12345` o `shop:listing:electronics:page:3`. La jerarquía hace tres cosas por ti: agrupa visualmente en cualquier herramienta de inspección, permite invalidar familias con claves versionadas por prefijo, y evita colisiones cuando dos servicios comparten el mismo Redis (algo que no deberías hacer con cachés grandes, pero que ocurre en todas partes). Mantén las claves cortas pero legibles —cada byte de clave se multiplica por millones de entradas y viaja en cada operación— y nunca metas en ellas datos sin sanear: una clave construida con input del usuario sin normalizar (`search:{query}` con espacios, mayúsculas o caracteres de control) fragmenta la caché en variantes duplicadas y, en el peor caso, permite a un atacante inflar tu memoria a base de misses artificiales.

Versión de esquema. El bug más traicionero de una caché no lo causa Redis, sino un despliegue: la versión N+1 de tu código lee una entrada serializada por la versión N con otra forma (un campo renombrado, un tipo cambiado) y explota —o peor, no explota y procesa datos mal interpretados—. La solución cuesta un prefijo: incluye la versión del esquema serializado en la clave, y cámbiala cada vez que cambies la forma del dato cacheado:

```
SCHEMA_VERSION = "v3"   # súbelo al cambiar la forma del dato

def product_key(product_id: int) -> str:
    return f"shop:product:{SCHEMA_VERSION}:{product_id}"

# El despliegue de v4 simplemente deja de leer las claves v3:
# misses fríos controlados en vez de errores de deserialización.
# Las claves v3 huérfanas expiran solas por TTL.
```

El efecto de subir la versión es una caché fría instantánea para ese tipo de dato, así que combínalo con lo que ya sabes: despliega en horario de valle o precalienta antes de enrutar tráfico. Es un coste planificado y visible, infinitamente preferible al error de deserialización aleatorio que aparece solo para las claves escritas por la versión anterior y desaparece al expirar, dejándote un stack trace intermitente imposible de reproducir.

Multi-tenancy y datos por usuario. Si tu aplicación sirve a varios tenants o cachea datos que dependen del usuario, el tenant o el usuario *tienen que estar en la clave*: `shop:{tenant}:product:v3:123`. El fallo

contrario es uno de los incidentes de seguridad más comunes relacionados con cachés: un endpoint que cachea la respuesta bajo `profile:latest` en vez de `profile:{user_id}:latest` y sirve los datos del usuario A al usuario B durante el TTL. La revisión sistemática es sencilla de enunciar: por cada clave, pregunta "¿de qué depende exactamente este valor?" — y todo lo que aparezca en la respuesta (usuario, tenant, idioma, moneda, flags de feature activos) debe aparecer en la clave. Si la lista de dependencias se hace larga, es señal de que ese dato no es buen candidato a caché compartida: cachéalo más abajo (la consulta a base de datos que no depende del usuario) en lugar de la respuesta completa que sí depende.

Con los tenants en la clave, además, las operaciones administrativas se vuelven triviales: invalidar todo lo de un tenant tras una migración es incrementar su versión (`tenant:acme:version` como parte del prefijo, el truco de claves versionadas que ya vimos), no un `SCAN` peligroso sobre todo el keyspace. Y en Redis Cluster, poner el tenant en un hash tag (`shop:{acme}:...`) garantiza que sus claves conviven en el mismo slot, con la contrapartida de que un tenant enorme puede desequilibrar un shard: elige el hash tag según tu distribución real de tamaños, no por estética.

Una última recomendación transversal: trata estas convenciones como código, no como tradición oral. Un módulo `cache_keys.py` (o su equivalente TypeScript) con funciones constructoras de claves —nunca f-strings sueltos repartidos por el código— convierte todas las reglas anteriores en algo que el type checker y la revisión de código pueden vigilar, y hace que subir una versión de esquema sea un cambio de una línea en un solo sitio.

Elegir el motor: Redis, Valkey o Memcached

Todos los patrones de esta guía hablan de "Redis", pero en 2026 esa palabra esconde una decisión con tres opciones. Un poco de contexto: en marzo de 2024 Redis abandonó la licencia BSD por licencias source-available, y la comunidad respondió el mismo mes con **Valkey**, un fork alojado en la Linux Foundation y respaldado por AWS, Google y Oracle, que arrancó de Redis 7.2.4 y mantiene licencia BSD. Redis contraatacó en 2025: desde la versión 8.0 vuelve a ofrecer una licencia open source (AGPLv3, en tri-licencia con SSPL y RSAL). El resultado práctico es que hoy conviven dos proyectos maduros y compatibles a nivel de protocolo, con cadencias de release propias.

¿Qué significa para tu caché? Menos de lo que parece y más de lo que admiten los debates de Internet. **Todo lo que has leído en esta guía funciona igual en ambos**: cache-aside, SET NX EX, scripts Lua, pipelines, políticas de evicción LFU, client-side caching. Si usas un servicio gestionado, la decisión probablemente ya está tomada por tu proveedor: ElastiCache y MemoryDB de AWS usan Valkey por defecto; otros proveedores ofrecen ambos. Si te autoalojas, los criterios reales son la licencia (si AGPL encaja en tu política de dependencias) y qué características recientes necesitas de cada línea de desarrollo, no el rendimiento: para el caso de uso caché, ambos están sobrados.

¿Y **Memcached**? Sigue vivo y sigue teniendo un nicho legítimo: es multihilo (escala verticalmente en un solo nodo mejor que un proceso Redis), hace una sola cosa —caché LRU con TTL— y la hace con una simplicidad operativa difícil de igualar. Si tu necesidad es exactamente "cachear bytes con TTL detrás de un load balancer" y nada más, Memcached es una respuesta perfectamente seria. Lo descartas en cuanto necesitas cualquier otra pieza de esta guía que dependa de estructuras o atomicidad del lado del servidor: locks con SET NX EX, scripts Lua, pub/sub para invalidación L1, ZSETs de popularidad para el warmer. En cache-aside moderno, esas piezas

aparecen pronto, y esa es la razón silenciosa por la que Redis/Valkey se convirtió en el estándar: no porque cachee mejor, sino porque la caja de herramientas alrededor de la caché es más completa.

El consejo accionable: elige el motor por restricciones (proveedor, licencia, operación) y no por benchmarks de foro; escribe tu capa de caché contra una interfaz propia (las funciones `cache_get` / `cache_set` con fail-open que ya construimos) para que cambiar de motor sea un cambio de configuración; y si hoy estás en Redis 7.x autoalojado sin contrato de soporte, planifica la migración a Redis 8+ o Valkey con calma — ambas son actualizaciones directas desde 7.2, y quedarse en una versión sin parches de seguridad sí es una decisión mala en todos los escenarios.

Cómo testear una caché (y sus defensas contra estampidas)

El código de caché tiene fama de difícil de testear porque mezcla red, tiempo y concurrencia. Se domestica con dos herramientas. Para tests unitarios rápidos, **fakeredis** implementa la API de Redis en memoria, incluidos TTLs y scripts Lua, sin levantar ningún proceso. Para tests de integración que deben ser fieles (pipelines, cluster, comportamiento de expiración real), **Testcontainers** arranca un Redis efímero en Docker por sesión de test.

Lo primero que hay que testear no son los hits —esos funcionan solos— sino los comportamientos que solo aparecen bajo presión. Este test verifica la propiedad central de toda la guía: *N lecturas concurrentes de una clave fría deben producir una sola recomputación*:

```
import asyncio
import pytest

@pytest.mark.asyncio
async def test_stampede_produces_single_recompute(cache):
    calls = 0

    async def slow_recompute():
        nonlocal calls
        calls += 1
        await asyncio.sleep(0.2)      # simula una consulta lenta
        return {"id": 1, "name": "hot product"}

    # 100 peticiones concurrentes contra una clave fría
    results = await asyncio.gather(*[
        cache.get("product:1", slow_recompute) for _ in range(100)
    ])

    assert calls == 1                          # una sola recomputación
    assert all(r == results[0] for r in results)

@pytest.mark.asyncio
```

```
async def test_cache_failure_fails_open(cache, broken_redis):
    # con Redis caído, la petición degrada a la BD, no a un error
    result = await cache.get("product:1", lambda: {"id": 1})
    assert result == {"id": 1}

@pytest.mark.asyncio
async def test_negative_result_is_cached(cache, db_spy):
    for _ in range(10):
        assert await cache.get("product:999", db_spy.fetch_missing) is None
    assert db_spy.calls == 1          # "no existe" también se cachea
```

Los otros dos tests cubren los modos de fallo que de verdad despiertan a alguien de madrugada: que un Redis roto no rompa las peticiones (fail-open) y que los resultados negativos se cacheen. Completa la batería con un test de expiración usando un reloj falso o TTLs de milisegundos, y un test de invalidación (escribir → borrar → siguiente lectura recomputa). Si además quieres validar el comportamiento bajo carga real, una prueba con `k6` o `locust` contra staging, matando la clave caliente a mitad de la prueba, te enseña la estampida —o su ausencia— en las gráficas.

Implementación completa en TypeScript

Para cerrar la parte de código, una clase que apila todas las defensas de esta guía —single-flight, lock distribuido, TTL con jitter, negative caching, fail-open y métricas— detrás de una única llamada `getOrCompute()`. Es el equivalente TypeScript de lo que hemos construido por piezas en Python, listo para adaptar:

```
import Redis from "ioredis";
import { randomUUID } from "crypto";

interface CacheOptions {
  ttlSeconds: number;
  jitterSeconds?: number;      // por defecto 20 % del TTL
  lockTtlSeconds?: number;     // por defecto 10 s
  negativeTtlSeconds?: number; // por defecto 30 s
}

const RELEASE_LUA = `
if redis.call("get", KEYS[1]) == ARGV[1] then
  return redis.call("del", KEYS[1])
end
return 0`;

const NEGATIVE = "__NULL__";   // centinela para "no existe"
```

```
export class ProductionCache {
  private inFlight = new Map<string, Promise<unknown>>();

  constructor(
    private redis: Redis,
    private onMetric: (event: string) => void = () => {},
  ) {}

  async getOrCompute<T>(
    key: string,
    compute: () => Promise<T | null>,
    opts: CacheOptions,
  ): Promise<T | null> {
    // 1) single-flight: dedupe local por clave
    const existing = this.inFlight.get(key);
    if (existing) return existing as Promise<T | null>;

    const p = this.load(key, compute, opts)
      .finally(() => this.inFlight.delete(key));
    this.inFlight.set(key, p);
    return p;
  }

  private async load<T>(
    key: string,
    compute: () => Promise<T | null>,
    opts: CacheOptions,
  ): Promise<T | null> {
    // 2) lectura con fail-open
    const cached = await this.safeGet(key);
    if (cached !== undefined) {
      if (cached === NEGATIVE) { this.onMetric("negative_hit"); return null; }
      if (cached !== null) { this.onMetric("hit"); return JSON.parse(cached) as T; }
    }
    this.onMetric("miss");

    // 3) lock distribuido: solo un proceso recomputa
    const lockKey = `lock:${key}`;
    const token = randomUUID();
    const lockTtl = opts.lockTtlSeconds ?? 10;
    const acquired = await this.redis
      .set(lockKey, token, "EX", lockTtl, "NX")
      .catch(() => null);            // Redis roto: recomputa sin lock
```

```
  if (acquired) {
    try {
      const value = await compute();
      await this.safeSet(key, value, opts);
      return value;
    } finally {
      await this.redis
        .eval(RELEASE_LUA, 1, lockKey, token)
        .catch(() => {});
    }
  }

  // 4) no hay lock: espera acotada a que otro repueble
  const deadline = Date.now() + lockTtl * 1000;
  while (Date.now() < deadline) {
    await new Promise((res) => setTimeout(res, 50));
    const repopulated = await this.safeGet(key);
    if (repopulated === NEGATIVE) return null;
    if (repopulated) {
      this.onMetric("stampede_wait_hit");
      return JSON.parse(repopulated) as T;
    }
  }
  this.onMetric("stampede_wait_timeout");
  return compute();                // último recurso: consulta directa

}

private async safeGet(key: string): Promise<string | null | undefined> {
  try { return await this.redis.get(key); }
  catch { this.onMetric("error"); return undefined; }
}

private async safeSet<T>(
  key: string, value: T | null, opts: CacheOptions,
): Promise<void> {
  const jitter = opts.jitterSeconds ?? Math.ceil(opts.ttlSeconds * 0.2);
  const ttl = opts.ttlSeconds + Math.floor(Math.random() * jitter);
  try {
    if (value === null) {
      // 5) negative caching con TTL corto
      await this.redis.set(key, NEGATIVE, "EX", opts.negativeTtlSeconds ?? 30);
```

```
      } else {
        await this.redis.set(key, JSON.stringify(value), "EX", ttl);
      }
    } catch { this.onMetric("error"); }
  }
}

// Uso
const cache = new ProductionCache(new Redis(), (e) => metrics.inc(e));

const product = await cache.getOrCompute(
  `product:${id}`,
  () => db.products.findById(id),   // devuelve null si no existe
  { ttlSeconds: 300 },
);
```

Merece la pena leerla con calma una vez: cada bloque numerado corresponde a una sección de esta guía, y el orden de las defensas no es casual —el single-flight envuelve todo (es lo más barato), la lectura va antes que el lock (el caso feliz no debe pagar coordinación) y cada llamada a Redis está envuelta en fail-open (la caché nunca rompe la petición).

Caso práctico: dimensionar la caché de un catálogo

Aterricemos los números con un servicio concreto: un catálogo de e-commerce con 2 millones de productos, 8.000 peticiones por segundo en pico, y una consulta de producto que cuesta 25 ms de media en PostgreSQL (con joins a precios, stock e imágenes).

¿Cuánta memoria? La vista de producto serializada ocupa unos 2 KB en JSON compacto (600 bytes con msgpack + gzip). Cachear el catálogo completo costaría 4 GB en JSON, pero no hace falta: el análisis de accesos muestra que el 10 % de productos concentra el 92 % de las lecturas. Con `maxmemory 1gb` y `allkeys-lfu`, la caché converge sola hacia ese working set: LFU expulsa la cola larga y protege las estrellas. Regla general: dimensiona para el working set medido, no para el dataset completo, y deja que la política de evicción haga su trabajo.

¿Qué TTL? Los precios cambian con un proceso batch cada hora y ediciones manuales ocasionales. TTL base de 300 s + jitter de 60: la inconsistencia máxima por TTL es de 6 minutos, aceptable para fichas de producto; las ediciones manuales borran la clave al guardar, así que se reflejan en segundos. Para el carrito y el stock en tiempo real: *no se cachean*. Decidir qué NO entra en la caché es la mitad del diseño.

¿Aguanta la base de datos un flush? 8.000 rps × 25 ms = 200 conexiones ocupadas solo en productos si la caché desaparece: no aguanta. Conclusión operativa: la caché es *crítica*, así que hay warmer para despliegues (top 50.000 claves, 40 segundos con concurrencia 16), SWR con TTL duro de una hora como colchón ante incidentes de base de datos, y una alerta si el hit ratio de productos baja del 85 %. Con todo en su sitio, PostgreSQL ve unas 300 consultas por segundo (las 8.000 menos el ~96 % de hit ratio efectivo), el p99 del

endpoint queda en 4 ms, y la expiración de cualquier clave —incluida la del producto viral del día— es un no-evento.

Checklist de producción

Antes de dar por terminada una capa de caché, repasa: cada TTL lleva jitter; el single-flight envuelve toda lectura; las claves caras tienen lock distribuido o SWR; los resultados negativos se cachean con TTL corto; la invalidación borra (y si hay carreras que duelan, doble borrado o CDC); los timeouts de Redis son de milisegundos y todo fallo de caché degrada a miss; `maxmemory` y la política de evicción están configuradas explícitamente; el hit ratio, los errores y las evicciones tienen dashboard y alerta; existe un test que demuestra que 100 lecturas concurrentes producen una recomputación; y alguien ha apagado Redis en staging para ver qué pasa. Si puedes marcar las diez casillas, tu caché ya no es la parte frágil del sistema: es la que aguanta cuando todo lo demás tiembla.

Cache-Aside in Production: TTLs, Invalidation, and How to Prevent Cache Stampedes

ENGLISH VERSION

Why a single key can take down your database

Your busiest endpoint serves 5,000 requests per second from Redis without breaking a sweat. At 2:03 pm, the key expires. During the 800 ms the query takes to recompute, **every** request misses the cache and goes straight to PostgreSQL: 4,000 identical queries running at once, a saturated connection pool, latencies through the roof and, in the worst case, a cascading failure. That is a *cache stampede* (also known as a *thundering herd*), and it is one of the most common — and most avoidable — incidents in systems that rely on caching.

This guide covers the cache-aside pattern done right, plus the three stampede defenses production systems actually use: distributed locks, single-flight, and probabilistic early expiration.

The cache-aside pattern in 30 seconds

In cache-aside, the application is responsible for reading and populating the cache. You read the key; on a miss, you query the database and write the result back with a TTL:

```
import json
import redis

r = redis.Redis(decode_responses=True)
TTL = 300  # 5 minutes

def get_product(product_id: int) -> dict:
    key = f"product:{product_id}"
    cached = r.get(key)
    if cached is not None:
        return json.loads(cached)

    product = fetch_product_from_db(product_id)  # expensive query
    r.set(key, json.dumps(product), ex=TTL)
    return product
```

This code is perfectly fine at moderate traffic. The problem shows up exactly when the cache matters most: under heavy load, at the instant the key expires.

Defense 1: a distributed lock with Redis

The idea: when the key is missing, only one process acquires a lock and recomputes; everyone else waits for the cache to be repopulated. The lock is acquired with `SET NX EX`, which is atomic, and released with a Lua script that checks the lock is still yours:

```
import time
import uuid

LOCK_TTL = 10        # seconds: greater than the query's p99
WAIT_INTERVAL = 0.05 # 50 ms between checks

RELEASE_SCRIPT = """
if redis.call("get", KEYS[1]) == ARGV[1] then
    return redis.call("del", KEYS[1])
end
return 0
"""
release = r.register_script(RELEASE_SCRIPT)

def get_product_with_lock(product_id: int) -> dict:
    key = f"product:{product_id}"
    lock_key = f"lock:{key}"

    cached = r.get(key)
    if cached is not None:
        return json.loads(cached)

    token = str(uuid.uuid4())
    # SET NX EX is atomic: never use separate SETNX + EXPIRE calls
    if r.set(lock_key, token, nx=True, ex=LOCK_TTL):
        try:
            product = fetch_product_from_db(product_id)
            r.set(key, json.dumps(product), ex=TTL)
            return product
        finally:
            release(keys=[lock_key], args=[token])

    # We didn't get the lock: another process is recomputing.
    # Wait with a deadline instead of hammering the database.
    deadline = time.monotonic() + LOCK_TTL
    while time.monotonic() < deadline:
        time.sleep(WAIT_INTERVAL)
```

```
        cached = r.get(key)
        if cached is not None:
            return json.loads(cached)

    # Last resort: one direct query beats returning an error
    return fetch_product_from_db(product_id)
```

Three details make all the difference: the unique token prevents you from deleting someone else's lock if your process overruns `LOCK_TTL`; the wait has a deadline (never wait on a lock indefinitely); and the final fallback degrades to a direct query instead of surfacing an error to the user.

Defense 2: single-flight inside the process

The distributed lock coordinates across instances, but within a single instance handling 500 concurrent requests you would still issue 500 GETs to Redis and 500 lock attempts. The single-flight pattern deduplicates the work in memory: concurrent requests for the same key share one promise:

```
const inFlight = new Map<string, Promise<unknown>>();

export function singleFlight<T>(key: string, fn: () => Promise<T>): Promise<T> {
  const existing = inFlight.get(key);
  if (existing) return existing as Promise<T>;

  const p = fn().finally(() => inFlight.delete(key));
  inFlight.set(key, p);
  return p;
}

// Usage: 500 concurrent requests -> a single query
const product = await singleFlight(`product:${id}`, async () => {
  const cached = await redis.get(key);
  if (cached) return JSON.parse(cached);
  const fresh = await db.products.findById(id);
  await redis.set(key, JSON.stringify(fresh), "EX", 300);
  return fresh;
});
```

Go ships this pattern in the standard ecosystem as `golang.org/x/sync/singleflight`; in Python you get the same effect with a dictionary of `asyncio.Task`s. It is so cheap you should always use it, even combined with the other defenses: single-flight removes local contention while the distributed lock coordinates across instances.

Defense 3: probabilistic early expiration (XFetch)

The two defenses above react after the key has already expired. XFetch (from the paper *Optimal Probabilistic Cache Stampede Prevention* by Vattani, Chierichetti, and Lowenstein) prevents it from expiring at all: on every read, with a probability that grows as expiration approaches, the caller decides whether it should refresh the key early. The result is that a hot key gets renewed before it expires, with no coordination and no synchronized refreshes:

```
import math
import random
import time

BETA = 1.0  # > 1.0 = more aggressive refreshing

def get_with_early_refresh(key: str, ttl: int, recompute) -> dict:
    raw = r.get(key)
    if raw is not None:
        value = json.loads(raw)
        delta = value["_delta"]    # how long the last recompute took
        expiry = value["_expiry"]  # logical expiration timestamp

        # Refresh probability grows near expiration; the more
        # expensive the query (delta), the earlier it refreshes.
        if time.time() - delta * BETA * math.log(random.random()) < expiry:
            return value["data"]

    start = time.time()
    data = recompute()
    delta = time.time() - start
    value = {"data": data, "_delta": delta, "_expiry": time.time() + ttl}
    r.set(key, json.dumps(value), ex=ttl + 60)  # margin over the logical TTL
    return data
```

Note that the real Redis TTL carries extra margin: the "logical" expiration lives inside the value, so even when one request draws the short straw and refreshes, everyone else keeps serving the previous data in the meantime. XFetch shines on hot keys with constant traffic; for rarely-read keys it adds nothing, because nobody reads them close to expiration.

Jittered TTLs: avoid synchronized expiration

If you write a thousand keys with the same TTL after a deploy or a bulk load, a thousand keys will expire in the same second five minutes later. The fix is trivial and almost nobody applies it:

```
ttl = 300 + random.randint(0, 60)  # 300 s + random jitter
r.set(key, payload, ex=ttl)
```

Invalidation: delete, don't update

When data changes, the temptation is to update the cache from the write path. Don't: two concurrent writes can permanently leave a stale value in the cache (the "old" write wins the race against the "new" one). Deleting is safer, because the worst case is just a miss:

```
def update_product(product_id: int, changes: dict) -> None:
    save_to_db(product_id, changes)
    r.delete(f"product:{product_id}")  # next reader recomputes
```

If you need to invalidate whole families of keys (say, every listing that includes a product), use versioned keys: store a `version:products` counter, embed it in derived keys (`listing:v42:page:3`), and increment it on writes. Old keys become orphans and quietly expire via TTL.

Common pitfalls

Separate SETNX and EXPIRE calls. If the process dies between the two, the lock has no TTL and blocks the key forever. `SET key value NX EX ttl` is a single atomic operation.

Releasing the lock with a bare DEL. If your recompute took longer than `LOCK_TTL`, the lock now belongs to another process and you would be deleting theirs. Compare the token in a Lua script.

Waiting on the lock without a deadline. An unbounded wait turns a database incident into a whole-application incident.

Not caching empty results. If "not found" is never cached, every request for a nonexistent ID hits the database. Cache the negative with a short TTL (10-60 s).

Identical TTLs after bulk loads. Without jitter, you schedule periodic, perfectly synchronized stampedes.

Updating the cache on writes. Races between concurrent writes leave stale data with no expiration date. Delete, and let the read path recompute.

Which defense to use

Single-flight: always. It is local, free, and has no side effects.

Distributed lock: when recomputation is expensive and you run multiple instances. The most robust defense for genuine misses.

XFetch: for hot, constantly-read keys where slightly early refreshes are acceptable. It eliminates the miss before it happens.

Jittered TTLs: whenever you write many keys at once.

In a real system, the defenses stack: jitter on every TTL, single-flight in every instance, a distributed lock for the cold miss and, for the handful of truly hot keys, XFetch. With those four pieces in place, a key expiring goes from being a risk event to an implementation detail nobody notices.

Cache-aside vs. read-through, write-through, and write-behind

Cache-aside is not the only way to organize a cache, and it pays to know what you are turning down when you pick it. In **read-through**, the caching layer itself knows how to load data: the application asks for the key and, on a miss, it is the cache (or a library wrapping it) that queries the database. In **write-through**, every write goes through the cache and is synchronously propagated to the database, so the cache is never out of date. In **write-behind** (or write-back), the write is recorded in the cache and flushed to the database asynchronously, in batches.

So why does cache-aside dominate in production? Three practical reasons. First, it **tolerates cache failures without losing data**: since the database is always the source of truth and writes go straight to it, a dead Redis degrades latency but corrupts nothing. With write-behind, by contrast, a dead Redis can take down writes that had not been flushed yet. Second, it **only caches what is actually read**: write-through pays the cost of caching data nobody may ever query, which is especially painful on write-heavy, read-light tables. Third, **control**: because it lives in application code, you can apply different TTLs per data type, cache aggregated results that do not map to any single row, and decide case by case what gets cached and what does not.

The price is well known: the inconsistency window between the database write and the invalidation, cold misses, and all the extra logic you are reading about in this guide. As a quick rule: use cache-aside by default; consider read-through if your framework hands it to you (say, Spring Cache or a DAO layer with annotations) and you do not need custom TTLs; reserve write-through for data that is read immediately after being written; and treat write-behind as an aggressive optimization that demands durable queues and tolerance for losing the occasional write.

Stale-while-revalidate: serve the old while you recompute

XFetch decides *probabilistically* when to refresh. There is an alternative that is easier to reason about and widely used: separating logical freshness from actual expiration — the **stale-while-revalidate** (SWR) pattern, also known as soft TTL. Each entry stores two timestamps: a soft TTL (when the data starts being considered "stale") and a hard TTL (when it stops being servable). Between the two, a request returns the stale data *immediately* and kicks off a background refresh. The user never waits; the database receives at most one recompute per freshness window.

```
import asyncio
import json
import time

SOFT_TTL = 300   # after 5 min the data becomes "stale"
```

```
HARD_TTL = 3600  # after an hour it stops being served

_refreshing: dict[str, asyncio.Task] = {}  # local single-flight

async def get_swr(key: str, recompute) -> dict:
    raw = await r.get(key)
    now = time.time()

    if raw is not None:
        entry = json.loads(raw)
        if now < entry["fresh_until"]:
            return entry["data"]              # fresh: serve and done
        # stale but servable: serve now, refresh in the background
        if key not in _refreshing:
            _refreshing[key] = asyncio.create_task(
                _refresh(key, recompute)
            )
        return entry["data"]

    # hard miss: no choice but to block
    return await _refresh(key, recompute)

async def _refresh(key: str, recompute) -> dict:
    try:
        data = await recompute()
        entry = {"data": data, "fresh_until": time.time() + SOFT_TTL}
        await r.set(key, json.dumps(entry), ex=HARD_TTL)
        return data
    finally:
        _refreshing.pop(key, None)
```

Note the relationship between the two TTLs: the hard one should be considerably larger than the soft one (here 12×), because it is your cushion during incidents. If the database goes down for 40 minutes, every key whose hard TTL has not yet expired keeps being served stale, and your site stays up while the on-call engineer fixes the real problem. That behavior — degrading to old data instead of to errors — is the difference between a scare and an incident with a status page.

When SWR and when XFetch? SWR is easier to explain, to test, and to debug, and it gives a clear guarantee: nobody waits as long as there is *something* servable. XFetch spreads refreshes better when you have a very large number of hot keys, because it does not need the in-flight task dictionary. In practice, SWR plus single-flight covers 95% of cases and is what libraries like `stale-while-revalidate-cache` in Node implement — and the HTTP header `Cache-Control: stale-while-revalidate`, standardized in RFC 5861, is exactly this pattern at the CDN level.

When Redis goes down: fail-open, timeouts, and a circuit breaker around the cache

Everything above assumes Redis answers. The uncomfortable question is: what does your endpoint do when Redis takes 5 seconds to reply or refuses connections? If the answer is "it propagates the exception," your cache — an optimization — has become a single point of failure. The golden rule: **a broken cache must never break the request**. This is called *fail-open*: if Redis fails, you behave as if it were a miss and go to the database.

```
import logging
from redis.exceptions import RedisError

logger = logging.getLogger(__name__)

# Aggressive timeouts: a cache must answer in milliseconds.
r = redis.Redis(
    decode_responses=True,
    socket_connect_timeout=0.1,   # 100 ms to connect
    socket_timeout=0.15,          # 150 ms per operation
    retry_on_timeout=False,       # retries are YOUR code's job here
)

def cache_get(key: str) -> str | None:
    try:
        return r.get(key)
    except RedisError:
        logger.warning("cache_get failed, failing open", exc_info=True)
        return None   # treated as a miss

def cache_set(key: str, value: str, ttl: int) -> None:
    try:
        r.set(key, value, ex=ttl)
    except RedisError:
        logger.warning("cache_set failed, skipping", exc_info=True)
        # don't propagate: the response was computed, it just isn't cached
```

Timeouts deserve attention: the defaults in most clients (seconds, or infinite) are absurd for a cache. If Redis normally answers in under 1 ms, waiting more than 150 ms is already a sign of serious trouble, and blocking a worker for half a second per operation multiplies the damage. With short timeouts and fail-open, a degraded Redis costs each request at most 150 extra milliseconds — not a cascade of blocked workers.

There is a trap in pure fail-open: if Redis dies completely, 100% of your traffic suddenly hits the database — the ultimate stampede. That is why it pays to wrap the cache in a **circuit breaker**: after N consecutive failures, stop trying to talk to Redis for a while (avoiding paying the timeout on every request) and, if your database

cannot take the traffic uncached, switch on an explicit degradation mode (serve simplified responses, enable aggressive rate limiting, or serve from an emergency local cache). What matters is having decided the plan B *before* the incident — and having tested it by turning Redis off in staging. If you have never run that experiment, your plan B is a hypothesis.

```
import time

class CacheBreaker:
    """Minimal circuit breaker around the cache."""
    def __init__(self, failure_threshold: int = 5, cooldown: float = 30.0):
        self.failures = 0
        self.threshold = failure_threshold
        self.cooldown = cooldown
        self.opened_at: float | None = None

    def available(self) -> bool:
        if self.opened_at is None:
            return True
        if time.monotonic() - self.opened_at >= self.cooldown:
            self.opened_at = None      # half-open: try again
            self.failures = 0
            return True
        return False                   # open: don't even try

    def record_success(self) -> None:
        self.failures = 0

    def record_failure(self) -> None:
        self.failures += 1
        if self.failures >= self.threshold:
            self.opened_at = time.monotonic()

breaker = CacheBreaker()

def cache_get_cb(key: str) -> str | None:
    if not breaker.available():
        return None                    # circuit open: straight to miss
    try:
        value = r.get(key)
        breaker.record_success()
        return value
    except RedisError:
```

```
        breaker.record_failure()
        return None
```

Two-tier caching: local memory + Redis

Redis is fast, but it is still a network hop: 0.5–1 ms in the best case, plus serialization. For very hot keys (configuration, feature flags, the category catalog) it is worth having a first tier **in the process's own memory**, with Redis as the second tier. An L1 read costs nanoseconds and completely eliminates network traffic for the most requested data.

```
# pip install cachetools
from cachetools import TTLCache

l1 = TTLCache(maxsize=10_000, ttl=30)   # 30 s: short on purpose

def get_two_tier(key: str, recompute) -> dict:
    # Tier 1: process memory (nanoseconds)
    if key in l1:
        return l1[key]

    # Tier 2: Redis (sub-millisecond)
    cached = cache_get(key)
    if cached is not None:
        value = json.loads(cached)
        l1[key] = value
        return value

    # Tier 3: database
    value = recompute()
    cache_set(key, json.dumps(value), ttl=300)
    l1[key] = value
    return value
```

The trade-off is invalidation: when data changes, you delete the key from Redis, but each instance keeps its L1 copy for up to 30 seconds. That is why the L1 TTL must be short: it is your upper bound on cross-instance inconsistency. If you need immediate L1 invalidation, you have two options. The hand-rolled one: publish deletions on a Redis pub/sub channel all instances subscribe to, and have each one drop its local entry on receipt. The native one: the **server-assisted client-side caching** Redis has offered since version 6 with `CLIENT TRACKING` over the RESP3 protocol — the server remembers which keys each connection has read and pushes an invalidation message when someone modifies them.

```
# Client-side caching with redis-py (requires RESP3, Redis >= 7.4)
# The client keeps a local cache that Redis invalidates on its own.
# redis-py exposes it as CacheConfig in recent versions:
from redis.cache import CacheConfig

r_tracked = redis.Redis(
    protocol=3,
    decode_responses=True,
    cache_config=CacheConfig(max_size=5_000),
)

value = r_tracked.get("config:flags")   # 1st time: goes to Redis
value = r_tracked.get("config:flags")   # 2nd time: served locally
# If another client runs SET config:flags, Redis pushes the
# invalidation over this same connection and the local copy is dropped.
```

Two warnings before adopting it: default-mode tracking consumes server memory (the invalidation table grows with tracked keys; there is a broadcasting mode with prefixes that costs no memory but over-invalidates), and managed services do not always support it well behind their proxies. Test it against your specific provider before betting on it; hand-rolled pub/sub remains the most portable option.

Advanced consistency: the race that "delete, don't update" doesn't cover

The invalidation section recommended deleting the key after writing to the database. That eliminates the race between two writes, but a second, subtler race remains — between a concurrent read and write:

1. Request A (a read) misses and queries the database: it gets the *old* value.
2. Request B (a write) updates the database and deletes the cache key.
3. Request A — paused by the scheduler, a GC pause, or plain bad luck — wakes up and writes into the cache the old value it read in step 1.

Result: the cache holds stale data *with no natural expiry event*, because B's invalidation landed before A's `SET`. It is a rare race (it requires a read to overlap a write exactly and also get delayed), which is why plenty of teams go years without seeing it… until the cached value is a balance or a price.

There are three mitigations, from least to most effort. First: **sensible TTLs**. If every key expires within minutes, the damage from this race is bounded; this is the actual defense of most systems. Second: **delayed double delete**, popularized by teams operating caches at very large scale: delete the key, write to the database, and schedule a second delete a few hundred milliseconds later, which sweeps away whatever stale value an overlapping read may have written:

```
async def update_product_ddd(product_id: int, changes: dict) -> None:
    key = f"product:{product_id}"
```

```
    await r.delete(key)              # 1st delete
    await save_to_db(product_id, changes)
    await asyncio.sleep(0.5)         # > p99 of a reader's (query + set)
    await r.delete(key)              # 2nd delete: sweeps the race

# In production, the sleep + delete runs outside the request:
# enqueue {"key": key, "delete_at": now + 0.5} on a queue or use
# a delayed worker, so the user's write isn't slowed down.
```

The third, for those who cannot tolerate even that window: **CDC-driven invalidation** (change data capture). Instead of the application deleting keys, a process reads the database's replication log (Debezium on PostgreSQL's WAL or MySQL's binlog) and translates every committed change into cache deletions. Because invalidation is derived from the log, it is impossible to forget to invalidate from some code path (the classic data-fix script that updates rows without going through the service), and ordering follows commit order. Facebook described this architecture in its memcache paper, and it is the norm in systems where the cache fronts critical data. The cost: extra infrastructure and an invalidation latency of tens to hundreds of milliseconds — the window does not disappear, it just becomes systematic and monitorable.

Hot keys: when the problem isn't expiration but popularity

A **hot key** is a key that concentrates a disproportionate share of traffic: the home page, the viral product, the global config every request reads. Hot keys cause two problems of their own. In a Redis cluster, each key lives on exactly one shard, so a key taking 200,000 reads per second saturates one node while the others yawn — adding shards does not help at all. And when a hot key expires, the stampede is proportional to its popularity.

To **detect** them you have three tools. With the LFU eviction policy enabled, `redis-cli --hotkeys` scans the keyspace and shows you the keys with the highest access frequency. `MONITOR` over a short window (seconds — never sustained in production: it can cost up to half your throughput) lets you sample which keys show up most. And the most sustainable: instrument your own caching layer to record per-key accesses into a histogram or an approximate top-K (a count-min-style sketch) exported to your metrics.

To **mitigate** them, three techniques that combine well. The first we have already seen: a local L1 cache, which absorbs the vast majority of reads for the hot key inside the process itself. The second: **read replicas** — routing GETs to the shard's replicas spreads read traffic, at the price of reading data with replication lag. The third, for the extreme case: **key splitting**. Instead of one `trending:home` key, write N copies (`trending:home:0` … `trending:home:7`) and have each reader pick one at random. You multiply memory and writes by N, but divide the hottest shard's traffic by N:

```
import random

N_SPLITS = 8

def get_hot(key: str) -> str | None:
    return r.get(f"{key}:{random.randrange(N_SPLITS)}")
```

```
def set_hot(key: str, value: str, ttl: int) -> None:
    pipe = r.pipeline()
    for i in range(N_SPLITS):
        # per-copy jitter: don't let all 8 expire at once
        pipe.set(f"{key}:{i}", value, ex=ttl + random.randint(0, 30))
    pipe.execute()
```

Memory and eviction: the cache fills up too

A cache with no memory limit is a time bomb: Redis will grow until the operating system kills the process. In production, always configure `maxmemory` (leave headroom over the machine's RAM: copy-on-write during RDB snapshots can temporarily double consumption) and an explicit eviction policy:

```
# redis.conf for a dedicated cache node
maxmemory 6gb
maxmemory-policy allkeys-lfu
maxmemory-samples 10        # more samples = more accurate eviction
lfu-decay-time 1            # frequency counter decays per minute
```

For a pure cache, there are two sensible policies: `allkeys-lru` (evicts the least recently used) and `allkeys-lfu` (evicts the least *frequently* used, available since Redis 4). LRU is the safe default; LFU wins when the access pattern is skewed — a few very hot keys and a long tail of lukewarm ones — because a one-off scan of cold keys will not evict your stars. Avoid the `volatile-*` variants for a cache: they only consider keys with a TTL and, if all keys have one, they behave the same but with more work; and if one day you write keys without TTLs, Redis will start returning out-of-memory errors instead of evicting.

Watch two metrics from `INFO stats`: `evicted_keys` (if it grows steadily, your cache is too small for your working set, and you are paying for misses you thought were cached) and `mem_fragmentation_ratio`. And one piece of simple hygiene that saves gigabytes: do not serialize fields you never read on the read path (blobs, long descriptions, audit fields). A cache entry is not a database row; it is the *view* your endpoint needs.

Serialization: JSON is fine, until it isn't

JSON is the right starting point: readable, universal, debuggable with `redis-cli GET`. But past a certain volume, payload size matters twice: in Redis's RAM (multiplied by the number of keys) and on the network (multiplied by reads per second). Two cheap levers:

```
import gzip, json

def serialize(value: dict) -> bytes:
    raw = json.dumps(value, separators=(",", ":")).encode()
    # compress only when it pays off: gzip costs CPU
```

```
    if len(raw) > 1024:
        return b"\x1f" + gzip.compress(raw, compresslevel=1)
    return b"\x00" + raw

def deserialize(blob: bytes) -> dict:
    if blob[0:1] == b"\x1f":
        return json.loads(gzip.decompress(blob[1:]))
    return json.loads(blob[1:])
```

The prefix byte distinguishes compressed entries from uncompressed ones, which lets you migrate gradually and compress only large payloads (compressing 200 bytes is counterproductive). On typical JSON documents, gzip at level 1 shrinks payloads 5–10× for a CPU cost measured in microseconds. If operation volume is high as well, consider **msgpack** instead of JSON: binary serialization 2–4× more compact and faster to parse, at the price of losing direct readability in `redis-cli`. Msgpack plus conditional compression is the de facto standard in high-traffic caches.

Batched reads: MGET and pipelines

An endpoint rendering a list of 30 products must not issue 30 sequential GETs: that is 30 round-trips, and network time dominates completely. Redis gives you two tools. `MGET` reads N keys in a single operation; a **pipeline** groups heterogeneous operations into a single trip:

```
def get_products_batch(ids: list[int]) -> dict[int, dict]:
    keys = [f"product:{i}" for i in ids]
    values = r.mget(keys)             # 1 round-trip for N keys

    found, missing = {}, []
    for pid, raw in zip(ids, values):
        if raw is not None:
            found[pid] = json.loads(raw)
        else:
            missing.append(pid)

    if missing:
        # one single query with WHERE id IN (...), never a loop
        rows = fetch_products_from_db(missing)
        pipe = r.pipeline()
        for pid, row in rows.items():
            pipe.set(f"product:{pid}", json.dumps(row),
                     ex=300 + random.randint(0, 60))
        pipe.execute()                # 1 round-trip for the sets
        found.update(rows)
```

```
    return found
```

The structure is always the same: one MGET, split hits and misses, *one* batched database query for the misses, and one pipeline to write them back. A detail for clusters: MGET requires all keys to land in the same slot; if you use Redis Cluster, group with hash tags (`product:{catalog}:123`) or let the client split the MGET per slot (good clients do it on their own).

Observability: if you don't measure the hit ratio, you don't have a cache — you have faith

Three numbers tell a cache's whole story: the **hit ratio** (what fraction of reads is served from cache), the **recompute latency** (what each miss costs), and the **stampedes avoided** (how many times the defenses did their job). Instrument them in your caching layer; do not rely solely on Redis server metrics — Redis's global hit ratio blends all your data types together and hides the one that is misbehaving:

```
from prometheus_client import Counter, Histogram

CACHE_OPS = Counter(
    "cache_ops_total",
    "Cache operations by result",
    ["cache_name", "result"],   # result: hit | miss | stale | error
)
RECOMPUTE_SECONDS = Histogram(
    "cache_recompute_seconds",
    "Recompute duration after a miss",
    ["cache_name"],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1, 2.5, 5],
)
STAMPEDE_WAITS = Counter(
    "cache_stampede_waits_total",
    "Requests that waited on the lock instead of recomputing",
    ["cache_name"],
)

def get_product_observed(product_id: int) -> dict:
    key = f"product:{product_id}"
    cached = cache_get(key)
    if cached is not None:
        CACHE_OPS.labels("products", "hit").inc()
        return json.loads(cached)

    CACHE_OPS.labels("products", "miss").inc()
```

```
    with RECOMPUTE_SECONDS.labels("products").time():
        product = fetch_product_from_db(product_id)
    cache_set(key, json.dumps(product), ttl=300)
    return product
```

With that in place, the per-data-type hit ratio is one PromQL query: `sum(rate(cache_ops_total{result="hit"}[5m])) by (cache_name) / sum(rate(cache_ops_total{result=~"hit|miss"}[5m])) by (cache_name)`. Alert on *changes*, not absolute values: a 60% hit ratio may be excellent for one data type and disastrous for another; what is always a bad sign is a 20-point drop after a deploy (did someone change the key format so everything is now a miss?) or `cache_ops_total{result="error"}` ceasing to be zero. Add a panel with `evicted_keys` and Redis's own memory usage too: a hit ratio that slowly degrades while evictions climb is the unmistakable signature of a cache that has outgrown its size.

Cache warming: don't go to production with a cold cache

Every deploy with a local cache, every Redis failover, and every accidental flush leaves you with a cold cache: the first minute of traffic hits the database at maximum miss rate. If your database can take the traffic uncached, you need nothing special (and you should know that with certainty, not assume it). If it cannot, pre-warm: before routing traffic to the new version, a script walks the N most popular keys from the previous day (that list comes from your metrics or from a popularity `ZSET`) and recomputes them in a controlled way, with bounded concurrency:

```
async def warm_cache(top_keys: list[str], concurrency: int = 8) -> None:
    sem = asyncio.Semaphore(concurrency)   # don't stampede your own DB

    async def warm_one(key: str) -> None:
        async with sem:
            if await r.exists(key):
                return                     # already warm
            entity_id = int(key.split(":")[1])
            data = await fetch_product_from_db_async(entity_id)
            await r.set(key, json.dumps(data),
                        ex=300 + random.randint(0, 60))

    await asyncio.gather(*(warm_one(k) for k in top_keys))
```

The semaphore is the important part: a warmer with unbounded concurrency is, quite literally, a self-inflicted stampede. And do not try to pre-warm everything: the top 5% of keys usually covers the large majority of traffic; the long tail warms itself.

Key design: naming, schema versions, and multi-tenancy

Key design looks like a cosmetic detail until it causes your first incident. Three decisions worth making on day one rather than improvising later.

Naming convention. Adopt a hierarchical format with a fixed separator and document it: `app:type:identifier` — for example `shop:product:12345` or `shop:listing:electronics:page:3`. The hierarchy does three things for you: it groups entries visually in any inspection tool, it enables invalidating whole families via prefix-versioned keys, and it prevents collisions when two services share the same Redis (something you should not do with large caches, but which happens everywhere). Keep keys short but readable — every key byte is multiplied by millions of entries and travels with every operation — and never build them from unsanitized data: a key constructed from unnormalized user input (`search:{query}` with stray spaces, mixed case, or control characters) fragments the cache into duplicate variants and, in the worst case, lets an attacker inflate your memory with artificial misses.

Schema version. The most treacherous cache bug is not caused by Redis but by a deploy: version N+1 of your code reads an entry serialized by version N with a different shape (a renamed field, a changed type) and blows up — or worse, does not blow up and processes misinterpreted data. The fix costs one prefix: include the serialized schema's version in the key, and bump it every time you change the shape of the cached data:

```
SCHEMA_VERSION = "v3"   # bump when the data shape changes

def product_key(product_id: int) -> str:
    return f"shop:product:{SCHEMA_VERSION}:{product_id}"

# Deploying v4 simply stops reading the v3 keys:
# controlled cold misses instead of deserialization errors.
# Orphaned v3 keys quietly expire via TTL.
```

The effect of bumping the version is an instant cold cache for that data type, so combine it with what you already know: deploy during off-peak hours or pre-warm before routing traffic. It is a planned, visible cost — infinitely preferable to the random deserialization error that shows up only for keys written by the previous version and vanishes as they expire, leaving you an intermittent stack trace that is impossible to reproduce.

Multi-tenancy and per-user data. If your application serves multiple tenants or caches data that depends on the user, the tenant or the user *must be in the key*: `shop:{tenant}:product:v3:123`. Getting this wrong is one of the most common cache-related security incidents: an endpoint that caches its response under `profile:latest` instead of `profile:{user_id}:latest` and serves user A's data to user B for the duration of the TTL. The systematic review is easy to state: for every key, ask "what exactly does this value depend on?" — and everything that shows up in the answer (user, tenant, language, currency, active feature flags) must show up in the key. If the dependency list gets long, that is a sign the data is a poor candidate for a shared cache: cache one level lower (the user-independent database query) instead of the full, user-dependent response.

With tenants in the key, administrative operations also become trivial: invalidating everything belonging to one tenant after a migration means incrementing its version (`tenant:acme:version` as part of the prefix — the versioned-keys trick we saw earlier), not running a dangerous `SCAN` across the whole keyspace. And on Redis

Cluster, putting the tenant in a hash tag (`shop:{acme}:...`) guarantees its keys live together in the same slot, with the trade-off that one huge tenant can unbalance a shard: choose the hash tag based on your actual size distribution, not aesthetics.

One last cross-cutting recommendation: treat these conventions as code, not oral tradition. A `cache_keys.py` module (or its TypeScript equivalent) with key-builder functions — never loose f-strings scattered through the codebase — turns all the rules above into something the type checker and code review can enforce, and makes bumping a schema version a one-line change in a single place.

Choosing the engine: Redis, Valkey, or Memcached

Every pattern in this guide says "Redis," but in 2026 that word hides a three-way decision. Some context: in March 2024 Redis left the BSD license for source-available licenses, and the community answered the same month with **Valkey**, a fork hosted at the Linux Foundation and backed by AWS, Google, and Oracle, which started from Redis 7.2.4 and keeps the BSD license. Redis countered in 2025: as of version 8.0 it once again offers an open source license (AGPLv3, tri-licensed alongside SSPL and RSAL). The practical result is that two mature, protocol-compatible projects now coexist, each with its own release cadence.

What does that mean for your cache? Less than it seems and more than Internet debates admit. **Everything you have read in this guide works identically on both**: cache-aside, SET NX EX, Lua scripts, pipelines, LFU eviction policies, client-side caching. If you use a managed service, the decision has probably been made by your provider already: AWS ElastiCache and MemoryDB default to Valkey; other providers offer both. If you self-host, the real criteria are the license (whether AGPL fits your dependency policy) and which recent features you need from each development line — not performance: for the caching use case, both have headroom to spare.

And **Memcached**? Still alive, and still owning a legitimate niche: it is multithreaded (it scales vertically on a single node better than one Redis process), it does exactly one thing — LRU caching with TTLs — and it does it with an operational simplicity that is hard to match. If your need is precisely "cache bytes with a TTL behind a load balancer" and nothing else, Memcached is a perfectly serious answer. You rule it out the moment you need any other piece of this guide that depends on server-side data structures or atomicity: locks with SET NX EX, Lua scripts, pub/sub for L1 invalidation, popularity ZSETs for the warmer. In modern cache-aside, those pieces show up early — and that is the quiet reason Redis/Valkey became the standard: not because it caches better, but because the toolbox around the cache is more complete.

The actionable advice: pick the engine based on constraints (provider, license, operations), not forum benchmarks; write your caching layer against your own interface (the fail-open `cache_get` / `cache_set` functions we already built) so that switching engines is a configuration change; and if you are on self-hosted Redis 7.x today without a support contract, plan a calm migration to Redis 8+ or Valkey — both are direct upgrades from 7.2, and staying on an unpatched version is the one choice that is bad in every scenario.

How to test a cache (and its stampede defenses)

Cache code has a reputation for being hard to test because it mixes networking, time, and concurrency. It is tamed with two tools. For fast unit tests, **fakeredis** implements the Redis API in memory — TTLs and Lua

scripts included — without spinning up any process. For integration tests that must be faithful (pipelines, cluster behavior, real expiration semantics), **Testcontainers** starts an ephemeral Redis in Docker per test session.

The first thing to test is not the hits — those work on their own — but the behaviors that only show up under pressure. This test verifies the central property of this whole guide: *N concurrent reads of a cold key must produce exactly one recompute*:

```
import asyncio
import pytest

@pytest.mark.asyncio
async def test_stampede_produces_single_recompute(cache):
    calls = 0

    async def slow_recompute():
        nonlocal calls
        calls += 1
        await asyncio.sleep(0.2)      # simulates a slow query
        return {"id": 1, "name": "hot product"}

    # 100 concurrent requests against a cold key
    results = await asyncio.gather(*[
        cache.get("product:1", slow_recompute) for _ in range(100)
    ])

    assert calls == 1                          # a single recompute
    assert all(r == results[0] for r in results)

@pytest.mark.asyncio
async def test_cache_failure_fails_open(cache, broken_redis):
    # with Redis down, the request degrades to the DB, not to an error
    result = await cache.get("product:1", lambda: {"id": 1})
    assert result == {"id": 1}

@pytest.mark.asyncio
async def test_negative_result_is_cached(cache, db_spy):
    for _ in range(10):
        assert await cache.get("product:999", db_spy.fetch_missing) is None
    assert db_spy.calls == 1          # "not found" gets cached too
```

The other two tests cover the failure modes that actually wake someone up at 3 am: a broken Redis must not break requests (fail-open), and negative results must be cached. Round out the suite with an expiration test using a fake clock or millisecond TTLs, and an invalidation test (write → delete → next read recomputes). If

you also want to validate behavior under real load, a `k6` or `locust` run against staging, killing the hot key mid-test, shows you the stampede — or its absence — right on the graphs.

A complete TypeScript implementation

To close out the code, here is a class that stacks every defense in this guide — single-flight, distributed lock, jittered TTL, negative caching, fail-open, and metrics — behind a single `getOrCompute()` call. It is the TypeScript equivalent of what we built piece by piece in Python, ready to adapt:

```
import Redis from "ioredis";
import { randomUUID } from "crypto";

interface CacheOptions {
  ttlSeconds: number;
  jitterSeconds?: number;      // defaults to 20% of the TTL
  lockTtlSeconds?: number;     // defaults to 10 s
  negativeTtlSeconds?: number; // defaults to 30 s
}

const RELEASE_LUA = `
if redis.call("get", KEYS[1]) == ARGV[1] then
  return redis.call("del", KEYS[1])
end
return 0`;

const NEGATIVE = "__NULL__";   // sentinel for "not found"

export class ProductionCache {
  private inFlight = new Map<string, Promise<unknown>>();

  constructor(
    private redis: Redis,
    private onMetric: (event: string) => void = () => {},
  ) {}

  async getOrCompute<T>(
    key: string,
    compute: () => Promise<T | null>,
    opts: CacheOptions,
  ): Promise<T | null> {
    // 1) single-flight: local dedupe per key
    const existing = this.inFlight.get(key);
    if (existing) return existing as Promise<T | null>;
```

```
  const p = this.load(key, compute, opts)
    .finally(() => this.inFlight.delete(key));
  this.inFlight.set(key, p);
  return p;
}

private async load<T>(
  key: string,
  compute: () => Promise<T | null>,
  opts: CacheOptions,
): Promise<T | null> {
  // 2) read with fail-open
  const cached = await this.safeGet(key);
  if (cached !== undefined) {
    if (cached === NEGATIVE) { this.onMetric("negative_hit"); return null; }
    if (cached !== null) { this.onMetric("hit"); return JSON.parse(cached) as T; }
  }
  this.onMetric("miss");

  // 3) distributed lock: only one process recomputes
  const lockKey = `lock:${key}`;
  const token = randomUUID();
  const lockTtl = opts.lockTtlSeconds ?? 10;
  const acquired = await this.redis
    .set(lockKey, token, "EX", lockTtl, "NX")
    .catch(() => null);            // Redis broken: recompute without a lock

  if (acquired) {
    try {
      const value = await compute();
      await this.safeSet(key, value, opts);
      return value;
    } finally {
      await this.redis
        .eval(RELEASE_LUA, 1, lockKey, token)
        .catch(() => {});
    }
  }

  // 4) no lock: bounded wait for someone else to repopulate
  const deadline = Date.now() + lockTtl * 1000;
  while (Date.now() < deadline) {
```

```
      await new Promise((res) => setTimeout(res, 50));
      const repopulated = await this.safeGet(key);
      if (repopulated === NEGATIVE) return null;
      if (repopulated) {
        this.onMetric("stampede_wait_hit");
        return JSON.parse(repopulated) as T;
      }
    }
    this.onMetric("stampede_wait_timeout");
    return compute();               // last resort: direct query

  }

  private async safeGet(key: string): Promise<string | null | undefined> {
    try { return await this.redis.get(key); }
    catch { this.onMetric("error"); return undefined; }
  }

  private async safeSet<T>(
    key: string, value: T | null, opts: CacheOptions,
  ): Promise<void> {
    const jitter = opts.jitterSeconds ?? Math.ceil(opts.ttlSeconds * 0.2);
    const ttl = opts.ttlSeconds + Math.floor(Math.random() * jitter);
    try {
      if (value === null) {
        // 5) negative caching with a short TTL
        await this.redis.set(key, NEGATIVE, "EX", opts.negativeTtlSeconds ?? 30);
      } else {
        await this.redis.set(key, JSON.stringify(value), "EX", ttl);
      }
    } catch { this.onMetric("error"); }
  }
}

// Usage
const cache = new ProductionCache(new Redis(), (e) => metrics.inc(e));

const product = await cache.getOrCompute(
  `product:${id}`,
  () => db.products.findById(id),   // returns null when not found
  { ttlSeconds: 300 },
);
```

It is worth reading through slowly once: each numbered block maps to a section of this guide, and the ordering of the defenses is not accidental — single-flight wraps everything (it is the cheapest), the read comes before the lock (the happy path must not pay for coordination), and every Redis call is wrapped in fail-open (the cache never breaks the request).

A worked example: sizing the cache for a catalog

Let's ground the numbers with a concrete service: an e-commerce catalog with 2 million products, 8,000 requests per second at peak, and a product query that averages 25 ms in PostgreSQL (with joins to prices, stock, and images).

How much memory? The serialized product view is about 2 KB in compact JSON (600 bytes with msgpack + gzip). Caching the full catalog would cost 4 GB in JSON — but you don't need to: access analysis shows the top 10% of products takes 92% of the reads. With `maxmemory 1gb` and `allkeys-lfu`, the cache converges toward that working set on its own: LFU evicts the long tail and protects the stars. General rule: size for the measured working set, not the full dataset, and let the eviction policy do its job.

Which TTL? Prices change via an hourly batch process plus the occasional manual edit. Base TTL of 300 s + 60 of jitter: the maximum TTL-driven inconsistency is 6 minutes, acceptable for product pages; manual edits delete the key on save, so they show up within seconds. For the cart and real-time stock: *not cached at all*. Deciding what does NOT go in the cache is half the design.

Can the database survive a flush? 8,000 rps × 25 ms = 200 connections busy on products alone if the cache vanishes: it cannot. Operational conclusion: the cache is *critical*, so there is a warmer for deploys (top 50,000 keys, 40 seconds at concurrency 16), SWR with a one-hour hard TTL as a cushion against database incidents, and an alert if the products hit ratio drops below 85%. With everything in place, PostgreSQL sees about 300 queries per second (the 8,000 minus an effective ~96% hit ratio), the endpoint's p99 sits at 4 ms, and any key expiring — including the day's viral product — is a non-event.

Production checklist

Before calling a caching layer done, run through this: every TTL carries jitter; single-flight wraps every read; expensive keys have a distributed lock or SWR; negative results are cached with a short TTL; invalidation deletes (and where races would hurt, delayed double delete or CDC); Redis timeouts are in milliseconds and every cache failure degrades to a miss; `maxmemory` and the eviction policy are explicitly configured; hit ratio, errors, and evictions have a dashboard and an alert; there is a test proving that 100 concurrent reads produce one recompute; and someone has switched Redis off in staging to see what happens. If you can tick all ten boxes, your cache is no longer the fragile part of the system — it is the part that holds when everything else shakes.